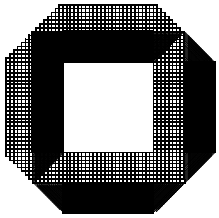


UNIVERSITÄT KARLSRUHE
FAKULTÄT FÜR INFORMATIK

Postfach 6980, D-76128 Karlsruhe

Ausgewählte Kapitel der Implementierung objektorientierter Datenbanksysteme+

Institut für Programmstrukturen und Datenorganisation
Universität Karlsruhe
Fakultät für Informatik
D-76128 Karlsruhe, F. R. G.

Editiert von

Christoph Kilger, Hans-Dirk Walter und Andreas Zachmann

Interner Bericht Nr. 18/94

+ Dieser Bericht faßt die Ausarbeitungen eines Seminars gleichen Titels zusammen, das im Sommersemester 1994 am Institut für Programmstrukturen und Datenorganisation gehalten wurde.

Zusammenfassung

Objektorientierte Datenbanksysteme bieten gegenüber klassischen Datenbanksystemen, wie bspw. den Relationalen Systemen oder den Netzwerk-Systemen, eine erweiterte Funktionalität, die in vielen Anwendungsbereichen benötigt wird. Beispiele für Anwendungen objektorientierter Datenbanksysteme sind technische Informationssysteme, Entwurfsdatenbanken oder Multi-Media-Datenbanken.
Die erweiterte Funktionalität objektorientierter Datenbanksysteme erfordert neue Implementierungskonzepte. Das Ziel dieses Seminars, das im Sommersemester 1994 am Institut für Programmstrukturen und Datenorganisation der Universität Karlsruhe gehalten wurde, ist die Untersuchung ausgewählter Kapitel bei der Implementierung objektorientierter Datenbanksysteme.
Das Seminar ist in vier Teile gegliedert. Im ersten Teil wird — quasi aus Benutzersicht — in die Konzepte objektorientierter Datenmodelle eingeführt. Aus Implementierungssicht stellen diese Konzepte die Spezifikation für ein zu implementierendes objektorientiertes Datenbanksystem dar. Der zweite Teil beschäftigt sich mit der Speicherung von Objekten auf dem Hintergrundspeicher sowie mit dem Umgang mit persistenten Objekten zur Laufzeit. In Teil drei werden zwei spezielle Synchronisations-Verfahren vorgestellt, die im Hinblick auf objektorientierte Datenbanksysteme entworfen wurden. In Teil vier sind zwei Themen zusammengefaßt, die einen Überblick geben über Evolution in objektorientierten Datenbanksystemen und über aktive objektorientierte Datenbanksysteme.

Inhaltsverzeichnis

Teil I

Einführung

Kapitel 1

Objektorientierte Datenmodelle (*Bethina Schmitt*)

1.1 Einleitung

Am weitesten verbreitet sind heutzutage sogenannte relationale Datenbanksysteme. Sie erlauben es, die Strukturen der potentiell anfallenden Daten mittels Tupeln zu definieren und diese zu Relationen, das sind Mengen solcher Tupel, zusammenzufassen. Ein Tupel besteht aus einer Reihe von Attributen, deren Werte aus einigen wenigen vordefinierten Wertebereichen stammen müssen. Diese tabellenartigen Konstrukte können dann unter Anwendung fest vorgegebener generischer Funktionen manipuliert werden. Der "Charme" des Relationenmodells liegt in seiner Einfachheit, die eine weitgehende Formalisierung ermöglichte.

So geeignet solche Systeme für ihre ursprünglichen Anwendungsgebiete auch sind, zur Beschreibung der im Entwurfs- und Produktionsprozeß eines Betriebs anfallenden Daten stellen sie nicht genügend ausdrucksstarke Mittel zur Verfügung. Zwar sind auch hier große Datenmengen zu bewältigen, die sich aber durch eine viel reichhaltigere, mittels Tupelmengen nur unzulänglich zu beschreibende Struktur auszeichnen, und die auf weitaus spezifischere und vielfältigere Art manipuliert werden müssen, als es die generischen Funktionen relationaler Systeme ermöglichen. Insgesamt ist die Welt des industriellen Betriebes zu komplex, um durch so einfache, wenig Abstraktionsmöglichkeiten bietende Modelle, wie das Relationenmodell ausreichend erfaßt zu werden.

In diese Lücke stoßen objektorientierte Konzepte, die in den unterschiedlichsten Bereichen der Informationsverarbeitung mit Enthusiasmus aufgenommen wurden. Bei der Erstellung komplexer Programmsysteme mit höchsten Ansprüchen an Sicherheit, Wartbarkeit und Erweiterbarkeit haben sie sich schon glänzend bewährt. Objektorientierte Modelle bieten hohe Expressivität, wobei sie immer noch auf einer relativ einfachen Begriffswelt basieren. Der Grundgedanke der Objektorientierung besteht darin, Entitäten aus der Realität, z.B. einen Gegenstand oder, allgemeiner, einen Sachverhalt, als Individuum mir eigener, unverwechselbarer Identität in die Informationswelt aufzunehmen, und dies als Objekt durch einen sich über die Zeit ändernden Zustand und ein spezifisches Verhaltensmuster zu beschreiben. Es herrscht allgemeiner Konsens, daß objektorientierte Modellierungsmethoden für technische Anwendungen allen anderen heute bekannten Modellen, wie hierarchischen, netzwerkartigen oder relationalen Modellen, vorzuziehen sind.

Im folgenden soll nun eine Definition eines objektorientierten Datenbanksystems versucht werden, was gar nicht so einfach ist, da es im Gegensatz zu relationalen Datenbanksystemen für objektorientierte bislang keine eindeutige Spezifikation gibt. D.h. nicht, daß kein vollständiges objektorientiertes Datenmodell existiert, sondern daß so viele verschiedene Vorschläge darüber in der Literatur zu finden sind, man sich aber noch nicht auf eine einzige Klassifikationsmölichkeit einigen konnte.

In Anlehnung an [ABD+89] werden wir hier die Merkmale von objektorientierten Datenbanksystemen in 2 Kategorien einteilen:

- obligatorische, zwingend erforderliche Eigenschaften
- optionale, wünschenswerte Eigenschaften

Entsprechend ist der Artikel im folgenden auch gegliedert:

Kapitel 2 zählt alle Eigenschaften auf, die ein Datenbanksystem besitzen muß, damit es sich auch objektorientiert nennen darf. Und in Kapitel 3 werden zusätzliche Merkmale besprochen, die ein System besser machen, aber nicht unbedingt erforderlich sind.

1.2 Obligatorische Eigenschaften

Ein objektorientiertes Datenbanksystem muß zwei Kriterien genügen: Es sollte ein Datenbanksystem sein, und es sollte ein objektorientiertes System sein. Aus dem ersten Kriterium ergeben sich fünf Forderungen: nach Persistenz, Hintergrundspeichermanagement, Nebenläufigkeit, Recovery und nach einer Ad Hoc Anfragesprache. Aus dem zweiten ergeben sich acht Forderungen: nach komplexen Objekten, Objektgleichheit, Kapselung, Typisierung, Vererbung, Überladen von Operationen, Turingvollständigkeit und Erweiterbarkeit. Wir wollen uns hier hauptsächlich mit den acht objektorientierten Charakteristika beschäftigen, die anderen sollten hinreichend von traditionellen Datenbanksystemen bekannt sein.

1.2.1 Komplexe Objekte

Ein Objekt ist eine Gesamtheit von Datenspeicher und Verhaltensrepertoire mit einer zustandsunabhängigen Identität. Man gibt hier also mit Absicht die bisher übliche strikte Trennung zwischen Daten und Abläufen auf und handhabt sie als Einheit.

Jedes Objekt besitzt eine innere Struktur in Form von Attributen, die von außen nicht einsehbar sind. Die aktuelle Belegung dieser Attribute macht den Objektzustand aus. Neben atomaren Werten (Zahlen, Zeichenketten, etc) können die Attribute eines Objektes Referenzen auf andere Objekte enthalten, die dann als Unterobjekte dieses Objektes bezeichnet werden. Diesen Sachverhalt bezeichnet man als Aggregierung. Auf diese Weise können beliebig komplexe Objektnetze entstehen.

Atomare Typen: Ausprägungen atomarer Typen sind Werte. Werte stellen sich selbst beschreibende Daten ohne Identifikator oder andere Zusatzangeben dar, die nicht verändert werden können. Sie können nicht eigenständig in einer Datenbasis existieren.

Komplexe Typen: Ausprägungen komplexer Typen können als Objekte mit eigener Identität eigenständig in Datenbasen existieren oder als unselbständiger Bestandteil (komplexer Wert) der Struktur eines Objektes auftreten. Ihr Aufbau kann beispielweise mit Hilfe der folgenden drei Typkonstruktoren geschehen:

- *Tupeltypen* $[A_1 : t_1, ..., A_n : t_n]$: Jede Ausprägung eines Tupeltyps ist ein Tupel mit den speziellen Attributen und einer Belegung dieser Attribute. Die Belegung eines Attributes A_i ist auf einen Wert oder ein Objekt des Typs t_i eingeschränkt.

- *Mengentypen* $\{t\}$: Eine Ausprägung dieses Typs ist eine Menge von Objekten des Typs t.

- *Listentypen* $\langle t \rangle$: Eine Ausprägung dieses Typs ist eine Liste von Objekten des Typs t, d.h. hier besteht eine Ordnung auf den Elementen.

Diese Typkonstruktoren müssen orthogonal sein: Jeder beliebige Konstruktor darf auf beliebige Typen angewendet werden. Die Konstruktoren des relationalen Modells sind zum Vergleich nicht orthogonal, weil der Mengenkonstruktor nur auf Tupel, und der Tupelkonstruktor nur auf atomare Werte angewendet werden darf.

1.2.2 Objektidentität und Objektgleichheit

Jedes Objekt besitzt eine eindeutige Identität. Die Identität eines Objektes ist während seiner Lebensdauer unveränderlich. Idealerweise gelten außerdem folgende Eigenshaften:

- Die Identität eines Objektes ist unabhängig vom Speicherungsort des Objektes und von Objektzustand.

- Nach dem Löschen eines Objektes aus dem System wird es kein anderes Objekt geben, das jemals die gleiche Identität wie die des gelöschten Objektes aufweist.

Dieser Identitätsbegriff ist grundverschieden von den Identifikationskonzepten anderer Datenmodelle. Die Identität ist hierbei erstmals eine Eigenschaft eines Informationselementes, die zu dessen anderen Eigenschaften völlig orthogonal ist. Man betrachte zum Vergleich beispielsweise das relationale Modell, in dem ein Tupel eindeutig durch eine Teilmenge seiner Attributwerte identifiziert wird.

Eng verbunden mit der Frage nach der Identifizierung von Objekten ist die Frage nach Vergleichsmöglichkeiten für Objekte, also Objektgleichheit. Im Unterschied zu anderen Datenmodellen muß man in der Objektorientierung zwischen verschiedenen Objektgleichheits-Begriffen unterscheiden. Der Grund hierfür liegt in der Tatsache, daß zur Definition der Gleichheit nun zwei Ausgangskonzepte zur Verfügung stehen: zum einen die Identifikatoren der Objekte (Objektgleichheit), zum anderen die Werte der Objektattribute (Wertegleichheit).

Beispiel: Eine Person hat einen Namen, ein Alter und eine Menge von Kindern. Angenommen Peter und Susan haben beide einen 15-jährigen Sohn namens John. Im wirklichen Leben können nun zwei Konstellationen auftreten: Susan und Peter sind die Eltern des gleichen Kindes, oder es sind zwei Jungen beteiligt. In einem System ohne Identitäten sieht die Repräsentation folgendermaßen aus:

(peter, 40, { (john, 15, { }) })
(susan, 41, { (john, 15, { }) })

Somit gibt es keine Möglichkeit auszudrücken, ob Susan und Peter die Eltern des gleichen Kindes sind. In einem Modell, das auf Identität basiert, kann man beide Situationen modellieren.

Angenommen Susan und Peter sind tatsächlich die Eltern von John. In diesem Fall werden alle Änderungen, die Susans Sohn betreffen, am Objekt John vorgenommen und somit auch an Peters Sohn. In einem werte-basierten System müßten beide Unterobjekte getrennt aktualisiert werden. Objektidentiät ist also ein sehr mächtiges Werkzeug bei der Manipulation von Mengen, Tupeln und rekursiv definierten komplexen Objekten.

1.2.3 Kapselung

Kapselung bedeutet das Verbergen des inneren Zustands eines Objektes und der Implementierung seiner Operationen (Algorithmen, Hilfsdatenstrukturen, etc.) vor dem Benutzer (information hiding). Kapselung garantiert also, daß externe Nutzer eines Objektes nur über dessen Verhaltensrepertoire mit ihm interagieren können. Die direkte Manipulation interner Strukturen wird unterbunden; die Schnittstelle und die dahinterstehende Implementierung für das Verhalten sind voneinander entkoppelt, was Änderungen der Implementierung ohne Modifikation der Schnittstelle ermöglicht.

In der objektorientierten Programmierung liegt der Schwerpunkt auf dem Objektverhalten. Damit haben Objektschnittstelle und Kapselung eine zentrale Bedeutung. Dies erklärt auch die wachsende Bedeutung sogenannter Objektbibliotheken. Für den Datenbankeinsatz spielt hingegen auch die Struktur der Objekte eine wesentliche Rolle: Ohne Struktur kann es schlechterdings keine Implementierung der Operationen geben, und ohne diese sind die aufgestellten Schnittstellen ohne Wert. Während dem reinen (End-)Benutzer also die Kenntnis der Schnittstellen genügt, müssen sich Datenbankentwerfer und Systemverwalter sehr wohl mit der Objektstrukturierung befassen.

Beispiel: Betrachten wir einmal einen Quader. Die interne Repräsentation eines Quader-Objektes bestehe beispielsweise aus den Koordinaten der acht Begrenzungspunkte. Die Objektkapselung verbietet aber den direkten Zugriff auf diese Koordinaten, was ja auch leicht zu einem inkonsistenten Zustand der Quader-Repräsentation führen könnte. Vielmehr muß jeder Zugriff und jede Objektmanipulation über die Menge der Schnittstellenoperationen durchgeführt werden. In diesem Fall stehen als Mutatoren die wohlbekannten geometrischen Transformationsoperationen *rotate*, *scale* und *translate* zur Verfügung; als Observierer gibt es weitere Zustandsabfrage-Operationen wie *volume*.

In Anlehnung an GOM [KKM+92] würde die Typdefinition etwa folgendermaßen aussehen:

```
type Quader supertype ANY is

public rotate, scale, translate, volume
body [p1, p2, p3, p4, p5, p6, p7, p8: Punkt;]
operations
    declare volume: ⟶ float;
    declare rotate: float, char ⟶ void;
    declare scale: float ⟶ void;
    declare translate: float, float, float ⟶ void;
implementation
    define volume is
    ...
    end define volume;
    define rotate(angle, axis) is
```

```
        ...
        end define rotate;
        ...
end type Quader;
```

In der **public**-Klausel werden die Operationen aufgeführt, die den sogenannten Klienten des Typs zum Zugriff auf Objekte bzw. zum Modifizieren des Zustands der Objekte des jeweiligen Typs zur Verfügung gestellt werden. Insofern stellt die **public**-Klausel die Spezifikation der Schnittstelle nach außen dar.

In der **body**-Klausel wird die interne Struktur des Objekttyps genauer spezifiziert, in diesem Fall ist Quader ein tupelstrukturierter Typ.

In der **operations**-Klausel werden die abstrakten Signaturen der dem Typ zugeordneten Operationen aufgeführt. Allerdings können einige der hier aufgeführten Operationen auch *private* Operationen sein, die nicht in der **public**-Klausel aufgeführt werden. Die privaten Operationen werden lediglich verwendet, um die Implementierung der öffentlichen Operationen modular durchführen zu können.

Zu den in der **operations**-Klausel aufgeführten Operationen wird die Realisierung (Programmierung) im **implementation**-Teil der Typdefinition vorgenommen.

1.2.4 Typprüfung

Ein Objekttyp spezifiziert die Struktur, die Schnittstelle und das Verhalten von Objekten. Dabei wird die Struktur in Form von Attributen festgelegt; die Schnittstelle besteht aus der Deklaration von ausführbaren Operationen. Das Verhalten ist die Realisierung dieser Operationen mit programmiersprachlichen Mitteln. Die Bedeutung des Objekttyps liegt neben seiner Funktion als Objektschablone bei der Typprüfung, die auch für Datenbanksysteme eine gewichtige Rolle spielt. Mit Typprüfung ist gemeint, daß man entweder zur Übersetzungszeit oder zur Laufzeit die Typkonsistenz von Ausdrücken des Programms (also die Verträglichkeit der im Ausdruck vorkommenden Objekte) sicherstellt. Dabei nimmt der Übersetzer die Rolle eines Benutzers ein und beachtet bei der Sicherstellung der Typkonsistenz nur die Schnittstellen.

Dem Zeitpunkt zur Feststellung der Typisierung und zur Sicherstellung der Typkonsistenz entsprechend unterscheidet man:

- *schwache Typisierung*: Ein einzelnes Objekt gehört während seiner Lebensdauer stets ein und demselben Typ an; Variablen und Objektattribute können jedoch zu unterschiedlichen Zeitpunkten uneingeschränkt Objekte unterschiedlichen Typs referenzieren. Die Typkonsistenz von Ausdrücken eines Programms ist daher erst zur Laufzeit unmittelbar vor der Auswertung feststellbar.

- *strenge Typisierung*: Auch hier gehört ein einzelnes Objekt immer dem gleichen Typ an; Variablen und Objektattribute können immer noch zu unterschiedlichen Zeitpunkten Objekte unterschiedlichen Typs referenzieren. Diese wechselnde Referenzierbarkeit wird jedoch derart eingeschränkt, daß man bereits zur Übersetzungszeit Typkonsistenz sicherstellen kann. Der tatsächliche Typ des Wertes wird allerdings erst zur Laufzeit festgestellt.

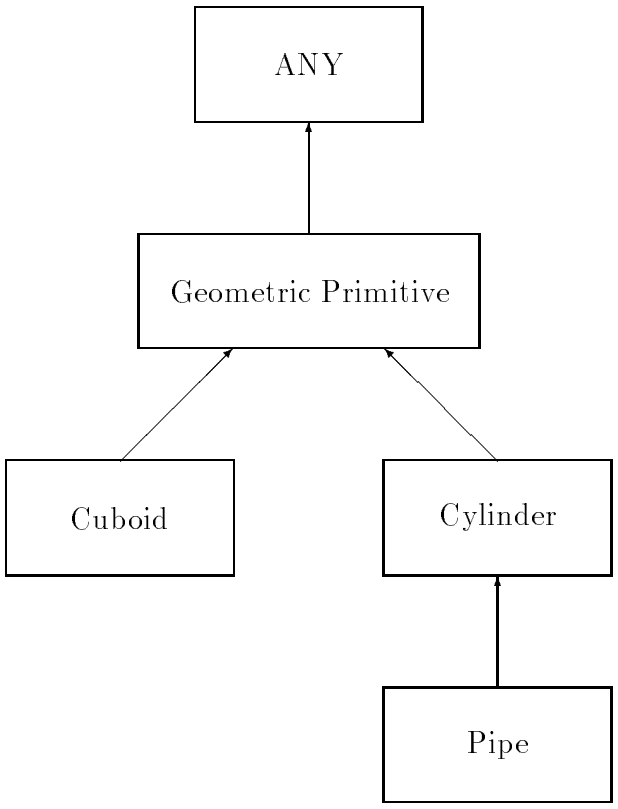
- *strikte Typisierung*: Diese Typisierung gehorcht der Forderung, daß die tatsächlichen Typen aller Objekte bereits zur Übersetzungszeit ermittelt werden können. Die Typisierung ist also zu allen Zeitpunkten dieselbe.

1.2.5 Vererbung

Generalisierung und (einfache) Vererbung: Objekttypen lassen sich in einem gerichteten azyklischen Graphen anordnen. Ein Typ erbt von seinem Obertypen dessen strukturelle und verhaltensmäßige Eigenschaften. Ein Objekttyp vererbt an alle seine Untertypen alle auf ihm definierten strukturellen und verhaltensmäßigen Eigenschaften. Umgekehrt gesehen werden die Untertypen zu einem Obertypen verallgemeinert (generalisiert).

Bezüglich des Typgraphen erlassen wir zunächst die Beschränkung, daß jeder Typ höchstens einen direkten Obertyp besitzen darf. Dieses Prinzip nennt man auch Einfachvererbung oder single inheritance. Mit dieser Bildungsregel entsteht eine Typhierarchie. Diese Hierarchie kann mehrstufig sein; ein Typ kann somit mehrere (indirekte) Obertypen besitzen. Wurzel der Typhierarchie ist ein vordefinierter Typ, z.B. ANY. Damit gilt natürlich, daß jedes Objekt insbesondere vom Typ ANY ist.

Wir wollen Vererbung anhand folgender Typhierarchie erläutern:



Der Typ *GeometricPrimitive* ist als direkter Untertyp der gemeinsamen Wurzel *ANY* definiert. *GeometricPrimitive* ist weiter verfeinert zu den Typen *Cuboid* und *Cylinder*, wobei *Cylinder* selbst wieder einen Untertyp *Pipe* besitzt.

Die Typdefinitionsrahmen für *GeometricPrimitive* und *Cylinder* könnten etwa wie in Abbildung 1.1 dargestellt aussehen. Der Objekttyp *GeometricPrimitive* ist als tupelstrukturierter Typ definiert. Er besitzt drei Attribute:

- *GeoID* vom Typ string, ein vom Benutzer definierter Identifikator des geometrischen Objektes. Dieses Attribut ist nicht mit dem Objektidentifikator, der jedem komplexen Objekt zugeordnet wird, zu verwechseln.
- *Color* ebenfalls vom Typ string
- *Material* ein objektwertiges Attribut, dem ein Objekt vom Typ Material zugeordnet werden kann.

Der Objekttyp *Cylinder* ist als direkter Untertyp von *GeometricPrimitive* definiert. Zusätzlich zu den im Typ *GeometricPrimitive* öffentlich gemachten Operationen bietet der *Cylinder*-Typ noch die Operationen *Radius*, *length*, *weight*, *volume*, sowie die geometrischen Transformationen *rotate* und *translate*.

1.2.6 Überladen und Verfeinern von Operationen und dynamisches Binden

Überladen von Operationen: In manchen Fällen scheint es sinnvoll, daß verschiedene Operationen den gleichen Namen haben. Z.B. die *display*-Operation: Sie hat ein Objekt als Eingabe und zeigt dieses auf dem Bildschirm an. In Abhängigkeit vom Objekttyp will man natürlich verschiedene Anzeigemechanismen verwenden. Ist das Objekt ein Bild, soll es auf dem Bildschirm erschienen. Wenn das Objekt eine Person ist, soll irgendein Tupel ausgedruckt werden. Und ist das Objekt schließlich ein Graph, verlangen wir seine graphische Repräsentation.

Verfeinern von Operationen: In unserer Typhierarchie aus dem vorigen Abschnitt ist *Pipe* zur Modellierung von Röhren als weiterer Objekttyp enthalten. Dieser Objekttyp unterscheidet sich von *Cylinder* dadurch, daß er über ein weiteres Attribut *InnerRadius* vom Typ *float* verfügt. Die Struktur einer *Pipe*-Instanz ist also beschrieben durch die acht Attribute

- *GeoID* und *Color* vom Typ *string*,
- das Attribut *Mat* vom Typ *Material*,
- *Center1* und *Center2*, beide vom Typ *Vertex*,
- *Radius* und *Length*, jeweils von der Sorte *float*, und
- *InnerRadius*, auch vom Typ *float*.

Die ersten drei Attribute wurden vom indirekten Obertyp *GeometricPrimitive* geerbt, die nächsten vier vom direkten Obertyp *Cylinder*, und nur das letzte Attribut *InnerRadius* ist direkt in *Pipe* definiert worden. Die Typdefinition sieht dann folgendermaßen aus:

type Pipe **supertype** Cylinder **is**

```
type GeometricPrimitive supertype ANY is
public paint, SpecWeight, GeoID, Color →
body [GeoID, Color: string; Mat: Material;]
operations
      declare SpecWeight: ⟶ float;
      declare paint: string ⟶ void;
implementation
      define paint(c) is
         self.Color := c;
      end define paint;
      define SpecWeight is
         return self.Mat.SpecWeight;
      end define SpecWeight;
end type GeometricPrimitive;

type Cylinder supertype GeometricPrimitive is
public Radius →, weight, volume, translate, rotate
body [Radius: float, Center1, Center2: Vertex;]
operations
      declare weight: ⟶ float;
      declare translate: Vertex ⟶ void;
      declare volume: ⟶ float code CylinderVolumeCode;
      declare Cylinder: string, string, Material, float, Vertex, Vertex ⟶ void;
      declare length: ⟶ float;
implementation
      define Cylinder(g, c, m, r, c1, c2)
      begin
         self.GeoID := g;
         self.Color := c;
         self.Material := m;
         self.Radius := r;
         self.Center1 := c1;
         self.Center2 := c2;
         return self;
      end define Cylinder;
      define length is
         return self.Center1.distance(self.Center2);
      define translate(t)
      begin
         Center1.translate(t);
         Center2.translate(t);
      end define translate;
      define weight is
         return self.volume * self.SpecWeight;
      define CylinderVolumeCode is
         return (self.Radius * self.Radius * 3.14 * self.length);
end type Cylinder;
```

Abbildung 1.1: Definition der Typen *GeometricPrimitive* und *Cylinder*

```
public InnerRadius, connect
body [InnerRadius: float;]
operations
    declare connect: Pipe ⟶ Pipe;
    refine volume: ⟶ code PipeVolumeCode;
    refine weight: ⟶ float code PipeWeightCode;
implementation
    define PipeVolumeCode is
      (super.volume - self.InnerRadius * self.InnerRadius * 3.14 * self.Length);
    define PipeWeightCode is
      return self.volume * self.SpecWeight;
    define connect(otherpipe) is
    ...
end type Pipe;
```

Das Bemerkenswerte an der obigen Typdefinition *Pipe* sind die Verfeinerungen der Operationen *volume* und *weight*. Da die Semantik der ererbten Eigenschaften vom Typ *Cylinder* nicht genau auf die Eigenschaften des Untertyps *Pipe* paßt, müssen die Eigenschaften im Untertyp modifiziert werden. Die Operation *volume* ist jetzt so implementiert, daß das Hohlraumvolumen einer Röhre von dem Gesamtvolumen abgezogen wird.

Dynamisches Binden: Wegen der Verfeinerungsmöglichkeit werden Operationen prinzipiell dynamisch gebunden. Dynamisches Binden bedeutet, daß zu einem Operationsaufruf zur Laufzeit eine Implementierung (unter mehreren, die zur Verfügung stehen) ausgewählt wird

Die Notwendigkeit, verfeinerte Operationen dynamisch zu binden, soll durch ein Beispiel klar werden:

```
var c: Cylinder;

var p: Pipe;


p := Pipe.create();
c := p;
c.volume();
```

Da *Pipe* ein Untertyp von *Cylinder* ist, ist es völlig legitim, daß einer Variablen vom Typ *Cylinder* auch eine Ausprägung des Typs *Pipe* zugewiesen werden darf. Man bezeichnet dies im Zusammenhang objektorientierter Modelle als *Substituierbarkeit*: Eine Untertyp-Instanz ist überall dort legal einsetzbar, wo eine Obertyp-Instanz gefordert ist. Würde man in diesem Programmfragment die Operation *volume* statisch (zur Übersetzungszeit) binden, so würde die *CylinderVolumeCode*-Version ausgeführt. Es muß aber die *PipeVolumeCode*-Version gebunden werden. Genau dies wird durch dynamisches Binden sichergestellt. Verfeinerte Operationen werden zur Laufzeit entsprechend dem direkten Typ des Empfängerobjektes gebunden. Einmal verfeinerte Operationen können durchaus nochmal verfeinert werden, so daß man im allgemeinen eine Verfeinerungshierarchie erhält.

Durch dynamisches Binden wird garantiert, daß ausgehend vom Typ des Empfängerobjektes immer die spezifischste Version einer verfeinerten Operation zur Ausführung kommt. Dies geschieht ganz einfach dadurch, daß zur Laufzeit der direkte Typ des Empfängerobjektes bestimmt wird und dann innerhalb der Typhierarchie – beginnend beim Typ des Empfängerobjektes – in Richtung der Wurzel ANY nach einer Operation des gegebenen Namens gesucht wird. Dies könnte entweder eine Verfeinerung der Operation sein oder aber die Originalversion.

1.2.7 Turingvollständigkeit

Für Programmiersprachen ist diese Eigenschaft offensichtlich: Man meint damit einfach, daß man jede berechenbare Funktion ausdrücken kann. Und genau dazu sollte man nun auch mit Hilfe der DML des Datenbanksystems in der Lage sein. Für eine Datenbasis ist das eine Neuheit, zumal SQL zum Beispiel nicht turingvollständig ist. Entwickler von objektorientierten Datenbanken sollen nun aber keine neuen Programmiersprachen erfinden, sondern die Turingvollständigkeit durch Anbindung an existierende Programmiersprachen erreichen, was die meisten ja auch beherzigen.

1.2.8 Erweiterbarkeit

Datenbanksysteme besitzen vordefinierte Typen. Diese Typen können von den Programmierern verwendet werden, wenn sie ihre Anwendungen schreiben. Aber die Menge dieser Typen muß im folgenden Sinne erweiterbar sein: Es muß Mittel geben, um neue Typen zu definieren, und es darf keinen Unterschied bei der Verwendung der vom System definierten und der selbst definierten Typen geben, zumindest nicht für die Sicht des Benutzers. Natürlich wird das System die vordefinierten Typen anders unterstützen als die vom Benutzer selbst definierten Typen, aber das sollte sowohl für die Anwendung, als auch für den Anwendungsprogrammierer unsichtbar bleiben. Diese Typdefinition schließt auch die Definition von Operationen auf den Typen mit ein. Dabei ist es aber nicht notwenig, daß die Menge der Typkonstruktoren erweiterbar ist.

1.2.9 Persistenz

Unter Persistenz versteht man das Überleben von Programmkomponenten über die Ausführungszeit des Programmes hinweg. Dazu müssen diese Komponenten natürlich auf dem Hintergrundspeicher abgelegt werden.

Beispielsweise im relationalen Datenmodell wird stillschweigend unterstellt, daß alle dem Schema gehorchenden Daten auch persistent sind. Dagegen spielt Persistenz in Programmiersprachen keine solch ausgezeichnete Rolle. Erst die Verwendung eines objektorientierten Programmiermodells als Datenmodell wirft die Frage der Persistenz auf. Generell gibt es heute (noch) keinen Standard, der diese Fragestellung einheitlich beantworten würde.

Eine Lösung für das Persistenzproblem könnte darin bestehen, kurzerhand alle Typen sowie Objekte als stets persistent zu erklären. Dieses bewährte Prinzip wirkt an dieser Stelle trotzdem deplaziert, weil es außer acht läßt, daß sie starke programmiersprachliche Prägung zum ausschließlichen Arbeiten auf persistenten Datenbasiselementen führen würde.

Eine differenzierte Vorgehensweise besteht darin, dem Datenbanksystem ausdrücklich anzuzeigen, welche Elemente eines objektorientierten Programmes persistent sein sollen. Dies läßt sich durch ein bestimmtes Schlüsselwort für Variablenvereinbarungen oder durch die Verfügbarkeit spezieller Persistenzoperationen bewerkstelligen.

Auf jeden Fall sollte Persistenz orthogonal sein, d.h. jedes Objekt, unabhängig von seinem Typ, sollte als solches persistent werden können.

1.2.10 Hintergrundspeichermanagement

Hintergrundspeichermanagement ist eine klassische Eigenschaft von Datenbankmanagementsystemen. Es wird üblicherweise von mehreren Mechanismen unterstützt, einschließlich Index-Management, Datencluster, Datenpuffer, Zugriffspfadauswahl und Anfrageoptimierung. Nichts von alledem ist für den Benutzer sichtbar. Es dient nur zur Verbesserung der Performanz des Systems. Und ohne diese Zusätze wären manche Aufgaben gar nicht ausführbar, einfach weil sie zu lange dauern würden.

Der wichtige Punkt ist, daß diese Unterstützungen unsichtbar sind. Der Anwendungsprogrammierer sollte also keinen Quellcode schreiben müssen, um Indizes beizubehalten, Plattenspeicher anzufordern oder Daten von der Platte in den Hauptspeicher zu bewegen. Somit sollten die logische und die physikalische Ebene des Systems völlig unabhängig voneinander sein.

1.2.11 Nebenläufigkeit

Mehrere Benutzer sollten simultan auf der Datenbasis arbeiten können, wie das bei anderen aktuellen Systemen auch der Fall ist. Ein objektorientiertes Datenbanksystem sollte also den gleichen Grad an Diensten vorsehen.

Da aktuelle Datenbanksysteme Dienste bereitstellen, um damit mehreren Benutzern ein gleichzeitiges Arbeiten auf einer gemeinsamen Datenbasis zu ermöglichen, sollten auch objektorientierte Datenbanksysteme in etwa die gleichen Dienstleistungen bezüglich Nebenläufigkeit anbieten, um ein harmonisches "Nebeneinander" zwischen den Benutzern sicherzustellen. Dazu sollte das System die vier Transaktionsparadigmen (ACID) umsetzen: Atomizität, Konsistenz, Isolation und Dauerhaftigkeit. Ebenso sollte Serialisierbarkeit von Operationen angeboten werden, wenn auch in nicht ganz so strenger Form.

1.2.12 Recovery

Auch hier sollte das objektorientierte Datenbanksystem in etwa die gleichen Dienstleistungen anbieten wie derzeitige Datenbanksysteme. Daher sollte das System im Falle von eintretenden Hardware- oder Softwarefehlern wiederanlauffähig sein und einen konsistenten Datenzustand annehmen. Dabei schließen Hardwarefehler sowohl Prozessor- als auch Plattenfehler mit ein.

1.2.13 Ad Hoc Anfragesprache

Das Hauptproblem hier ist, die Funktionalität einer Ad Hoc Anfragesprache vorzusehen. Es muß nicht unbedingt in Form einer Anfragesprache sein, sondern nur den Dienst anbieten. Beispielsweise würde ein graphischer Browser diese Funktionalität ausreichend

erfüllen. Der Dienst besteht darin, dem Benutzer einfache Anfragen an die Datenbasis zu erlauben.

Zum Vergleich werden natürlich relationale Systeme herangezogen. Getestet kann werden, indem man eine Anzahl von repräsentativen relationalen Anfragen nimmt und schaut, ob sie ungefähr mit dem gleichen Arbeitsaufwand bearbeitet werden können.

Drei Kriterien sollte eine Anfragemöglichkeit aber trotzdem genügen:

- Sie sollte auf einem hohen Niveau sein, d.h. man sollte die Möglichkeit haben, mit wenigen Worten oder Maus-Klicks nicht triviale Anfragen präzise zu formulieren. Das impliziert vernünftigerweise, daß eine Anfragesprache deklarativ sein und das *Was* und nicht das *Wie* betonen sollte.

- Sie sollte effizient sein, d.h. daß nach der Formulierung der Anfrage diese in irgendeiner Form optimiert werden sollte.

- Sie sollte anwendungsunabhängig sein, d.h. man sollte mit ihr auf jeder beliebigen anderen Datenbasis arbeiten können.

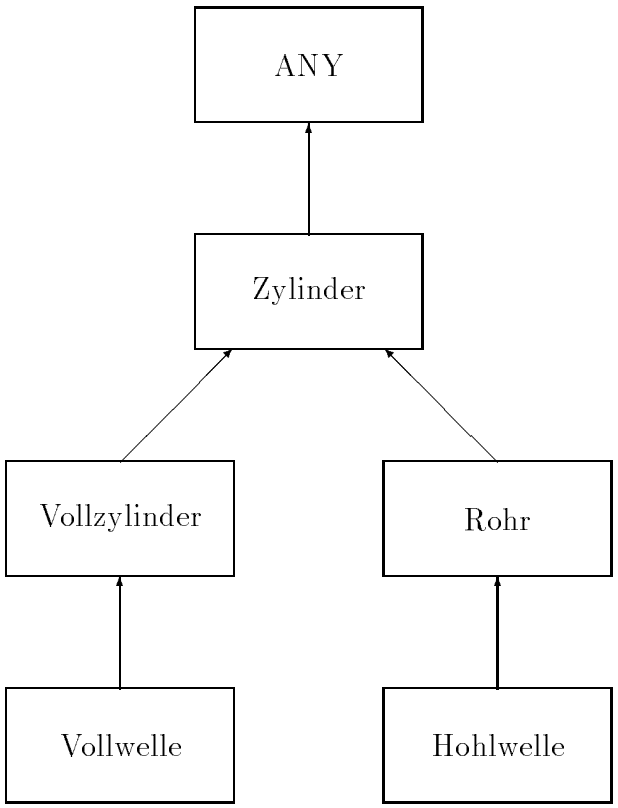
1.3 optionale Eigenschaften

Hier werden Eigenschaften besprochen, die ein System eindeutig verbessern, also wünschenswert, aber nicht unbedingt notwendig für ein objektorientiertes System sind. Die Eigenschaften sind bei den meisten Systemen standardmäßig vorhanden. Sie sind teilweise objektorientierter Natur, teilweise sind es aber auch typische Eigenschaften von traditionellen Datenbanken.

1.3.1 Mehrfachvererbung

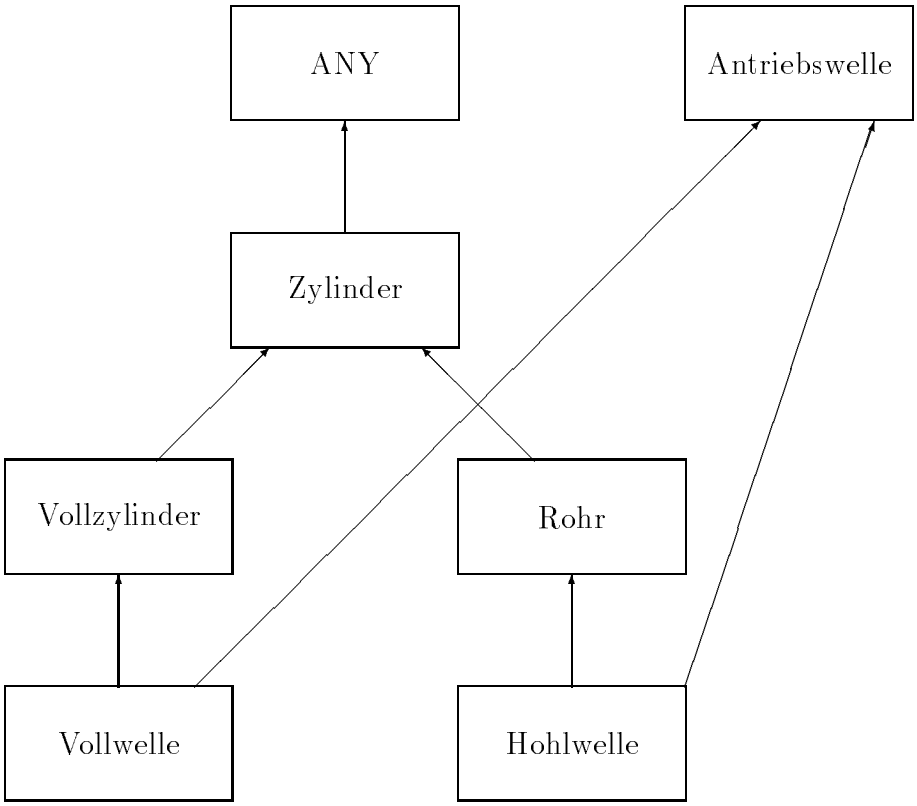
Jeder Typ darf mehrere direkte Obertypen besitzen, so daß sich eine Typheterarchie (nicht mehr nur -hierarchie) ausbilden kann. Ein Typ erbt dabei die Attribute und Operationen aller seiner Obertypen.

Beispiel: In der industriellen Praxis lassen sich sowohl Vollzylinder als auch Rohre (Hohlzylinder) als Antriebswellen nutzen. Antriebswellen ließen sich daher als entsprechende Spezialfälle auffassen. Besteht man auf Einfachvererbung, so bietet sich bei Einbringung von Wellen in den Typgraphen folgende Anordnung an:



Dabei wird unterhalb des Typs *Vollzylinder* der Typ *Vollwelle* eingeführt; analog dazu führt man *Hohlwelle* als Untertyp von *Rohr* ein. Für beide Wellen-Typen gilt, daß die bereits für *Vollzylinder* bzw. *Rohr* gültigen Eigenschaften wegen der Vererbung automatisch zur Verfügung stehen und nicht dupliziert werden müssen. Ärgerlich ist an dieser Modellierung allerdings die Tatsache, daß die Eigenschaften einer Antriebswelle zweimal gehalten werden.

Daher greift man auf das Prinzip der Mehrfachvererbung oder multiple inheritance zurück. Mehrfachvererbung gestattet eine gegenüber Einfachvererbung redundanzvermindernde Modellierung für die Darstellung von Antriebswellen. Hinzugekommen ist ein Typ *Antriebswelle* mit Angaben zur Dynamik. Vollwellen vereinigen nun die Eigenschaften von Vollzylindern und Antriebswellen, Hohlwellen die Eigenschaften von Rohren und Antriebswellen. Die minimale syntaktische Erweiterung des Typdefinitionsrahmens gegenüber der bisherigen Notation besteht darin, daß in der **supertype**-Klausel nun mehr als ein Typ spezifiziert werden kann.



1.3.2 Typisierung

Der Grad der Typisierung, den das System zur Übersetzungszeit ausführt, wird offengelassen, trotzdem ist es besser, sie möglichst streng durchzuführen. Die optimale Situation wäre, daß in einem Programm, wenn es einmal vom Compiler akzeptiert worden ist, keine Typfehler zur Laufzeit mehr auftreten können.

1.3.3 Verteilung

Ein System heißt verteilt, wenn sich seine Komponenten an räumlich getrennten Stellen befinden, hierdurch aber die Funktionalität des Gesamtsystems nicht beeinträchtigt wird.

Es sollte klar sein, daß diese Eigenschaft orthogonal zu Systemen objektorientierter Natur ist. Insofern kann ein Datenbanksystem verteilt sein oder auch nicht.

1.3.4 Transaktionen

In vielen neueren Anwendungen ist die Modellierung von Transaktionen nicht zufriedenstellend: Transaktionen, bei Design-Aufgaben beispielsweise, neigen dazu, ziemlich lang zu werden, und das übliche Serialisierbarkeitskriterium ist dann nur noch schwer zu erfüllen. Deshalb unterstützen viele objektorientierte Datenbanksysteme lange oder geschachtelte Transaktionen.

Objektorientierte Datenbanksysteme haben die Möglichkeit, mehr Nebenläufigkeit als die traditionellen Ansätze, die hauptsächlich auf der Lese-Schreib-Semantik beruhen, zu gewähren. Ein objektorientiertes Datenbanksystem weiß nämlich mehr über die Operationen, die ausgeführt werden sollen. Diese Operationen gehen weit über die Semantik des Lesens und Schreibens hinaus. Für den Datentyp *Schlange* könnte es die Operationen *Einfügen* und *Entfernen* geben. Diese kann man als ein Lesen und ein Schreiben auffassen, oder aber man zieht die spezielle Semantik dieser Operationen in Betracht und kann damit einen höheren Grad an Nebenläufigkeit erreichen.

Betrachten wir einmal ein Objekt Q vom Typ *Schlange* und zwei Transaktionen *T1* und *T2*. *T1* hat ein Element in Q eingefügt, und *T2* wird jetzt - gemäß der Lese-Schreib-Semantik - so lange am Entfernen eines Elementes aus Q gehindert, bis *T1* abgeschlossen ist (commit). Auf jeden Fall bemerken wir, daß diese zwei Operationen jeweils nicht das Ergebnis der anderen beeinflussen, wenn Q nicht leer ist und es sich beim einzufügenden und zu entfernenden Element nicht um dasselbe handelt. Sie könnten daher gleichzeitig ausgeführt werden.

1.3.5 Versionen

Oft finden Diskussionen über Versionen im Zusammenhang mit objektorientierten Datenbanken statt. Das hat aber weniger mit dem Modell an sich, als vielmehr mit den Anwendungsgebieten zu tun, für die dieses Modell entwickelt wurde. Die meisten neueren Anwendungen (CAD/CAM und CASE) sind mit Design-Aufgaben verbunden und benötigen somit eine Form von Versionenhaltung. Denn Versionen stellen einen Weg dar, um die Historie eines Objektes festzuhalten. Erwartungsgemäß wird heute auch von vielen objektorientierten Datenbanksystemen die Versionenhaltung unterstützt.

1.4 Zusammenfassung

Nach diesen Ausführungen sollte der Unterschied zwischen traditionellen und objektorientierten Datenbanksystemen klar geworden sein. Das CODASYL-Datenbanksystem besitzt teilweise die Eigenschaften 1 und 2. Manche Leute behaupten, daß objektorientierte Datenbanksysteme nichts anderes als CODASYL-Systeme sind. Es sollte angemerkt werden, daß CODASYL-Systeme nicht exakt die beiden Eigenschaften besitzen: Die Objektkonstruktoren sind nicht orthogonal, und Objektgleichheit ist nicht einheitlich behandelt, da Beziehungen auf 1:n beschränkt sind. Außerdem treffen die Eigenschaften 3, 5, 6, 8 und 12 nicht zu.

Über einige Eigenschaften konnte man sich immer noch nicht einigen, ob sie zu den obligatorischen oder den optionalen gezählt werden sollten:

- Sichtendefinitionen und abgeleitete Daten
- Datenbasisverwaltungshilfen
- Integritätsbedingungen
- Schemaevolutionsmöglichkeiten

Teil II

Speicherverwaltung

Kapitel 2

Speicherstrukturen für große Objekte (*Berthold Weiss*)

2.1 Einleitung

Durch eine Vielzahl neuer Anwendungen in den letzten Jahren wurde der Wunsch nach Datenbank - Systemen deutlicher, die den Anforderungen dieser Applikationen genügen. Zum einen trat das Problem auf, daß unterschiedliche Anwendungen meist auch unterschiedliche Operationen bzw. Datentypen benötigen, zum anderen fielen oft große Datenmengen (später als Objekte interpretiert) an. Ein Beispiel dafür findet man im CAD - Bereich, bei statistischen Anwendungen, Expertensystemen oder Multimedia - Anwendungen. Was man nun forderte, war die effiziente Manipulation dieser Daten, die Unterstützung beliebig großer Datenmengen und Operationen auch auf Bytefolgen. Desweiteren war das Ziel eine gute Performance zu erreichen, insbesondere kurze Objekterzeugungszeiten, schnelle Byte - Operationen (z.B. sollte das Lesen von der Platte nahe der Transferrate liegen) und eine gute Speicherauslastung.
Diese Arbeit beschäftigt sich mit den Speicherverwaltungssystemen der drei Datenbank - Systeme EXODUS [CDRS86], EOS [Bil92] und Starburst [LL89], d.h. mit der Kunst, Objekte geeignet zu speichern bzw. zu verwalten. Dabei werden die Effizienz und Performance der Systeme miteinander verglichen. Alle drei Systeme sind segmentbasiert, d.h. die Daten sind auf physikalisch adjazenten Blöcken gespeichert.
Für das Verständnis des folgenden sollten erst einige Begriffe geklärt werden. Eine Seite ist eine physikalische Einheit auf der Platte, die in einem Schritt gelesen wird. Mit einem Segment bezeichnet man eine Sequenz von physikalisch adjazenten Seiten. Bei der Einführung des Objektbegriffes unterscheidet man zunächst zwischen logischen und physikalischen Objekten. Ein logisches Objekt ist eine Datenansammlung, die einen Zustand und eine bestimmte Identität (OID - object identity) aus der Anwendersicht besitzt. Ein physikalisches Objekt ist die Repräsentation eines logischen Objektes auf einem Hintergrundspeicher. Wenn im folgenden der Begriff des Objektes fällt, so ist damit immer das physikalische Objekt gemeint. Ein Objekt ist klein, wenn es in einer Seite untergebracht werden kann, ansonsten wird es als großes Objekt betrachtet und auf soviele Seiten abgebildet wie nötig sind, um das Objekt zu halten. Eine Datei ist eine Sequenz von Datensätzen. Eine Datenbasis ist eine Sammlung von Dateien und gewöhlichen Objekten und wird in einem Speicherbereich erzeugt, z.B. in einem Unix - file (Volume). Die Objekte werden in sogenannten 'slotted pages' gespeichert, womit man mittels Page - Nummer

und Slot - Nummer auf das Objekt zugreifen kann. Man erhält also durch ein Quadrupel (volume, page, slot, unique) die Identität (OID) eines Objektes. Ein Datensatz ist lediglich ein Bytecontainer mit einer Identität TID (page, slot). Man kann einen Datensatz, dessen Attribute eventuell große Datenmengen vorsehen, entweder insgesamt als großes Objekt ansehen oder als kleines Objekt mit Deskriptoren auf große Felder (wie es z.B. Starburst macht).

2.2 Objekt- und Dateiverwaltung im EXODUS - Datenbank System

2.2.1 Überblick

Das Datenbank System EXODUS wurde 1986 an der Universität Wisconsin entwickelt. Das Hauptziel war, ein modulares und erweiterbares System zu erzeugen, das insbesondere die Verwaltung großer dynamischer Objekte effizient beherrscht. Der einzige feste Bestandteil ist das Speichersystem, der Kern des Systems mit dem sich dieses Kapitel weitgehend befaßt. Durch dieses Architekturprinzip ist genügend Flexibilität vorhanden, um eine große Anzahl von Anwendungen zu befriedigen. Desweiteren bietet EXODUS eine Versionsverwaltung, d.h mehrere Objekte, die einander ähnlich sind, können effizient gespeichert werden, eine Pufferverwaltung sowie Mechanismen, die Concurrency und Recovery ermöglichen. Der Entwickler erhält mit EXODUS eine Bibliothek mit nützlichen Routinen sowie die Möglichkeit, Zugriffsmethoden und Operationen zu konstruieren, indem die low-level Funktionen des Speichersystems benützt werden. Desweiteren besteht die Möglichkeit, eine spezielle Anfragesprache zu erzeugen, die auf die eigene Anwendung zugeschnitten sein kann. Das System kann insbesondere durch einen generischen Anfrage-Optimierer erweitert werden, indem vom Entwickler Transformations - Regeln geliefert werden.

Aus der Sicht des Speichersystems ist ein Objekt eine uninterpretierte Bytefolge unbegrenzter Größe. Das Speichersystem bietet wie oben schon erwähnt Concurrency- und Recovery - Mechanismen sowie Mittel zur Versionsverwaltung. Um Performance zu steigern, gibt es die Möglichkeit des direkten Zugriffs auf ein Speicherobjekt im Puffer, d.h. statt die Bytes zu kopieren, genügt ein Zeigerzugriff. Desweiteren, insbesondere um die Erweiterbarkeit des Systems zu gewährleisten, wurde versucht mit minimaler Semantik auszukommen, d.h. dem System so wenig wie möglich Information geben zu müssen. Allein sogenannte 'hints' sind erlaubt, um dem System Hinweise geben zu können, die die Leistung erhöht.

2.2.2 Schnittstellen

Das Speichersystem enthält eine prozedurale Schnittstelle. Die Objekte werden in Dateien gesammelt. Für diese Dateien und für die darin enthaltenen Speicherobjekte existieren folgende Funktionen:

- Erzeugen, Freigeben, Öffnen und Schließen von Dateien,
- ein Aufruf, der die Objekt-Id des nächsten Objektes innerhalb einer Datei liefert,

- Erzeugen und Freigeben von einzelnen Objekten,
- eine Funktion, die einen Zeiger auf eine gewünschte Bytefolge eines gegebenen Objektes zurückgibt (Lesen),
- eine Funktion, die dem System mitteilt, daß ein Teil der gelesenen Bytes modifiziert wurde,
- einen Aufruf, der die Bytes im Puffer freigibt,
- Operationen zum Einfügen/Löschen/Anhängen innerhalb eines Objekts,
- für die Transaktionsverwaltung die Prozeduren 'begin', 'commit' und 'end'.

Desweiteren besteht noch die Möglichkeit, die Performance des Systems durch 'hints' zu steigern, z.B einen Hinweis, wo ein neues Objekt plaziert werden soll oder einen Hinweis über die Größe des Objekts.

2.2.3 Datenstruktur

Große vs. kleine Objekte

Aus der Sicht des Anwenders besteht kein Unterschied, ob ein Objekt groß oder klein ist (klein bedeutet, daß es auf eine Platten - Seite paßt). Der Anwender erhält einfach eine Objektidentifikation zurück, die aus Volume - Id, Seiten - Nummer, Slot - Nummer und einem unique - Teil besteht. Intern befindet sich an dieser Adresse für große Objekte ein Zeiger auf einen 'header', der die Wurzel des Objekts bildet und für kleine Objekte ein Zeiger auf auf das Objekt selber. Sollte ein kleines Objekt zu groß werden, wird es automatisch in ein großes Objekt umgewandelt.

Datenstruktur für große Objekte

Die Datenstruktur besteht aus einem B+ - Baum mit Byte - Position als Schlüssel und Byte - Inhalt als Eintrag (Abb. 2.1). Diese Struktur unterscheidet zwischen internen Knoten und Blättern. Die internen Knoten bestehen aus (count, pointer) - Paaren, wobei count(i) - count(i-1) die Anzahl Bytes im i-ten Unterbaum angibt und pointer(i) der Zeiger darauf ist. Damit gibt die am weitesten rechts gelegene 'count' - Komponente der Wurzel die Größe des Objekts an (im Beispiel 1830 Bytes).

Die Blätter bestehen aus Segmenten fester Länge (n Seiten), enthalten nur reine Nutzdaten und sind halb bis ganz voll. Damit kann man vorab schon einsehen, daß für Objekte, die oft verändert werden, Blätter der Länge 1 Seite am günstigsten sind, da dadurch die I/O - Kosten minimiert werden und weniger Aufwand durch Hin- und Herbewegen von Bytes erforderlich ist. Für mehr statische Objekte sind allerdings mehrere Seiten geeigneter, da hier beim Lesen des Objektes weniger I/O - Kosten anfallen (jeder Segmentwechsel bedeutet Mehrkosten an Zugriffszeit).

2.2.4 Algorithmen

Die Standardalgorithmen für B - Bäume sehen das Einfügen bzw. Löschen eines einzelnen Datensatzes vor. Die Analogie dafür ist in unserem Fall das Einfügen bzw. Löschen

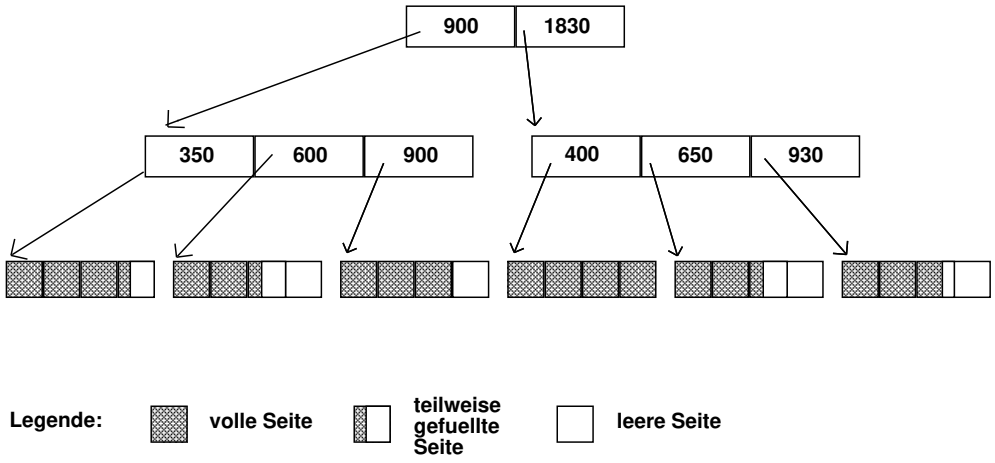


Abbildung 2.1: Die EXODUS Speicherstruktur für große Objekte

eines einzelnen Bytes. Update - Operationen von ganzen Bytefolgen entspräche dem mehrmaligen Ausführen dieser 'Ein-Byte-Operation', was zu ineffizient ist. Darum werden im folgenden einige Operationen auf große Speicherobjekte angegeben.

Suchen

Diese Operation liest eine Sequenz von N Bytes ab einer Startposition S. Es seien c(i), p(i) die Komponenten 'count' und 'pointer' eines Knoten und $c(0) = 0$.

- (1) start := S; P := Wurzel;
- (2) Solange P kein Blatt ist, führe folgendes aus: Speichere die Adresse von P auf dem Keller und suche binär in P nach dem kleinsten c(i), so daß start $\leq$ c(i).
  start := start - c(i-1)
  P := p(i)
- (3) Das erste gewünschte Byte beginnt ab Position 'start'.
- (4) Um den Rest der N Bytes zu erhalten (falls in diesem Segment nicht alle enthalten sind), traversiere den Baum weiter mit Hilfe des in (2) aufgebauten Stacks.

Einfügen

Diese Operation fügt eine Sequenz von N Bytes ab einer Startpostion S ein.

- (1) Traversiere den Baum wie beim Suchen. Korrigiere zusätzlich die 'count' - Komponenten.
- (2) Sei L das Blatt, in dem eingefügt wird. Wenn beim Einfügen kein Überlauf auftritt, ist alles o.k.
- (3) Andernfalls allokiere so viele Blätter wie nötig, um die neuen Bytes unterzubringen und verteile diese (und die von L) gleichmäßig.
- (4) Korrigiere die 'count' und 'pointer' - Komponenten rückwärts, indem der Keller benützt wird.

Dieser Algorithmus minimiert die I/O - Kosten, weil er die kleinstmöglichste Anzahl von internen Knoten und Blättern liest, allerdings hat er eine sehr schlechte Speichernutzung. Um dem zu entgegnen, kann der Schritt (3) modifiziert werden:

- (3') M sei der Nachbar von L mit am meisten freien Speicher (das steht im Vorgänger - Knoten von L) und B sei die Anzahl Bytes pro Blatt.
  Falls Free (L) + Free (M) $\leq$ N mod B, dann allokiere N/B neue Knoten und verteile die Daten gleichmäßig zwischen diesen Knoten und zwischen M und L. Dadurch kann eventuell die Reservierung eines unnötigen Knotens erspart bleiben. Der zusätzliche Aufwand ist auch vertretbar, da auf den Nachbarknoten M erst dann zugegriffen werden muß, wenn man weiß, daß die Verteilung erfolgreich sein wird. Im Erfolgsfall hat man zwar höhere I/O - Kosten, aber dafür muß ein Knoten weniger allokiert werden.

Anhängen

Diese Operation hängt N Bytes ans Ende des Objekts, was eigentlich eine spezielle Form des Einfügens ist. Sie wird aber aus Performance - Gründen extra behandelt. Der Unterschied besteht darin, daß beim Anhängen von Daten darauf geachtet wird, die Blätter vollständig zu füllen, um damit eine maximale Speichernutzung zu erzielen.

Löschen

Diese Operation löscht N Bytes ab einer Startposition S in zwei Phasen. In der ersten wird der geforderte Bereich gelöscht und in der zweiten der Baum wieder balanciert. Durch das Löschen wird eventuell ein ganzer Unterbaum gelöscht. Betrachtet man die dabei betroffenen Knoten wird i.a. ein Weg von der Wurzel bis zu einem gemeinsamen Knoten *lca* (lowest common ancestor), der die Wurzel des gelöschten Unterbaums darstellt, beschrieben (Abb. 2.2). Dieser Weg nennt sich hier 'cut-path'. Die rechte Grenze entlang des gelöschten Unterbaumes 'right cut-path', die linke 'left cut-path'. Man stellt fest, daß wenn irgendein Knoten einen Unterlauf hat, so muß er notwendigerweise im 'cut-path' enthalten sein. In der Rebalancierungs Phase wird dann der 'cut-path' top-down durchlaufen, wobei dieser Pfad im Hauptspeicher gehalten wird, um die I/O - Kosten zu senken. Wenn ein Knoten nun in Gefahr ist zu unterlaufen, wird dieser mit einem Nachbarknoten vereint, bis er sicher ist. Mit der folgenden Definition des potentiellen Unterlaufs eines Knotens, reicht ein Durchlauf.

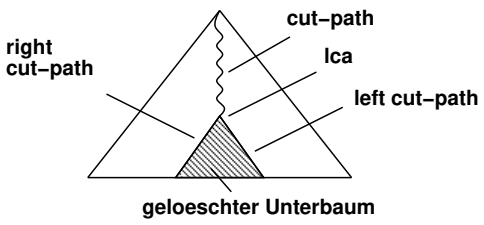


Abbildung 2.2: Terminologie für den Lösch - Algorithmus

Sei m die max. Anzahl von Paaren in einem Knoten und n := m/2;
Ein Knoten ist in Gefahr zu unterlaufen, wenn

- der Knoten echt unterlaufen ist, d.h wenn er weniger als n Einträge enthält,
- der Knoten ist ein innerer Knoten mit genau n Einträgen und einer seiner Nachfolger (im 'cut-path') ist in Gefahr,
- der Knoten ist der *lca* - Knoten und er hat genau n+1 Einträge und beide seiner Nachfolger sind in Gefahr.

Die Lösch - Phase sieht nun folgendermaßen aus:

- (1) Durchlaufe das Objekt gemäß der linken u. rechten Löschgrenzen. Alle Unterbäume, die komplett darin enthalten sind, werden gelöscht.
  Die 'count' - Komponente wird korrigiert.
  Erzeuge eine Datenstruktur 'path', die den 'cut-path' beschreibt (Adresse der Knoten, sowie Anzahl der Nachfolger).
- (2) Traversiere die 'path' - Struktur bottom-up, markiere jeden Knoten, der in Gefahr ist zu unterlaufen.

In der Rebalancierungs - Phase sind folgende Schritte notwendig:

- (1) Ist die Wurzel nicht in Gefahr, gehe nach (2). Hat die Wurzel nur einen Nachfolger, so mache diesen zur neuen Wurzel und gehe nach (1).
  Sonst vermische die Nachfolger der Wurzel, die in Gefahr sind und gehe nach (1).
- (2) Betrachte den nächsten Knoten im 'path' (falls vorhanden, sonst fertig).
- (3) Wenn der Knoten in Gefahr ist, mische ihn mit einem Nachbarn.
- (4) Gehe nach (2).

In der Lösch - Phase muß immer nur ein Blatt angeschaut werden, denn Knoten, die ganz gelöscht werden, können direkt freigegeben werden (ihre Adressen stehen im Elternknoten) und für den rechten cut-path muß nur die 'count' - Komponente im Elternknoten dekrementiert werden. Also muß nur das Blatt des 'left cut-path' gelesen bzw. geschrieben werden.

2.2.5 Besonderheiten

Versionsverwaltung

Das EXODUS - System bietet die Möglichkeit, von seinen Objekten verschiedene Versionen zu halten. Eine Version ist dann die aktuelle, alle vorangehenden werden im 'object header' als alte Versionen markiert (Abb. 2.3).

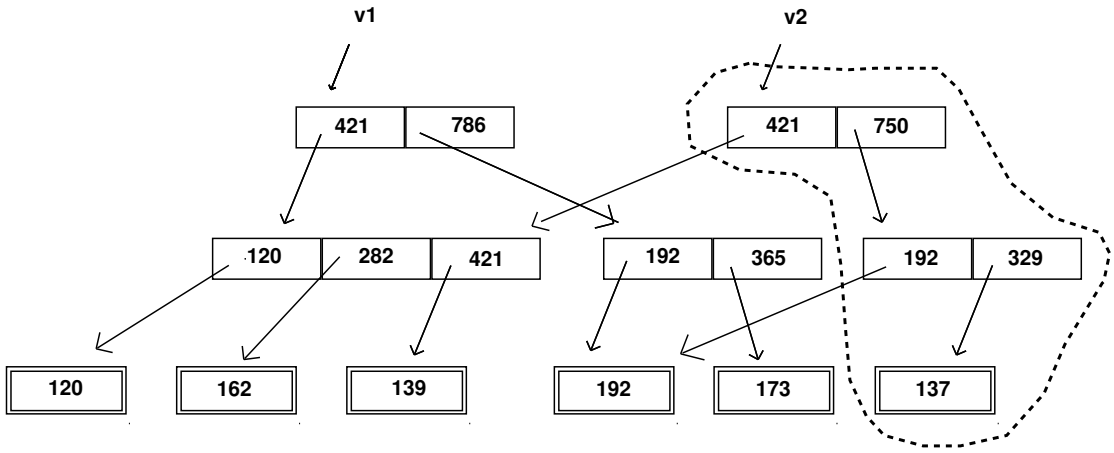


Abbildung 2.3: Zwei Versionen eines Speicherobjekts

Wenn ein Speicherobjekt modifiziert wird, wird der alte 'header' kopiert und die neue Adresse zurückgegeben. Die alte Version des 'header' kann dann überschrieben werden. Die Knoten bzw. Blätter, die sich unterscheiden, müssen kopiert und modifiziert werden. Im wesentlichen ist der Aufwand dann proportional zu diesem Unterschied, also wesentlich effizienter, da insbesondere nicht ganze Kopien von großen Objekten gehalten werden müssen.
Beim Löschen einer Version muß beachtet werden, daß ein Knoten zu mehreren Versionen gehören kann. Eine naive Methode würde folgendermaßen vorgehen: will man eine Version V löschen, dann markiert man alle Knoten, die aus Nachfogerversionen von V entstanden sind. Dann kann V traversiert werden und jeder unmarkierte Knoten gelöscht werden. Der Aufwand dafür ist allerdings recht hoch. Um die Anzahl der Knoten, die besucht werden müssen, zu drücken, genügt es nur die Knoten des direkten Nachfolgers und Vorgängers zu besuchen. Ebenso reicht es, wenn ein Knoten von zwei Versionen geteilt wird, nur diesen zu besuchen, da der ganze Unterbaum ebenso zu beiden Versionen gehören muß.

Concurrency & Recovery

Das EXODUS - Speichersystem bietet sowohl Concurrency- als auch Recovery - Mechanismen. Die Kontrolle der Concurrency besteht aus der Möglichkeit das ganze Objekt zu sperren, was für kleine Objekte sicherlich ausreicht. Für große Objekte gibt es zusätzlich die Möglichkeit des Sperrens von Bytefolgen. Dadurch können z.B. in verschiedenen Gebieten des Objektes Updates, die allerdings die Länge des Objektes nicht verändern dürfen, durchgeführt werden.
Recovery für kleine Objekte wird so gehandhabt, daß der Zustand vorher und nachher aufgezeichnet wird. Für große Objekte wird eine Kombination aus der 'shadow' - Technik und dem Aufzeichnungsverfahren benutzt (Erläuterung folgt in Abschnitt 2.3.5).

Pufferverwaltung

Wie bereits erwähnt, erhält ein Anwender die Möglichkeit direkt auf den Pufferinhalt zuzugreifen, womit er sich das Kopieren dieser Daten in seinen Anwenderspeicher erspart. Erstreckt sich der Umfang der angeforderten Daten über mehrere Seiten, so regelt die Pufferverwaltung das Problem auf folgende Weise. Zuerst kann Pufferspeicher in Blöcken,

deren Größe variabel ist, reserviert werden. Dann werden in einem solchen Pufferblock die nichtleeren Anteile der Blätter so kopiert, daß die Daten aufeinanderfolgend sind (die Blätter müssen nicht gefüllt sein). Zur Verwaltung des Pufferblocks existiert ein sogenannter 'scan descriptor', der die OID des Objekts, ein Zeiger auf den Puffer, die Größe der Daten, die eben kopiert wurden, einen Zeiger auf das erste geforderte Byte und Informationen, woher der Inhalt kam (für Replacement), enthält. Der Anwender erhält nun einen Zeiger auf den Pufferinhalt.

File - Objekte

Im EXODUS - Speichersystem können Objekte in einer Datei (File - Objekt) gesammelt werden. Das hat den Vorteil, daß alle Objekte auf einfache Art und Weise sequentiell gescannt werden können. Ferner werden solche Objekte auf den Seiten der Platte abgelegt, die für diese Datei reserviert worden ist, d.h. sie können physikalisch zusammengelegt werden.
Die Organisation solcher Dateien ist ähnlich wie die der Objekte selber. Es existiert für das File - Objekt eine OID und wieder eine B+ - Baum Struktur, wobei der Index der Knoten aus den Seitennummern besteht. Die Blätter enthalten eine Sammlung von Seitennummern für die 'slotted pages', die in diesem File enthalten sind. Bei der Erzeugung eines Objektes kann dieses in die Nähe eines anderen Objektes plaziert werden, d.h entweder auf die gleiche Seite oder in eine neue Seite, die physikalisch nahe liegt.

2.3 Der Starburst Long Field Manager

2.3.1 Überblick

Starburst ist ein Datenbank - Verwaltungssystem, das insbesondere durch Erweiterbarkeit, Unterstützung für Wissensbasen und große Objekte geprägt ist. An dieser Stelle wird nur auf die Speicherverwaltung von Starburst eingegangen, auch 'long field manager' genannt. Das System wurde insbesondere entwickelt, um Multimedia - Anforderungen zu genügen, beispielsweise das schnelle Lesen von Bildinformation, um Animation in Echtzeit zu erreichen.
Die wesentlichen Ziele beim Entwurf des Speichersystems waren eine effiziente Speicher(de)allokierung (im Bereich von 100 MB), ausgezeichnete I/O - Performance (im Bereich der Plattenübertragungsrate) und ein Recovery - Mechanismus, der die Leistung des Systems nur unwesentlich drückt.

2.3.2 Schnittstellen

Starburst bietet folgende Operationen an:

- **Clear** : löscht ein großes Feld.
- **Truncate** : löscht ab einer bestimmten Position.
- **Read** : Liest ab einer Position Bytes vom Feld in den Puffer.
- **Append** : Hängt Bytes vom Puffer an das Ende des Feldes an.

- **Replace** : überschreibt einen bestimmten Bereich.
- **Length** : Gibt die Größe des Feldes zurück.

Diese Operationen können auch über eine SQL - Schnittstelle angesprochen werden. Die Syntax lautet dann: EXEC SQL APPLY *operation*;

2.3.3 Datenstruktur

Speicherstruktur

Gemäß Abb. 2.4 wird nicht das Objekt selber in der Relation gespeichert, sondern ein 'long field' - Deskriptor. Dieser Deskriptor enthält komplett die Speicherinformation für das Objekt. Obwohl er kleiner als 255 Bytes ist, kann er Information für Objekte der Größe bis 100 MByte tragen. Er enthält neben Größe des Objekts, Anzahl von Segmenten, Größe des ersten und letzten Segments auch die Zeiger auf die Segmente, was den direkten Zugriff auf die Daten ermöglicht. Die Größen der übrigen Segmente sind implizit gegeben, da sie in Zweierpotenzen wachsen. Sei z.B die Größe des ersten Segmentes zwei Seiten, dann weiß man, daß das zweite Segment vier Seiten groß ist, das dritte 8 Seiten, usw. Nur das letzte Segment fällt aus der Reihe. Die Segmente zusammen mit einem Verzeichnis (allocation page), das den Belegungszustand der Segmente wiederspiegelt, bilden den 'buddy space', der wiederum einem größeren Speicherbereich (DB Space) entstammt.

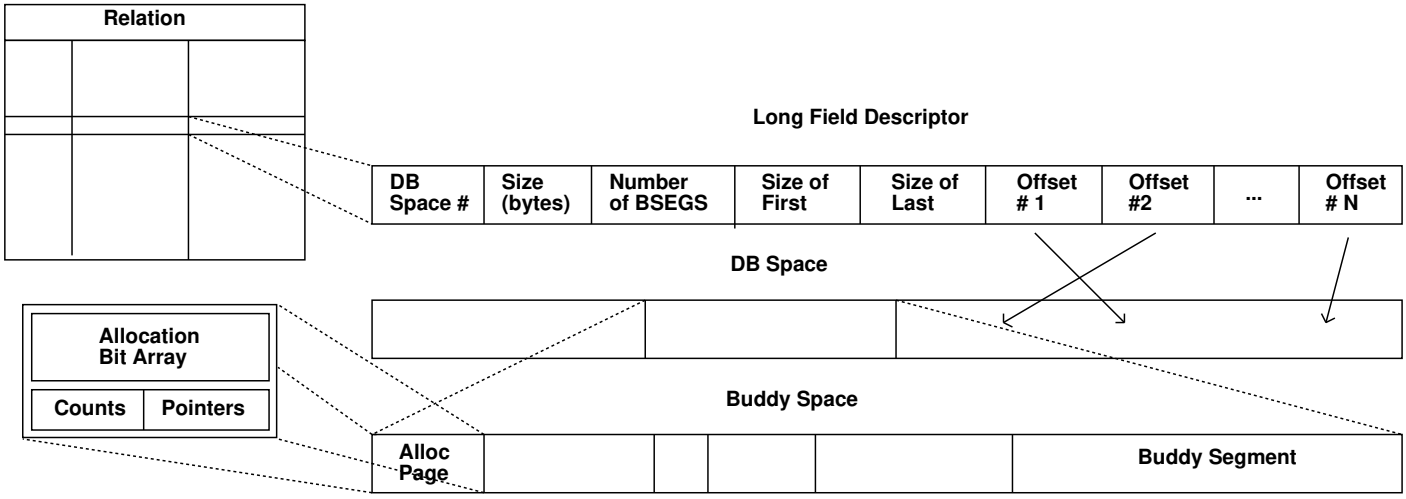


Abbildung 2.4: Die Speicherstruktur des 'long field managers'

Das buddy System

Das (binäre) buddy System verwaltet den Freispeicher so, daß nur Blöcke der Größen 2^k, $k \in [1..max]$ belegt werden können. Man sieht hieran sofort, daß der Speicher intern fragmentiert wird, denn die Speicheranforderungen müssen zur nächsthöheren Zweierpotenz aufgerundet werden. Benötigt man nun eine Speichergröße von 2^n, so wird der Freispeicher solange halbiert, bis zwei Blöcke dieser Größe entstehen. Diese beiden Blöcke nennt man 'buddies'. Beim Freigeben des Speichers schaut das System, welchen Zustand der

'buddy' des freizugebenden Blockes hat. Ist dieser frei, so können die beiden Blöcke wieder vereinigt werden. Die Adresse des zugehörigen 'buddies' kann auf folgende einfache Art berechnet werden: adr (Block2) = adr (Block1) XOR size (Block1). Der Vorteil des buddy Systems liegt in der schnellen Allokierung und Deallokierung von Speicher. Allerdings erzielt er eine hohe externe Fragmentierungsrate, wenn immer nur kleine Blöcke angefordert werden. Benutzt man allerdings bei der Allokierung Segmente unterschiedlicher Größe und verwendet das nachfolgend beschriebene 'Trimming', so kann sowohl die externe als auch die interne Fragmentierung deutlich verbessert werden. Mit dem buddy System kann genau die richtige Größe für die buddy Segmente reserviert werden, so daß, wenn die Einheit der Segmente eine Seite beträgt, weniger als eine Seite interner Speicherverschnitt entsteht.

Plattenspeicher - Verwaltung

Bei der Allokierung von Plattenspeicher werden sogenannte 'allocation pages', die ungefähr 16 MB an Speicher verwalten können, benutzt (Abb. 2.5). Eine 'allocation page' zusammen mit den Buddy - Segmenten bilden den 'buddy space'. Mehrere solcher Einheiten wiederum bilden einen Speicherbereich, der die Datenbasis umfaßt. Die 'allocation page' besteht aus einer Bitmap, die angibt, ob die Segmente frei oder belegt sind und aus einer Tabelle, die einmal Einträge enthält wieviel Segmente bestimmter Größe verfügbar sind und zum andern Zeiger auf Segmente besitzt, ab der die Suche nach der richtigen Segmentgröße sinnvoll ist. Diese zwei Komponenten der Tabelle erhöhen die Effizienz bei der Suche eines passenden Segmentes. Die Zeiger liefern allerdings nur einen Hinweis und keine absolute Position, d.h. sie zeigen auf eine Stelle, wo zuletzt ein verfügbares Segment gesehen wurde (das Segment kann also auch bereits belegt sein). Wenn ein Segment allokiert wird, wird die Anzahl für diese Segmentgröße korrigiert und der entsprechende Zeiger auf dieses Segment gesetzt. Wird ein Segment freigegeben, wird wieder die Anzahl korrigiert und der Zeiger auf das vorderste entsprechende Segment gesetzt.
Für die Bitmap wird eine Kodierung verwendet, die sowohl den Zustand (frei, belegt) als auch die Größe der Segmente wiederspiegelt. Man möchte gern mit maximal 8 bit einen Bereich von einer bis 32768 (2^{15}) Seiten adressieren können. Die ersten vier Bits enthalten Informationen entsprechend Abb. 2.6, die letzten vier die Größe als Zweierlogarithmus. Bei einem belegten Block, der größer als 4 Seiten ist, beginnt die Kodierung also immer mit 1001. Beträgt die Segmentgröße nur eine Seite so genügen zwei Bit bzw. bei zwei Seiten genügen vier Bit. Mit dieser Kodierung hat man den Vorteil, daß man maximal 8 Bit anschauen muß, um die Größe bzw. den Status des Segments zu bekommen. Als Beispiel wird die letzte Bitfolge in Abb. 2.6 (00011000) erläutert. Das erste Bit (eine 0) sagt aus, daß das Segment noch frei ist. Das zweite und das dritte Bit (jeweils eine 0) sagen aus, daß die Segmentgröße größer als eins bzw. zwei ist und das Check Bit (hier eine 1) bestätigt dies. Damit hat man also eine Bitfolge, die aus mehr als vier Bits besteht (pro Seite zwei Bits). Um die Größe des Segments zu bekommen, müssen jetzt nur noch die letzten vier Bits logarithmisch interpretiert werden. Hier ist die Länge des Segments also 2^8.

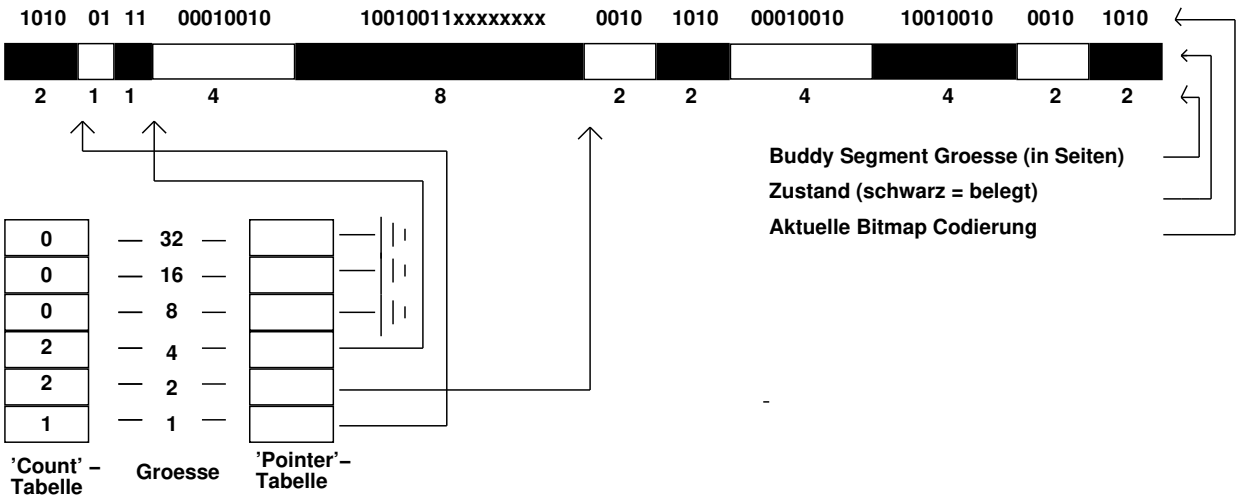


Abbildung 2.5: Struktur der 'allocation page'

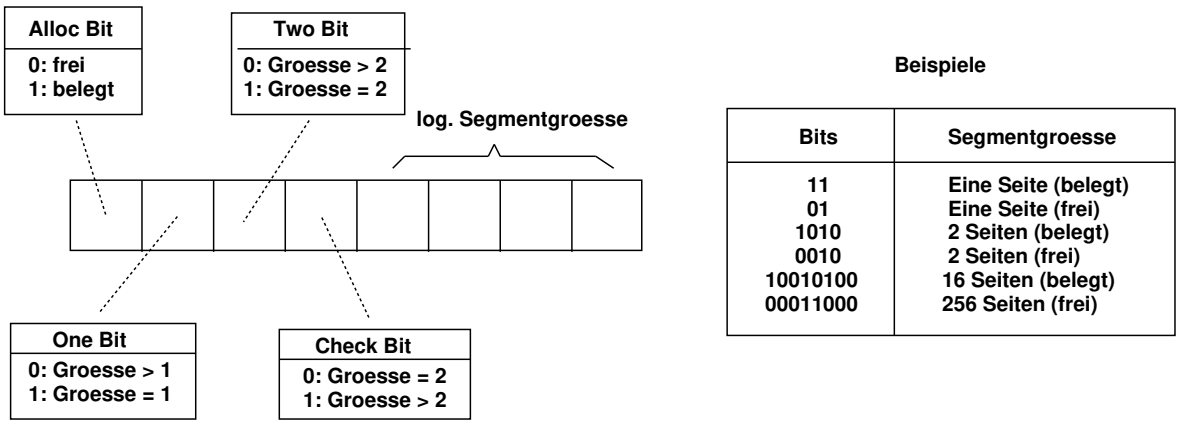


Abbildung 2.6: Bitmap Codierung

2.3.4 Algorithmen

Erzeugen eines Objektes

Beim Erzeugen eines großen Feldes wird der Deskripor erzeugt, buddy Segmente reserviert und die Zeiger des Deskriptors auf die Segmente gesetzt. Wenn die Größe des zu speichernden Objekts a priori nicht bekannt ist, dann verdoppeln sich die Größen aufeinanderfolgender Segmente sukzessive, bis die maximale Segmentgröße erreicht ist. Dann wird solange die maximale Segmentgröße allokiert bis das Objekt untergebracht ist. Hat man die Objektgröße bereits, werden gleich maximale Segmente benutzt. Der Vorteil Segmentgrößen zu benutzten, die sich stetig verdoppeln, liegt zum einen darin, daß nicht unnötigerweise große Blöcke allokiert werden, die schließlich wieder in kleinere Teile gebrochen werden müssen, zum anderen reduzieren kleinere Portionen die externe Fragmentierung und schließlich spart man sich im Deskriptor Information, weil die Größen implizit gegeben sind.

Trimming

Beim sogenannten 'Trimming' wird unnötig reservierter Speicherplatz wieder freigegeben, um die interne Fragmentierung zu reduzieren, unabhängig davon, ob die Größe des Objekts bereits bekannt ist oder nicht. Hat man z.B. ein Segment mit 16 Seiten, das Objekt belegt aber nur 11 Seiten, so kann ein Segment der Größe 4 und ein Segment der Größe 1 freigegeben werden (das sieht man leicht an der binären Repräsentation der Objektgröße). Das Trimming wird nach jeder Update - Operation durchgeführt.

Anhängen von Daten an ein Objekt

Das oben beschriebene Trimming von Segmenten verursacht folgendes Problem. Will man Daten anhängen, die eigentlich noch in das orginale 'ungetrimmte' Segment gepaßt hätten, so muß man ein neues Segment allokieren und alle Daten des getrimmten Segments dorthin kopieren (und natürlich auch die neuen Daten). Das vermeidet man, inden man das getrimmte Segment in einzelne Segmente bricht und den Zeiger auf das Segment zu einem Verzeichnis für die neue Menge von Segmenten umfunktioniert.

Lesen

Das Lesen eines Objektes ist sehr leicht zu realisieren, denn die gewünschte Start- und Endposition der gesuchten Bytes kann durch einfache Adreßrechnung bestimmt werden (man erinnere sich, daß die Größen aller Segmente gegeben sind und der Baum nur die Tiefe eins hat).

Einfügen/Löschen

Diese Update - Operationen sind bei Starburst sehr aufwendig. Werden neue Daten eingefügt bzw. wird ein Teil gelöscht, so muß ab der Updateposition sämtliche Segmente rechts davon neu organisiert werden, damit das Konzept des Systems gewahrt bleibt, nämlich das Wachsen der Segmente in Zweierpotenzen.

2.3.5 Besonderheiten

Concurrency & Recovery

Der Concurrency - Mechanismus wird eine Abstraktionsstufe höher gehandhabt (das Datenobjekt ist in Starburst die Komponente eines Datensatzes), braucht hier also nicht betrachtet werden. Für Recovery gibt es zwei wesentliche Techniken: zum einen das Aufzeichnen von Vorgängen und zum anderen die 'shadow' - Technik. Die letztere reserviert immer erst eine neue Seite und beschreibt diese und verwirft die alte erst dann, wenn diese nicht mehr gebraucht wird. Beim Aufzeichnen werden die Daten bei Updates zwar sofort überschrieben, aber die alten und neuen Werte festgehalten. Die 'shadow' - Technik hat den Vorteil, daß sie recht einfach ist und weniger I/O - Operationen benötigt. Starburst verwendet für die eigentlichen Daten die 'shadow-Technik'. Modifiziert man ein Objekt, entsteht eine neue Kopie der Datensegmente und der Deskriptor und die 'allocation page' werden korrigiert, wobei diese Veränderungen aufgezeichnet werden. Die alten Segmente bleiben als Kopien erhalten. Beim Löschen wird der entsprechende Speicher zwar als frei

markiert, aber gegen alle Transaktionen gesperrt, bis die Löschoperation vollendet ist. Somit kann das Löschen wieder rückgängig gemacht werden, da sowohl die alten Segmente noch bestehen als auch die Informationen für den Deskriptor und die 'allocation page' noch vorhanden sind.

2.4 Das EOS - Datenbank System

2.4.1 Überblick

Das Speicherverwaltungssytem von EOS ist im Prinzip eine Mischung der Speicherverwaltungssysteme von EXODUS und Starburst. Man versuchte hier die Vorteile beider Systeme zu vereinen.

2.4.2 Datenstruktur

Im Gegensatz zu EXODUS, das auf festen Segmenten basiert bzw. Starburst, das Segmente in Zweierpotenzen allokiert, erlaubt EOS variable Segmentgrößen (Abb. 2.7). Diese Segmente bestehen aus physikalisch aufeinanderfolgenden Platten - Seiten, die wiederum mit dem buddy System verwaltet werden. Um zu verhindert, daß bei häufigen Updates Segmente entstehen, die nur aus einer Seite bestehen, führte man eine Schwellgröße T (Treshold) ein. Diese Größe gibt eine untere Grenze an, ab der es nicht möglich ist, daß Daten in zwei adjazenten Segmenten verteilt sind, wenn sie in einem untergebracht werden könnten. Ein Beispiel dafür findet der Leser unter Abschnitt 4.4.

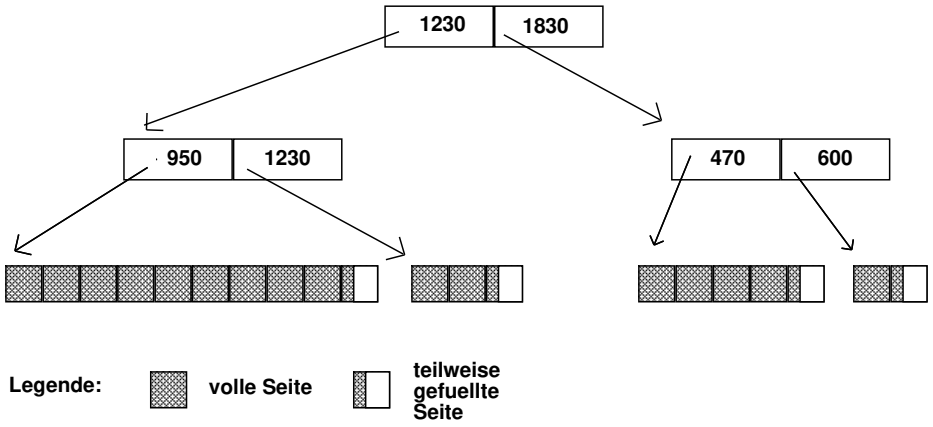


Abbildung 2.7: Speicherstruktur von EOS

2.4.3 Schnittstellen

Das EOS - System bietet eine objektorientierte und eine prozedurale Schnittstelle. Es werden verschiedene Klassen von Objekten unterstützt sowie die üblichen Funktionen auf Objekten (wie Erzeugen, Freigeben, Lesen, Schreiben, Löschen, Einfügen, Anhängen).

2.4.4 Algorithmen

Suchen

Dieser Algorithmus ist analog zu dem Verfahren in ESM.

Einfügen/Löschen

Bei Update - Operationen auf der Datenstruktur des EOS - Systems muß der Schwellwert beachtet werden. Zur Erläuterung dieses Vorgehens betrachte man Abb. 2.8. Hier sei der Schwellwert auf die Größe 4 gesetzt. Man möchte nun in ein bestehendes Segment neue Daten einfügen (Situation (a)). Normalerweise würden sich dann, wie in Situation (b) aufgezeigt, drei neue Segmente ergeben, deren Größen aber unter dem Schwellwert liegen. Also faßt man die Segmente so zusammen, daß der Schwellwert nicht unterschritten wird (Situation (b')).

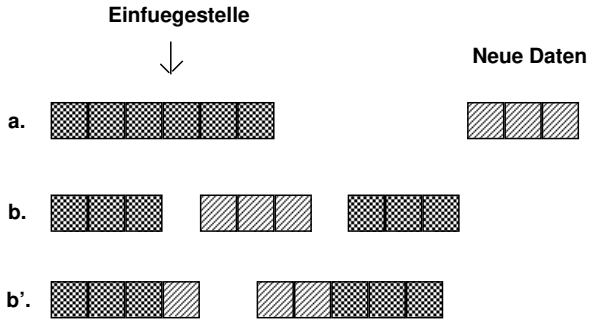


Abbildung 2.8: Einfügeoperation mit Schwellwert 4

2.4.5 Besonderheiten

Es sind hier keine Besonderheiten erwähnenswert.

2.5 EXODUS, STARBURST und EOS im direkten Vergleich

Im folgenden werden zwei Testreihen präsentiert, die unabhängig voneinander aufgestellt wurden. Die erste Testreihe wurde von den Entwicklern des EXODUS - Systems aufgestellt und beinhaltet nur Ergebnisse bezüglich des eigenen Systems. Die zweite ist ein Vergleich der drei Systeme, der von AT&T Bell Laboratories durchgeführt wurde.

2.5.1 Erste Testreihe

Implementierungsdetails & Parametergrößen

Für die Testreihe benutzten die Entwickler als Seitengröße 4 kB und für den Puffer 12 Seiten Umfang. Dabei wurden I/O - Zeiten und Speichernutzung getestet, indem einmal für die Blätter eine Größe von einer Seite und einmal eine Größe von 4 Seiten verwendet wurde. Desweiteren verglich man zwei Algorithmen zum Einfügen von Daten, einen

sogenannten 'basic insert' und einen 'improved insert' - Algorithmus, die oben bereits vorgestellt wurden. Um die Auswirkung der Operationsgröße (d.h der Umfang der Bytefolge für die Operation) sowohl auf Speicherfragmentierung als auch auf I/O - Zeiten festzustellen, wurde einmal mit 100 Byte und einmal mit 10k Byte gearbeitet. Der Test selber sah so aus, daß man zu Beginn ein 10 MB - Objekt erzeugte mit nahezu 100% Speicherauslastung. Danach wurden zufällig gemischte Operationen darauf losgelassen mit 40% Suchen, 30% Einfügen und 30% Löschen. Alle Updates waren gleichmäßig über das Objekt verteilt.

Speichernutzung

Bei der Betrachtung der Ergebnisse sieht man (Abb. 2.9), daß der verbesserte Algorithmus den Speicher eindeutig besser nutzt, was nicht verwundert, denn dafür wurde er ja konzipiert. Betrachtet man nur mal den Grundalgorithmus mit der kleineren Operationsgröße, so sieht man, daß er für Blätter, die nur aus einer Seite bestehen, leicht besser ausfällt als für 4-seitige Blätter. Den Grund findet man darin, daß bei größeren Blättern natürlich auch größere Teile gebrochen werden und also mehr freier Speicher übrig bleibt. Beim 'improved insert' verschwindet dieser Unterschied, da dieser Algorithmus versucht, das Teilen zu vermeiden. Zuletzt kann man noch feststellen, daß bei der größeren Operationsgrößen die einseitigen Blätter klar besser abschneiden als die 4-seitigen, da 3 bis 4 kleine, fast volle Blätter erzeugt werden (10 kB entspricht 2,5 Seiten). Bei den größeren Blättern muß mit Sicherheit ein neuer erzeugt werden, die dann beide relativ leer sind.

Abbildung 2.9: Speicherfragmentierung: links 100 Byte, rechts 10KB - Operationen

I/O - Kosten

Betrachten wir erst einmal die Operation 'Suchen' (Abb. 2.10). Man stellt fest, daß die I/O - Kosten unabhängig von der Größe der Blätter und der Art des Algorithmus sind, wenn wir eine Operationsgröße von 100 Bytes wählen, da diese meist durch eine Seite befriedigt werden können. Sucht man aber nach 10 kB, dann haben die 4-seitigen Segmente einen klaren Vorteil, da die 2,5 Seiten meist in ein Segment passen. Man braucht beim Lesen also nicht das Blatt zu wechseln.

Beim Einfügen von Daten stellt sich heraus, daß der modifizierte Algorithmus 3-10% teurer kommt. Für große Operationen sind 4-seitige Blätter 10-15% besser, weil dabei weniger Blätter angetastet werden müssen (Abnahme für zufälliges Lesen überwiegt die Zunahme für sequentielles Lesen). Für kleine Operationen ist 'basic insert' mit Blockgröße eins die bessere Alternative, da keine Nachbarknoten angeschaut werden müssen und keine sequentiellen Lesekosten entstehen.
Beim Löschen von Bytefolgen führt der verbesserte Algorithmus zu geringeren Kosten (ca. 20%), was wieder mit einer weniger starken Fragmentierung des Speichers zu erklären ist. Denn dies senkt die Wahrscheinlichkeit, daß man Daten von einem Blatt holen muß, um Lücken in teilweise gelöschten Blättern zu füllen. Weiterhin erkennt man, daß Segmente mit einer Seite Umfang einen Vorteil beim Löschen gegenüber den größeren Segmenten aufweisen, da bei diesen mit zusätzlichem sequentiellem Leseaufwand gerechnet werden muß. Dies gilt vor allem für den 'basic insert' - Algorithmus. Und noch stärker werden die Unterschiede für große Operationen (20% bzw. 12%).

Abbildung 2.10: Suchkosten: links 100 Byte-, rechts 10 KB - Operationen

Zusammenfassung

Zusammenfassend lässt sich sagen, daß mehrseitige Blöcke einen Vorteil bei großen Leseaktionen aufweisen, aber schlechter bei Updates abschneiden und auch eine schlechtere Speichernutzung hervorweisen. Sie sind also eher geeignet für große, statische Objekte mit nahezu 100% Speicherauslastung.

2.5.2 Zweite Testreihe

Was wird gemessen

Die drei Systeme werden bezüglich Objekterzeugungszeit, sequentielles Suchen, Fragmentierung und I/O - Kosten für Lesen, Einfügen und Löschen verglichen.

Implementierungsaspekte

Die Messungen wurden mit 4 kB - Seiten durchgeführt. Die Größe des Puffers beträgt 12 Seiten und das größte Segment, das in einem Schritt eingelesen werden kann, wurde auf

4 Seiten gesetzt. Die Größe des Objekts beträgt 10 MB. Für EOS wurde der Schwellwert auf 1, 4, 16 und 64 Seiten gesetzt, für EXODUS dieselben Werte für die Segmentgrößen.

Um die Ergebnisse richtig deuten zu können, werden im folgenden noch einige Implementierungsdetails erläutert. Für die Implementierung wurden zwei low-end Komponenten des EOS Speichersystems benutzt. Einmal das buddy System zur Verwaltung von Plattenspeicher und zum andern gepufferte I/O. Beim Allokieren von Speicher muß lediglich die 'allocation page' untersucht werden. Ist das Objekt aber entsprechend groß, so kann es sein, daß mehrere solcher Seiten existieren. Um zu gewährleisten, daß ein Plattenzugriff trotzdem reicht, wird ein Super - Directory (im Hauptspeicher) angelegt, daß die Information über die Größe der größten freien Segmente in den 'allocation pages' trägt.
Bei der Pufferung von Daten wird hier folgendes Verfahren eingesetzt: Wenn die Blöcke im Puffer untergebracht werden können, dann werden sie gepuffert, sonst wird direkt in den Anwenderspeicher kopiert. Dabei muß allerdings beachtet werden, daß meist die Blockgrenzen nicht übereinstimmen. In diesem Fall sind dann drei I/O - Schritte notwendig, d.h im ersten und dritten Schritt werden die gebrochenen Blöcke kopiert und im zweiten Schritt komplette Blöcke.
Für Segmente, die aus mehreren Blöcken bestehen, werden nur die Blöcke gelesen, die auch benötigt werden, also nicht das ganze Segment, was u.U. sequentielle Lesekosten erspart. Ebenso werden beim Zurückschreiben nur die 'dirty pages' berücksichtigt. Hier muß noch erwähnt werden, daß durch die 'shadow' - Technik hohe Kosten entstehen, weil immer ganze Segmente allokiert werden müssen (sonst wird die physikalische Kontinuität der Blöcke zerstört).

Erzeugen eines Objekts

Im folgenden wird die Zeit für die Erzeugung eines 10 MB - Objektes präsentiert (Abb. 2.11). Das Objekt wird durch sukzessives Anhängen von kleineren Datenblöcken erzeugt, wobei die I/O - Kosten in Abhängigkeit der Größen dieser Datenblöcke aufgezeigt wird. Für größere Blöcke bekommt man natürlich kürzere Zeiten. Es fällt jedoch auf, daß die Erzeugungszeiten zum Teil sehr stark hin- und herschwanken. Das liegt zum einen daran, daß die Blockgröße und die Größe der anzuhängenden Daten differieren und zum anderen an der Organisation der Operation selber. In EXODUS z.B. werden bei einem Überlauf des Blattes, das am weitesten rechts liegt, die neuen Bytes, die Bytes des Blattes und seinen linken Nachbars so verteilt, daß alle bis auf die letzten zwei total gefüllt sind. Dies muß immer dann ausgeführt werden, wenn die Größe der neuen Bytes kein Vielfaches der Blockgröße ist. In EOS bzw. Starburst ist das Problem nicht ganz so gravierend, da hier nur angehängt wird. Allgemein läßt sich sagen, hält man die Größe der anzuhängenden Bytes fest, so erzielt EOS/Starburst leicht bessere Ergebnisse als EXODUS.

Abbildung 2.11: Objekterzeugungszeit

Sequentielles Scannen

Bei diesem Test wird die Zeit gemessen, die benötigt wird, um das ganze Objekt sequentiell zu lesen (Abb. 2.12). Dabei werden genau solche Leseblöcke verwendet, die auch für das Erzeugen des Objektes verwendet wurden (bei Starburst/EOS hängt die Struktur davon ab).
Alle drei Techniken liefern das gleiche Ergebnis, wenn die Scan - Größe kleiner ist als die Seitengröße. Dann wird die Seite nämlich einfach in den Puffer gelesen. Man erhält allerdings das schlechteste Ergebnis, weil jedes mal die Zugriffszeit anfällt (das gilt insbesondere auch für EXODUS mit einseitigen Segmenten). Für EXODUS erhält man keine besseren Ergebnisse, wenn die Scan - Größe die Segmentgröße überschreitet. In Starburst dagegen produzieren größere Abtastgrößen auch bessere Ergebnisse, da hier ja auch die Segmentgrößen wachsen und keine erneuten Zugriffe nötig sind. Wieder gewinnt Starburst/EOS das Duell gegen EXODUS.

Random Read & Updates

Nun wenden wir uns der Frage zu, wie sich Updates auf das Objekt bezüglich Speicherfragmentierung und Performance auswirken. Wie in den früheren Tests wurde für die Aktionen eine Mischung aus 40% Lesen, 30% Einfügen und 30% Löschen angewendet. Dabei werden Operationsgrößen von 100 Bytes, 10 kB und 100 kB betrachtet.
Betrachten wir erst einmal die **Speicherauslastung** (Abb. 2.13 und 2.14). Starburst

Abbildung 2.12: Sequentielle Abtastzeit

braucht hier nicht weiter analysiert werden, weil dieses System bei Updates alle Segmente (bis auf das letzte) neu organisiert und komplett füllt; also tritt hier praktisch keine Fragmentierung auf.
In EXODUS hängt die Speicherfragmentierung stark von der Operationsgröße ab. Ist die Operationsgröße klein, sind die Segmentgrößen praktisch egal, aber für große Operationsgrößen bekommt man starke Unterschiede bei verschiedenen Segmentgrößen. Der Grund liegt darin, daß das Einfügen bzw. Löschen von kleinen Bytemengen sich innerhalb weniger Segmente abspielt. Größere Operationen umfassen mehrere Segmente (je kleiner die Segmente umso mehr) und somit entstehen viele nicht ganz gefüllte Blöcke.
In EOS liefern größere Schwellgrößen bessere Ergebnisse, denn in diesem System kann unbenutzter Speicher nur in der letzten Seite der Segmente vorkommen, d.h. für größere Schwellgrößen bleiben größere Restsegmente, was die Quote bezüglich belegtem und freien Speicher verbessert.
Vergleicht man beide Systeme, so sieht man, daß sie für einseitige Segmente bzw. einseitigem Schwellwert quasi die gleiche Auswirkung aufweisen.

Abbildung 2.13: ESM Speicherfragmentierung

Abbildung 2.14: EOS Speicherfragmentierung

Betrachtet man nun die **Lesekosten** der Systeme (Abb. 2.15 und 2.16) , so kann Starburst wieder außer Acht gelassen werden, da seine Struktur nach jedem Update wieder frisch organisiert wird.
Die Markierungen in den Abbildungen repräsentieren die Lesekosten, die seit der letzten Markierung entstanden sind. In EOS fällt nun auf, daß die I/O - Kosten anfänglich unabhängig von der Schwellgröße sind und für spätere Zeitpunkte schließlich doch noch differieren. Das liegt daran, daß zu Beginn die Segmente alle sehr groß sind (zum Zeitpunkt des Erzeugens) und später durch die Updates zu Segmenten degenerieren, die im

Größenbereich des Schwellwertes liegen.
Vergleicht man nun die Unterschiede, die sich durch verschiedene Segment- bzw. Schwellgrößen ergeben, kann man klar sagen, daß die einseitigen Segmente einen Nachteil gegenüber den mehrseitigen aufweisen. Es gibt zwei Gründe dafür. Zum einen sorgen größere Blätter dafür, daß die Anzahl interner Knoten kleiner wird, d.h. daß die Wahrscheinlichkeit sinkt, daß ein interner Knoten im Puffer nicht mehr vorhanden ist (und also ein Plattenzugriff mehr nötig ist). Zum anderen muß bei kleineren Segmenten auf mehrere Seiten (bei großen Operationen) zugegriffen werden.
Hat man eine Operationsgröße von 10 kB, so kann man feststellen, daß EOS eine bessere Performance aufweist als EXODUS, da bei EOS die neuen Bytes in ein Segment eingefügt werden (da hier variable Längen möglich sind), wogegen bei EXODUS drei verschiedene Blätter nötig sind. Bei einer Operationsgröße von 100 kB wird dieser Unterschied noch deutlicher.
Zusammenfassend kann man sagen, daß größere Segmente bei Leseoperationen einen Vorteil bieten. Wählt man für EOS die Schwellgröße von 16 Seiten, so erreicht man die Performance von Starburst.

Bei den **Update - Kosten** (Abb. 2.17 und 2.18). ist Starburst unabhängig von der Operationsgröße. Wie erwartet liefert das Verfahren hier sehr schlechte Ergebnisse. Der Grund liegt darin, daß beim Einfügen bzw. Löschen die Segmente wieder restauriert werden, d.h. es muß nahezu das ganze Objekt kopiert werden. Bei EXODUS und bei der Einfügegröße von 100 kB bemerkt man, daß die 16 - seitigen Segmente das beste Ergebnis erzielen, weil diese Segmentgröße nämlich am dichtesten bei der Einfügegröße liegt, d.h. es muß weniger Ausgleich zwischen benachbarten Knoten stattfinden. Die einseitigen Segmente schneiden ganz schlecht ab, weil hier 25 Seiten auf die Platte zu schreiben sind (zufällig). Die 64 - seitigen Segmente kommen bei den zwei kleineren Einfügegrößen sehr schlecht weg, weil hier große Teile der Segmente auf die Platte geschrieben werden müssen, d.h. die Zunahme für sequentielles Schreiben überwiegt die Abnahme von zufälligem Schreiben.
Betrachtet man nun die Graphen für EOS, so zeigen die Ergebnisse, daß kein Unterschied beim Einfügen besteht, ob man nun einen Schwellwert von einer oder vier Seiten benutzt. EOS fügt die neuen Bytes nämlich in soviele Seiten ein wie nötig sind, wenn die Anzahl dieser Seiten größer als die Schwelle ist. Steigt dieser Wert über 4, steigen auch die Kosten fürs Einfügen, da nun die Bytes wieder verteilt werden müssen.
Vergleicht man EOS und EXODUS miteinander, stellt man fest, daß für Segmentgrößen kleiner als 16, EOS das bessere Resultat aufweist, für Werte größer oder gleich 16 bekommt man gemischte Ergebnisse, wobei EXODUS besser bei kleinen und EOS besser für große Einfügegrößen ist. Für Löschoperationen ergeben sich genau die gleichen Resultate.
Grob gesprochen kann man folgendes sagen: Starburst schneidet beim Einfügen bzw. Löschen sehr schlecht ab. Für EOS und EXODUS sind die Update - Operationen im Gegensatz zu Starburst unabhängig von der Objektgröße.

Abbildung 2.15: ESM Lesekosten

Abbildung 2.16: EOS Lesekosten

Zusammenfassung der Meßergebnisse

Starburst erreicht exzellente Performance in allen Disziplinen bis auf die Einfüge- und Löschoperationen, worin es äußerst schlecht abschneidet.
Der Vorteil von EXODUS ist der, daß dieses System unabhängig von der Objektgröße arbeitet; die Performance hängt von der Größe der Blätter ab, was aber vom Anwender als Hinweis übergeben werden kann. Allerdings ist es sehr schwierig das richtige Maß zu bestimmen, da die Blattgröße sich auf Speicherauslastung und Performance beim Lesen entgegengesetzt auswirkt.
Die Güte von EOS hängt von dem gewählten Schwellwert ab. Größere Segmente liefern eine bessere Speicherauslastung, kleinere Lese - Kosten aber höhere Update - Kosten.

Beim Setzen der Schwellwertgröße muß man auf folgendes achten: Erstens sind Segmente kleiner als vier Blöcke zu vermeiden und zweitens sollte für oft veränderte Objekte dieser Schwellwert etwas größer gewählt werden als die Größe der Suchoperationen.

Abbildung 2.17: ESM Einfügekosten

Abbildung 2.18: EOS Einfügekosten

2.6 Zusammenfassung

Um die richtige Wahl für ein Speicherverwaltungssystem zu treffen, sollte man wissen, welche Ziele man sich gesteckt hat. Hat man Objekte, die meist nur gelesen werden, so setzt man am besten Starburst ein. Dieses System liefert dann eine exzellente Performance.

Oder man wählt EOS, das annähernd diese Leistung erreicht, aber zusätzlich bei Updates (Einfügen, Löschen) das Starburst - System um Längen übertrifft.
EXODUS kommt mit allen Operationen sehr gut klar und zwar unabhängig von der Objektgröße. Für bessere Lese - Performance sollte die Segmentgröße erhöht werden.

Kapitel 3

Implementierungstechniken für komplexe Objekte (*Ansgar Zwick*)

3.1 Einleitung

In dieser Seminararbeit soll auf drei Aspekte der Implementierung von komplexen Objekten eingegangen werden, wobei auch der Begriff des komplexen Objekts näher erläutert wird. Der erste Aspekt bezieht sich auf verschiedene Speichermodelle, deren Charakteristiken und deren Vor- und Nachteile, wobei hier im wesentlichen keine speziell objektorientierten Merkmale behandelt werden. Beim zweiten Aspekt steht vornehmlich die Implementierung von Vererbung im Vordergrund. Und zwar geht es hierbei um die Frage, wo geerbte Attribute in der Klassenhierarchie abgelegt werden sollen. Hierzu werden sechs Speichermodelle vorgestellt und auf Speicheraufwand untersucht. Der dritte Aspekt liegt noch eine Stufe näher an der Maschine und beschäftigt sich damit, wie ein Compiler innerhalb eines Objekts möglichst effizient ein Feld adressieren kann. Es werden dazu Objektadressierungsmechanismen für statisch getypte Sprachen mit Mehrfachvererbung präsentiert.

Entsprechend der inhaltlichen Gliederung erfolgt auch die formale in drei Kapitel. Danach findet der Leser noch ein Kapitel mit einer Zusammenfassung der Arbeit.

3.2 Implementierungstechniken für komplexe Objekte

In diesem Kapitel soll erörtert werden, wie sich die Struktur einer Datenbank an den Bedürfnissen der Anwendung orientieren sollte. Es werden drei verschiedene Modelle einer physischen Repräsentation von Datenbanken vorgestellt und ihre charakteristischen Merkmale und ihr bevorzugter Einsatzzweck erläutert.

Die Modelle entstammen den Arbeiten von Valduriez, Khoshafian und Copeland [VKC86], [CK85].

3.2.1 Ein Modell für komplexe Objekte

Im folgenden soll eine grobe Vorstellung vermittelt werden, was unter einem komplexen Objekt zu verstehen ist.

Ein Objekt ist rekursiv definiert wie folgt:

1. Integers, reals, booleans und strings sind *atomare* Objekte
2. Sind $O_1, O_2, \ldots, O_n$ Objekte und $a_1, a_2, \ldots, a_n$ unterschiedliche Attributnamen, dann ist $[a_1 : O_1, a_2 : O_2, \ldots, a_n : O_n]$ ein sogenanntes *Tupel*-Objekt
3. Sind $O_1, O_2, \ldots, O_n$ Objekte, dann ist $\{O_1, O_2, \ldots, O_n\}$ ein sogenanntes *Mengen*-Objekt.

Tupel können dabei als Attribute atomare Objekte, Tupel oder Mengen haben.

3.2.2 Ein Beispiel

Im folgenden soll nun ein bewußt einfach gehaltenes Beispiel vorgestellt werden, das auch in den folgenden Kapiteln verwendet wird. Es wurde in der Sprache GOMpl notiert, sollte aber auch für Nicht-GOMpl'er verständlich sein.

```
type Person is [ Name: string ];
type Student supertype Person is [ Mat: int ];
type Autor supertype Person is [ Thema: string ];
type Seminarist supertypes Student,Autor;
type Hiwi supertype Student is
  [ Anstellung: Vertraege ];
type Vertraege is
  { [StdAnz: int,
     Taetigkeit: string ]; }
```

Hiermit wird ein einfacher Sachverhalt modelliert: Eine Person werde durch ihren Namen beschrieben. Ein Student sei eine Person mit einer Matrikelnummer. Ein Autor sei eine Person mit einem bestimmten Thema und ein Seminarist erbe die Eigenschaften von Student und Autor und wird somit durch die Attribute Name, Mat.-Nr und Thema charakterisiert. Ein Hiwi sei ein Student mit einer Menge von Verträgen, die jeweils aus der Studenanzahl und der Art der Tätigkeit bestehen. Folgende Instanzen seien angelegt:

```
   Person( Otto )
Student( Gabi, 567 )
Autor( Goethe, Faust )
Seminarist( Willi, 123, DB )
Seminarist( Ulla, 987, KI )
Seminarist( Heinz, 555, CAD )
```

3.2.3 Begriffsklärung

Der folgende Abschnitt soll kurz die wichtigsten verwendeten Begriffe aus der Datenhaltung klären.

Ein Tupel wird physisch durch einen Datensatz repräsentiert. Mehrere gleichartige Datensätze können zu einer Datei zusammengefaßt werden. Eine Menge von gleichartigen Tupeln wird auch als Relation bezeichnet. Dabei müssen die Anzahl der Attribute, die

Art der Attribute und die Domänen (Wertebereiche) der Attribute eines Tupels gleich sein. Eine Datei bzw eine Relation kann auf zweierlei Arten zerlegt werden: erstens durch horizontale Partitionierung in Teilmengen von Datensätzen bzw. Tupeln, und zweitens durch vertikale Partitionierung in disjunkte Mengen von Attributen.

3.2.4 Direktes Speichermodell

Im direkten Speichermodell werden komplexe Objekte genau so gespeichert wie sie im Datenbankschema definiert sind. Jeder Datensatz enthält also ein vollständiges komplexes Objekt.

Es gibt für ein Objekt i.a. verschiedene Anordnungen für die Abspeicherung der Attribute, je nach Ordnung die man der Hierarchie auferlegt (z.B: präfix, infix, postfix). Für obiges Beispiel Hiwi mit einfacher präfix-Ordnung hätte das Speicher-Schema folgende Form (OID sei Objekt-Identifikator) :

```
sort Dir-Hiwi-Record is
  [ self: OID,
    state: [ Name: string,
             Mat: int,
             Anstellung: [ self: OID,
                           state: { [ StdAnz: int,
                                      Taetigkeit: string ] } ] ] ];
```

(Anmerkung: eine Instanz von `type` besteht aus ID und Zustand, während eine Instanz von `sort` lediglich aus einem Zustand besteht.)

3.2.5 NSM (N-ary Storage Model)

Dies ist der meist verwendete Ansatz für Datenbanksysteme. Jede Relation wird in einer eigenen Datei gespeichert. Die vertikale Partitionierung ist also trivial (da eine Relation nicht zerlegt wird). Das Einbringen eines Tupels ist sehr effizient, da nur eine einzige Datei davon betroffen ist. Die Relation Seminarist aus unserem Beispiel hätte beispielsweise folgendes Schema (Zur eindeutigen Identifizierung wurde jeder Datensatz zusätzlich mit einem Schlüssel (ID) versehen):

Seminarist	ID	Name	Mat	Thema
	id_1	Willi	123	DB
	id_2	Ulla	987	KI
	id_3	Heinz	555	CAD

In GOMpl würden die zugehörigen Deklarationen so aussehen:

```
sort NSM-Seminarist-Record is
  [ self: OID,
    Name: string,
    Mat: int,
    Thema: string ];
```

Die effizienteste Operation im Vergleich zu anderen Modellen ist eine Projektion auf viele Attribute, da bei einem Zugriff auf einen Datensatz sowieso alle Attribute zur Verfügung stehen. Die Projektion auf wenige Attribute dagegen ist weniger effizient, da trotz weniger benötigter Attribute der ganze Datensatz gelesen werden muß und somit das Verhältnis von benötigten Daten zu angeforderten Daten schlecht ausfällt.

3.2.6 DSM (Decomposition Storage Model)

Bei diesem Ansatz wird für jedes Attribut eine eigene Datei eingerichtet. Sie enthält die Werte des Attributs und zu jedem Wert den Schlüssel des zugehörigen Tupels. Man hält dabei jeweils zwei Kopien pro Attribut (zweite Kopie nicht abgebildet), eine wird nach dem Attributwert geordnet und eine nach dem Schlüssel des Tupels. Dies erhöht die Geschwindigkeit bei Operationen mit Suchvorgängen für Attribute.

Name	ID	val
	id_1	Willi
	id_2	Ulla
	id_3	Heinz

...

Mat	ID	val
	id_1	123
	id_2	987
	id_3	555

...

Thema	ID	val
	id_1	DB
	id_2	KI
	id_3	CAD

Die Vorteile hierbei liegen in einer höheren Sicherheit durch zwei Kopien und in einer effizienten Suche oder Projektion bezüglich weniger Attribute. Jedoch ist der Speicherverbrauch höher und man bekommt einen schlechteren Aufwand für das Einfügen oder Löschen eines Datensatzes.

Die Darstellung des Beispiels würde in der Sprache GOMpl folgendermaßen aussehen.

```
sort DSM-Seminarist-Record-Name is
  [ self: OID,
    Name: string ];

sort DSM-Seminarist-Record-Mat is
  [ self: OID,
    Mat: int ];

sort DSM-Seminarist-Record-Thema is
  [ self: OID,
    Thema: string ];
```

3.2.7 P-DSM (Partial DSM)

P-DSM ist eine Mischung aus NSM und DSM. Man hat hierbei versucht, die Vorteile beider Speichermodelle zu kombinieren. Bei P-DSM werden diejenigen Attribute in einer Datei zusammengefaßt, die auch bei Verbindungen oder beim Suchen häufig zusammen auftreten. Für eine solche Zusammenstellung wird allerdings zusätzliches Wissen der Anwendung benötigt. Sofern man die Art und Häufigkeit der Nutzung von Attributen relativ gut vorhersagen kann, kann mit diesem Modell ein Gewinn erzielt werden. Weitere Verbesserungen könnten darin bestehen, wie bei DSM für jedes Attribut zusätzlich eine binäre Relation anzulegen, die nach dem Attributwert geordnet ist. Damit könnten Suchoperationen bezüglich der Attribute beschleunigt werden.

Insgesamt wird bei diesem Modell also Speicherplatz und erhöhter Aufwand für das Einbringen von Datensätzen investiert in die Hoffnung auf effizientes Suchen und Verbinden für die vorhergesagten Häufigkeiten und Korrelationen von Attributen, wobei die Genauigkeit der vertikalen Partitionierung ein Schlüsselfaktor für die Effizienz darstellt.

3.2.8 Aufwands-Analyse

In [COPE85] wurde ein analytisches Modell für ein zwei-Kopien DSM vorgestellt. D.h. eine Kopie wurde nach Schlüssel geordnet, die andere nach den Werten des Attributs. Dabei wurde dieses Modell mit einem NSM Modell verglichen. Unter der Annahme von typischen Parametern wurde errechnet, daß unter Verwendung von Daten-Komprimierung der Speicheraufwand von DSM etwa 2.1 mal so hoch ist wie bei NSM wegen der zwei Kopien und einer minimal schlechteren Komprimierbarkeit. (Verwendete Parameter: Attribute pro Relation: 10 , durchschnittliche Attributgröße: 15 , Schlüsselgröße: 5) Insgesamt lag der Faktor zwischen 1 und 4. Zweitens wurde dargelegt, daß der durchschnittliche Aufwand für Aktualisierung bei DSM im Durchschnitt drei mal so hoch ist.

Am interessantesten war jedoch der Vergleich bezüglich des Such-Aufwandes für die beiden Modelle. Es wurde beobachtet, daß DSM dabei vergleichsweise besser abschneidet, falls die Selektivität S (Anzahl der betrachteten Tupel) / (Anzahl der Tupel der Basisrelation) jenseits einer bestimmten Schranke liegt (meistens etwa 1%). Der Vorteil von DSM gegenüber NSM zeigte ein Maximum bei einer Selektivität von etwa 10% .

Zusammenfassend kann man also sagen, daß sich für Verbindungen, die sich nur auf wenige Attribute beziehen und die mehr als 1 % der Tupel der Basisrelation zum Ergebnis haben, der Einsatz von DSM im Hinblick auf den Zeitaufwand lohnt. Je verzweigter eine Datenbank ist und je mehr auszuführende Verbindungen den obigen Bedingungen entsprechen, desto größer wird der Gewinn sein. Wenn auch der Speicherbedarf von DSM durchschnittlich etwa doppelt so hoch ist wie bei NSM, so muß hier bedacht werden, daß in Zukunft der Speicherverbrauch immer weniger kritisch wird, während der Bedarf an Schnelligkeit von Programmen zunehmen wird. Im übrigen wurde deutlich, daß nicht pauschal ein Modell bevorzugt werden sollte, sondern daß ein effizientes System sich nach den Bedürfnissen und Charakteristiken der Anwendung richten muß.

3.3 Speichermodelle für Klassenhierarchien

In diesem Kapitel wird ein theoretisches Modell für Klassenhierarchien für Vererbung vorgestellt und danach werden dafür sechs Speichermodelle präsentiert und bezüglich ihres Speicherplatzbedarfes miteinander verglichen.

Das Modell wurde von Willshire in [Wil91] beschrieben.

3.3.1 Modell für Klassenhierachien

Eine objektorientierte Datenbankhierarchie kann durch einen gerichteten azyklischen Graphen dargestellt werden. (s. Abb 3.1) Jeder Knoten markiert dabei eine Klasse. Eine Unterklassenbeziehung wird durch einen Pfeil dargestellt. Zu jeder Klasse können mehrere Instanzen gehören. Andererseits muß jede Instanz zu genau einer speziellen Klasse zugeordnet werden können, der sog. *home class*. Zusätzlich gehört jede Instanz allen über-

geordneten Klassen an. Zur Identifizierung wird jede Instanz in der Datenbank durch einen sog. Identifikator eindeutig gekennzeichnet.

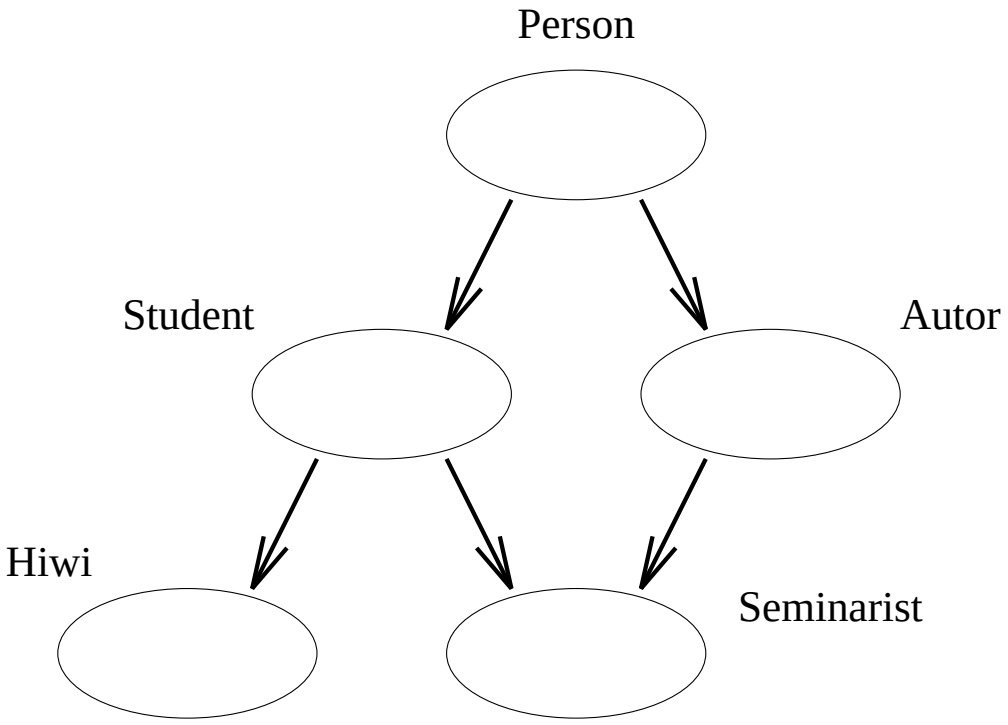


Abbildung 3.1: Graph einer Klassenhierarchie

Im folgenden soll nun im speziellen der Speicherbedarf für sechs Modelle untersucht werden, die sich jeweils darin unterscheiden, wo sie die einzelnen Attribute eines Objektes in der Klassenhierarchie ablegen. Dies kann entweder so weit unten wie möglich geschehen (d.h. in der home class des Objekts), so weit oben wie möglich (d.h. in der Klasse, von der das Attribut geerbt wurde), oder sowohl in der home class als auch in allen übergeordneten Klassen. Darüber hinaus gibt es noch die Möglichkeit alle Attribute zentral abzulegen.

3.3.2 Sechs Speichermodelle

Home Class Model – HC

In diesem Speichermodell taucht jede Instanz genau einmal innerhalb einer Datenbank auf. Eine Instanz wird dabei komplett in ihrer home class abgelegt. D.h. sowohl die speziellen Attribute der home class, als auch die von anderen Klassen geerbten Attribute sind gemeinsam in der home class zu finden. Es findet keine Duplizierung von Attributen statt. In GOMpl ergäben sich für Person und Seminarist folgende Deklarationen:

```
sort HC-Person is
  [ self: OID,
    Name: string ];

sort HC-Student is
  [ self: OID,
    Name:string,
    Mat: int ];

sort HC-Autor is
  [ self: OID,
    Name: string,
    Thema: string ];

sort HC-Seminarist is
  [ self: OID,
    Name: string,
    Mat: int,
    Thema: string ];
```

Für das Beispiel Seminarist(Willi, 123, DB) ergäbe sich folgende Ausprägung :

HC – Seminarist			
self	Name	Mat	Thema
id_1	Willi	123	DB

Leaf Overlap Model – LO

Im obigen Home Class Model kann jede Instanz nur einer Klasse angehören. Wird eine neue Instanz in die Datenbank eingefügt, so muß der Anwender entscheiden welcher Klasse die neue Instanz angehören soll. Nun kann es aber berechtigt sein, daß eine Instanz nicht nur eine home class haben soll, sondern mehrere. Im LO ist dies möglich. Abgesehen von den für diesen Zweck erstellten Duplikaten ist das LO mit HC identisch. In der Praxis hat dieses Modell aber im allgemeinen nur eine geringe Bedeutung.

Split Instance Model – SI

In diesem Modell werden die Attribute eines Objekts so weit oben wie möglich zusammen mit einem Identifikator abgelegt. Dies bedeutet, daß allein der Identifikator dupliziert

werden muß, aber wie beim HC werden keine Daten an sich mehrfach gespeichert. Man erhält hier eine teilweise vertikale Partitionierung der Instanzen in einer Datenbank, was mit P-DSM vergleichbar ist. In GOMpl würde unser Beispiel für SI so aussehen:

```
sort HC-Person is
  [ self: OID,
    Name: string ];

sort HC-Student is
  [ self: OID,
    Mat: int ];

sort HC-Autor is
  [ self: OID,
    Thema: string];

sort HC-Seminarist is
  [ self: OID ];
```

Und eine Ausprägung könnte sein:

SI – Person	
self	Name
id_1	Willi

SI – Student	
self	Mat
id_1	123

SI – Autor	
self	Thema
id_1	DB

SI – Seminarist
self
id_1

Repeat Class Model – RC

Gehört eine Instanz einer Datenbank mehreren Klassen an, so wird die Instanz beim RC in jeder dieser Klasse abgelegt, und zwar mit allen zu dieser Klasse gehörigen und mit allen geerbten Attributen. D.h. das RC implementiert direkt die logische Sicht einer Datenbank. Die Deklaration in GOMpl sieht genau gleich aus wie beim HC, allerdings wird bei der Ausprägung ein Objekt auch in allen Oberklassen angelegt wie in folgendem Beispiel ersichtlich:

RC – Person	
self	Name
id_1	Willi

RC – Student		
self	Name	Mat
id_1	Willi	123

RC – Autor		
self	Name	Thema
id_1	Willi	DB

RC – Seminarist			
self	Name	Mat	Thema
id_1	Willi	123	DB

Universal Class Model – UC

In einer relationalen Datenbank wird oft die Annahme gemacht, daß die Datenbank auf einer allumfassenden Universal-Relation basiert. UC nun ist die objektorienierte Version dieser Annahme. D.h. für jedes Objekt werden alle (gegebenenfalls leere) Attribute angelegt. Ein Vorteil von UC ist, daß alle Daten an einer zentralen Stelle verfügbar sind. Jedoch werden für gegebene Instanzen auch viele leere Variablen angelegt und somit Speicherplatz verschwendet. Es folgt die Deklaration in GOMpl und ein Beispiel.

```
sort UC is
  [ self: OID,
    Name: string,
    Mat: int,
    Thema: string,
    Anstellung: [ self: OID,
                  state: { [ StdAnz: int,
                             Taetigkeit: string ] } ] ];
```

UC				
self	Name	Mat	Thema	Anstellung
id_1	Otto			
id_2	Gabi	567		
id_4	Goethe		Faust	
id_3	Willi	123	DB	

Value Triple Model – VT

Dieses Modell ist eine Variante von UC. Wie beim UC werden auch hier alle Daten in einer Tabelle gespeichert. Die Identifikatoren der Instanzen bestehen aus der Kombination des Identifikators der home class und einer Seriennummer. Zusätzlich wird eine binäre Relation MinClass erzeugt, die ein Objekt seiner home class zuordnet. Daten werden als Tripel gespeichert mit der Form:

`Value (objectID, instanceVariableID, value)`

Wobei Value eine ID eines anderen Objekts sein kann, ein Integer- oder Real-Wert oder ein String (zur Optimierung können für Strings auch Referenzen eingesetzt werden). In der GOMpl-Notation bekäme man folgende Deklaration:

```
sort VT-Record is
  [ self: OID,
    var-Name: string,
    var-value: any ];
```

3.3.3 Die Simulation

In dem im folgenden vorgestellten Experiment ging es vor allem darum, allgemeine Trends im Verhalten der Modelle aufzuzeigen und den Designern von Datenbanken Faustregeln

in die Hand zu geben, und nicht darum, möglichst genaue Resultate zu erhalten. Die Untersuchungen erstreckten sich dabei insbesondere auf folgende drei Aspekte: Die Form des Graphen, der der Klassenhierarchie zugrunde liegt, die Anzahl der Instanzen pro Knoten (Klasse) und die Art der Verteilung von Instanzen über den Graphen. Für alle Kombinationen dieser Parameter wurde der Speicherplatz in Abhängigkeit von der Anzahl der Knoten des Graphen berechnet.

Um die Modelle besser vergleichen zu können, wurden dabei folgende Annahmen gemacht:

- Jede Instanz hat einen Identifikator (ID) mit fester Länge.
- Jede Klasse des Graphen benötigt eine feste Länge für das Einrichten neuer Instanzen.
- Werden Instanzen der Datenbank hinzugefügt, so werden sie in der spezialisiertesten Klasse untergebracht, die erlaubt ist (d.h. so weit unten wie möglich) und die Daten der Instanz werden je nach Modell abgespeichert.
- Die Größe der Datenbank ist proportional zu der Anzahl der Knoten.

Anzahl der Instanzen

Bei der Untersuchung des Speicherplatzes in Abhängigkeit von der Anzahl der Instanzen wurde noch die zusätzliche Annahme gemacht, daß jede Instanz genau einer home class angehört. Unter dieser Annahme stellte sich heraus, daß der Speicherbedarf für alle Modelle, alle Graphenformen und alle Verteilungen der Instanzen über den Graphen proportional zu der Anzahl der Instanzen ist.
(Anzahl der Instanzen bedeutet hier die Anzahl von Instanzen, die dieselbe Klasse als home class haben und nicht die Anzahl der Instanzen, die physikalisch in einer Klasse gespeichert sind.)

Graphenform

Bei den Formen des Graphen wurden drei unterschieden: flach, linear und ausgeglichen. Ein flacher Graph hat eine Wurzel und alle anderen Knoten sind deren direkte Nachfolger. Für die zugehörige Datenbank bedeutet dies eine geringe Spezialisierung. Das Gegenteil dazu ist ein linearer Graph, wo jeder Knoten – abgesehen vom letzen – genau einen Nachfolger hat und jeder Knoten, der hinzugefügt wird, wird an den letzten Knoten angehängt, was eine extreme Spezialisierung darstellt. Als Zwischenform wurde ein ausgeglichener Graph gewählt unter der Annahme, daß der Graph mit zunehmender Knotenzahl ausgeglichen bleibt.

Verteilung der Instanzen

Es wurden zwei verschiedene Instanzenverteilungen untersucht. Zum einen ist es möglich, daß jeder Knoten im Durchschnitt die gleiche Anzahl von Instanzen beherbergt, was in einer *gleichmäßigen* Datenverteilung über den Graphen resultiert. Zum anderen wurde angenommen, daß Nachfolgerknoten grundsätzlich mehr Instanzen besitzen als ihre Vorgänger (*fußlastig*). Dies bedeutet, daß die spezialisierteren Knoten auch über mehr

detaillierte Informationen verfügen und somit auch ein größeres Stück der Datenbank enthalten.

Die Parameter

Insgesamt ergaben sich für das Experiment also drei zu variierende Faktoren:

1. Graphenform – flach, lang oder ausgeglichen
2. Speichermodell – von HC bis VT
3. Instanzenverteilung – gleichmäßig oder fußlastig

Für die Konstanten wurde angenommen, daß die Länge einer ID konstant 8 Bytes beträgt, daß die Überlappung bei LO 30% beträgt und daß bei VT 30% der Instanzen-Variablen 24 Bytes lange Strings sind (der Rest Integers, Reals oder IDs).

3.3.4 Die Beobachtungen

Das absolute Minimum an Speicherbedarf benötigt das Modell HC. Es werden hierbei keine Daten dupliziert und es wird kein nutzloser Speicher angefordert. Das nächstbeste Modell ist SI, bei dem einzig die Identifikatoren mehrfach angelegt werden und somit die Länge der IDs und die Tiefe des Graphen ins Gewicht fallen können, da mit zunehmender Tiefe auch die IDs öfters gespeichert werden müssen.

Das Modell VT bewegt sich zwischen HC und SI bei hoher Speicherausnutzung, und bei RC und UC im schlechtesten Fall. Auch LO ist eher ein Zwischenmodell. Im besten Fall wie HC, nimmt der Speicherverbrauch proportional zu, je mehr home classes eine Instanz zugeordnet bekommt.

Am Ende der Skala sind RC und UC zu finden. Während RC durch exzessive Vervielfachung zu seinem hohen Speicherbedarf gelangt, gelingt dies UC durch das Anlegen von leeren Variablen.

3.3.5 Die Ergebnisse

Aus obigen Experimenten kann man letztendlich folgende Schlußfolgerungen ziehen:

1. **Graphenform:** Alle Modelle außer UC brauchen mit zunehmender Graphentiefe auch zunehmmend Speicherplatz. Dieser Effekt wird verstärkt, falls Daten in der Hierarchie mehrfach gespeichert werden. Je tiefer also der Graph, desto weniger kann Duplizierung akzeptiert werden.

2. **Anzahl der Instanzen** Für alle Modelle ist der Speicherbedarf proportional zu der Anzahl der Instanzen. Zwei Instanzen der gleichen Art benötigen also auch zweimal soviel Speicher wie eine Instanz dieser Art.

3. **Verteilung der Instanzen** Je spezialisierter eine Datenbank ist, je mehr Informationen also in den Blättern sitzt, desto ungünstiger werden Modelle, die Daten mehrfach anlegen, wie z.B. RC.

4. **Speichermodell** Unter den getesteten Bedingungen brauchen die Modelle HC, SI und LO weniger Speicherplatz als VT, RC oder UC.

Insgesamt läßt sich aus diesen Beobachtungen allerdings noch nicht sehr viel ableiten, da der Faktor Zeit nicht in die Untersuchungen mit einging. Es gilt hier noch weitere Forschungen abzuwarten. Es kann aber wiederum festgestellt werden, daß für eine geeignete Auswahl eines Modells die Art der Anwendung und die zur Verfügung stehenden technischen Mittel zu berücksichtigen sind.

3.4 Objektadressierung in statisch getypten Sprachen

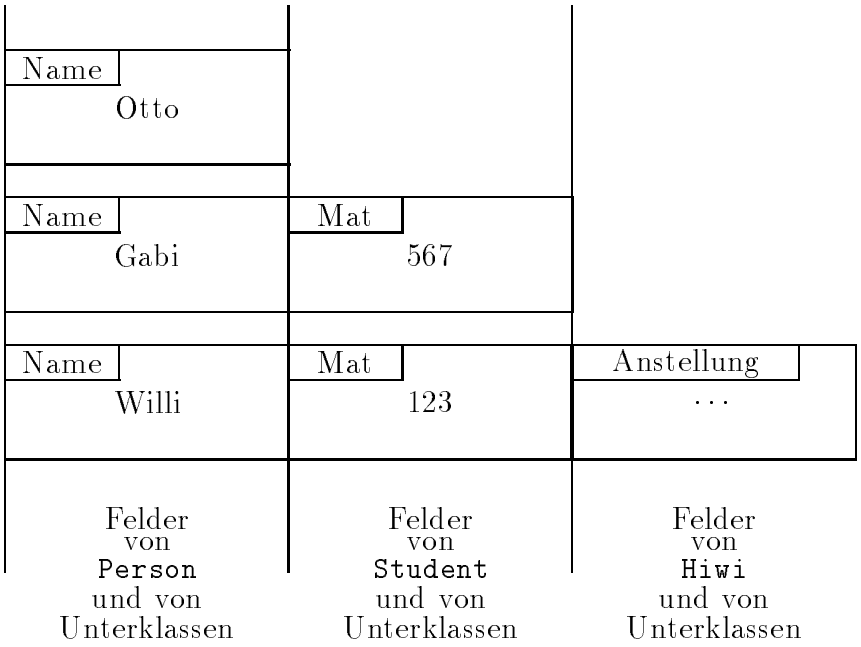
In diesem Kapitel geht es um Adressierungsmechanismen für statisch getypte Sprachen mit Mehrfachvererbung. Diese Mechanismen werden für eine effiziente Feldsuche benötigt, was bei Objektorientierung gleichzusetzen ist mit dem Zugriff auf Methoden. Und mit dem schnellen Zugriff auf Methoden steht und fällt auch die Schnelligkeit einer objektorientierten Sprache insgesamt. Es wird im folgenden gezeigt, daß Adressen in solchen Sprachen nicht immer statisch berechnet werden können, daß aber zumindest die Adressen der Adressen statisch bestimmt werden können. Weiterhin wird ein Weg aufgezeigt, wie man zu diesem Zweck Adreß-Tabellen einsetzen kann.

Der Objektadressierungs-Mechanismus ist in [CDMB89] beschrieben.

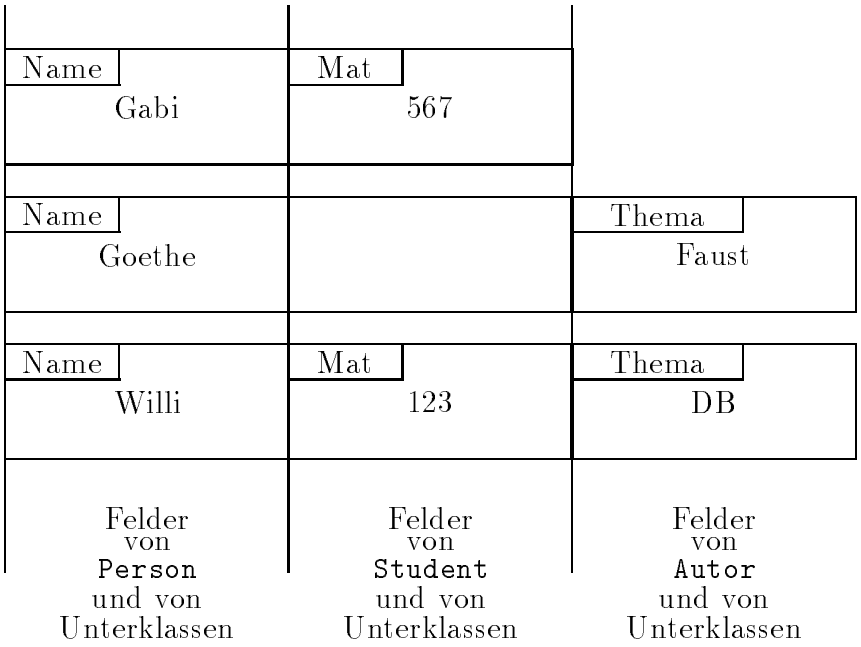
3.4.1 Implementierung von Unterklassen

Man betrachte nun die oben definierte Methode `Name`. (Ein Attribut kann immer auch als Methode aufgefaßt werden. Name z.B. als `Name@Person: -> string`) Die Methode `Name` kann auf Objekte der Klasse `Person` oder einer Unterklasse angewandt werden. Dies bedeutet, daß sie wissen muß, an welcher Position des Objekts die Variable `Name` zu finden ist. Eine einfache Implementierung würde z.B. die Variable `Name` stets an derselben Stelle in allen Instanzen von Unterklassen von `Person` speichern.

Führt man dies für `Person, Student` und `Hiwi` durch, so ergäbe sich eine Ausprägung wie folgt :



Man beachte, daß hier die Position eines Feldes schon statisch bekannt ist und daß somit eine hohe Effizienz bei Zugriffen erreicht wird. Allerdings ist diese Technik im allgemeinen nur geeignet für einfache Vererbung. Bei Mehrfachvererbung würde sich beispielsweise folgendes ergeben:



Man erkennt, daß hier Lücken auftreten, die Speicherplatz vergeuden. Und je mehr Unterklassen einer Klasse existieren desto größer ist dieser Effekt. In der Praxis können zwar nicht unendlich viele Unterklassen auftreten, aber doch eine sehr große Anzahl, so daß der Speicherverbrauch bei dieser Technik gewöhnlich für Mehrfachvererbung nicht akzeptabel ist und hierfür nach anderen Lösungen gesucht werden muß.

3.4.2 Implementierung von Mehrfachvererbung mittels Adreßtabellen

Eine einfache effiziente Implementierung von objektorientierten Systemen mit Mehrfachvererbung kann durch den Einsatz von Adreßtabellen erreicht werden. Eine solche Adreßtabelle, die beispielsweise am Anfang eines jeden Objekts liegen kann, enthält die Positionen aller Felder des Objekts. Für `Seminarist( Willi, 123, DB )` beispielsweise würde man eine Ausprägung nach Abb. 3.2 erhalten.

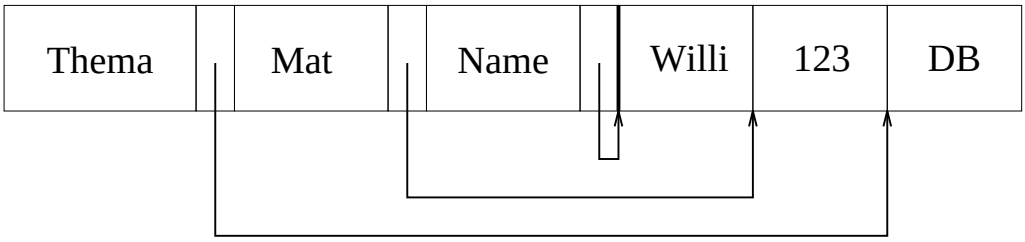


Abbildung 3.2: Objekt mit eigener Adreßtabelle

Um Speicher zu sparen könnte man eventuell auch nur eine Adreßtabelle für jeden Typ von Objekten anlegen und im Objekt einen Zeiger auf diese Tabelle halten, wie es in Abbildung 3.3 dargestellt ist.

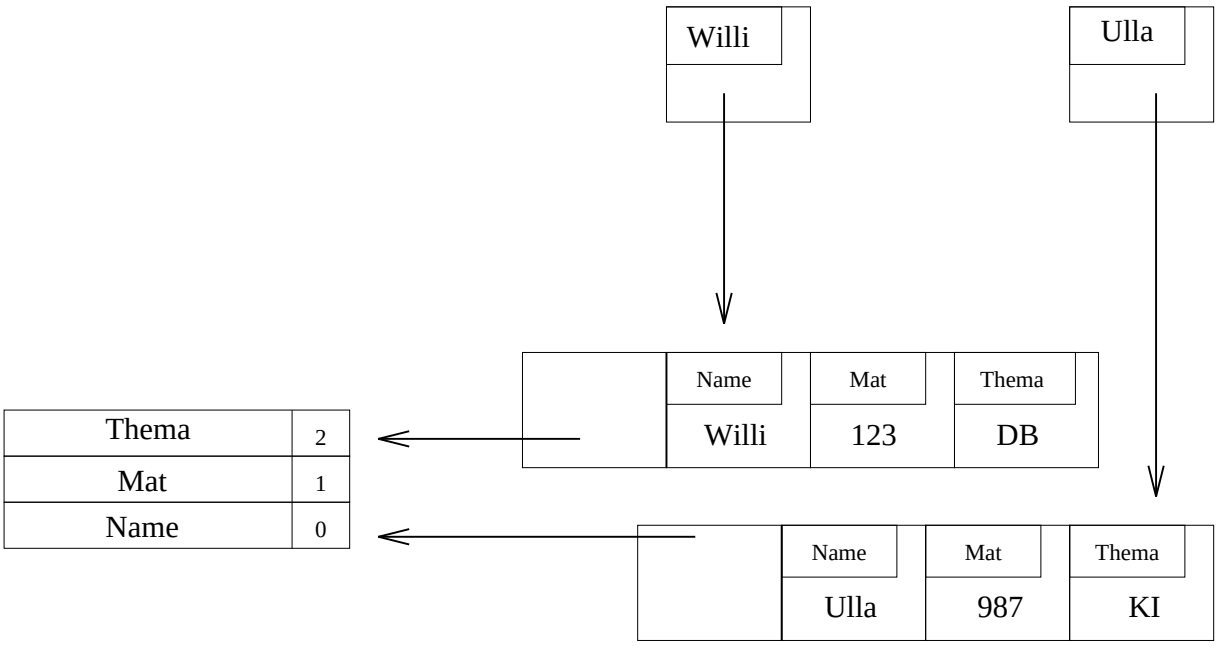


Abbildung 3.3: Gemeinsam benutzte Adreßtabelle

Eine weitere Einsparung von Speicherplatz kann erreicht werden, indem man für die Bezeichnung der Felder einen Integer-Schlüssel benutzt anstelle eines strings. Dafür kann der Compiler eigens ein zentrales Verzeichnis der Feldnamen anlegen, wozu er dann statisch den Platz für einen Integerwert pro Feld alloziieren kann.

3.4.3 Implementierung von Substituierbarkeit

Die obigen Mechanismen beruhen darauf, daß der Compiler schon vorab den genauen Typ eines Objekts kennt. Wird aber Substituierbarkeit zugelassen, so kann dies nicht mehr vorausgesetzt werden. Im folgenden wird nun auch für dieses Problem eine Lösung angeboten.

Substituierbarkeit bedeutet hier, daß einer Variable einer bestimmten Klasse der Wert einer Unterklasse zugewiesen werden darf. Dies sei durch folgendes Beispiel illustriert:

```
   var a := Person( Otto )
var b := Student( Gabi, 567 )
a := b
var d = a.Name
```

Die letzte Zeile ist hier erlaubt, wohingegen

```
var e = a.Mat
```

verboten wäre, da das Feld `Mat` nicht über a als Instanz der Klasse `Person` angesprochen werden kann. Es ist also für den Compiler nicht mehr länger möglich, die Position von Feldern schon statisch zu bestimmen. Dadurch müssen noch einmal weitere Adreßinformationen angelegt werden, da vorab nicht der genaue Typ eines Objekts bestimmt werden kann.

Eine einfache und direkte Lösung besteht darin, für ein Objekt nicht nur ein, sondern zwei Zeiger anzulegen. Einen Zeiger auf das Objekt an sich und einen Zeiger für ein Adreßverzeichnis der Felder. Dies erscheint sehr ähnlich zu herkömmlichen Adreßtabellen. Es sei jedoch auf folgendes hingewiesen:

- Die Adreßtabelle ist kein Teil des Objekts, sondern beinhaltet Informationen über Adressen. Ein Objekt kann durch mehrere verschiedene Adreßtabellen betrachtet werden (siehe Abb. 3.4).
- Die Adreßtabelle verweist nur auf Felder des ursprünglichen Typ des Objekts, nicht aber auf mögliche andere Felder des momentan realen Objekts, die eventuell hinzugekommen sein könnten (siehe Abb. 3.4).
- Dieselbe Adreßtabelle kann unter Umständen auch mehrfach benutzt werden, auch von Objekten unterschiedlicher Klassen.

Mit Hilfe dieser Tabellen ist es nun möglich, zur Laufzeit effizient die Adresse von Feldern zu berechnen. Dafür verwendet man die Adresse des Objekts, die in der Adreßtabelle steht, und addiert dazu den Offset für das gewünschte Feld. So erhält man die genaue Adresse des Feldes bzw. des Attributs des Objekts.

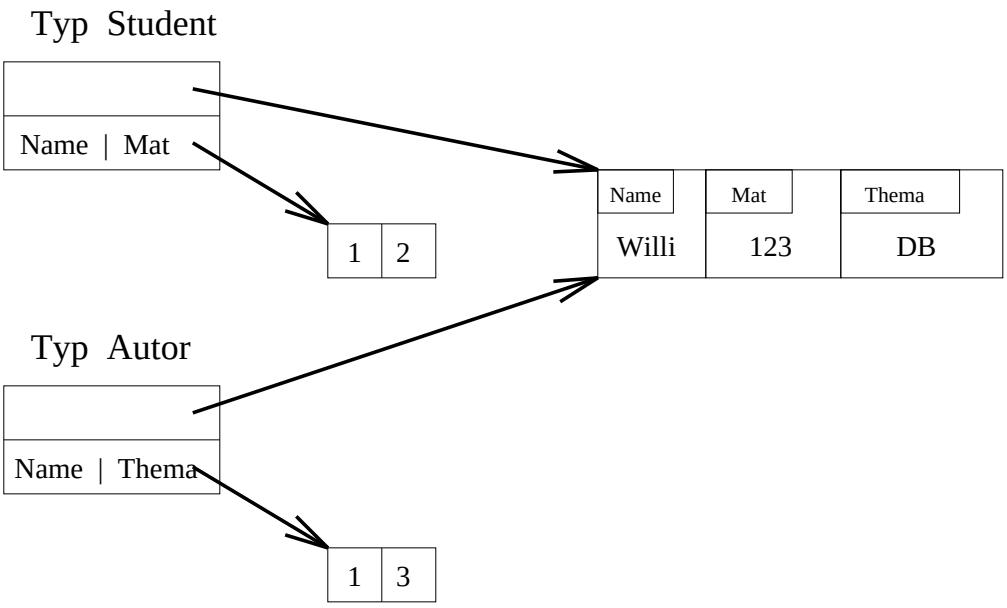


Abbildung 3.4: Adreßtabelle für Substituierbarkeit

3.5 Zusammenfassung

Diese Seminararbeit sollte Einblicke geben, welche Techniken verwendet werden können, um komplexe Objekte zu implementieren. Dabei war der erste Teil eher Strategien für eine Implementierung von herkömmlichen Datenbanken gewidmet als speziellen objektorientierten Aspekten. Es stellte sich beim Vergleich der zwei Modelle DSM und NSM heraus, daß in den meisten Fällen das verbreitete NSM dem speicheraufwendigeren DSM vorzuziehen ist, daß es aber unter gewissen Bedingungen hinsichtlich einer Zeitoptimierung sinnvoll sein kann, NSM einzusetzen.

Im zweiten Teil wurden sechs Modelle vorgestellt, die auf unterschiedliche Weise die Felder von Objekten in einer Vererbungshierarchie unterbringen. Allerdings war hier allein der Speicheraufwand Gegenstand der Untersuchungen, was für sich gesehen nicht sehr aussagekräftig ist. Zusammen mit Ergebnissen über den Zeitaufwand der Modelle aber hätte man beim Entwurf von Datenbanken einige gute Richtlinien zur Hand. Es gilt also weitere Forschungen auf diesem Gebiet abzuwarten.

Der dritte Teil behandelte Adressierungsmechanismen für statisch getypte Sprachen mit Mehrfachvererbung. Eine einfache Adressierungstechnik für Einfachvererbung wurde erweitert für Mehrfachvererbung durch den Einsatz einer Adreßtabelle, so daß die Adresse eines Objekts statisch berechnet werden konnte. Sollte auch Substituierbarkeit von Objekten durch Objekte von Unterklassen möglich sein, wurde gezeigt, daß es nicht mehr länger möglich ist, die Adresse eines Feldes in einem Objekt statisch zu bestimmen, daß aber mit Hilfe einer weiteren Referenzebene zur Laufzeit diese Adresse effizient berechnet werden kann.

Kapitel 4

Pointer Swizzling (*Andreas Neukirch*)

4.1 Einleitung

'To swizzle or not to swizzle' ist der Untertitel des Artikels [Mos90], der die Hauptquelle für diese Seminararbeit ist. Die Ethymologie des Begriffes 'Swizzling' ist den Autoren nicht bekannt und wenn man es übersetzen will, gibt es nur ein Substantiv 'swizzle', das ein alkoholisches Mischgetränk sein soll. 'Pointer Swizzling' jedenfalls ist eine bestimmte Repräsentation persistenter Objekte im transienten Bereich. Eine andere Möglichkeit bietet sich mit dem ObjectStore-Konzept an. Im Folgenden werden beide Ansätze erklärt und weitere Verfeinerungen des Swizzlings anhand von Tests gegeneinander ausgespielt.

Das Problem stellt sich so: Eine Anwendung will auf persistente Objekte zugreifen, die ihrerseits wieder auf andere persistente Objekte verweisen. Dazu müssen die Objekte von dem beständigen Speicher, z.B. der Festplatte des Servers, in den flüchtigen Speicher, z.B. den RAM einer Arbeitsstation geladen werden. Im beständigen Speicher werden die Objekte durch eindeutige Objektidentifikatoren (Oid) repräsentiert. Der Objektmanager kann nun, zum Beispiel so wie beim Mneme-Konzept, anhand des Oids das Objekt in dem persistenten Speicher lokalisieren. Er lädt dann das Objekt in den transienten Speicher. Wie die Anwendung danach darauf zugreift, ob weiterhin über den Objektmanager, durch direkte Adressen eines virtuellen Speichers etc., entscheidet eben der Ansatz. Welcher dabei der günstigste ist, ist von Parametern wie der durchschnittlichen Größe von Objekten abhängig.

Einsatzgebiete dieser Überlegungen sind Sprachen, bei denen ein Bereich des Speichers (z.B. ein Objekte-Heap) persistent ist (z.B. E, Alltalk), Datenbankprogrammiersprachen, die außerdem noch große Datenmengen verwalten können (z.B. O^{++}, CO_2) , objektorientierte Datenbanken (z.B. Exodus, DAMOKLES, O_2), Persistent Object Stores (z.B. Exodus, Mneme) und Objektserver (z.B. ObServer, Gemstone).

4.2 Die Konzepte

4.2.1 Mneme

Der Zugriff auf den persistenten Speicher ist bei allen Varianten des Artikels [Mos90] gleich. Objekte bestehen dabei aus Benutzerdaten und aus Slots, in denen (sofern initia-

lisiert) Verweise auf andere Objekte sind. Wenn die Anwendung auf einen Objektidentifikator (Oid) als einem solchen Verweis auf ein Objekt stößt, liefert es diesen an den Objektmanager (OM), der dann das Objekt lokalisiert (object lookup) und, wenn es nicht resident ist, es lädt. Die Swizzlingvarianten nehmen nun vom OM die Adresse des Objektes im transienten Speicher entgegen, um mit damit zu arbeiten. Beim Mneme-Konzept gibt es zwei Möglichkeiten. Entweder übernimmt der OM auch die Bearbeitung des Objektes und schon beim nächsten Zugriff auf das Objekt findet die gleiche Prozedur statt (call interface). Oder die Anwendung läßt sich den Zeiger auf das gefundene Objekt in einen Puffer des OM legen, um dann das Objekt unmittelbar zu manipulieren (pointer interface). Dieser direkte Zugriff gilt nur für eine gewisse Dauer, zum Beispiel eine Prozedur, und wird Besuch (visit) genannt. Danach wird der Puffer,der die direkte Adresse enthält, wieder überschrieben.

In Abbildung 4.1 ist der Oid zu sehen, der ähnlich aufgebaut ist wie bei der Technik des virtuellen Speichers. Ein 30-Bit-Feld wird in 3 10-Bit-Felder aufgeteilt die jeweils Tabellen indizieren.

Abbildung 4.1: Format des Mneme-Objektidentifizierers

Gemäß Abbildung 4.2 indiziert das erste Feld eine Dateitabelle, um die Startadresse der Datei bzw. deren Segmenttabelle zu liefern. Das zweite Feld indiziert eben diese und ergibt entweder die Startadresse des Segmentes bzw. der Objekttabelle, falls dieses resident ist. Ansonsten wird mit anderweitiger Information der Ort auf der Festplatte festgestellt und das Segment geladen. Jedenfalls zeigen die niederwertigsten 10 Bit auf die Adresse des Objektes in der Objekttabelle, in der dann die Adresse des Objekt im RAM steht.

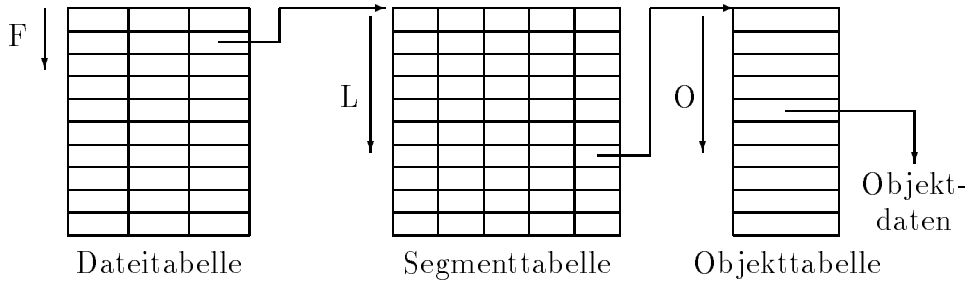


Abbildung 4.2: lookup-Schema für Objekte

4.2.2 Swizzling

Hier herrscht folgender Gedanke: Wenn die Objekte erst einmal resident sind, also von der Festplatte in den Speicher geladen sind, dann könnte man ja gleich sämtliche Verweise auf diese Objekte, also die Oids in den Slots anderer Objekte, über die man in Zukunft auf dieses Objekt zugreifen wird, durch direkte Zeiger ersetzen. Dieser Vorgang heißt

'Pointer Swizzling'. Wenn dann am Ende einer Sitzung veränderte Objekte wieder in den persistenten Speicher geschrieben werden sollen, müssen diese Zeiger entsprechend wieder in Oids umgewandelt werden, was man 'Unswizzling' nennt. Der anfängliche und am Ende zu Buche schlagende Mehraufwand des Swizzlings und Unswizzlings kann sich durch Zeitersparnis bei mehrmaligem Zugriff auf die Objekte auszahlen.

Beim Swizzling gibt es zwei Verfeinerungen, die, jeweils kombiniert, vier verschiedene Varianten ergeben. Dies sind *eager* (fleißig) gegenüber *lazy* (faul) und *copy* gegenüber *in-place* (vor Ort).

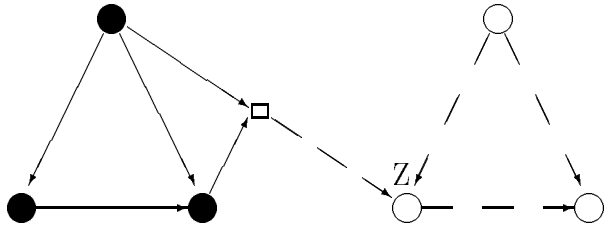
Von Eager-Swizzling spricht man, wenn die Slots des Objektes, nachdem es geladen ist, sofort nach Oids durchsucht werden und die Objekte, auf die damit verwiesen wird, gleich geladen werden und ein Swizzling ausgeführt wird. Dieser Vorgang zieht sich rekursiv oder iterativ duch die Verweise.

Beim Lazy-Swizzling werden die Objekte erst beim konkreten ersten Zugriff geladen. Dies kann wiederum auf zwei Arten geschehen. Zum einen kann man den Inhalt der slots erst einmal als Oid belassen und als 'unswizzled' markieren (*edge marking*). Wenn man dann bei einem Zugriff merkt, daß auf der Referenz noch gar kein Swizzling ausgeführt ist, wird das nachgeholt, der Oid durch die Adresse ersetzt und der Slot als 'swizzled' markiert. Ein Nachteil ist,s daß der Verweis, anhand dessen auf das Objekt zugegriffen wurde, von seinem ursprünglichen Slot über ein paar nicht mehr nachverfolgbare Stationen gewandert sein kann. Um jetzt zu garantieren, daß auf dem urprünglichen Oid, der auf dieses Objekt verweist, ein Swizzling ausgeführt wird, müssen alle Slots durchgeschaut werden, was natürlich sehr aufwendig ist, auch wenn es nur gelegentlich mit gleich mehreren Oids durchgeführt wird. Eager- und Lazy-Swizzling können auch kombiniert werden, so z.B. Eager innerhalb einer bestimmten Menge von Objekten (z.B. eine Baugruppe beim CAD) und Lazy zwischen Mengen.

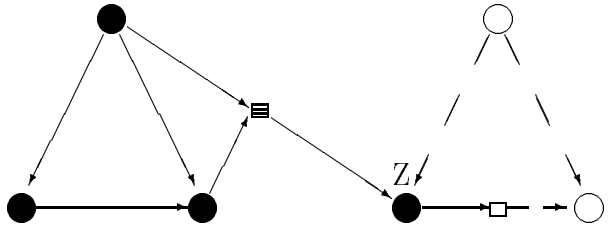
Bei der anderen Art (*node marking*) werden auch, sobald ein Objekt geladen ist, dessen Slots nach Oids durchsucht. Für jeden Verweis wird ein Pseudoobjekt, ein sogenannter Fault-Block, der als solcher erkennbar ist und der aus nur einem Slot besteht, erzeugt. Wie in Abbildung 4.3 (a) zu sehen ist, wird der Oid des Zielobjektes Z in den Slot des Fault-Blocks geschrieben (gestrichelter Pfeil) und der Slot des Objektes mit einem direkten Zeiger auf den Fault-Block besetzt (durchgezogener Pfeil). Es ergibt sich also etwas aehnliches wie ein indirekter Zeiger. Wenn nun auf das persistente Objekt Z zugegriffen werden soll, die Anwendung also ueber den direkten Zeiger auf den Fault-Block auf den Oid in dessen Slot stößt, wird dieser Oid des Objektes Z an den Objektmanager uebergeben, der dann das Objekt lädt und den Zeiger darauf zurückliefert. Zunächst wird dieser Zeiger in den Slot des Fault-Blocks geschrieben und es ergibt sich ein normaler indirekter Zeiger. Der Fault-Block ist dann ein indirekter Block (Abbildung 4.3 (b)). Jetzt können gelegentlich (der Artikel [Mos90] gibt auch keine genauere Angabe) die Objekte gescannt werden, um die indirekten Blöcke zu eliminieren und die Zeiger direkt zu setzen, wie das in Abbildung 4.3 (c) zu sehen ist.

Beim In-Place-Swizzling werden sämtliche Oids überschrieben und damit werden auch alle Objekte verändert. Am Ende, wenn der Inhalt des Puffers wieder zurückgeschrieben wird (seitenweise), muß bei jedem Objekt ein Unswizzling gemacht werden.

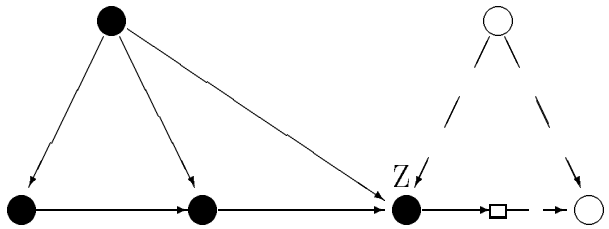
Das Copy-Swizzling hingegen macht eine Kopie des Objektes aus dem Puffer und führt das Swizzling auf der Kopie aus. Wenn Dieses Objekt nicht verändert wird, dann muß hinterher nur das Original, das ja noch die Oids in den Slots hat, aus dem Puffer auf



(a) Fault block stellvertretend für nicht residente Objekte



(b) Zielobjekt (Z) eingeholt



(c) Der indirekte Block ist durch einen garbage collector beseitigt

● residentes Objekt ○ nichtresidentes Objekt □ Fault-Block ▤ Indirekter Block

Abbildung 4.3: Knotenmarkierung fürs lazy swizzling

den persistenten Speicher zurückgeschrieben werden. Das Unswizzling muß also nicht auf allen Objekten ausgeführt werden. Im Folgenden ist abstrakter Beispielcode für die Kombinationen von Eager- mit In-Place-Swizzling und Eager- mit Copy-Swizzling abgebildet.

```
EagerInPlace(id) returns (pointer)
  p: pointer := OMLookup(id)
  if not Swizzled(p) then
    MarkSwizzled(p, id)
    for x := each slot offset in p do
      if p[x] holds an oid then
        p[x] := EagerInPlace(p[x])
return p
```

(a) EIS (Eager In-place Swizzling)

```
EagerCopy(id) returns (pointer)
  p: pointer := CopyTableLookup(id)
  if p is nil then
    q: pointer := OMLookup(id)
    p := AllocateCopy(SizeOf(q))
    CopyTableEnter(id, p)
    Copy contents of q to contents of p
    MarkSwizzled (p, id)
    for x := each slot offset in p do
      if p[x] holds an oid then
        p[x] := EagerCopy(p[x])
return p
```

(b) ECS (Eager Copy Swizzling)

4.2.3 ObjectStore

Das ObjectStore-Konzept [Obj90, LLOW91] geht von folgender Gliederung des Speichers aus. Der persistente Speicher teilt sich in Dateisysteme auf, die in Datenbanken, die in Segmente und die wiederum in Seiten. Die kleinste Stufe ist also die Seite (page), die auf eine Seite des virtuellen Speichers abgebildet werden kann. Dabei geht man von Client-Server-Architekturen aus, wobei zwischen Server und Client nur Seiten transferiert werden. Jeder Client hat einen Cache mit Cachemanager.

Die Oids sind nun Adressen des virtuellen Speichers und Zugriffe auf sie werden wie bei der gewöhnlichen Verwaltung virtuellen Speichers gehandhabt. Eine Anwendung will auf ein Objekt zugreifen und verweist dabei mit dem Oid, der Adresse, auf den virtuellen Speicher. Entweder das Objekt ist an eben dieser Adresse vorhanden und der Zugriff direkt und schnell. Oder die Hardware erkennt eine Schutzverletzung, da die betreffende Seite mit dem Objekt noch nicht auf den virtuellen Speicher abgebildet wurde und auf 'no access' gesetzt ist, und meldet das an ObjectStore zurück. Die Behandlung der Verletzungsmeldung geht nun folgendermaßen vor sich: Es wird die Seite vom Server angefordert, in den Cache geladen und dann auf den virtuellen Speicher abgebildet. Erst jetzt ist das Objekt, das sich irgendwo auf der Seite befindet, im RAM. Die Seite wird von ObjectStore auf read only gesetzt und der vorher fehlgeschlagene Zugriff wird wiederholt und kann jetzt erfolgreich durchgeführt werden. Bei Schreibzugriffen erfolgt noch einmal eine Fehlermeldung und die Seite muß auf 'read/write' gesetzt werden.

Es kann vorkommen, z.B wenn der virtuelle Speicher zu klein ist, daß es eine Kollision von zwei Seiten an der selben Adresse gibt. Dann muß eine Art Swizzling durchgeführt werden, also sämtliche Oids, die ja Adressen sind, auf der Seite durch die neuen Adressen auf der verschobenen Seite ersetzt werden.

Ein wichtiger Vorteil von ObjectStore ist, daß praktisch nicht unterschieden werden muß zwischen persistenten und nicht persistenten Objekten. Somit können ganze Bibliotheken ohne Modifikation übernommen werden. Ob ein Geschwindigkeitsvorteil gegenüber den anderen Konzepten besteht, konnte ich leider nicht feststellen. Bei dem OO7-Benchmark (s.u.) war ObjectStore zwar dabei, aber die Firma, die es vertreibt, hat kurz vor dem Druck des Artikels kundgetan, daß sie mit der Art des Benchmarks nicht zufrieden sei und hat mit Prozessen gedroht.

4.3 Vergleichende Benchmarktests

4.3.1 OO7-Benchmarktest

Bei dem OO7-Benchmarktest in [CDN93] kann man gut sehen, wie sich das Verfahren des Pointer-Swizzlings auswirkt. Beim ersten Durchgang des Testes wurden zwei Situationen betrachtet. Zum einen das erstmalige 'kalte' Durchwandern der Objektgruppen direkt und innerhalb der Gruppen über Verweise. Dabei waren alle drei objektorientierten Datenbankmanagementsysteme (OODBMS) ungefähr gleich schnell. Zum anderen wurde ein 'heißes' Durchlaufen der Objekte durchgeführt und dabei festgestellt, daß die beiden OODBMSs Ontos und Exodus, die Pointer-Swizzling machen, erheblich schneller sind als das OODBMS Objectivity. Das läßt sich leicht erklären, da bei späteren Besuchen der Objekte, auf denen ein Swizzling durchgeführt wurde, das Wandern über Verweise eine simple Addressierung ist und kein Nachschauen in Tabellen des Objektmanagers erfolgt.

4.3.2 Benchmarktests mit den verschiedenen Strategien

Vorraussetzungen

In [Mos90] werden das Mnemekonzept und die verschiedenen Varianten des Swizzling-Konzeptes bezüglich der Geschwindigkeit verglichen. Damit bei der Messung die Unterschiede der Zeiten bei den verschiedenen Varianten deutlich werden und nicht relativ zu anderen Kosten nahezu verschwinden, ging man von diesen Randbedingungen aus:

- Die Ein/Ausgabekosten sind minimal
- Es findet kein spürbares Paging oder Puffern statt
- Die Steuerung der Nebenläufigkeit fällt kaum ins Gewicht

Um außer der Variation der Konzepte auch eine Variation der Daten zu erreichen, wurden verschiedene Datensätze verwendet. Sie unterschieden sich folgendermaßen:

- Anzahl an Objekten in der Kollektion
- durchschnittliche Anzahl an Referenzen pro Objekt
- durchschnittliche Größe der Benutzerdaten pro Objekt
- Anteil veränderter Objekte
- durchschnittliche Anzahl an Besuchen pro Objekt
- durchschnittlicher Aufwand pro Objektbesuch

Die verwendeten Varianten der Konzepte sind:

- 'non-swizzling' (NS): reines Mnemeformat mit Pointer Interface
- 'non-persistent-C' (NPC): Objektorientiertes C ohne Persistenz
- 'eager in-place swizzling' (EIS)

- 'eager copy swizzling' (ECS)
- 'lazy in-place swizzling' (LIS)
- 'lazy copy swizzling' (LCS)

Die Objekte waren in Kollektionen (hier Mengen von Objekten, die von einer Anwendung verwendet werden) zusammengefaßt. Eine Kollektion war als binärer Baum aufgebaut. Es wurden zwei Sitzungen getestet, und zwar ein Erzeugen-Arbeiten-Sichern-Zyklus (EASZ) und ein Laden-Arbeiten-Sichern-Zyklus (LASZ). Beim EASZ wurden die Objekte und Kollektionen erzeugt, deren Struktur danach fest war und der Einfachheit halber im LASZ nicht mehr geändert wurde. Der Arbeitsteil wurde verschieden durchgeführt, entweder ein einfaches Durchlaufen der Objekte über die Verweise in den Slots oder zusätzliches Manipulieren der Benutzerdaten und Slots. Der Ladeteil beinhaltete gegebenenfalls auch das Swizzling, der Sichernteil das Unswizzling. Ausgeführt wurde der Test im Single-User-Mode ohne Paging und Swapping. Die Daten entstammen nicht aus einer realen Datenbank, sondern wurden fuer den Test erzeugt.

Einzelmessung

Aus den verschiedenen Einzelmessungen zum Erzeugen, Laden, Vorbereiten zum Schreiben, etc. sollen hier nur zwei erscheinen, bei denen ein Unterschied zwischen den verschiedenen Schemata deutlich wird.

Beim Laden ergab sich, daß das Copy-Swizzling, ob Eager- oder Lazy-Variante, bei kleinen Objekten etwas schneller ist, als das In-Place-Swizzling, das seinerseits aber wieder bei größeren Objekten (>30 Byte) besser abschneidet. Das kommt daher, daß beim In-Place-Swizzling das Objekt um 4 Byte größer ist. Diese 4 Byte speichern den Oid und werden für das Unswizzling benötigt, zumindest in der hier vorliegenden Implementation (wenn man diese 4 Byte umgehen könnte, wäre In-Place-Swizzling immer schneller als Copy-Swizzling). Der Fleiß stach die Faulheit immer aus (eager vs lazy). Gar kein Swizzling erwies sich erwartungsgemäß als schnellste Variante, zumindest für Objekte mit mehr als 8 Byte Benutzerdaten. In der Tabelle 4.1 kann man die Auswertung sehen.

	Verhältnis zu NS (%)				
	Anzahl Benutzerbytes				
Schema	8	24	50	200	∞
LCS	115	125	133	141	145
LIS	115	122	126	131	134
ECS	100	108	113	120	123
EIS	105	108	111	114	115

Tabelle 4.1: Kosten für das Laden bei verschiedenen Objektgrössen

Die Tabelle 4.2 zeigt das Verhältnis der Meßwerte von NPC zu denen der anderen Varianten, wenn die Anwendung die Kollektion durchläuft, ohne Arbeit zu verrichten. Beim NS findet ein zweiter Aufruf des Objektmanagers statt, um das Objekt wieder

freizugeben. Wenn man diesen wegläßt, ergibt sich der Wert von NS (angeglichen). Bei den Swizzling-Varianten wurde außerdem gemessen, wenn ein Check, ob das Objekt resident ist, unterbleibt.

Schema	Verhältnis zu NPC
NS	1.70
NS (angeglichen)	1.50
IS/CS (mit Check)	1.11
IS/CS (ohne Check)	1.02

Tabelle 4.2: Durchlauf ohne Arbeit

Ergebnisse

Als ein Ergebnis der Testreihe läßt sich feststellen: Es zeigt sich, daß sich mit der richtigen Wahl der Variante ein Geschwindigkeitsvorteil erreichen läßt.

Im Einzelnen (über ganz Zyklen und mehrere verschiedene Datensätze gesehen): Lazy-Swizzling ist immer langsamer als Eager-Swizzling. Es ist allerdings nicht viel, max. 16-18%, und wird außerdem von anderen unvermeidbaren Kosten (Laden, Dereferenzieren, Manipulieren, etc.) überdeckt, so daß es nicht groß ins Gewicht fällt. Erklärbar wird der Unterschied durch laufende Prüfungen, ob die Objekte resident sind, oder nicht, was bei Eager-Swizzling entfällt.

Der Vergleich von Copy-Swizzling mit In-Place-Swizzling ist von mehreren Parametern abhängig. Man kann feststellen, daß beim Copy-Swizzling die Kosten pro Byte um ca. 6-8% höher sind. Das ist verständlich, da jedes Objekt kopiert werden muß. Schwieriger wird es da bei den Kosten pro Objekt. Überraschenderweise ist es bei kleinen Objekten so, daß das In-Place-Swizzling mehr Kosten verursacht. Der Grund hierfür ist, wie schon vorher erwähnt, daß das Objekt zusätzliche 4 Byte braucht, damit das Unswizzling durchgeführt werden kann. Tatsächlich wäre das In-Place-Swizzling schneller als das Copy-Cwizzling, wenn man eben diese 4 Byte eliminieren könnte, bzw. eine separate Tabelle erstellte, die den Adressen wieder den Oid zuweist. Deren Verwaltungskosten müßten allerdings sehr niedrig sein. Bei großen Objekten gewinnt wieder In-Place-Swizzling die Oberhand, mit der eben erwähnten Optimierung sogar bis zu 45%. Das Absuchen der Kollektion nach modifizierten Objekten beim Copy-Swizzling kostet so viel, daß bei dieser Implementation der Vorteil, auf den nicht modifizierten Objekten kein Unswizzling durchführen zu müssen, fast verfällt. Daß das Kopieren der Objekte beim Ladevorgang keine großen Zusatzkosten verursacht, liegt wohl auch an dem Rechner im konkreten Testaufbau, der sein Cache-Memory effizient nutzt.

Das Abschätzen , ob Swizzling überhaupt einen Vorteil bringt, fällt noch schwerer. Im theoretischen Extremfall kann das Swizzling das zweieinhalbfache vom Normalen kosten. Als realistisches Beispiel wird jetzt einmal ein durchschnittliches Objekt von Smalltalk genommen: 45% der Größe der Objekte belegen die Slots. Allerdings sind in diesen Slots nicht nur Zeiger, sondern auch Daten. Insgesamt besteht das Objekt zu 20% aus initialisierten Zeigern. In der Tabelle 4.3 wird aufgezeigt, wie viele Besuche bei einem solchen Objekt der Größe b stattfinden müssen, damit sich der Mehraufwand des Swizzlings

lohnt. Ab dieser Anzahl ist es vorteilhaft, auf die entsprechende Variante des Swizzling-Konzeptes umzusteigen (In diesem Versuch). Hierbei werden auch Schemata in Betracht gezogen, die aus den gemessenen Werten berechnet wurden, auch wenn man sie nicht konkret implementiert hat. Bei LCSP und ECSP wird von LCS und ECS die Zeit abgezogen, die man für das Durchlaufen der Objekte braucht, um die modifizierten unter ihnen zu finden. LIS8 und EIS8 gehen davon aus, daß man es geschafft hat, die zusätzlichen 4 Byte streichen zu können.

Schema	Anzahl Besuche bei Größe b			
	b=20	b=40	b=100	b=200
LCS	7,5	10,9	21	38
LCSP	5,3	8,7	19	36
LIS	7,0	9,6	17	30
LIS8	4,9	7,5	15	28
ECS	4,0	5,4	10	17
ECSP	2,2	3,6	8	15
EIS	4,2	5,1	8	13
EIS8	2,7	3,7	7	11

Tabelle 4.3: Anzahl der Besuche für den break-even point

Generell kann man sagen, daß sich das Swizzling erst nach einer gewissen Anzahl von Besuchen lohnt. Normalerweise läßt sich aber nicht mehr als 30% gewinnen oder verlieren, was bei entsprechend hohen anderen Kosten (z.B. aufwendige Berechnungen aus den Benutzerdaten) verschwindend gering sein kann.

4.4 Zusammenfassung

Zur anfänglichen Frage 'To swizzle or not to swizzle' läßt sich folgendes sagen:

- Die Zeitersparnis, die sich mit Swizzling erreichen läßt, beträgt bis zu 30% für Anwendungen, die oft an Verweisen auf andere Objekte entlang gehen, aber die Objekte wenig bearbeiten. Die Objekte müssen oft besucht werden, damit sich das Swizzling lohnt.

- Die Kosten des Swizzlings wachsen mit größeren Objekten und einer steigenden Anzahl an Zeigern, die berücksichtigt werden müssen und auf denen ein Swizzling ausgeführt werden muß. Daher bestimmt auch die Struktur der Objekte den Nutzen des Swizzlings.

- Wenn genügend Hauptspeicher vorhanden ist, sollte man sich für Copy-Swizzling entscheiden, da es allgemeinere Transformationen von Datenstrukturen wie Fließkommazahlen zwischen Speicher und Festplatte erlaubt. Es kann eine Abbildung zwischen dem Original im Puffer, das dann der Darstellung der Daten auf der Festplattenstruktur mehr entspricht, und der Kopie, auf der das Swizzling ausgeführt wird, definiert werden.

- Es können auch ganz andere Faktoren als die obigen den Ausschlag für Swizzling oder kein Swizzling bzw. für eine der Variationen des Swizzlings geben, da die Kostengewinne und -verluste nicht allzu hoch sind.

Kapitel 5

Client-Server Caching (*Lars Petersson*)

5.1 Einleitung

Nach einer nun schon mehrere Jahrzehnte anhaltenden kontinuierlichen Steigerung der Leistungsfähigkeit von Rechenanlagen sowie einer ähnlich dramatischen Entwicklung auf dem Gebiet der Vernetzung dieser Anlagen, hat der Einsatz leistungsfähiger, miteinander vernetzter Arbeitsplatzrechner inzwischen weite Verbreitung in vielen Bereichen der Informationsverarbeitung gefunden. Dabei erfolgt die Vernetzung in vielen Fällen nach dem Client-Server-Modell, bei dem ein ausgewählter Rechnerknoten (Server) den anderen Rechnerknoten (Clients) auf deren Anforderung hin Daten übermittelt oder für sie Dienste erbringt.

Im Datenbankbereich existieren zahlreiche Client-Server-Ansätze insbesondere für relationale und für objektorientierte DBMS. In solchen Architekturen bietet es sich an, einen Teil der Daten in den lokalen Hauptspeichern (Cache) der Client-Rechner zu halten. Hierdurch sollte der Verkehr auf dem Netz verringert, und die Leistung der Client-CPU sowie des Client-Hauptspeichers zumindest teilweise in den Dienst des Datenbanksystems gestellt werden können. Dabei muß jedoch in jedem Zeitpunkt die Konsistenz der Daten in den Hauptspeichern der Clients und des Servers gewährleistet sein, was mit einem nicht unerheblichen Aufwand verbunden sein kann. Mit der Abwägung der Vor- und Nachteile verschiedener Sperren-basierter Algorithmen zur Wahrung der Cache-Konsistenz wird sich der zweite Punkt der vorliegenden Abhandlung befassen. Er basiert auf den Artikeln [CFLS91] und [FC94].

Unabhängig vom gewählten Verfahren zur Gewährleistung der Cache-Konsistenz ist bei objektorientierten DBMS das sog. Dual Buffering, die Aufteilung der Puffer der Clients in einen nach Seiten und einen nach isolierten Objekten organisierten Bereich eine weitere potentiell leistungssteigernde Maßnahme. Da das Anbieten akzeptabler Leistung als eine entscheidende Voraussetzung für den zukünftigen Erfolg objektorientierter DBMS angesehen wird, sollen im dritten Punkt verschiedene Varianten des Dual Buffering betrachtet werden. Die Grundlage hierzu bildet [KK93].

5.2 Sperren-basierte Verfahren zur Cache-Konsistenz

Vor der Beschreibung der Algorithmen in 2.1 soll eine grundlegende Darstellung der bei der Behandlung der Algorithmen unterstellten, und in nachfolgenden Abschnitten (2.2, 2.3) mit Hilfe eines Simulationsmodells untersuchten Architektur des betrachteten Client-Server-Datenbanksystems mit Caching gegeben werden (vgl. Abbildung 1).

Das System besteht aus einem Datenbankserver, der mit Hilfe eines lokalen Netzwerks mit einer Anzahl von Client-Workstations verbunden ist. Auf jeder dieser Client-Workstations läuft genau ein Client-Datenbankprozeß (CD) und mindestens eine vom Datenbanksystem unabhängige Client-Anwendung (CA). Diese CAs können aufgrund des Vorhandenseins eines Teils der Datenbank-Software beim Client die Datenbank als einen lokalen Diensterbringer ansehen, und mit ihr auf geeignete Weise lokal interagieren. Jede Client-Workstation verfügt über einen nach Seiten organisierten Pufferbereich für die Aufnahme von Datenbankseiten. Da die Client-Datenbankprozesse die von ihnen durchgeführten Transaktionen überdauern, können sie Daten während einer Transaktion, jedoch auch über Transaktionsgrenzen hinweg in ihrem Cache halten.

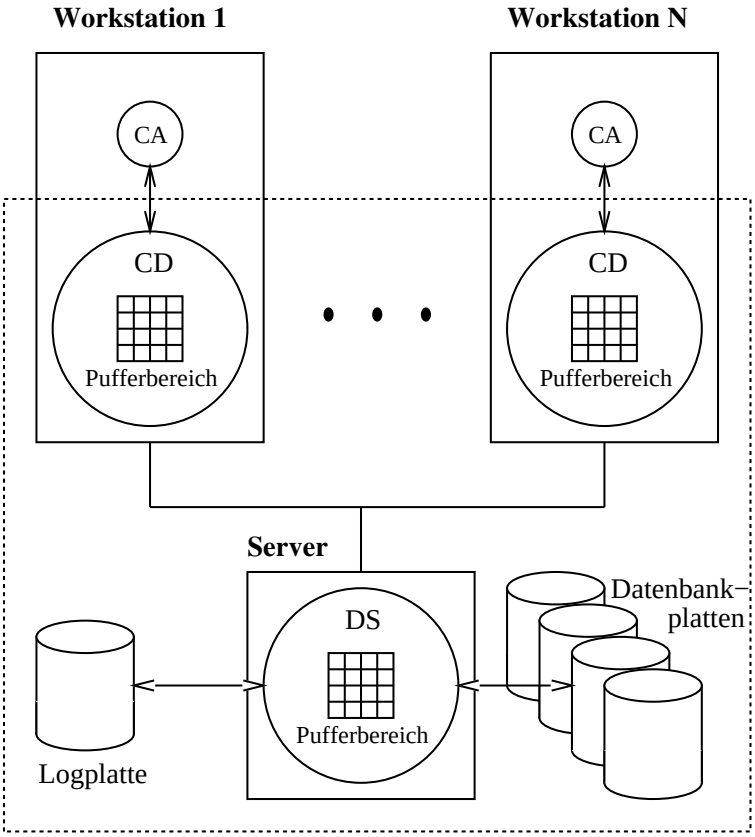


Abbildung 5.1: Architektur eines Client-Server-DBMS

Seitens der Server-Workstation läuft genau ein Server-Datenbankprozeß, der ganz ähnlich dem Client-Prozeß einen nach Seiten organisierten Pufferbereich besitzt. Ferner verfügt der Serverprozeß als einzige Komponente des DBMS über einen Zugriff auf die Platten, auf denen die Datenbasis abgelegt ist. In der hier betrachteten Konfiguration dient der Serverprozeß als Seitenserver (im Unterschied zu einem in einem verteilten objektorientierten DBMS ebenfalls denkbaren Objektserver), d. h. , daß Datenbankseiten die Grundeinheit des Datenaustauschs und der im Rahmen der Algorithmen angewandten Sperroperationen bilden.

5.2.1 Beschreibung der Verfahren

Im folgenden werden einige Verfahren zur Wahrung der Cache-Konsistenz beschrieben.

Basic Two-Phase Locking (B2PL)

Dieses Verfahren ist dadurch gekennzeichnet, daß die beim Client gehaltenen Daten nicht über das Ende einer Transaktion hinaus im Cache verbleiben, und daß sämtliche Sperroperationen sich auf Seiten im Pufferbereich des Servers beziehen. Der Server-Pufferbereich dient als sog. primary copy.

Es existieren zwei Arten Sperren:

Die Anforderung von Lesesperren durch den Client ist zugleich eine Anforderung der entsprechenden Seiten. Der Server übermittelt die fraglichen Seiten an den Client erst, nachdem er auf ihnen Lesesperren zugunsten des Clients gesetzt hat.

Schreibsperren gehen grundsätzlich aus Lesesperren durch Aufwertung hervor. Da sich die betroffene Seite schon beim Client befindet, genügt zur Anforderung einer Schreibsperre eine kurze, vom Server zu quittierende Mitteilung.

Alle Sperren werden bis zum Ende der Transaktion (Commit oder Abort) gehalten. Beim Commit werden Kopien der aktualisierten Seiten an den Server geschickt.

Die Erkennung von Verklemmungen wird bei Bedarf vom Server zentral durchgeführt. Zu ihrer Auflösung wird die an der Verklemmung beteiligte Transaktion mit dem am kürzesten zurückliegenden Startzeitpunkt abgebrochen.

Caching Two-Phase Locking (C2PL)

Der zweite Algorithmus ist insoweit eine verbesserte Variante des B2PL, als er das Caching von Daten über Transaktionsgrenzen hinweg gestattet. Auch hier werden sämtliche Sperroperationen sowie die Erkennung von Verklemmungen vom Server zentral duchgeführt.

Das wesentliche Problem bei diesem Verfahren besteht darin, dafür zu sorgen, daß die in den Puffern der Clients über Transaktionsgrenzen hinweg gehaltenen Daten nur dann auch später verwendet werden, wenn sie noch gültig, d. h. aktuell sind. Zu diesem Zweck wird jede Seite mit einer log sequence number (LSN) versehen. Wenn eine Seite im Pufferbereich des Servers aktualisiert wird, ändert sich auch ihre LSN. Beim Anfordern einer Lesesperre auf eine sich in seinem Cache befindliche Seite übermittelt der Client dem Server die LSN seiner Kopie der Seite. Falls diese LSN nicht mit der LSN der serverseitigen Kopie der Seite übereinstimmt, wird die aktuelle Version dem Client im Rahmen der Antwort auf seine Anforderung übermittelt (piggybacking). Andernfalls wird die geforderte Seite mit einer Lesesperre versehen, und der Client von der erfolgreichen Einrichtung der Sperre unterrichtet.

Optimistic Two-Phase Locking (O2PL)

Die Klasse der im folgenden vorgestellten Algorithmen unterscheidet sich von B2PL und C2PL dadurch, daß bis zum Beginn der Commit-Phase Lese- und Schreibsperren grundsätzlich nur bei den Clients gesetzt werden. Der Server besitzt lediglich eine Buchführung darüber, in welchen Client-Caches sich aktuelle Kopien welcher Seiten befinden.

Um die bis dahin nur lokal bei einem Client durchgeführten Änderungen einer Transaktion persistent zu machen, schickt der Client an den Server eine Nachricht, die insbesondere eine Kopie aller von der Transaktion aktualisierten Seiten enthält. Der Server setzt nun den Schreibsperren nicht unähnliche update-copy-Sperren auf die serverseitigen Kopien dieser Seiten, und schickt bei Erfolg prepare-to-commit-Nachrichten an alle Clients, die Kopien der aktualisierten Seiten in ihren Caches halten. Nun werden auch bei den Clients die von der Aktualisierung betroffenen Seiten mit einer update-copy-Sperre belegt.

Nachdem nun alle Sperren gesetzt worden sind, gibt es mehrere Möglichkeiten, den Algorithmus fortzusetzen. Vier davon sollen im folgenden beschrieben werden.

- **Update Invalidation (O2PL-I)**

  Bei dieser Variante von O2PL werden die in den Caches der anderen Clients (von denen das Commit nicht ausging) gehaltenen Kopien der aktualisierten Seiten verworfen.

Beim Server können unter dem Schutz der update-copy-Sperren die aktualisierten Versionen der geänderten Seiten abgelegt werden. Nachdem alle Kopien der geänderten Seiten bei den anderen Clients verworfen worden sind, entsperren diese Clients ihre update-copy-Sperren, und schicken eine prepared-to-commit-Nachricht an den Server. Erst wenn der Server diese Nachricht von allen Clients erhalten hat, können die Änderungen auf Seiten des Servers festgeschrieben werden. Zu diesem Zeitpunkt besitzt lediglich der Server sowie der Client, von dem die Änderung ausging, Kopien der aktualisierten Seiten.

- **Update Propagation (O2PL-P)**

  Hier werden alle bei anderen Clients im Cache gehaltenen Kopien der bei einem Client geänderten Seiten durch die aktuelle Version ersetzt. Da die aktuelle Version der geänderten Seiten im Rahmen einer atomaren Befehlsfolge auf dem Server und allen die Aktualisierung nicht initiierenden Clients abgelegt werden soll, wird in dieser Version von O2PL anders als im Invalidation-Fall ein Zwei-Phasen-Commit durchgeführt. Dies bedeutet, daß der Server die Meldung abwarten muß, daß auf allen Clients die angeforderten update-lock-Sperren gesetzt wurden. Erst dann kann der Server den Clients signalisieren, daß sie die alte Kopie der betroffenen Seiten durch die im Rahmen der Signalisierung gewissermaßen huckepack an die Clients übermittelte aktuelle Kopie ersetzen dürfen.

- **Ein dynamischer Algorithmus (O2PL-D)**

  Die Entwicklung der beiden bisher vorgestellten Vertreter der Klasse O2PL war durch das Ziel der Spezialisierung auf verschiedene Lastprofile motiviert. Der dynamische Ansatz soll sich eigenständig möglichst vorteilhaft an die an ihn herangetragenen Anforderungen anpassen. Er leitet Aktualisierungen solange weiter, bis er feststellt, daß er dies zu oft bzw. mit zu geringem Erfolg tut. Ein Client verfährt nach O2PL-I anstatt nach O2PL-P, wenn

  1. bereits eine Aktualisierung der Seite weitergeleitet worden ist, und
  2. auf die Seite seitdem nicht wieder zugegriffen wurde.

- **Ein neuer dynamischer Algorithmus (O2PL-ND)**

  Hier wird der zu O2PL-D gewissermaßen entgegengesetzte Ansatz verfolgt: es wird grundsätzlich nach O2PL-I verfahren. Erst wenn die folgenden beiden Bedingungen erfüllt sind, findet eine Weiterleitung aktualisierter Seiten statt:

  1. die Seite ist bereits vorher weitergeleitet worden, und auf die Seite ist seitdem wieder zugegriffen worden und
  2. die Seite ist vorher verworfen worden und diese Verwerfung hat sich als Fehler erwiesen.

  Diese beiden Bedingungen stellen sicher, daß O2PL-ND eine Seite mindestens einmal verwirft, bevor er aktuelle Seiten weiterleitet.

  Die größte Schwierigkeit bei der Implementierung des neuen dynamischen Algorithmus besteht in der Feststellung, daß die Verwerfung einer Seite ein Fehler war. Da bei der Verwerfung einer Seite auch die zugehörigen Cache-Beschreibungsinformationen

entfernt wurden, muß eine gesonderte Datenstruktur angelegt werden, in der die letzten n verworfenen Seiten vermerkt sind. Auf diese Datenstruktur wird bei der Darstellung der Ergebnisse der Simulationsexperimente unter dem Namen invalidate window Bezug genommen.

Callback Locking (CB)

Anders als O2PL in seinen verschiedenen Ausprägungen verfolgt diese Klasse von Algorithmen zur Wahrung der Cache-Konsistenz einen eher pessimistischen Ansatz. Will ein Client lesend oder schreibend auf eine Seite in seinem Cache zugreifen, die noch nicht zu seinen Gunsten mit einer passenden Sperre versehen ist, so muß er vorher (und nicht erst beim Commit) eine entsprechende Sperre beim Server anfordern. Steht eine solche Sperranforderung eines Clients im Widerspruch zu bestehenden Sperren bei anderen Clients, ersucht der Server diese anderen Clients um eine Rücknahme ihrer Sperren. Dies setzt voraus, daß der Server ähnlich wie in O2PL über Informationen darüber verfügt, in welchen Client-Caches sich Kopien welcher Seiten befinden. Der Sperranforderung des einen Clients kann erst stattgegeben werden, wenn der Server festgestellt hat, daß alle anderen zu der Anforderung in Konflikt stehenden Sperren anderer Clients zurückgesetzt worden sind. Das Setzen der Sperre selber erfolgt dann jedoch nicht beim Server, sondern auf der Kopie der Seite im Cache des Clients, der die Sperre angefordert hatte. Auf diese Weise wird bereits beim Setzen der Sperren sichergestellt, daß Transaktionen später in der Commit-Phase keinen Aufwand mehr zur Aufrechterhaltung der Konsistenz leisten müssen. Im folgenden sollen zwei Varianten des Callback-Locking beschrieben werden: Callback-Read und Callback-All.

- **Callback-Read (CB-Read)**

  Anders als bei allen bisher beschriebenen Verfahren ist es bei den CB-Verfahren möglich, Sperren über das Ende von Transaktionen hinaus bestehen zu lassen, bei CB-Read gilt dies allerdings nur für Lesesperren.

  Nachdem der Server einen Client dazu aufgefordert hat, zu einer Schreibsperranforderung eines anderen Client in Konflikt stehende Sperren aufzuheben, wird diese Aufforderung beim Client als Anforderung exklusiver Sperren auf den betroffenen Seiten angesehen. Falls der Client dieser Aufforderung aufgrund eines Konflikts nicht umgehend stattgeben kann, teilt er dem Server mit, daß die Seite(n) sich in Gebrauch befindet. Andernfalls entfernt er die Seite aus seinem Pufferbereich und schickt eine Bestätigung an den Server ...

  Nachdem der Server dem anfordernden Client das Setzen einer Schreibsperre eingeräumt hat, blockiert er alle bei ihm eingehenden Anforderungen von Lese- und Schreibsperren auf der Seite, bis die Transaktion die Schreibsperre aufgehoben hat. Darüberhinaus schickt der Client bei Ende der Transaktion Kopien der aktualisierten Seiten an den Server. In seinem Cache behält er nach Beendigung der Transaktion ebenfalls aktuelle Kopien der Seiten sowie implizite Lesesperren auf ihnen.

- **Callback-All (CB-All)**

  Dieser Algorithmus funktioniert ähnlich wie CB-Read, mit dem Unterschied, daß auch Schreibsperren bei Beendigung der Transaktion nicht freigegeben werden. Die Informationen des Servers über die bei den Clients im Cache gehaltenen Seiten

werden dahingehend erweitert, daß die Kopie einer Seite bei einem Client als exklusiv eingestuft werden kann. Analog hierzu führen die Clients darüber Buch, welche der bei ihnen im Cache gehaltenen Seiten sich exklusiv bei ihnen befinden. Geht nun der Wunsch nach einer Lesesperre auf eine bei einem anderen Client exclusiv im Cache gehaltenen Seite beim Server ein, so wird der Client mit der exklusiven Kopie der Seite ersucht, die Schreibsperre auf eine Lesesperre herabzustufen. Nicht-exklusive Kopien werden wie im CB-Read-Algorithmus behandelt. Obgleich in CB-All die exklusiven Sperren über das Ende der Transaktion hinaus gesetzt bleiben, schickt der Client beim Commit der Transaktion Kopien der aktualisierten Seite(n) an den Server. Dies dient jedoch nicht der Konsistenzwahrung, sondern der Erleichterung der Recovery.

5.2.2 Das Simulationsmodell

Um Aussagen zur Leistungsfähigkeit der in 2.1 vorgestellten Algorithmen bei verschiedenen Lastanforderungen und Systemkonfigurationen machen zu können, wurde ein detailliertes Simulationsmodell entwickelt, welches nun in drei Teilen vorgestellt werden soll.

Modellierung der Datenbank und der Lastprofile

Die Datenbank wird beschrieben durch die Anzahl in ihr enthaltener Seiten, *DatabaseSize*, sowie die Größe dieser Seiten, *PageSize*. Die Gesamtlast des Systems wird von einer festzulegenden Anzahl *NumClients* von Clients generiert.

Jede Client-Workstation generiert einen Strom von Transaktionen, wobei zwischen zwei Transaktionen eine exponentiell verteilte "Denkpause" liegt, deren Mittelwert *ThinkTime* beträgt. Eine auf einem Client ablaufende Transaktion liest zwischen 0,5* *TransactionSize* und 1,5* *TransactionSize* verschiedene Seiten aus der Datenbank, wobei im Mittel *PerPageInst* CPU-Anweisungen zum Bearbeiten einer gelesenen, und doppelt soviele zum Bearbeiten einer zu beschreibenden Seite veranschlagt werden; auch *PerPageInst* ist eine exponentialverteilte Größe. Um darstellen zu können, daß Clients bei gewissen Anwendungen bevorzugt auf bestimmte, vergleichsweise kleine Datenbankbereiche zugreifen, lassen sich durch *HotBounds* und *ColdBounds* begrenzte, heiße und kalte Regionen innerhalb der Datenbasis bestimmen. Die Wahrscheinlichkeit für den Zugriff auf eine Seite in einer heißen Region beträgt *HotAccessProb*. *HotWriteProb* und *ColdWriteProb* geben für eine gelesene Seite in Abhängigkeit von der Region, aus der sie stammt, die Wahrscheinlichkeit an, mit der sie geändert wird.

Modellierung der Clients und des Servers

In der Simulation besteht ein Client aus einer Transaction Source, einem Client Manager, einem CC Manager (von *Con*currency), einem Buffer Manager und einem Resource Manager. Die Transaction Source generiert einen Strom von Transaktionen, die jeweils als eine Zeichenkette mit geeignet kodierten Seitenzugriffen dargestellt werden. Innerhalb einer Transaktion wird dabei niemals öfter als einmal auf eine Seite zugegriffen. Der Client Manager führt die Transaktionen in Abhängigkeit des gewählten Algorithmus zur Wahrung der Cache-Konsistenz aus. Im Falle des Abbruchs einer Transaktion aufgrund von beispielsweise einer Verklemmung, führt der Client Manager den Abbruch durch, beauftragt den Buffer Manager, alle geänderten Seiten zu löschen und versucht erneut, dieselbe

Transaktion durchzuführen. Der CC Manager ist für die Sperren, und der Buffer Manager für den Cache des Clients zuständig. Der Resource Manager verwaltet die anderen Ressourcen der Clientworkstation. Der Aufbau des Servers unterscheidet sich von dem des Clients nur dadurch, daß er von einem Server Manager kontrolliert wird, der nach den Regeln des jeweiligen Algorithmus auf die Anforderungen der Clients reagieren muß.

Im folgenden soll ein Überblick über die den Datenaustausch zwischen den Clients und dem Server beschreibenden Parameter gegeben werden. Die Anzahl der CPU-Anweisungen zum Senden oder Empfangen einer Nachricht setzt sich aus der festen Komponente *FixedMsgInst* sowie *PerByteMsgInst* Anweisungen für jedes Byte der Nachricht zusammen. Eine Kontrollnachricht (z. B. zur Sperrenanforderung) hat die Größe *ControlMsgSize*. *LockInst* bezeichnet die insgesamt benötigte Anzahl der CPU-Anweisungen zum Setzen und dazugehörigen Aufheben einer Sperre. *DeadlockInterval* gibt an, in welchen Abständen der Server bei den O2PL-Algorithmen eine globale Verklemmungsbehandlung durchführt, zu welchem Zweck er von allen Clients deren waits-for-Graphen anfordert.

Modellierung der physikalischen Ressourcen

Die Leistungsfähigkeit der im System vorhandenen CPUs wird durch *ClientCPU* und *ServerCPU* in MIPS angegeben. *ClientBufSize* und *ServerBufSize* beschreiben die Größe der jeweiligen Pufferbereiche als Anteil an der Gesamtgröße der Datenbasis. Nachrichten werden von den Client- und Server-CPUs nach einer FIFO-Strategie bearbeitet, andere Aufgaben quasi-parallel. Die Bearbeitung von Nachrichten verdrängt andere Aktivitätsströme von der CPU. Die Pufferbereiche beim Client und beim Server werden nach der LRU-Strategie (*L*east *R*ecently *U*sed) verwaltet. Der Server schreibt veränderte Seiten nur dann auf die Platte, wenn sie in seinem Pufferbereich ersetzt werden sollen. Der Parameter *ServerDisk* spezifiziert die Anzahl der dem Server zugeordneten Festplatten, deren Zugriffszeit als zwischen *MinDiskTime* und *MaxDiskTime* gleichverteilt angenommen wird. Für Plattenoperationen wird jeweils zufällig eine Platte ausgewählt, die angeforderten Zugriffe auf eine Platte werden jeweils nach der FIFO-Strategie durchgeführt.

Die Komponente Network Manager des Simulationsmodells modelliert das verwendete Netzwerk als eine einfache FIFO-Struktur mit der Bandbreite *NetworkBandwidth*. Dies erscheint vertretbar, da in dem unterstellten lokalen Netzwerk die Nachrichten sich nur während einer sehr kurzen Zeitspanne tatsächlich auf dem Netz befinden, und die Netzwerkkosten hauptsächlich durch die Beanspruchung der CPUs beim Senden und Empfangen von Nachrichten entstehen.

5.2.3 Versuchsergebnisse

Die wichtigste Leistungskenngröße im Rahmen dieser Untersuchung bildet der Durchsatz des verteilten DBMS, d. h. , die Anzahl der in einer Zeiteinheit erfolgreich beendeten Transaktionen. Die die Datenbasis beschreibenden Parameter wurden für alle beschriebenen Versuche einheitlich mit den folgenden Werten belegt:

DatabaseSize	1250 Seiten, entspr. 5 MB
PageSize	4096 Byte
NumClients	1, ..., 25

ThinkTime	0 Sek.
PerPageInst	30000 Anweisungen

Die Datenbasis ist mit 5 MB relativ klein gewählt, um den Aufwand der Simulation zu begrenzen. Dies erscheint auch durchaus vertretbar, da für die vorliegenden Untersuchungen die Verhältnisse der Größen der Pufferbereiche zur Größe der Datenbasis bestimmender ist als die absolute Größe der Datenbasis.

Die Parameter zur Beschreibung des bei der Kommunikation über das Netzwerk entstehenden Aufwands sowie der physikalischen Ressourcen wurden, sofern nicht ausdrücklich anders angegeben, wie folgt gewählt:

FixedMsgInst	10000 Anweisungen
PerByteMsgInst	5000 Anweisungen pro Seite
ControlMsgSize	256 Byte
LockInst	300 Anweisungen
DeadlockInterval	1 Sek.
ClientCPU	5 MIPS
ServerCPU	10 MIPS
ClientBufSize	5% oder 25% der Datenbankgröße
ServerBufSize	25% der Datenbankgröße
ServerDisks	2
MinDiskTime	10 ms
MaxDiskTime	30 ms
NetworkBandwidth	32 megabit pro Sekunde

Im folgenden soll die Leistungsfähigkeit der beschriebenen Algorithmen für verschiedene Lastprofile in jeweils einem eigenen Punkt beschrieben werden. Diese Lastprofile wurden eigens für die vorliegende Untersuchung entwickelt, und wurden nicht aus real existierenden Anwendungen abgeleitet. Aufgrund der Vielzahl von Informationen, die sich aus der Untersuchung eines Lastprofils für die verschiedenen Algorithmen und eine variable Anzahl von Client-Workstations gewinnen läßt, sollen im Rahmen dieser Abhandlung nur einige grundlegende Lehren aus den einzelnen Versuchen exemplarisch dargestellt werden.

Lastprofil HOTCOLD

Das Lastprofil HOTCOLD wird durch folgende Parameter beschrieben:

TransactionSize	20 Seiten
HotBounds	Seiten $p, \ldots, p+49$, wobei $p = 50 * (n-1) + 1$ für Client n
ColdBounds	Rest der Datenbasis
HotAccessProb	0,8
ColdAccessProb	0,2
HotWriteProb	0,2

ColdWriteProb 0,2

Jeder Client verfügt über eine eigene, 50 Seiten umfassende heiße Region in der Datenbasis, auf die sich 80% seiner Zugriffe beziehen. Es wird somit eine Situation modelliert, in der die Clients disjunkte Regionen der Datenbank bevorzugen, in der jedoch durchaus auch Überlappungen existieren.

- *ClientBufSize*=5% für B2PL, C2PL, O2PL-I, O2PL-P, O2PL-D

  Im Falle eines kleinen Pufferbereichs beim Client schneiden grundsätzlich die drei O2PL-Algorithmen am besten ab, gefolgt von C2PL und schließlich B2PL. Im Bereich von einem bis fünf Clients steigt der Durchsatz bei allen fünf betrachteten Algorithmen annähernd linear an, was auch in der Konstanz ihres Antwortszeitverhaltens zum Ausdruck kommt. Bei einer Clientanzahl in diesem unteren Bereich wird die Überlegenheit von O2PL auf die bei diesen Algorithmen geringere Anzahl von bei Sperroperationen zu übermittelnden Nachrichten zurückgeführt. Diese ist zum einen dadurch zu erklären, daß die Sperren ausschießlich lokal bei den Clients gesetzt werden, darüberhinaus aber auch damit, daß die heißen Seiten eines Client (4% der Datenbasis) sich mit hoher Wahrscheinlichkeit im Pufferbereich des Clients befinden, und nicht erst durch eine Interaktion mit dem Server beschafft werden müssen. Die Vorteilhaftigkeit von C2PL gegenüber B2PL ergibt sich aus den bei diesem Verfahren verwendeten, im Mittel kürzeren Nachrichten. Wächst nun die Anzahl der Clients über den Wert von fünf, bleibt die Vorteilhaftigkeit der drei Varianten von O2PL zwar erhalten, jedoch wächst die Leistungsfähigkeit aller Algorithmen bis zur Clientanzahl von zehn nur noch sublinear. Ab hier sinkt der Durchsatz sogar zunächst, um schließlich (außer bei B2PL) im Bereich von 20 bis 25 Clients annähernd konstant zu werden. Der Leistungsabfall jenseits einer Clientanzahl von zehn läßt sich damit erklären, daß der Server zunehmend weniger in der Lage ist, alle heißen Seiten aller Clients in seinem Pufferbereich zu halten, sondern mehr und mehr kostspielige Plattenzugriffe tätigen muß. Die E/A-Operationen werden zum Engpaß für den Server.

  Von diesem Abfall der Trefferquote des Servers beim Hinzufügen neuer Clients ist B2PL am stärksten betroffen, da dieses Verfahren kein Caching über Transaktionsgrenzen hinweg betreibt. Unter den O2PL-Algorithmen ist O2PL-I bei einer hohen Clientanzahl aufgrund einer höheren Trefferquote des Servers überlegen. Hier werden heiße Seiten im Rahmen des Invalidation-Prozesses aus dem Pufferbereich des Clients entfernt. Wenn der Client auf diese Seiten wieder zugreifen muß, befinden sie sich mit großer Wahrscheinlichkeit noch im Pufferbereich des Servers, da das für die Invalidation verantwortliche Commit eine aktuelle Version der Seite beim Server abgelegt hat. In den beiden anderen Versionen von O2PL (P, D) werden dagegen Seiten nur im Rahmen der LRU-Strategie aus dem Pufferbereich des Clients entfernt, und können vielfach nicht aus dem Pufferbereich des Servers, sondern nur von der Platte wiedergewonnen werden.

  Die Vorteilhaftigkeit von O2PL-I gegenüber C2PL läßt sich damit erklären, daß in C2PL nicht länger aktuelle Seiten solange einen Pufferbereich beim Client beanspruchen, bis sie im Rahmen der LRU-Strategie entfernt werden. Demgegenüber werden solche Seiten in O2PL-I verworfen, sobald sie als veraltet bekannt werden. Es ist

somit in C2PL eine größere Gefahr der Verwerfung aktueller, insbesondere heißer aktueller Seiten gegeben.

- *ClientBufSize*=25% für B2PL, C2PL, O2PL-I, O2PL-P, O2PL-D

  Die Erweiterung der aggregierten Pufferbereiche des Gesamtsystems führt bei allen fünf Algorithmen außer B2PL und O2PL-P zu einer Erhöhung des Durchsatzes. B2PL verhält sich genauso wie im Fall *ClientBufSize*=5%, da die durch die Vergrößerung des Client-Pufferbereichs verbesserten Möglichkeiten zum Caching innerhalb einzelner Transaktionen aufgrund der gewählten Modellierung der Transaktionen nicht genutzt werden können.

  O2PL-P zeigt nun zumindest bei einer großen Anzahl von Clients eine deutlicher nachlassende Leistung, da aufgrund der vergrößerten Client-Pufferbereiche heiße Seiten eines Clients auf mehr anderen Clients als kalte Seiten vorhanden sind. Im Falle der Aktualisierung einer solchen Seite muß die Seite an alle anderen Clients gesendet werden, obgleich sie dort mit nicht geringer Wahrscheinlichkeit nie wieder gelesen wird. Durch diesen unnützen Kommunikationsaufwand wird die Server-CPU zum Flaschenhals des Systems. Im oben betrachteten Fall (*ClientBufSize*=5%) wurden solche kalten Seiten schneller im Rahmen der LRU-Strategie aus dem Client-Pufferbereich entfernt, es fanden somit weniger unnütze Propagierungen statt.

  Da O2PL-I unproduktive Kopien heißer Seiten bei anderen Clients verwirft, wird die Server-CPU in geringerem Maße als bei O2PL-P durch die Kommunikation in Anspruch genommen. Vielmehr entwickeln sich für O2PL die Zugriffe zu den Platten zum Engpaß, da aktualisierte Versionen von sich im Server-Pufferbereich befindlichen Seiten auf die Platten geschrieben werden müssen, bevor sie aus dem Server-Pufferbereich entfernt werden können. Im Vergleich zum ersten betrachteten Fall des HOTCOLD-Lastprofils spielt hier das Lesen von der Platte eine geringere Rolle, da einmal gelesene Seiten, insbesondere heiße, länger im Client-Pufferbereich verbleiben können.

  Da der dynamische Algorithmus O2PL-D aktualisierte Seiten solange weiterreicht, bis er dieses Verhalten als unproduktiv erkennt, ist seine Leistungsfähigkeit unterhalb der von O2PL-I angesiedelt, der von O2PL-P jedoch deutlich überlegen.

  C2PL profitiert eindeutig von der Erweiterung der Client-Pufferbereiche, passen doch nun neben den heißen Seiten eines Clients noch ein sehr viel größerer Teil der kalten Seiten (21% der Datenbasis) in die Client-Pufferbereiche. Aufgrund der häufigen Nachrichten zur Sperranforderung entwickelt sich ein Flaschenhals seitens der Server-CPU. Unter Leistungsgesichtspunkten ist C2PL bei einer hohen Anzahl von Clients zwischen O2PL-D und O2PL-P anzusiedeln.

  An dieser Stelle sei eine interessante Erscheinung erwähnt, die beim Vorhandensein einer kleinen Zahl von Clients auftritt: die hohe Korrelation zwischen den Inhalten der Pufferbereiche von Clients und des Servers. Findet beispielsweise im 1-Client-Fall der Client eine kalte Seite nicht in seinem Pufferbereich, so beträgt die Wahrscheinlichkeit, sie im Pufferbereich des Servers zu finden (obwohl der Server die Hälfte der Datenbasis in seinem Pufferbereich hält) nur 1/3, da der Server-Pufferbereich zur Hälfte die Seiten des Client-Pufferbereichs enthält. Mit einer zunehmendem Zahl von Clients wächst diese Wahrscheinlichkeit aufgrund der Tatsache, daß der Server-

Pufferbereich dann eine zufällige Auswahl der gesamten Datenbasis widergibt, stark an.

- weitere Experimente mit C2PL, O2PL, CB

  In weiteren Experimenten mit dem HOTCOLD-Lastprofil sollte auch O2PL-ND sowie die beiden CB-Algorithmen in die Untersuchung einbezogen werden. Da hierzu (geringfügig) andere Parameter gewählt wurden, erscheint eine getrennte Beschreibung dieser Experimente vertretbar. Für die Leistungsfähigkeit der im Client-Server-System verwendeten Rechner wurde *ClientCPU*=15 MIPS und *ServerCPU*=30 MIPS gewählt. Die Bandbreite *NetworkBandwidth* des Netzwerks beträgt statt 32 Mbit/s nun wahlweise 8 Mbit/s (langsames Netzwerk) oder 80 Mbit/s (schnelles Netzwerk), ferner wurden die Kosten für die Kommunikation verändert, *FixedMsgInst* wurde auf 20000 verdoppelt, für *PerByteMsgInst* wurde der Wert 2,44 Anweisungen/Byte gewählt. Wie in den bereits beschriebenen Experimenten wurden Versuche mit einem Client-Pufferbereich von 5% und von 25% der Datenbasis durchgeführt.

  Falls das langsame Netzwerk und eine Größe des Client-Pufferbereichs von 5% der Datenbasis gewählt wird, zeigt sich O2PL-ND genauso leistungsfähig wie O2PL-I, und damit insbesondere leistungsfähiger als O2PL-D, dessen Leistung mit der von O2PL-P vergleichbar ist. Die Gleichheit von O2PL-ND und O2PL-I erklärt sich aus der Tatsache, daß im Falle von Aktualisierungen O2PL-ND meistens die Alternative Invalidation wählt, und sich somit genauso verhält wie O2PL-I. Erhöht man die Größe des invalidate windows, so steigt die Gefahr, daß eine Seite fälschlicherweise verworfen wird, stetig an, und es finden zunehmend mehr Propagierungen von Seiten statt. Dabei nähert sich O2PL-ND in seinem Verhalten an O2PL-D an.

  Wird nun die Größe der Client-Pufferbereiche auf 25% der Größe der Datenbasis erhöht, so entwickelt sich nun aufgrund der geänderten Systemparameter (langsameres Netzwerk, schnellere CPUs als in den ersten beiden Punkten zu HOTCOLD) das Netzwerk zum Engpaß, wodurch die relativen Kosten nutzloser Propagierungen ansteigen. Naturgemäß leiden hierunter O2PL-I und O2PL-ND am wenigsten, O2PL-D relativ stark und O2PL-P am stärksten.

  In der beschriebenen Umgebung (langsames Netzwerk, großer Client-Pufferbereich) zeigen die CB-Algorithmen eine untereinander praktisch identische, leicht geringere Leistung als O2PL-ND. Dies liegt am großen Bedarf der CB-Algorithmen nach Datenaustausch. Während O2PL-ND nur am Ende einer Transaktion Maßnahmen zur Gewährleistung der Cache-Konsistenz ergreift, führt CB-Read konsistenzwahrende Operationen auf einzelnen Seiten gegebenenfalls auch zu früheren Zeitpunkten durch. Da CB-All im Unterschied zu CB-Read Schreibsperren im Cache halten kann, sind bei diesem Algorithmus bei einer kleinen Anzahl von Clients weniger Nachrichten über das Netzwerk zu schicken als bei CB-Read. Erst wenn die Anzahl von zehn Clients überschritten wird, beansprucht CB-All das Netzwerk in stärkerem Maße als CB-Read.

  Wird nun das schnelle Netzwerk zusammen mit kleinen Client-Caches eingesetzt, haben immer noch O2PL-I und O2PL-ND sowie O2PL-P und O2PL-D jeweils eine ähnliche Leistung, jedoch liegt der Engpaß des Systems nun nicht mehr teilweise beim Netzwerk und teilweise bei den Plattenzugriffen, sondern verschiebt sich

vollständig hin zu letzteren. Auch für CB-Read und CB-All fallen beim schnellen Netzwerk die Beschränkungen durch die begrenzte Bandbreite weg, die beiden Algorithmen bekommen einen E/A-Engpaß und zeigen eine ähnliche Leistung wie O2PL-ND.

Lastprofil PRIVATE

Das Lastprofil PRIVATE wird durch folgende Parameter beschrieben:

TransactionSize	16 Seiten
HotBounds	Seiten $p, \ldots, p+24$, wobei $p = 25 * (n-1) + 1$ für Client n
ColdBounds	Seiten 626, ...,1250 für alle Clients
HotAccessProb	0,5
ColdAccessProb	0,5
HotWriteProb	0,2
ColdWriteProb	0,0

Neben einer 25 Seiten umfassenden Region, in die 50% der Zugriffe gerichtet sind, geht die andere Hälfte der Zugriffe eines Clients in eine für alle Clients gleiche, 625 Seiten große Region, aus der ausschließlich gelesen wird. Somit ist es ausgeschlossen, daß auf verschiedenen Clients ausgeführte Transaktionen lesend und schreibend auf ein Datenelement zugreifen wollen. Das Lastprofil PRIVATE soll eine Situation modellieren, bei der beispielsweise im Rahmen eines großen CAD-Projekts jeder Ingenieur einen disjunkten Teil des Gesamtentwurfs erstellt, und zu diesem Zweck lesend auf eine Bibliothek mit Standardkomponenten zugreift. Bei den Experimenten zur Beurteilung der Leistungsfähigkeit der einzelnen Verfahren unter der durch PRIVATE modellierten Last wurde stets *ClientBufSize*=25% gewählt. Aufgrund der großen Ähnlichkeit des vorliegenden Lastprofils zu HOTCOLD unterscheiden sich die Ergebnisse dieses Versuchs nicht wesentlich von denen des vorherigen.

B2PL besitzt wieder einen E/A-Flaschenhals aufgrund der geringen Trefferquote beim Zugriff auf den Server-Pufferbereich, während für C2PL die Server-CPU aufgrund der häufigen Sperranforderungen einen Engpaß darstellt. Da sich niemals eine veraltete Kopie einer im Pufferbereich eines Clients aktualisierten Seite im Pufferbereich eines anderen Clients befinden kann, finden bei den O2PL-Algorithmen weder Verwerfungen noch Propagierungen statt, so daß alle drei Varianten von O2PL exakt das gleiche Laufzeitverhalten aufweisen. Dieses ist gegenüber B2PL und C2PL deutlich verbessert. Da O2PL mit sehr wenigen auszutauschenden Nachrichten auskommt, liegt der Engpaß für diese Verfahren bei den E/A-Operationen.

Lastprofil FEED

Das Lastprofil FEED wird durch folgende Parameter beschrieben:

TransactionSize	5 Seiten
HotBounds	Seiten $1, \ldots, 50$
ColdBounds	Rest der Datenbasis

HotAccessProb	0,8
ColdAccessProb	0,2
HotWriteProb	1,0 (schreibender Client), 0,0 (lesende Clients)
ColdWriteProb	0,0 (schreibender Client), 0,0 (lesende Clients)

Dabei soll der Client Nr. 1 im Rahmen einer Erzeuger-Verbraucher-Beziehung den anderen Clients Daten zur Verfügung stellen, auf die diese nur lesend zugreifen. Auf diese Weise läßt sich beispielsweise eine Anwendung aus dem Finanzbereich charakterisieren, bei der eine Workstation eine Kursdatenbank auf dem aktuellen Stand hält, die von den anderen Workstations aus in starkem Maße lesend genutzt wird. Hierbei gehen die Zugriffe aller Clients vorzugsweise in eine 50 Seiten umfassende Region der Datenbank.

- *ClientBufSize*=25% für B2PL, C2PL, O2PL-I, O2PL-P, O2PL-D

  B2PL und C2PL sind aufgrund der bei diesen Algorithmen notwendigen Intensität des Nachrichtenaustauschs ab einer Anzahl von ca. 10 bis 15 Clients nicht mehr in der Lage, den Gesamtdurchsatz des Systems weiter zu steigern. Sie bilden einen Engpaß bei der Server-CPU aus, und sind den O2PL-Algorithmen grundsätzlich unterlegen.

  Von den O2PL-Algorithmen verfügt O2PL-P über die beste Leistungsfähigkeit, gefolgt von O2PL-D. O2PL-I schneidet am schlechtesten ab, da jede Aktualisierung einer kräftig gelesenen Seite ihre Verwerfung bei den anderen Clients und damit in vielen Fällen ihre sofortige Wiederanforderung vom Server erforderlich macht. Hierdurch verursacht O2PL-I einen starken Verkehr auf dem Netz, und eine starke Beanspruchung der Server-CPU, die dadurch anders als in O2PL-P zum Engpaß wird.

  Das Hinzufügen neuer Clients zum System führt übrigens zu einer Verringerung des Durchsatzes des einen schreibenden Clients, da die neuen Clients nun in den Wettbewerb um die Ressourcen des Servers eintreten.

- weitere Experimente mit C2PL, O2PL, CB

  Hier wird wieder auf die leicht veränderte Modellwelt Bezug genommen, wie sie bei HOTCOLD unter dem Punkt "weitere Experimente" eingeführt wurde. Im folgenden wird grundsätzlich von einem langsamen Netzwerk mit *NetworkBandwidth*=8 Mbit/s ausgegangen. *ClientBufSize* betrage zunächst 5%.

  Von den O2PL-Algorithmen hat O2PL-I bei mehr als 15 Clients den höchsten, und O2PL-P den niedrigsten Durchsatz. Die beiden dynamischen Algorithmen liegen in ihrer Leistungsfähigkeit zwischen den beiden statischen, wobei O2PL-ND gegenüber O2PL-D einen leichten Vorteil bietet. Diese Reihenfolge der Leistungsfähigkeiten der verschiedenen O2PL-Algorithmen ergibt sich aus der Kombination eines langsamen, Propagierungen verteuernden Netzwerks mit einem kleinen Client-Pufferbereich, aus dem propagierte Seiten tendentiell schneller wieder verdrängt werden.

  Das FEED-Lastprofil unter den gegebenen Systemparametern stellt übrigens die einzige Situation dar, in der die Leistungsfähigkeit von O2PL-ND merklich von der Größe des invalidate windows abhängt. Dies liegt an der geringen Größe des Client-Pufferbereichs und der geringen Anzahl aktualisierter Seiten. Im kleinen Pufferbereich wächst die Anzahl der im Rahmen der LRU-Strategie entfernten Seiten. Falls

eine solche Seite in einem Eintrag des invalidate windows verzeichnet ist, wächst mit kleiner werdendem invalidate window die Gefahr, daß dieser Eintrag bei einer Verwerfung weiterer Seiten des Pufferbereichs aus dem invalidate window verdrängt wird. Bei einer Fenstergröße von Null nähert sich O2PL-ND an O2PL-I an, bei einer Größe von 50 an O2PL-D.

Nun wird *ClientBufSize* auf 25% der Größe der Datenbasis erhöht. In diesem Umfeld ist O2PL-P deutlich leistungsfähiger als O2PL-I. Die Clients können ihre heißen Seiten vollständig in ihren Pufferbereichen halten, sodaß weniger Propagierungen vorkommen, die unvorteilhaft sind in dem Sinne, daß propagierte Seiten schon recht bald von der LRU-Strategie entfernt werden. Als Folge muß O2PL-I mehr Seitenanforderungen an den Server senden als O2PL-P. Die beiden dynamischen Varianten von O2PL erreichen nicht ganz die Leistung von O2PL-P, sind jedoch deutlich besser als O2PL-I. Da sich O2PL-ND fast immer für eine Propagierung der aktualisierten Seite entscheidet, sind die beiden dynamischen Varianten unter Leistungsgesichtspunkten praktisch gleichwertig. Aufgrund der großen Client-Pufferbereiche werden heiße Seiten nur selten verworfen. Diese wenigen verworfenen Seiten finden auch in einem recht kleinen invalidate window Platz, sodaß die Leistungsfähigkeit von O2PL-ND von der Größe dieser Datenstruktur weitgehend unabhängig ist.

CB-Read und CB-All haben ab einer Clientanzahl von 15 einen höheren Durchsatz als O2PL-ND. Dies liegt an der unterschiedlichen Nachfrage der Algorithmen nach Netzdienstleistungen. Bei den Callback-Algorithmen muß der schreibende Client bei CB-Read jede Schreibsperre vom Server anfordern, während er bei CB-All die zurückgerufenen Schreibsperren anfordern muß (dies sind praktisch alle Schreibsperren bei mehr als fünf Clients). Ferner müssen bei CB-All die lesenden Clients die beim Schreiber gecacheten Schreibsperren zurückrufen. Bezogen auf die Anzahl der Commits sendet und empfängt der Schreiber bei CB-All bis zu sechs mal mehr Nachrichten als bei CB-Read. Weiterhin wirkt es sich für CB-All nachteilig aus, daß das Caching von Schreibsperren unter dem betrachteten Lastprofil nur wenig Sinn hat. Die beschriebenen Seiten werden meistens schon sehr bald von den Lesern benötigt, die dann zu diesem Zweck die Schreibsperre zurückrufen müssen.

Resümee

Caching kann die Leistung für manche Lastprofile deutlich erhöhen. O2PL-I zeigte in vielen Fällen eine gute Leistung, während O2PL-P beim FEED-Lastprofil überlegen war. O2PL-D kam in allen Fällen nahe an die Leistung des besseren der beiden statischen O2PL-Algorithmen heran. Gleiches gilt in sogar höherem Maße auch für O2PL-ND. C2PL war im allgemeinen zumindest den besseren der vier Varianten von O2PL unterlegen. Die beiden CB-Algorithmen haben eine ähnliche, jedoch oft etwas geringere Leistungsfähigkeit als O2PL-ND. Sie führen die Maßnahmen zur Konsistenzwahrung auf der Ebene von Seiten durch, und stellen deshalb höhere Anforderungen an die Leistungsfähigkeit des Netzwerks.

5.3 Dual Buffering

In diesem Kapitel soll die Verbindung zweier Techniken zur Pufferwaltung in einem objektorientierten Client-Server DBMS beschrieben werden. Die erste ist die Segmentierung der Puffer in Bereiche mit spezifischen Ersetzungsstrategien. Insbesondere gibt es Pufferbereiche, die nach Seiten, und solche, die nach Objekten gegliedert sind. Dieses wird kombiniert mit dem eigentlichen Dual Buffering, welches die Entscheidung zum Inhalt hat, ob Objekte aus ihren Seiten heraus in nach Objekten gegliederte Speicherbereiche kopiert werden sollen, oder ob Seiten mit vielen interessierenden Objekten nur als Seite im Seitenbereich des Speichers abgelegt werden sollen. Bei schlecht geclusterten Datenbeständen soll durch die Extraktion der mit geringer Dichte über die Seiten verteilten interessierenden Objekte die Ausnutzung des Speichers erhöht werden.

Ähnlich wie bei der Beschreibung der Algorithmen zur Gewährleistung der Cache-Konsistenz wird in den folgenden Abschnitten davon ausgegangen, daß der Server als Page-Server fungiert, und über einen gewöhnlichen, nach Seiten organisierten Pufferbereich verfügt. Jeder Seitenfehler bei einem Client führt somit zur Anforderung der Seite vom Server. Diese vorliegende Architektur soll die Kommunikation zwischen Client und Server minimieren. Dual Buffering findet ausschließlich bei den Clients statt, die im übrigen über keine eigenen Platten verfügen sollen.

5.3.1 Strategien

Verschiedene Dual-Buffering-Strategien lassen sich durch zwei Größen klassifizieren:

copying time: Diese Größe bestimmt, zu welchem Zeitpunkt bzw. als Reaktion auf welches Ereignis ein Objekt von seiner sich in einem Seitenbereich befindlichen Heimatseite in einen Objektbereich kopiert wird.

relocation time: Diese Größe bestimmt, wann ein zu einem früheren Zeitpunkt aus seiner Heimatseite herauskopiertes, sich in einem Objektbereich befindliches Objekt aufgegeben wird. Falls das Objekt verändert wurde, muß es vorher auf seine sich in einem Seitenbereich befindliche Heimatseite zurückgeführt (relokiert) werden.

Zur Wahl jedes dieser beiden Zeitpunkte sind jeweils zwei Extreme vorstellbar, und zwar jeweils eine aggressive (eager) und eine konservative (lazy) Vorgehensweise. Aus der geeigneten Kombination beider Strategien für copying time und relocation time ergeben sich vier Grundstrategien zur Durchführung des Dual Buffering.

Darüberhinaus gibt es natürlich noch die Möglichkeit einer NOC-Strategie (no object copying), bei der grundsätzlich keine Objekte aus ihren Heimatseiten herauskopiert werden. Hieraus ergibt sich, daß auch keine Relokation erforderlich wird.

Eager Object Copying (EOC)

Im Rahmen des EOC wird ein Objekt aus seiner Heimatseite in einen Objektbereich kopiert, wenn zum ersten Mal darauf zugegriffen wird. Sämtliche Operationen auf dem Objekt beziehen sich ab diesem Zeitpunkt auf seine Kopie in einem Objektbereich. Deshalb sollten die Objektbereiche im Vergleich zu den Seitenbereichen groß gewählt werden, um den Speicherplatz nicht mit Seiten zu belegen, auf deren herauskopierte Objekte nicht mehr direkt zugegriffen wird.

In vielen Fällen kann die Anzahl der Seitenfehler mit EOC gegenüber NOC verringert werden. Wenn jedoch im Rahmen einer Anwendung große Datenmengen sequentiell durchlaufen werden sollen, kann die Anzahl der Seitenfehler grundsätzlich nicht reduziert werden. Das Kopieren von Objekten im Rahmen von EOC ist dann völlig unnötig, und kann sogar, falls die Objekte modifiziert wurden, bei der Relokation zu weiteren Seitenfehlern führen, wenn die Heimatseiten der Objekte inzwischen aus den Pufferbereichen entfernt wurden. Ferner erweist sich EOC als wenig leistungsfähig, wenn die Objektbasis bereits gut geclustert vorliegt. In diesem Fall kann das stets mit Aufwand verbundene Kopieren von Objekten die Nutzung der Pufferbereiche nicht grundlegend verbessern.

Lazy Object Copying (LOC)

Anders als bei EOC werden beim LOC Objekte so spät wie möglich aus ihrer Heimatseite in einen Objektbereich kopiert. Dies geschieht erst dann, wenn die Seite aus dem Puffer entfernt wird, die aktuelle Anwendung jedoch weiterhin auf das in ihr enthaltene Objekt zugreifen möchte.

Beim sequentiellen Durchlaufen von Datenmengen verhält sich LOC genauso wie das unter solchen Umständen besonders geeignete NOC. Da die bearbeiteten Objekte von der Anwendung freigegeben werden, bevor ihre Heimatseite entfernt wird, findet kein Kopieren von Objekten statt. Im Fall wenig geclusterter Daten, bei dem EOC gut abschneiden sollte, führt LOC die gleichen Kopieroperationen durch wie EOC, sie finden jedoch erst zu einem späteren Zeitpunkt statt. Deshalb sollte LOC unter diesen Umständen ähnlich gut abschneiden wie EOC. Ein leichter Vorteil ergibt sich für LOC aus der Tatsache, daß bei diesem Verfahren die Kopieroperationen unmittelbar nach einem aufgetretenen Seitenfehler durchgeführt werden, und somit während der Wartezeit auf die vom Server neu zu liefernde Seite stattfinden können. Diese leistungssteigernde Form interner Parallelität findet beim EOC nicht statt.

Im Falle der Modifikation von Objekten weist LOC nicht immer eine optimale Leistungsfähigkeit auf. Es wird deshalb zwischen frühen Aktualisierungen, zu einem Zeitpunkt, in dem sich die Heimatseite noch im Seitenpuffer befindet, und späten Aktualisierungen, nach dem Entfernen der Heimatseite, unterschieden. Werden an einem Objekt auf der Heimatseite frühe Aktualisierungen durchgeführt, werden die Veränderungen beim Entfernen der Seite in die Datenbasis geschrieben. Treten nach diesem Zeitpunkt weitere Änderungen am sich nun im Objektbereich befindlichen Objekt auf, muß die Seite ein weiteres Mal zum Server gesendet, und dort in die Datenbasis eingetragen werden.

Als Konsequenz aus der Unterteilung der Aktualisierungen von Objekten in frühe und späte wurden zwei Varianten von LOC entwickelt: LOC^- verhindert die Entfernung von Seiten aus dem Seitenbereich, die modifizierte Objekte enthalten, auf welche die Anwendung weiterhin zugreifen möchte. LOC^+ kopiert Objekte in einen Objektbereich, sobald sie das erste Mal verändert werden. Sowohl LOC^- als auch LOC^+ verhindern das Senden von Seiten mit frühen Aktualisierungen an den Server.

Eager Relocation (ERL)

Im Rahmen dieser Strategie wird ein Objekt unabhängig davon, ob es modifiziert wurde oder nicht, auf seine Heimatseite zurückgeführt, sobald die Heimatseite erneut in einen Seitenbereich des Puffers gebracht wird.

Wird ERL mit LOC kombiniert, existiert zu jedem Zeitpunkt höchstens eine Kopie eines Objekts bei einem Client, entweder im Rahmen seiner Heimatseite in einem Seitenbereich oder als Kopie in einem Objektbereich. Andererseits kann es bei ERL jedoch vorkommen, daß Objekte unnötigerweise in den Seitenbereich zurückgeführt werden. Dies trifft insbesondere dann zu, wenn kurze Zeit nach der Rückführung die Seite aus dem Seitenbereich entfernt wird, und das Objekt erneut in den Objektbereich kopiert werden muß.

Die Kombination von ERL mit EOC scheint einen Widerspruch zu enthalten: einerseits soll der Zugriff auf Objekte möglichst auf Kopien erfolgen (EOC), andererseits sollen diese Kopien zum frühestmöglichen Zeitpunkt aufgegeben werden (ERL). Aus diesem Grund wird die Kombination EOC/ERL in 3.2 und 3.3 nicht untersucht.

Lazy Relocation (LRL)

Bei der Strategie LRL ist man bestrebt, Objekte so spät wie möglich aus einem Objektbereich in seine Heimatseite zurückzuführen. Die Kopie des Objekts wird erst aufgegeben, wenn die Anwendung signalisiert hat, daß sie auf das Objekt nicht mehr zugreifen muß, und zusätzlich der von dem Objekt eingenommene Speicherplatz anderweitig verwendet werden soll.

Anders als ERL vermeidet also LRL die unnötige Rückführung von Objekten in ihre Heimatseite. Es besteht jedoch andererseits die Gefahr, daß beim Rückführen eines aktualisierten Objekts ein Seitenfehler auftritt, und seine Heimatseite erst aufwendig vom Server beschafft werden muß. Darüberhinaus ist beim LRL die Ausnutzung des Puffers i. allg. geringer, da im Falle des Vorhandenseins der Heimatseite eines Objekts das Objekt doppelt im Puffer vorkommt.

Verfeinerte Strategien für die Relokation

Die Leistungsfähigkeit der Vorgehensweisen bei der Rückführung von Objekten auf ihre Heimatseiten ist in starkem Maße davon abhängig, ob es sich um veränderte oder originale Kopien von Objekten handelt. Um diesem Umstand gerecht zu werden, sollen an dieser Stelle noch zwei zwischen LRL und ERL angesiedelte Strategien Erwähnung finden:

- Zurückführung unveränderter Objekte nach ERL und modifizierter Objekte nach LRL

  Hierbei muß eine Seite seltener in die Datenbasis geschrieben werden, da sie mit geringerer Wahrscheinlichkeit veränderte Objekte enthält. Soll sie aus einem Seitenbereich entfernt werden, so genügt es, sie dort zu löschen.

- Zurückführung modifizierter Objekte nach ERL und unveränderter Objekte nach LRL

  Diese Strategie reduziert die Anzahl der Seitenfehler, die dann entstehen können, wenn ein modifiziertes Objekt in die Datenbasis geschrieben werden soll, und sich seine Heimatseite nicht mehr im Seitenbereich des Puffers befindet.

5.3.2 Versuchsanordnung

Die Experimente zur Beurteilung der verschiedenen Varianten des Dual Buffering wurden auf zwei Sun Sparc 10 Workstations durchgeführt, die mittels eines Ethernet verbunden waren. Der Server verfügte über eine Platte der Größe 424 MB, und einen nach Seiten organisierten Pufferbereich. Der Pufferbereich umfaßte 1000 Seiten zu 4 KB, was der Hälfte der Datenbasis von 2000 Seiten entspricht.

Der Client wurde im single-user mode betrieben. Neben einem Objektbereich und einem Seitenbereich wurde nur noch ein weiterer Bereich eingerichtet, in den aus den anderen beiden Bereichen entfernte Seiten und Objekte kopiert werden, und von wo sie gelöscht werden, wenn der Speicherplatz anderweitig Verwendung finden soll. Der Objekt- und der Seitenbereich unterliegen einer LRU-Strategie, während für den dritten Bereich ein FIFO-Verfahren verwandt wurde.

Zur Leistungsmessung wurden die Durchläufe T1, T2a und T2b des OO7-Benchmarks zur Beurteilung objektorientierter DBMS verwendet (zu Details bez. dieses Benchmarkverfahrens vgl. [CDN93]). Dabei werden in T1 nur Daten gelesen, bei T2a gibt es wenige, und bei T2b viele Aktualisierungen. T1, T2a und T2b wurden auf die sog. kleine OO7-Datenbasis angewandt. Die gefundenen Ergebnisse lassen sich problemlos auf größere Objektbasen und größere Pufferbereiche übertragen, da lediglich das Verhältnis der Größe der Datenbasis zur Größe der Pufferbereiche letztlich von Bedeutung ist.

Die Clusterung von Daten hat einen großen Einfluß auf die Leistungsfähigkeit der Pufferverwaltung. Deshalb werden drei verschiedene Formen des Clusterings betrachtet:

Time-of-creation (TOC): Hierbei werden die Objekte in der Reihenfolge sequentiell in die Objektbasis eingetragen, in der sie angelegt wurden. Auf diese Weise kommt eine sehr gute, aber keine optimale Clusterung zustande. Beispielsweise mußten bei den Durchläufen T2a und T2b jeweils fast alle Seiten modifiziert werden.

Type-based (TB): Für jeden Objekttyp wird eine logische Datei angelegt, in welche die zugehörigen Objekte in der Reihenfolge ihrer Erzeugung eingetragen wurden. Eine TB-Clusterung ist besonders für die Durchläufe T2a und T2b als vorteilhaft anzusehen, da die hierbei zu modifizierenden Objekte aufgrund der Konstruktion des OO7-Benchmarks auf einem geringen Teil der insgesamt gelesenen Seiten nahe beieinanderliegen, und somit nur vergleichsweise wenige Seiten persistent gemacht werden müssen.

RANDOM: Hierbei werden die Objekte zufällig über die Seiten verteilt.

5.3.3 Ergebnisse

Im folgenden sollen die Versuchsergebnisse zur Leistungsmessung im Überblick dargestellt werden. Dabei werden die Kombinationen von LOC, LOC^- und LOC^+ mit LRL und ERL sowie von EOC mit LRL betrachtet, und mit der Strategie NOC verglichen.

T1-Durchlauf

Da hier keine Aktualisierungen stattfinden, sind die Ergebnisse für LOC, LOC^- und LOC^+ identisch. Zunächst seien die Pufferbereiche des Clients und des Servers zu Beginn leer. Beim Vergleich von NOC mit den Techniken des Dual Buffering für die TB-geclusterte

Datenbasis können in Abhängigkeit der Größe des Client-Pufferbereichs drei Fälle unterschieden werden.

Bei einer sehr geringen Puffergröße (weniger als 400 Seiten) wurden die meisten Objekte aus dem Objektbereich entfernt, bevor sie ein zweites Mal referenziert werden konnten. Aufgrund des hohen Kopieraufwands bei EOC/LRL erwies sich Dual Buffering in diesem Fall als leistungsmindernd. Bei den LOC-Strategien fanden hingegen nur wenige Kopiervorgänge statt, da die meisten Objekte nicht mehr benötigt wurden, wenn ihre Heimatseite ersetzt wurde.

Im Vergleich mit NOC war Dual Buffering besonders effizient, wenn der Client-Puffer einerseits einen großen Teil der von der Anwendung benötigten Objekte aufnehmen konnte, aber andererseits zu klein war für die entsprechenden Heimatseiten. Dies war im Bereich von 800 bis 1400 Seiten der Fall. Im Maximum konnte LOC/ERL die Anzahl der Seitenfehler um 40% und die Antwortszeit um 30% gegenüber NOC vermindern. Aufgrund der guten Clusterung der Daten und der geringeren Parallelität von EOC/LRL reichte die Leistungsfähigkeit dieses Verfahrens nicht an die der LOC-Verfahren heran.

Wurde schließlich der Client-Pufferbereich fast so groß wie die Objektbasis gewählt, konnten durch Dual Buffering allenfalls noch geringe Leistungszuwächse erzielt werden. Unter LOC wurden kaum Objekte kopiert, da nur wenige Seiten aus dem Client-Pufferbereich entfernt wurden. Aufgrund der vielen unnötigen Kopiervorgänge und der wegen der schlechten Pufferausnutzung großen Zahl von Seitenfehlern war EOC/LRL der NOC-Variante deutlich unterlegen.

Aufgrund der guten Clusterung der Daten kam es nur selten vor, daß die Heimatseite eines sich im Objektbereich befindlichen Objekts erneut in den Seitenbereich geladen werden mußte, um auf ein anderes Objekt zugreifen zu können. Deshalb kam es in den bisher beschriebenen Experimenten nur zu einem minimalen Vorsprung von LOC/ERL gegenüber LOC/LRL.

Bei einer schlechten Clusterung jedoch müssen Seiten mehrere Male geladen werden, bis alle benötigten Objekte kopiert sind. In einem solchen Umfeld führt ERL zur Vermeidung von Duplikaten und damit zu einer besseren Ausnutzung des Puffers und zu weniger Seitenfehlern. Konkret war bei der RANDOM-Clusterung LOC/ERL durchweg die beste Strategie. Bei einer Puffergröße von 800 Seiten verursachte sie einen bis zu zwölfmal höheren Aufwand als LOC/LRL durch das Kopieren von Objekten zwischen den Pufferbereichen. Gegenüber NOC konnte LOC/ERL bis zu 60% der Antwortszeit einsparen (bei TB-Clusterung bestenfalls 30%). Bei einer Puffergröße von 2000 Seiten sank die Ausnutzung der Puffer aufgrund der hohen Zahl von Duplikaten bei EOC/LRL und LOC/LRL unter den korrespondierenden Wert von NOC, was auch in einem vergleichsweise schlechten Antwortszeitverhalten zum Ausdruck kam.

Insgesamt stellt sich LOC/ERL als das unter den gegebenen Umständen überlegene Verfahren dar.

Um die Leistungsfähigkeit der Dual-Buffering-Algorithmen in einem warmgelaufenen System beurteilen zu können, wurde der T1-Durchlauf zweimal in Folge gestartet. Dabei stellte sich heraus, daß die meisten Kopiervorgänge in der ersten Phase durchgeführt wurden, und somit im zweiten Durchlauf die Leistungssteigerung gegenüber NOC am größten war.

T2a- und T2b-Durchläufe

Bei Anwendungen mit häufigen Aktualisierungen (T2b-Durchlauf bei TB-Clusterung) kann EOC/LRL aufgrund des Rückschreibens modifizierter Objekte beachtliche Leistungseinbußen erleiden. Falls der Client über einen kleinen Pufferbereich verfügt, ist die Heimatseite eines Objekts oft schon ersetzt worden, wenn das Objekt zurückgeführt werden soll. Bei einer Puffergröße von mehr als 1600 Seiten tritt dieses Phänomen sehr viel seltener auf, jedoch leidet EOC/LOC in diesem Fall immer noch unter den gleichen Nachteilen wie beim T1-Durchlauf. Lediglich bei einer mittleren Puffergröße von 1200 Seiten war EOC/LRL der Strategie NOC aufgrund der besseren Pufferausnutzung überlegen.

Die beste Variante von LOC, nämlich LOC^- war gegenüber NOC sogar noch deutlicher überlegen als beim T1-Durchlauf. Diesmal wurde nicht nur die Anzahl der Seitenfehler (gegenüber NOC) reduziert, sondern es wurde darüberhinaus die Anzahl der zum Server gesendeten, modifizierten Seiten deutlich vermindert. In diesem Umfeld (T2b-Durchlauf bei TB-Clusterung) stellte LOC^- die beste Variante der drei verschiedenen LOC-Strategien dar. Da die modifizierten Objekte sich auf gut geclusterten Seiten befinden, wird das Herauskopieren von Objekten aus ihrer Heimatseite in den Objektbereich hier zu Recht besonders restriktiv gehandhabt. Dieser Eindruck wird dadurch gestützt, daß LOC^+ die geringste Leistungsfähigkeit zeigte. Jedoch vermindern beide Abwandlungen des reinen LOC die Anzahl der zum Server gesendeten Seiten. LOC^-, indem es späte Aktualisierungen vermeidet, und LOC^+ dadurch, daß frühe Aktualisierungen nicht beim Server persistent gemacht werden.

Als ein mögliches Kriterium zur Beurteilung einer Pufferverwaltung kann das sog. space-time product angesehen werden, welches in einem multi-programming-System als das Integral des von einem Programm beanspruchten Speicherplatzes über die gesamte Programmlaufzeit definiert ist. Im vorliegenden Fall fester Puffergrößen ergibt sich das space-time product multiplikativ aus der Puffergröße und der Antwortszeit/Laufzeit des Programms. Die für die Durchläufe T2a und T2b über die nach verschiedenen Verfahren geclusterte Datenbasis zu untersuchenden Dual-Buffering-Strategien NOC, EOC/LRL, LOC^s/LRL und LOC^s/ERL ($s \in \{+, -, \epsilon\}$) wurden nun - hauptsächlich durch die Variation der Puffergröße sowie die Aufteilung des Puffers in Objekt- und Seitenbereiche - so konfiguriert, daß das space-time product minimal wurde. Im folgenden wird ein niedriges minimales space-time product als ein Maß der relativen Überlegenheit einer Strategie angesehen.

Bei warmen Durchläufen war jede gut konfigurierte Form des Dual Buffering dem NOC-Ansatz überlegen. Bei EOC/LRL waren allerdings die Zugewinne aufgrund der Schwierigkeiten beim Zurückschreiben modifizierter kopierter Objekte nicht so eindrucksvoll. So konnte beim T1-Durchlauf bei einer RANDOM-geclusterten Datenbasis EOC/LRL das space-time product um 84,5% reduzieren, beim T2b-Durchlauf jedoch nur um 19,7%. Auch die Unterschiede zwischen den LOC-Verfahren und NOC verringerten sich mit der Anzahl der Aktualisierungen, jedoch scheint die Leistungsfähigkeit von LOC sehr viel weniger sensibel auf Aktualisierungen zu reagieren als EOC.

Was den Zeitpunkt der Rückführung kopierter Objekte auf ihre Heimatseite anbelangt, so zeigt sich ERL im Vergleich zu LRL klar überlegen.

Unter den drei betrachteten Varianten von LOC kann keine als die eindeutig beste angesehen werden, die Unterschiede in ihren space-time products waren auch nicht sehr groß. Das reine LOC war selten das beste Verfahren. LOC^- kopierte oft zuwenige Objekte

(z.B. bei TOC-Clusterung), und LOC^+ zuviele (z.B. bei RANDOM-Clusterung).

Teil III

Synchronisation

Kapitel 6

Semantische Concurrency Control (*Torsten Ilse*)

6.1 Einführung

Diese Arbeit beschäftigt sich mit der Erweiterung bzw. Modifikation von Transaktionsverwaltungssystemen hinsichtlich der Erhöhung der Parallelität durch Ausnutzung von semantischem Wissen über die beteiligten Objekte und die sie manipulierenden Operationen. Die Aktualität dieses Themas ist durch die wachsende Bedeutung von objektorientierten Systemen, hier speziell Datenhaltungssystemen, gegeben. Gerade bei diesen Systemen kommt es aufgrund der nebenläufigen Arbeit vieler Benutzer auf gemeinsamen Datenobjekten auf die möglichst vollständige Ausnutzung der semantischen Informationen an, um die parallele Abarbeitung von Transaktionen nicht unnötig zu behindern. Bei dieser Ausarbeitung, die auf den Artikeln basiert, werden zur Veranschaulichung der wichtigsten Konzepte gemeinsame Abstrakte Datentypen verwendet, die im wesentlichen Erweiterungen der bekannten Typen Keller, Warteschlange und Verzeichnis mit den entsprechenden Operatoren sind. Es werden mehrere Ansätze zur Nutzung des semantischen Wissens über die Datentypen vorgestellt, die zwar auf diese Typen beschränkt sind, andererseits aber eine beträchtliche Erhöhung der Parallelität ermöglichen. Diese Konzepte können natürlich auch auf andere Typen übertragen werden. Ausgangspunkt für die Überlegungen ist das bekannte Leser-Schreiber-Schema und das Prinzip des Zwei-Phasen-Sperrens. Durch zunehmende Verfeinerung dieser Grundprinzipien erreicht man eine immer weitere Lockerung der Restriktionen bezüglich der Nebenläufigkeit. Hierbei versucht man, Erkenntnisse über die Art des Zugriffs auf die Objekte, die Art der Veränderung der Objekte, sowie Untersuchungen über die Lokalität der Auswirkungen von Operationen unnötige Beschränkungen zu vermeiden. Eine wichtige Frage ist hier die gegenseitige Beeinflussung von Transaktionen und die Konsequenzen für die Konsistenz der Datenbasis. Man muß sich bei der Analyse aller hier vorgestellten Konzepte im klaren darüber sein, daß eine Erhöhung der Parallelität im allgemeinen einen erhöhten Verwaltungsaufwand erfordert, da die Sperrgranularität des Synchronisationsmechanismus erhöht wird. Es wird nicht das ganze Objekt gesperrt, sondern durch Extraktion von semantischer Information versucht man mit Sperren auf Teilen des Objektes auszukommen. Im ungünstigsten Fall werden dadurch die Zeitgewinne, die durch Nebenläufigkeit erzielt wurden, wieder zunichte gemacht. Bei der Implementierung von entsprechenden Verfahren kommt es also darauf an, einen Kompromiß zwischen diesen Komponenten zu finden.

6.2 Grundlagen der Transationsverwaltung

Um bestimmte Erscheinungen später besser verstehen zu können, sollen noch einmal bestimmte Eigenschaften von Transaktionen erklärt werden:

Unter einer Transaktion versteht man eine Folge von Operationen auf Daten, die von BOT (begin of transaction) und EOT (end of transaction) eingegrenzt wird. Eine erfolgreich beendete Transaktion wird mit COMMIT bestätigt. Die Ergebnisse der Transaktion werden erst jetzt dauerhaft in der Datenbasis festgeschrieben. Sollte eine Transaktion vor Ende abbrechen müssen (ABORT), so werden alle bis dahin erfolgten Veränderungen an der Datenbasis wieder rückgängig gemacht, um die Konsistenz der Datenbasis nicht zu gefährden.

Ausgangspunkt der Betrachtungen werden die sogenannten ACID-Eigenschaften sein. ACID steht für:

- (A) **Atomizität**. Bei einer Transaktion werden entweder alle Operationen ausgeführt oder keine. Das ist vor allem wichtig, um keine halbfertigen Transaktionen, die unter Umständen einen temporär ungültigen Zustand der Datenbasis herbeiführen, ihre Veränderungen sichtbar werden zu lassen. Erst nach erfolgreichem Abschluß aller Operationen einer Transaktion (COMMIT) werden die Auswirkungen in der Datenbasis festgeschrieben und für andere Transaktionen sichtbar. Die Atomizitätsbedingung impliziert, daß es zu jeder Operation eine entsprechende UNDO-Operation geben muß. Damit können die Ergebnisse einer unvollständigen Transaktion nach einem Abbruch wieder rückgängig gemacht werden.

- (C) **Konsistenz (consistency)**. Die Transaktion setzt auf einem konsistenten Zustand der Datenbasis auf und muß die Datenbasis auch in einem konsistenten Zustand verlassen. Vor allem bei Abbrüchen muß beachtet werden, daß ev. inkonsistente Zwischenzustände, die zwischen BOT und EOT erlaubt sein können, da sie noch nicht nach außen sichtbar sind, wieder rückgängig gemacht werden.

- (I) **Isolation**. Die Auswirkungen einer Transaktion dürfen andere Transaktionen nicht in beeinflussen und eine Transaktion darf entsprechend nicht von anderen Transaktionen während der Abarbeitung gestört werden. Jede Transaktion arbeitet unter der Annahme, daß ihr die Datenbasis exklusiv zur Verfügung steht.

- (D) **Dauerhaftigkeit oder Permanenz**. Ergebnisse von Transaktionen, die mit COMMIT bestätigt wurden, sind in der Datenbasis festgeschrieben und können nicht mehr verloren gehen.

Oft wird die Forderung nach Serialisierbarkeit gestellt. Diese Forderung ist nicht dogmatisch zu sehen. Unter Beachtung einiger Nebenbedingungen ist es möglich, zur Erhöhung der Parallelität auf strenge Serialisierbarkeit zu verzichten. In diesem Fall wird die Einhaltung einer schwächeren Ordung auf der Gruppe der Transaktionen verlangt. Eine Verzögerung bei der Abarbeitung von Operationen können die Forderungen zur Vermeidung von kaskadierenden Abbrüchen mit sich bringen. Da keine Transaktion Ergebnisse einer anderen Transaktion sehen darf, die noch nicht festgeschrieben sind, kann es hier zu Behinderungen kommen, auf die aber noch eingegangen wird.

6.2.1 Zugriffsarten von Transaktionen und gegenseitige Beeinflussung

Bisher geht man davon aus, daß eine Überlappung von Transaktionen möglich ist, allerdings dürfen diese Transaktionen nicht das gleiche Objekt bearbeiten. Vor dem Hintergrund von kaskadierenden Abbrüchen ist diese Einschränkung in vielen Fällen sinnvoll. Man würde allerdings Parallelisierungspotential verschenken, wenn man diese Vorschrift undifferenziert immer anwenden würde. Man wird feststellen, daß es einige Fälle gibt, in denen es möglich ist, daß Transaktionen parallel auf einem Objekt arbeiten. An dieser Stelle ist es erforderlich, zu untersuchen, welche Arten von Zugriffen auf Objekte zu unterscheiden sind, und welche Beziehungen zwischen Operationen auftreten können, die auf demselben Objekt arbeiten. Zunächst werden die Operationen in Lese- und in Schreiboperationen unterteilt. Das Lesen und Schreiben muß sich nicht auf das gesamte Objekt beziehen. Genausogut ist es möglich, daß nur Teile des Objekts betroffen sind. Die Operationen auf dem Objekt sind teilweise zustandsabhängig. Je nach Zustand muß die entspechende Operation anderen Zugriffsklassen zugeordnet werden. So ist es vorstellbar, daß eine Operation bei einem Objektzustand s' einen lesenden Zugriff ausführt, bei Zustand s" ein schreibender Zugriff erfolgt und bei s"' der Zugriff nicht möglich ist. Außerdem gibt es Klassen von Operationen, die sowohl das Objekt lesen, als auch schreibend zugreifen.

Für die weitere Darstellung ist es vorteilhaft, an einem Beispiel arbeiten zu können. Besonders geeignet, stellvertretend für Abstrakte Datentypen, ist eine Kombination aus einer Warteschlange (Queue) und einem Keller (Stack). Für dieses Objekt wird der Name QStack verwendet.QStack hat eine begrenzte Kapazität. Die Elemente dieses Datentyps haben jeweils einen Wert oder eine Belegung.

Die folgende Tabelle enthält die auf QStack definierten Operationen und die zustandsunabhängige Zuordnung als (L)eser, (S)chreiber oder (L)eser-(S)chreiber.

POP	LS	DEQ	LS
PUSH	LS	SIZE	L
TOP	L	REPLACE	S
XTOP	LS		

Diese Operationen besitzen folgende Semantik:

push(e) → OK/NOK fügt das Element e an das Ende von QStack (bzw. aus Kellersicht auf den Keller) an. Bei Überschreitung der Kapazität (overflow) wird NOK zurückgegeben, ansonsten wird das Element e eingefügt und OK gemeldet. push ist ein Beispiel für eine zustandsabhängige Operation. Es findet auf jeden Fall eine Leseoperation statt, nämlich um zu überprüfen, ob QStack schon voll ist. Bei Overflow entspricht push einer (Nur)-Leseoperation; wenn hinzugefügt werden kann, ist push eine Lese-Schreib-Operation.

deq() → e/NOK löscht das Element am Anfang der Schlange (bzw. das sich am weitesten unten im Keller befindliche Element). Sollte QStack leer sein, wird NOK zurückgegeben, ansonsten wird das gelöschte Element geliefert.

pop() → e/NOK wie deq, nur wird die Operation am Ende der Schlange (bzw. auf dem obersten Element des Kellers ausgeführt)

top() → e/NOK wie pop, das Element wird aber nicht gelöscht.

size() n size liefert die Anzahl der Elemente in QStack

replace(e1,e2) → OK ersetzt alle Elemente e1 durch e2. Es wird immer OK zurückgegeben.

xtop() → OK/NOK die beiden Elemente am Ende der Schlange werden ausgetauscht. xtop gibt OK zurück, wenn mindestens zwei Elemente in QStack vorhanden sind.

6.2.2 Abhängigkeitsbeziehungen

Mit der Leser-Schreiber-Notation lassen sich jetzt folgende Abhängigkeiten gleichzeitig operierender Transaktionen beschreiben. D steht hier für eine Abhängigkeitsbeziehung, L für Lesezugriffe, S für Schreibzugriffe und LS für Lese-Schreibzugriffe.

$\mathbf{D}_1$: T_i L $\longrightarrow_O$ T_j L
$\mathbf{D}_2$: T_i L $\longrightarrow_O$ T_j S
$\mathbf{D}_3$: T_i S $\longrightarrow_O$ T_j L
$\mathbf{D}_4$: T_i S $\longrightarrow_O$ T_j S

Gelesen werden diese Abhängigeiten auf folgende Weise: Die Beziehung D1 beschreibt die Situation, daß die Transaktion T_i einen Lesezugriff auf dem Objekt O ausführt, der von einem Lesezugriff der Transaktion T_j auf demselben Objekt gefolgt wird. Man erkennt, daß durch absolutes Verbot von gleichzeitigen Operationen auf einem Objekt die erreichbare Parallelität unnötig eingeschränkt wird. Der gleichzeitige Zugriff zweier Leseoperationen auf ein Objekt entspricht allen geforderten Bedingungen. Ein Beispiel auf dem definierten QStack ist die gleichzeitige Ausführung von Top und Size. Es ist nicht möglich, anhand der Auswirkungen der Operationen auf den Zustand des Objektes eine Aussage über die Reihenfolge der Operationen zu treffen, da der Zustand durch die Leseoperationen nicht beeinflußt wird. Die Operationen sind also trotz der Abhängigkeitsbeziehung D_1 kommutativ. D_1 wird daher als nicht signifikant bezeichnet. Wenn man Transaktionen vor Ausführung eine der L-S-LS-Charakteristiken zuordnen kann, ist es in bestimmten Fällen also möglich, auch den parallelen Zugriff auf das Objekt zu gestatten. Daß man allerdings die o.g. Beziehungen sehr genau beachten muß, zeigt das folgende Beispiel, bei dem identische Operationen zweier verzahnter Transaktionen in unterschiedlicher Reihenfolge auf QStack arbeiten, und verschiedene Ergebnisse bewirken:

$T_1 : Top(QStack1) \longrightarrow e_1$ $\qquad$ $T_2 : Push(QStack1, e_2)$
$T_2 : Push(QStack1, e_2)$ $\qquad$ $T_1 : Top(QStack1) \longrightarrow e_1'$
$T_1 : Push(QStack1, e_3)$ $\qquad$ $T_1 : Push(QStack1, e_3)$

Offensichtlich erhält man unterschiedliche Rückgabewerte durch die Top-Operation in den beiden Abläufen. Die Operationen und damit die Transaktionen sind nicht kommutativ. Die Transaktionen bilden folgende Zyklen: erste Variante $T_1 \longrightarrow_{D2} T_2 \longrightarrow_{D4} T_1$ bzw. bei der zweiten Variante $T_2 \longrightarrow_{D3} T_1$. Als nicht signifikant gelten bisher aber nur Zyklen, die D_1 enthalten. Schlußfolgerung ist, daß die Abläufe mit D_2, D_3 und D_4 nicht zulässig sind.

6.2.3 Einteilung der Transaktions-Abhängigkeiten

Das Problem der kaskadierenden Abbrüche wurde bereits erwähnt. Der Abbruch einer nicht bestätigten Transaktion zieht den Abbruch weiterer Transaktionen nach. Ob solch eine Kaskade wirklich entstehen kann, hängt von den Beziehungen der beteiligten Transaktion ab. Unter Bezug auf die Untersuchungen des letzten Abschnitts kann es bei D_1-Beziehungen (L-L) nicht zu kaskadierenden Abbrüchen kommen. Das entscheidende Kriterium ist ein Informationsfluß von der ersten zur nachfolgenden Transaktion. Damit muß immer dann gerechnet werden, wenn die erste Transaktion etwas schreibt, was im Zugriff der zweiten Transaktion liegt und von dieser gelesen werden kann. Folgende problematische Konstellationen ergeben sich:

$$S \longrightarrow L \qquad S \longrightarrow LS \qquad LS \longrightarrow L \qquad LS \longrightarrow LS$$

Man erkennt, daß von der ersten Transaktion immer geschrieben werden kann, während die nachfolgende Transaktion diese Änderungen zu sehen bekommt. Wenn die erste Transaktion vor dem Bestätigen abbrechen muß, besteht die Gefahr, daß die nächste Transaktion die Veränderungen schon in irgendeiner Weise genutzt hat (immer vorausgesetzt, die Transaktionen arbeiten parallel auf dem gleichen Objekt. Diese Veränderungen sind nach dem Abbruch aber ungültig und müssen rückgängig gemacht werden, so daß auch die zweite Transaktion gezwungen ist, abzubrechen. Transaktionen, die in einer solchen Beziehung zueinander stehen, nennt man *abbruchabhängig* (abort-dependend).
Eine schwächere Beziehung besteht zwischen den Transaktionen folgenden Typs:

$$L \longrightarrow S \qquad L \longrightarrow LS \qquad LS \longrightarrow S \qquad S \longrightarrow S$$

Man sieht hier, daß kien Informationsflußvon der ersten zur zweiten Transaktion erfolgt. Die Ergebnisse der ersten Transaktion beeinflussen die zweite Transaktion nicht. Allerdings ergibt sich eine neue Forderung: die erste Transaktion muß vor der zweiten Transaktion bestätigt werden. Es ist also kein Überholen möglich. Die erste Transaktion kann möglicherweise die zweite warten lassen. Der Grund für diese neue Forderung ist, daß sonst Veränderungen der zweiten Transaktion schon vor Bestätigung für die erste Transaktion sichtbar würden, mit der Möglichkeit eines Informationsflusses von der zweiten zur ersten Transaktion und den erwähnten Gefahren bei kaskadierenden Abbrüchen. Transaktionen mit diesen Beziehungen nennt man *bestätigungsabhängig* (commit-dependend), d.h. vor Bestätigung der zweiten Transaktion muß die erste Transaktion beendet sein. Die Abbruchabhängigkeit schließt die Bestätigungsabhängigkeit ein, ist also strenger.

6.3 Erhöhung der Parallelität durch Strukturanalyse von Abstrakten Datentypen

In den letzten Abschnitten wurde die Parallelitätserhöhung allein durch die Untersuchung von Eigenschaften der Transaktionen erzielt. Hier soll jetzt erstmals die Struktur des Abstrakten Datentyps eine Rolle spielen. Genutzt wird wieder der Typ QStack, allerdings nur in seiner Eigenschaft als Warteschlange. Betrachtet werden hier nur die Operationen ENQ(e), entspricht Push(e), und DEQ(). Um kaskadierende Abbrüche zu vermeiden und um Serialisierbarkeit auf der Schlange garantieren zu können, müssen die typischen Bedingungen einer Warteschlange berücksichtigt werden. Für die folgenden Betrachtungen sind zwei Einschränkungen interessant:

Wenn eine Transaktion mehrere Elemente in eine Schlange einfügt, müssen diese Elemente zusammen und in der gleichen Reihenfolge am Ende der Schlange erscheinen.
Alle Einträge, die durch eine Transaktion in eine Schlange eingefügt werden, dürfen für andere Transaktionen erst sichtbar werden, wenn die Transaktion erfolgreich beendet wurde.

Wenn sich Transaktionen unkontrolliert überlappen, werden diese Forderungen schon durchbrochen. **Beispiel:** Unterschiedliche Transaktionen führen Einfügeoperationen (ENQ(e)) auf der Schlange aus. Es ist unwahrscheinlich, daß die Einträge der einzelnen Transaktionen jeweils als Block in der Warteschlange erscheinen.
Bei der Untersuchung der Abhängigkeiten zwischen den Operationen auf der Schlange ist es zum ersten Mal notwendig, individuelle Elemente in der Schlange zu unterscheiden. Es wird angenommen, daß jedes Element in der Schlange zur Unterscheidung einen eindeutigen Identifikator besitzt. Es können folgende Beziehungen auftreten:

$$
\begin{array}{lllll}
\mathbf{D}_1: & \mathrm{T}_i\ \mathrm{ENQ}(\sigma) & \longrightarrow_O & \mathrm{T}_j\ \mathrm{ENQ}(\sigma') \\
\mathbf{D}_2: & \mathrm{T}_i\ \mathrm{ENQ}(\sigma) & \longrightarrow_O & \mathrm{T}_j\ \mathrm{DEQ}(\sigma'\ ) \\
\mathbf{D}_3: & \mathrm{T}_i\ \mathrm{ENQ}(\sigma) & \longrightarrow_O & \mathrm{T}_j\ \mathrm{DEQ}(\sigma) \\
\mathbf{D}_4: & \mathrm{T}_i\ \mathrm{DEQ}(\sigma) & \longrightarrow_O & \mathrm{T}_j\ \mathrm{ENQ}(\sigma') \\
\mathbf{D}_5: & \mathrm{T}_i\ \mathrm{DEQ}(\sigma) & \longrightarrow_O & \mathrm{T}_j\ \mathrm{DEQ}(\sigma')
\end{array}
$$

Sowohl ENQ(e) als auch DEQ(e) entsprechen Lese-Schreib-Operationen. Deshalb jede Parallelität auf dem gemeinsamen Objekt zu verbieten, wäre verfrüht. Durch Berücksichtigung der Elemente ist es möglich, die Sperrgranularität der Synchronisation um eine Stufe heraufzusetzen. Sperren beziehen sich jetzt auf Elemente. Die Warteschlange ist auch bei schreibenden Zugriffen parallel bearbeitbar. Einige der genannten Abhängigkeiten werden sich wieder als nicht signifikant herausstellen. Bei Betrachtung der Abhängigkeit D_2, diese Beziehung entsteht, wenn eine Transaktion ein bestimmtes Element σ' aus der Warteschlange löscht, nachdem eine andere Transaktion vorher ein anderes Element σ in die Schlange eingefügt hat, sieht man, daß beide Operationen völlig unabhängig voneinander sind. Es kann nicht zum Informationsfluß kommen. Weder eine der beiden Transaktionen, noch eine dritte kann entscheiden, welche Transaktion als erste ausgeführt wurde; die beiden Transaktionen sind also kommutativ. Da die Beziehung D_4 die umgekehrte Beziehung zu D_2 darstellt, sind beide Beziehungen nicht signifikant und Abläufe, in denen Zyklen mit diesen Beziehungen auftreten, sind zulässig.

6.3.1 Semantische Änderungen zur Parallelitätserhöhung

Der letzte Abschnitt zeigte, daß die semantische Struktur eines Objekts ein bedeutendes Potential zur Parallelitätserhöhung bietet. Bei der formalen Objektbeschreibung wird das eine große Rolle spielen. Hier soll jedoch noch ein weiterer Aspekt zur Sprache gebracht werden, der die Parallelität unnötig beschränkt. Transaktionen unterliegen bestimmten typbezogenen Konsistenzzwängen. Diese Zwänge haben in vielen Fällen ihre Berechtigung. Oft kommt es jedoch vor, daß aufgrund einer Verallgemeinerung diese Zwänge auch auf Objekte angewandt werden, bei denen das gar nicht notwendig ist. Es ist möglich,

einige dieser Einschränkungen zu lockern oder sogar aufzuheben, allerdings nur unter der Voraussetzung, daß das System bzw. die Implementierung des Abstrakten Datentyps hier Gestaltungsmöglichkeiten eröffnen. Einige Ansätze sollen anhand eines geringfügig modifizierten QStack demonstriert werden. QStack wird wieder als Warteschlange genutzt, allerdings wird die strenge First-In-First-Out Forderung zum Teil aufgegeben.
Warteschlangen werden sehr oft als Puffer zwischen beliebigen Erzeugern und Konsumenten eingesetzt. Die genaue Reihenfolge der Auftragsabarbeitung spielt in den meisten Fällen keine große Rolle. Es wird nur erwartet, daß Einträge nicht auf zu lange Zeit in der Schlange bleiben, und daß Einträge ungefähr gleich schnell in der Schlange vorwärts kommen, d.h. daß eine faire Abarbeitung gewährleistet ist.
Der modifizierte QStack in seiner Funktion als Warteschlange bildet ebenfalls die zuletzt erwähnten Abhängigkeitsbeziehungen. Allerdings mit einer sehr positiven Veränderung: bei der Überschneidung von Transaktionen sind jetzt auch Zyklen mit den Beziehungen D_1 und D_5 erlaubt. Nur D_3 bleibt auch weiterhin verboten. Durch die Freigabe von D_1 wird das parallele Einfügen von Elementen möglich, der Verzicht auf D_5 gestattet das nebenläufige Löschen von Elementen. Um einerseits die Parallelität so weit wie möglich zu erhöhen, anderseits aber um die Gefahr von kaskadierenden Abbrüchen und andere Anomalien zu vermeiden, wird die Semantik der beiden Operatoren geringfügig der neuen Situation angepaßt. Der Operator ENQ wird für diese Anwendung umbenannt in M-ENQ (modifiziert) und DEQ in M-DEQ. Wenn die Transaktion, die den jetzt am weitesten vorn in der Schlange stehenden Eintrag einfügte, noch nicht bestätigt ist, kann dieses Element nicht ohne Konsistenzverletzung gelöscht werden. M-DEQ sucht also in der Schlange nach dem nächsten Eintrag, dessen Transaktion schon erfolgreich abgeschlossen wurde. Sollte kein bestätigter Eintrag mehr in der Warteschlange zu finden sein, sucht M-DEQ nach Einträgen, die von der Transaktion eingefügt wurden, die auch M-DEQ aufgerufen hatte. Ist die Suche auch hier erfolglos, wird M-DEQ blockiert, bis ein passender Eintrag zur Verfügung steht. Diese Änderung der Semantik stört die korrekte Abarbeitung der Einträge solange nicht, wie die folgenden Voraussetzungen gegeben sind: Kein Eintrag bleibt ewig in der Schlange, wenn

(1) die Transaktion, die den Eintrag hinzufügt, nach endlicher Zeit bestätigt wird
(2) die Transaktion, die den Eintrag löscht, nach endlicher Zeit terminiert
(3) nur eine begrenzte Zahl von Transaktionen versucht, den Eintrag zu löschen, und dann abbricht

Die Unterschiede zur strengen FIFO-Strategie werden am nächsten Beispiel deutlich. Vorher sind noch einige Erklärungen zur verwendeten Notation zu machen: Die Warteschlange wird durch eine Folge von Buchstaben repräsentiert, wobei links der Eintrag steht, der als nächstes gelöscht wird und rechts die Einfügeoperationen ablaufen. Die Buchstaben stellen folgende Einträge dar:

kleine kursive Buchstaben (a) stehen für Einträge, die eingefügt wurden, deren Transaktion aber noch nicht bestätigt ist
große Fettbuchstaben (A) stehen für Einträge, die eingefügt wurden und deren Transaktion erfolgreich abgeschlossen ist
GROSSE KURSIVE BUCHSTABEN (A) stehen für Einträge, auf denen schon

eine M-DEQ-Operation ausgeführt wurde, die Transaktion aber noch nicht bestätigt ist.

Hochgestellte Zahlen dokumentieren die Nummer der Transaktion, die die entsprechende Operation aufrief, die aber noch nicht beendet ist.
Beispiel: Ausgegangen wird von einer leeren Schlange. Mittels M-ENQ(a) und M-ENQ(b) werden nacheinander die Einträge a und b in die Schlange eingefügt. Das führt zu folgendem Zustand:

$[\ a^1\ ,\ b^2\ ]$

Jetzt wird ausgenutzt, daß Zyklen die Beziehung D_1 enthalten dürfen. Die Transaktion T_1 fügt also noch einen Eintrag c in die Warteschlange ein. Danach werden beide Transaktionen erfolgreich abgeschlossen. Diese beiden Schritte ergeben nacheinander folgenden Zustand:

$[\ a^1\ ,\ b^2\ ,\ c^1\ ]$

$[\ \mathbf{A}\ ,\ \mathbf{B}\ ,\ \mathbf{C}\ ]$

An dieser Stelle wird deutlich, daß die Transaktionen nicht mehr serialisierbar sind. Jetzt führt zuerst eine Transaktion T_3 ein M-DEQ aus, gefolgt von einer Transaktion T_4, die zwei M-DEQ-Operationen aufruft. Man erhält:

$[\ A^3\ ,\ B^4\ ,\ C^4\ ]$

Wenn T_3 an dieser Stelle gezwungen wird, abzubrechen, und T4 kann aber seine Operationen erfolgreich beenden, bekommt man folgenden Endzustand:

$[\ \mathbf{A}\ ]$

Das interessante an diesem Beispiel ist, daß A und C ihre Reihenfolge getauscht haben, obwohl sie von derselben Transaktion eingefügt wurden. Die Konsequenz aus der Ermöglichung von nichtserieller Arbeit auf dem Abstrakten Datentyp bewirkt, daß der Effekt einer aufgerufenen und später abgebrochenen Transaktion nicht notwendig derselbe sein muß, als wenn die Transaktion gar nicht aufgerufen worden wäre. Nachdem man gesehen hat, daß die Berücksichtigung der Lokalität der Transaktion, also die Konzentration auf die Wirkungsorte der Operationen, sehr wichtig ist, wird im nächsten Kapitel ein formales Modell eingeführt, das es gestattet, später das mögliche Parallelisierungspotential von Transaktionen methodisch zu ermitteln.

6.4 Der Objektansatz für Abstrakte Datentypen

Der Ansatz, die Sperrgranularität des Typs zu erhöhen, wird jetzt erweitert. Zusätzlich zur Struktur des Typs und der Operatoren, wird jetzt auch der Inhalt der Objektkomponenten analysiert. Das hat den Vorteil, daß bestimmte Ordnungseigenschaften zwischen den Elementen ausgenutzt werden können, eventuell ergeben sich neue Ansatzpunkte für zulässige Nebenläufigkeit.

6.4.1 Objektmodell und Objektgraph

Zum besseren Verständnis wird an dieser Stelle für Abstrakte Datentypen ein Objektmodell eingeführt. Ein Objekt kann man sich vorstellen als zusammengesetzt aus Unterobjekten und Komponenten, die in bestimmten Ordnungsbeziehungen zueinander stehen. Zeichnerisch wird ein solches Objekt durch einen Objektgraph dargestellt, auf welchem sich sehr vorteilhaft Lokalitätseffekte darstellen und ableiten lassen. Objekte können sowohl wieder aus anderen Objekten zusammengesetzt (komplexe Objekte), als auch einfache Komponenten (Datenwerte) sein. Ein Objekt besteht aus drei Mengen:

1. Menge S der Objekte bzw. Datenwerte - die Elemente dieser Menge entsprechen den Knoten im Objektgraph

2. Menge R der Ordnungsregeln - zwischen den Elementen des Abstrakten Datentypen bestehen bestimmte Odnungsbeziehungen, die in dieser Menge zusammengefaßt werden

3. Menge O vereinigt die Operationen, die zur Manipulation des Abstrakten Datentyps zur Verfügung stehen

Der Objektgraph stellt den logischen Aufbau des Abstrakten Datentyps dar. Die Knoten des Graphen, dessen Struktur je nach Datentyp sehr unterschiedlich sein kann, entsprechenentweder wieder Teilkomponenten, auf die man rekursiv die Definition für den Graphen anwenden kann oder stehen für Datenwerte. Zwischen den Komponenten können Ordnungsbeziehungen ausgeprägt sein. Diese Beziehungen entsprechen gerichteten Pfeilen auf der gleichen Ebene. Nur zwischen Objekten, die sich auf gleichem Niveau befinden, sind Ordnungsregeln definiert. Ein Pfeil von Objekt A zu Objekt B sagt aus, daß über den Zugriff auf A das Objekt B erreicht werden kann. Pfeile zwischen Objekten, die sich auf unterschiedlichen Ebenen befinden, repräsentieren "zusammengesetzt-aus" Beziehungen. Deshalb gilt auch die Einschränkung, daß diese Pfeile nur von einem oberen Niveau zu einem niedrigeren verlaufen dürfen. Zur Unterscheidung von Ordnungsregeln und Kompositionvorschriften stellt man sich den Gesamtgraphen zusammengesetzt aus zwei Untergraphen vor:

> dem Kompositionsgraphen G'_{Ob}, der aus allen Knoten k und den verbindenden "zusammengesetzt-aus" Kanten besteht (der Kompositionsgraph muß laut Definition ein Baum sein, und bildet eine Hierarchie zwischen den Objektkomponenten)
>
> dem Ordnungsgraphen G''_{Ob}, zu dem wieder alle Knoten k und zusätzlich alle Ordnungskanten gehören. Die Ordnungskanten zwischen den Knoten *eines Niveaus* können Zyklen bilden.

Der Wert eines Objektes *ob* kann rekursiv beschrieben werden: Wenn der Knoten k_i ein einfaches, nicht zusammengesetztes Element beschreibt, dann ergibt sich der Wert dieses Objektes aus dem Datenwert des Knotens. Wenn das Objekt wieder aus anderen Objekten zusammengesetzt ist, ergibt sich der Gesamtwert des Objekts aus den "zusammengesetzt-aus"-Kanten (Kompositionskanten) und den Werten der Knoten des Unterbaumes, dessen Wurzel der Objektknoten ist.

6.4.2 Lokalität von Operationen anhand des Objektgraphen

Informal wurde bereits in einem vorangegangenen Abschnitt die Lokalität von Operationen untersucht und auch ausgenutzt. Unter Zuhilfenahme des Objektgraphen lassen sich die vorstellbaren Auswirkungen von Operationen wie folgt beschreiben:

1. Änderungen des Knoteninhaltes; bei zusammengesetzten Objekten erreicht man das durch Aufrufen von Operationen, die das gesamte Objekt betreffen,

2. Hinzufügen und Entfernen von Knoten aus dem Graphen mit dem Effekt, daß auch zugehörige Kanten gelöscht werden

3. Änderung der Struktur durch Veränderungen an den Ordnungsanten des Graphen

4. Lesen des Inhaltes von Knoten

5. Lesen der Struktur des Graphen und Test auf das Vorhandensein von Knoten

Ein anschauliches Beispiel für die letzte Art von Zugriff liefert die Size-Operation auf dem Abstrakten Datentyp QStack. Size liefert die Anzahl der Elemente, die sich in QStack befinden. Die Anzahl der Elemente entspricht der Anzahl der Knoten im Objektgraphen, es muß also die Struktur gelesen und das Vorhandensein von Knoten überprüft werden.
Nachdem jetzt schon mehrere Male der Begriff der Lokalität verwendet wurde, ohne genau definiert zu sein, wird das jetzt anhand des Objektgraphen nachgeholt. Es bietet sich an, den Ort der Auswirkung einer Operation durch die betroffenen Knoten des Objektes zu beschreiben. Die Lokalität wird an dieser Stelle nochmal unterteilt, ähnlich der Unterteilung der Transaktionen in Leser und Schreiber. Man führt den Begriff der Strukturlokalität mit der Bezeichnung L_O^S und die Bezeichnung inhaltliche oder Inhalts-Lokalität L_O^I, jeweils bezogen auf das Objekt O, ein.

- Die Struktur-Lokalität L_O^S einer Operation *op* entspricht im Objektgraphen den Knoten, die gelöscht oder neu eingefügt, deren Ordnungskanten (vom oder zum Knoten) geändert oder gelesen, oder deren Vorhandensein getestet wurde.

- Die Inhalts-Lokalität L_O^I einer Operation *op* ergibt sich aus der Untermenge der Knoten des Objektgraphen, die eingefügt oder gelöscht wurden, oder deren Inhalt gelesen oder geändert wurde.

Die beiden Mengen können sich nach dieser Definition überschneiden.
Mit Hilfe dieser Begriffe sollen die früher benutzten Klassifikationen Leser - Schreiber verfeinert werden. Offensichtlich kann man vier verschiedene Arten von Auswirkungen unterscheiden (die natürlich oft kombiniert auftreten werden).

- Struktur-Leser mit der entsprechenden Lokalität L_O^{SL}

- Struktur-Schreiber (nicht nur schreiben, sondern im weiteren Sinne Verändern, also auch löschen) mit der Lokalität L_O^{SS}

- Inhalt-Leser mit der Lokalität L_O^{IL}

- Inhalt-Schreiber mit L_O^{IS}

Einige der Operatoren auf QStack sollen hinsichtlich ihrer Zugehörigkeit zu diesen Klassen untersucht werden.
Replace tauscht nur den Inhalt von Elementen, also hier Knoten. Egal welcher Zustand (leer, voll, Tauschelemente kommen nicht vor usw.) vorliegt, an der Struktur des QStack kann sich nichts ändern. Replace gehört also zu den Inhalt-Schreibern.
XTop ist etwas schwieriger zu beurteilen. Es wird zwar äußerlich der Inhalt der obersten Elemente geändert, aber insgesamt gesehen bleibt der Inhalt von QStack unverändert. Allerdings wird der Zeiger auf daß erste Element umgesetzt, nämlich auf den Nachfolger. Das ist gleichbedeutend mit einer Umordnung der Ordnungskanten. Man hat es hier mit einer Strukturveränderung zu tun, also mit einem Strukturschreiber.
Um die Charakteristik einer Operation herauszufinden, die die meiste Parallelität ermöglicht, müssen alle Klassenzugehörigkeiten berücksichtigt werden.
DEQ ändert sowohl den Inhalt von QStack als auch seine Struktur, da ja ein ganzer Knoten samt Kanten wegfällt. DEQ gehört also zu den Strukturschreibern und Inhalt-Schreibern.
Die Lokalität einer Operation wird durch das Aufzählen der betroffenen Knoten angegeben. Man unterscheidet nun, ob nur eine begrenzte Anzahl der Datenwerte (es wird also Bezug auf die nichtkomplexen Objekte des Strukturbaumes genommen) von der Operation berührt werden, oder ausnahmslos alle. Wenn alle einfachen Komponenten in die Ausführung der Operation einbezogen sind, spricht man von einer **globalen Operation**, ansonsten trivial nichtglobal. Anhand dieses Atrributes werden die definierten Klassen weiter unterteilt in globale und nichtglobale Strukturleser, globale und nichtglobale Strukturveränderer usw. Gerade dieses Kriterium ist zur Entscheidung, um Parallelität möglich ist oder nicht, von herausragender Bedeutung. So gehört Size mit Bestimmtheit zu den globalen Operationen, während Top eine nichtglobale Operation ist.

6.4.3 Zusammenhänge zwischen Lokalität und Abhängigkeitsbeziehungen

Die Definition der verschiedenen Lokalitätstypen solch jetzt helfen, möglichst im voraus schon Aussagen über erreichbare Parallelität machen und nutzen zu können. Der naheliegendste Fall, daß sich die speziellen Lokalitäten der beteiligten Operationen überscneiden, läßt schon auf Konfliktpotential schließen. Aber hier offenbart sich ein Problem: mit Hilfe der Lokalitäten lassen sich bestimmt Schlußfolgerungen auf Konflikte ziehen, aber die Lokalitäten der Operationen sind ja nicht immer im voraus bekannt. Oft ergeben sie sich auch erst während der Abarbeitung der Operation. Anhaltspunkte zur Bestimmung der L. ergeben sich oft auch aus den Eingangsparametern der Operationen und den Ordnungsregeln des Abstrakten Datentyps.
Ein wichtiges Hilfsmittel werden in den weiteren Untersuchungen die sogenannten Referenzen sein. Diese Referenzen sind eine Teilmenge der Kompositionskanten, die einen bestimmten Bezug zu den Ordnungsregeln des Objektes haben. Diese Referenzen beginnen immer an der Wurzel eines Kompositionsbaumes und werden als Teil des Objektzustandes aktuell gehalten. Man vermutet schon, daß es sich hier um Zeiger auf bestimmte ausgezeichnete Elemente des Objekts handelt. Als Referenzen werden im Fall des QStack die Zeiger auf das erste Element (Frontzeiger einer Schlange) und das letzte Element (Stackpointer bei Keller) geführt. Diese Referenzen sind nicht direkt manipulierbar, die unterliegen aber im Laufe von Operationen bestimmten Änderungen. Es ist eine Defi-

nitionsfrage, ob diese Referenzen auch gelöscht werden können. Hier wird angenommen, daß es einem Löschen beider Referenzen entspricht, wenn durch eine Pop- oder DEQ-Operation das letzte verbliebene Element in einem QStack gelöscht wird.
Bei beiden Referenzen handelt es sich um implizite referenzen. Der Unterschied zu expliziten Referenzen besteht darin, daß die iimpliziten Referenzen vom Objekt selber zur Verfügung gestellt und verwaltet werden; die expliziten Referenzen jedoch von der aufgerufenen Operation zur Verfügung gestellt werden, meist als Parameter.

6.4.4 Konfliktuntersuchung bei parallelen Operationen

Das bisher abgeleitete Wissen über die gegenseitige Behinderung von Operationen spiegelt die folgende Behauptung wieder:
Transaktionen, die zwei Operationen x und y auf dem Objekt ob aufgerufen haben, bilden keine gegenseitig behindernden Abhängigkeiten heraus, solange die Schnittmengen folgender Lokalitäten leer sind:

$$L_x^{IS} \cap L_y^{IL} \qquad L_x^{IL} \cap L_y^{IS} \qquad L_x^{IS} \cap L_y^{IS}$$

$$L_x^{SL} \cap L_y^{SS} \qquad L_x^{SS} \cap L_y^{SL} \qquad L_x^{SS} \cap L_y^{SS}$$

Als erste Schlußfolgerung erhält man, daß Operationen, die nur auf die Struktur Einfluß haben, nicht von Operationen behindert werden, die auf den Inhalt fixiert sind. Jetzt schließt sich der Kreis zu den schon bekannten Abbruch- und Bestätigungsabhängigkeiten. Operationen, bei denen Schnittmengen von anderen Kombinationen der erwähnten Lokalitäten nicht leer sind, können unter Umständen solche Abhängigkeiten bilden.

Es wird versucht, eine Methodik zu entwickeln, die nach der Entdeckung solcher Abhängigkeiten entscheidet, für welche Transaktionen eine parallele Aktivität zuzulassen ist. Zur Darstellung dieser Beziehungen zwischen verschiedenen Transaktionsarten wird eine Tableauform verwendet. Die in der horizontalen oberen Leiste zu findenden Bezeichnungen stehen für die momentan laufenden Transaktionen, während in der ersten vertikalen Leiste die Transaktionen aufgelistet sind, die sich um eine parallele Abarbeitung mit den schon aktiven Transaktionen bemühen. In den zugehörigen Schnittpunktfeldern kann die resultierende Abhängigkeit abgelesen werden. Es stehen

- **ND (NO DEPENDENCY) oder leeres Feld** für keine Abhängigkeit
- **CD (COMMIT DEPENDENCY)** für eine Bestätigungsabhängigkeit
- **AD (ABORT DEPENDENCY)** für Abbruchabhängigkeit

Hier werden die englischen Abkürzungen verwendet, da diese Bezeichnungen sehr gebräuchlich in diesem Zusammenhang sind. Als Ausgangspunkt, ohne Berücksichtigung eines bestimmten Datentyps, stellt die folgende Tabelle die Abhängigkeiten bezogen auf die Lokalität (getrennt nach Struktur und Inhalt) dar.

L	L_x^{sl}	L_x^{il}	L_x^{ss}	L_x^{is}
L_x^{SL}			AD	
L_x^{IL}				AD
L_x^{SS}	CD		CD	
L_x^{IS}		CD		CD

Man kann der Tabelle einfach entnehmen, daß die Abbruchabhängigkeit AD tatsächlich nur bei der Konstellation: aktive Transaktion schreibt und neue Transaktion will auf dem Objekt jeweils auch Struktur oder Inhalt lesen, auftritt.

6.4.5 Kompatibilität von Transaktionen auf Abstrakten Datentypen

Nachdem die erste Kompatibilitätstabelle von einer konkreten Objektausprägung abstrahiert hat, und nur auf der Transaktions-Lokalität aufgebaut hat, soll nun versucht werden, durch Betrachtung eines konkreten Abstrakten Datentyps die Verträglichkeiten der Transaktionen weiter zu verfeinern. Schrittweise wird immer mehr semantisches Wissen genutzt, um die Restriktionen lockern zu können und mehreren Transaktionen die gleichzeitige Aktivität auf einem Objekt zu ermöglichen . Der Ausgangspunkt ist ein Objekt, über dessen Struktur nichts bekannt ist, die Operationen können nicht weiter spezifiziert werden und müssen als Leser-Schreiber (ungünstigste Variante) eingeordnet werden. Herkömmliche Synchronisationsmechanismen, wie zum Beispiel das Zwei-Phasen-Sperrprotokoll, gehen von dieser Situation aus. Die nächste Tabelle modelliert dieses Szenarium:

	X(LS)
Y(LS)	AD

Der nächste Schritt war die Einteilung in Leser, Schreiber und Leser-Schreiber. Die Abhängigkeiten entsprechen den D_i-Beziehungen. Felder, in denen kein Eintrag steht, stehen für zulässige Abhängigkeiten, d.h. die Transaktionen behindern sich nicht. CD- und AD-Einträge stehen dagegen für Abhängigkeiten, die in Zyklen und Abläufen nicht erlaubt sind.

(X,Y)	L	S	LS
L		AD	AD
S	CD	CD	CD
L	CD	AD	AD

Nur die Leser-Leser-Abhängigkeit ist problemlos. Bei allen anderen Feldern müssen weitere Betrachtungen angestellt werden.
Die Fortsetzung mag zunächst wie ein Rückschritt aussehen, da im Grunde genommen auch keine Verfeinerung vorgenommen wird. Die Leser-Schreiber-Felder werden nicht weiter mitgeführt, da ihr Inhalt auf einfache Weise aus dem Inhalt der anderen Felder berechnet werden kann. Man nutzt die Ordnung ($AD > CD > ND$) auf den Abhängigkeiten aus, und betrachtet die Leser-Schreiber-Beziehungen als eine Funktion, die stets die restriktivere Abhängigkeit der zugehörigen Lese- oder Schreiberabhängigkeit zurückgibt.

Ein Beispiel soll das verdeutlichen: Gesucht ist die Abhängikeit zwischen einer laufenden Schreiber-Transaktion und einer Leser-Schreiber-Transaktion, die ebenfalls auf dieses Objekt zugreifen möchte. Man vergleicht also die Einträge von Schreiber $\longrightarrow$ Leser und Schreiber $\longrightarrow$ Schreiber. Laut obiger Tabelle erhält man $AD > CD$, die Berechnung muß AD ergeben, das Ergebnis stimmt mit dem (LS,S)-Eintrag in der Tabelle überein. Die reduzierte Tabelle hat also folgendes Aussehen:

(X,Y)	L	S
L		AD
S	CD	CD

Zur Erweiterung des Tableaus wird nun die Unterscheidung der Lokalität nach Struktur und Inhalt herangezogen. Für jedes der drei problematischen Felder in der letzten Tabelle wird nun ein separates Tableau entwickelt. Für das Feld (L,S), also Leseoperation soll Schreiboperation folgen, muß man Struktur-Lesen (SL), Inhalt-Lesen (IL) und Struktur-Inhalt-Lesen (SIL) mit Struktur-Schreiben (SS), Inhalt-Schreiben (IS) und Struktur-Inhalt-Schreiben (SIS) verknüpfen. Die anderen Kombinationen werden genauso gebildet. Unter Bezugnahme auf die zulässigen Abhängigkeiten, erhält man folgende drei Tableaus:

LESER - SCHREIBER

(L,S)	SS	IS	SIS
SL	AD		AD
IL		AD	AD
SIL	AD	AD	AD

SCHREIBER - SCHREIBER

(S,S)	SS	IS	SIS
SS	CD		CD
IS		CD	CD
SIS	CD	CD	CD

SCHREIBER - LESER

(S,L)	SL	IL	SIL
SS	CD		CD
IS		CD	CD
SIS	CD	CD	CD

Ein Test mit den auf QStack definierten Operationen soll die Nutzbarkeit dieser Tabellen zeigen. Die Operation Size ist ein Struktur-Leser und soll gerade aktiv sein. Kann parallel dazu eine DEQ-Transaktion zugelassen werden? Anwendung kann hier nur die dritte Tabelle finden. DEQ ändert sowohl die Struktur als auch den Inhalt des Objektes, ist also ein Struktur-Inhalt-Schreiber. In dem entsprechenden Feld kann man eine

Bestätigungsabhängigkeit (CD) ablesen, die Transaktion können also nicht problemlos parallelisiert werden.

Bisher hat man nur zustandsunabhängige Charakteristiken betrachtet. Die Untersuchung des Einflusses des momentanen Zustandes sowie die Nutzung von Wissen über die Eingabe und Ausgabe von Operationen, erfordert zwar sehr genaues Wissen über die Operatoren, bietet aber andererseits auch große Spielräume.

6.4.6 Einbeziehung von Zuständen und Ein-/Ausgabesemantik

In den Tabellen wurde der Zustand eines Objektes und die Auswirkungen auf die aktiven Transaktionen bisher nicht beachtet. Das kann behoben werden, indem jeder AD/CD-Eintrag durch eine zustandsabhängige Menge von Beziehungen ersetzt wird. Bei der Definition der Felder können zusätzlich auch die Lokalität der Operation und zusätzliche Referenzen des Abstrakten Datentyps als Bedingungen ins Spiel gebracht werden. Wie ein solcher erweiterter Eintrag aussieht, wird im nächsten Abschnitt an parallelen Push und DEQ-Operationen gezeigt.
Allerdings merkt man sehr deutlich, wie extrem typbezogen dieses Verfahren ist. Man benötigt für jeden Objekttyp andere Vorgaben und ist gezwungen, jeweils typbezogen die Auswirkungen der Operationen und die gegenseitigen Wechselwirkungen zu analysieren. Hier soll jedoch wieder anhand des Beispiels QStack die Vorgehensweise demonstriert werden.

Es wird der fall betrachtet, daß auf QStack gerade versucht wird, ein Element mit Push hinzuzufügen. Daraufhin wird eine andere Operation aufgerufen, die mit DEQ das letzte Element löschen soll. Der QStack ist zu diesem Zeitpunkt voll, push kann also nicht einfügen und gibt NOK zurück. Offensichtlich wird von push unter diesen Randbedingungen nur gelesen. Da DEQ aber das letzte Element löschen kann, hat man es mit einer Schreiber-Leser-Beziehung zu tun, die laut Tabelle einer CD-Abhängigkeit unterliegt. Der (unvollständige) Eintrag der Form (Abhängigkeitsform / Bedingung) sieht also wie folgt aus:

$$(CD , push_out = NOK)$$

Hier spielt also die Ausgabesemantik eine Rolle. Die Referenzen ergeben ebenfalls eine Bedingung. Wenn der Frontzeiger f und der Zeiger b auf das letzte Element definiert, aber nicht identisch sind, müssen mindestens zwei Elemente vorhanden sein. Die Lokalitäten von push und deq überschneiden sich nicht. Das bringt den Erfolg, daß in diesem Fall keine Abhängigkeit zwischen den Operationen besteht. Formal erhält man also:

$$(ND , Zeigerf \neq Zeigerb)$$

Auch im ersten Fall, bei einem overflow des QStack gilt $f \neq b$, so daß der zweite Fall den ersten mit einschließt. Man kann jetzt die schwächere der beiden Bedingungen wählen, also ND und den ersten Fall weglassen.

6.5 Algorithmus zur Bestimmung der Abhängigkeiten zwischen Transaktionen

Nach und nach wurden Schritte zur Ermittlung der Beziehungen zwischen Operationen durchgeführt. In diesem Abschnitt solch jetzt versucht werden, diese Schritte zu einem Algorithmus zusammenzufassen, der alle wichtigen Informationen für eine Parallelisierung liefert.

6.5.1 Herleitung des Algorithmus

Man unterscheidet fünf verschiedene Stufen bei der Untersuchung der Abhängigkeiten:

1. **Herleitung des Objektgraphen**. Mit Hilfe des Objektgraphen werden die Referenzen bestimmt. Jede der möglichen Operationen auf dem Abstrakten Datentypen wird untersucht hinsichtlich ihrer Auswirkungen auf den Objektgraphen.
2. **Klasseneinteilung**. Alle Operationen werden eingeteilt in Leser, Schreiber und Leser-Schreiber. Als weiteres Unterscheidungskriterium dient, ob Struktur, Inhalt oder beide betroffen sind. Welche Eingabe- und Ausgabeparameter gibt es? Handelt es sich um eine globale Operation und welche impliziten oder expliziten Referenzen treten auf?
3. **Kompatibilitätstabelle**. Aus den angegebenen Abhängigkeitstabellen wird eine Gesamttabelle gebildet, in der jeweils die Verträglichkeiten von allen Operationen auf dem Abstrakten Datentypen aufgeführt werden. Hier müssen auch die Leser-Schreiber expandiert werden.
4. **Ausnutzung der Ein- und Ausgabesemantik** zur Verfeinerung der Tabelleneinträge. Eine Verfeinerung ist natürlich nur sinnvoll, wenn wenigstens ein Paar (Abhängigkeitsform / Bedingung) weniger restriktiv als die Gesamtbeziehung ist.
5. **Lokalität und Referenzen**. Alle nichtglobalen Operationen werden untersucht, der Eintrag in der Tabelle wird ergänzt oder aufgeschlüsselt. Die Tabelle gibt jetzt ein ziemlich präzises Bild, welche Operationen unter welchen Bedingungen zusammenarbeiten können.

Das Ergebnis dieser fünf Schritte ist eine Übersicht in Tabellenform, die für jede Kombination aus den auf dem Abstrakten Datentyp definierten Operationen Einträge enthält, unter welchen Bedingungen eine parallele Abarbeitung des Operationspaares möglich ist. Diese Tabelle kann möglicherweise als Input für einen erweiterten Synchronisationsmechanismus dienen, der für unterschiedliche Objekttypen spezielle Parallelitätsprotokolle bereithält.

6.5.2 Anwendung des Algorithmus

Als abschließendes Beispiel soll noch einmal der Abstrakte Datentyp QStack dienen. Zur besseren Anschaulichkeit werden nicht alle möglichen Operationen betrachtet; die XTOP-Operation und die REPLACE-Operation bringen keine weiteren Erkenntnisse, lassen sich

aber äquivalent ableiten. Dadurch erreicht man schon einmal eine Reduzierung der Kompatibilitätstabelle von ursprünglich $7 * 7 = 49$ auf $5 * 5 = 25$ Einträge. Jetzt werden Schritt für Schritt die Punkte des angegebenen Algorithmus abgearbeitet:

1. Objektgraph
   Der Graph zur Darstellung des QStack besteht aus zwei Ebenen. Wurzel und auch einziges Element der obersten Ebene ist das Objekt selber. Von der Wurzel gehen die beiden Referenzzeiger f und b ab, die auf das erste und das letzte Element zeigen. Die Wurzel ist mit jedem Eintrag des Kellers bzw. der Schlange durch eine Kompositionskante verbunden. Auf der zweiten, also der unteren Ebene befinden sich die Einträge der Schlange, auf denen mit Hilfe der Ordnungskanten eine Reihenfolge definiert ist. Die Ordnungskante sind in Richtung des ersten Elements gerichtet, auf das die Referenz f zeigt.

2. Einteilung der Operationen
   Für die Operationen auf QStack wird eine Tabelle ermittelt, die alle notwendigen Informationen hinsichtlich der Auswirkungen usw. enthält. L steht für Leser, S für Schreiber , I für Inhalt und S für Struktur.

Name	L S LS	I S IS	Ausgabe	Lokalität	Referenz
Pop	LS	IS	Ergebnis / NOK	lokal	Zeiger f
Push	LS	IS	OK / NOK	lokal	Zeiger f
Deq	LS	IS	Ergebnis / NOK	lokal	Zeiger b
Size	L	S	Ergebnis	global	–
Top	L	IS	Ergebnis / NOK	lokal	Zeiger f

3. Kompatibilitätstabelle Für jedes Paar von Operationen werden die Tabellen der Abhängigkeiten für Leser, Schreiber, Struktur und Inhalt genutzt. Man erhält mehrere Abhängigkeiten, von denen die restriktivste in die Tabelle einzutragen ist.
   Die unvollständige QStack-Tabelle sieht wie folgt aus:

(OP_1, OP_2)	Push	Pop	Deq	Top	Size
Push	AD	AD	AD	CD	CD
Pop	AD	AD	AD	CD	CD
Deq	AD	AD	AD	CD	CD
Top	AD	AD	AD		
Size	AD	AD	AD		

   Es fällt auf, daß bei den meisten Paaren eine Abbruchabhängigkeit besteht. Das ist leicht verständlich, wenn man sich vor Augen führt, daß es sich bei den meisten Operationen um Leser-Schreiber mit globaler Lokalität handelt. Für die Parallelisierung ist eine recht pessimistische Ausgangsposition. Mit den folgenden Schritten wird sich Bild aber wandeln, viele Abbruchabhängigkeiten lassen sich aufbrechen, wenn man die Referenzen und die Ein-Ausgabesemantik nutzt.

4. Verfeinerung durch Nutzung der Ein-Ausgabesemantik
   Anhand des schon eingeführten Beispielpaares (Deq,Push) soll die Verfeinerung für einen Tabelleneintrag hergeleitet werden. Laut Eigenschaftstabelle haben beide Operationen Ausgabewerte. Aufgrund der Reihenfolge ist jedoch nur die Ausgabe von Push nutzbar. Push kann zwei Ausgabewerte, nämlich OK und NOK haben. Aus diesen beiden Werten lassen sich zwei (Abhängigkeit, Bedingung)-Paare bilden.

	Push
Deq	$(CD,\ Push_{out} = NOK)$ $(AD,\ Push_{out} = OK)$

   Die Erklärung für diese Abschwächung liegt in der Eigenschaft von Push, als Leser zu zählen, wenn aufgrund eines overflows kein Einfügen möglich ist. Eine noch stärkere Unterteilung erhält man bei Betrachtung von konkurrierenden Push-Operationen. Durch jeweils zwei mögliche Ausgabewerte ergeben sich vier Kombinationen mit unterschiedlichen Abhängigkeiten. In der Tabelle sind sie nach unterschiedlicher Restriktionsstufe geordnet.

	Push^x
$Push^y$	$(ND,\ Push^x_{out} = Push^y_{out} = NOK)$ $(CD,\ Push^x_{out} = Push^y_{out} = OK)$ $(CD,\ Push^x_{out} = NOK\ and\ Push^y_{out} = OK\ )$ $(AD,\ Push^x_{out} = OK\ and\ Push^y_{out} = NOK\ )$

   Bis jetzt wurde nur die Ausgabesemantik eingebracht. Genauso ist es jedoch möglich, die Eingabesemantik zu nutzen. Ein typischer Fall sind zwei Push-Operationen, die das gleiche Element versuchen einzutragen. Dabei spielt es keine Rolle, welche der push-Operationen zuerst einfügt. Es liegt also Kommutativität vor, keine Abhängigkeiten zwischen den Operationen.

	Push^x
$Push^y$	$(ND,\ Push^x_{out} = Push^y_{out} = NOK)$ $(CD,\ Push^x_{out} = Push^y_{out} = OK)$ $(CD,\ Push^x_{out} = NOK\ and\ Push^y_{out} = OK\ )$ $(AD,\ Push^x_{out} = OK\ and\ Push^y_{out} = NOK\ )$ $(ND,\ Push^x_{in} = Push^y_{in} = e)$

   Damit ist der vierte Schritt abgeschlossen.

5. Referenzuntersuchung und Lokalität
   Das Paar (Deq, Push) wird weiter untersucht. Beide Operationen sind nicht global,

unter der Voraussetzung, daß mindestens zwei Einträge in der Schlange sind.Das ist also der Fall bei der Beziehung $(AD,\ Push_out\ =\ OK)$. Man kann also die AD-Beziehung auf den Fall reduzieren, daß die Referenzen f und b identisch oder leer (dann sind die Lokalitäten nicht disjunkt) sind. Für den Fall, daß f und b auf verschiedene Elemente zeigen, ist die Schnittmenge der Lokalitäten leer und es besteht keine Abhängigkeit. Die erste Tabelle unter Punkt 4 wird konkretisiert zu:

	Push
Deq	$(CD,\ Push_{out}\ =\ NOK)$ $(AD,\ f\ =\ b\ )$ $(ND,\ f\ \neq\ b\ )$

Die Betrachtung der Einträge wird also immer spezieller. Zum Schluss wurde nur noch ein kleiner Ausschnitt aus der Menge der Operationen betrachtet. Die Ableitung der anderen Abhängigkeiten verläuft nach dem gleichen Muster.

6.5.3 Zusammenfassung und Ausblick

Es ist wohl einsichtig, daß mit diesem Verfahren in vielen Fällen die Parallelität erhöht werden kann. Wenn man sich allerdings vor Augen führt, welchen Aufwand die Ableitung nur eines kleinen Teils der Abhängigkeiten auf einem sehr einfachen Datentyp bereitet, stellt sich Frage nach der Notwendigkeit und dem Nutzen der Betrachtungen. Es kommt auf die Abschätzung des Verwaltungsaufwandes im Vergleich zur Häufigkeit des Auftretens von bestimmten Behinderungen an. Gerade im hier vorgestellten Fall handelt es sich jedoch um eine Konstruktion, die in Betriebssystemen eine sehr wichtige Rolle spielt. Wenn man außerdem noch in Betracht zieht, daß Transaktionen sehr umfangreich und zeitaufwendig sein können, dann bedeutet ein Warten auf die Beendigung einer Transaktion vor einer Sperre signifikante Einschränkungen und Zeitverlust, die den Verwaltungsaufwand rechtfertigen, um Sperren so weit wie möglich zu verhindern.

Kapitel 7

Ein Hybrider Sperr-Algorithmus (*Klaus–Dieter Spang*)

7.1 Motivation

Bei der Behandlung von fehlerbedingten Ausfällen (*failures*) und Nebenläufigkeiten (*concurrency*) in Datenbank–Systemen ist der Begriff der *Atomizität* von entscheidender Bedeutung. Verschiedene Algorithmen, die das Problem der Nebenläufigkeit und der Wiederanlaufbarkeit (*recovery*) behandeln, wurden mit der Zeit entwickelt.

Die ersten dieser algorithmische Entwicklungen verzichteten dabei auf die Einbeziehung von typgebundenen Objekten. Doch insbesondere die Einbeziehung typspezifischer Eigenschaften der Objekte in den Ablauf eines Algorithmus' selbst gaben Anlaß, bei der Erstellung von Algorithmen zur Sicherung von Nebenläufigkeit und Wiederanlaufbarkeit ein besonderes Augenmerk auf typisierte Objekte zu legen.

Die meisten dieser bisher entstandenen Algorithmen folgen dabei einem Sperr-Verfahren, in dem Konflikte im Kontext einer speziellen Kommutativitätseigenschaft behandelt werden: Sperren von Operationen, die kommutativ bezüglich der Konkatenation ihrer zeitlichen Ausführung sind, können nicht in Konflikt stehen.

Diese einfache Bedingung spiegelt sich in konventionellen Zwei-Phasen–Sperr-Verfahren (*two-phase locking*) folgendermaßen wider:

Erzeugen zwei Transaktionen beim Zusammentreffen an einem Objekt einen Sperr-Konflikt (*lock conflict*), so muß die eine Transaktion gezwungenermaßen auf die andere warten, um ihre Operation durchführen zu können; dadurch wird eine Verzögerung impliziert, so daß sich eine Serialisierung der beiden beteiligten Transaktionen ergibt.

Zwei-Phasen–Sperren bewirken somit eine Serialisierung der Transaktionen, was zu ihrer partiellen Ordnung führt: Transaktionen, die in keiner Beziehung zur transitiven Hülle eines entstandenen Sperr-Konfliktes stehen, können in beliebiger Reihenfolge in die Serialisierungsordnung eingebracht werden, sie können sogar auf verschiedene Datenobjekte in unterschiedlicher Ordnung serialisiert werden.

Sind die Operationen nebenläufiger Transaktionen kommutativ in obigem Sinne, dann sind alle lokalen Ordnungen äquivalent und vereinbar mit einer globalen, totalen Serialisierungsordnung.

Der Sperr–Algorithmus, der hier vorgestellt werden soll, dient der Nebenläufigkeitskon-

trolle (*concurrency control*) und Wiederanlaufbarkeit bei Verwendung typisierter Daten–Objekte. Motiviert wird dieses Vorgehen dabei durch die Überlegung, die vergleichsweise strengen Bedingungen, die andere bisher bekannte Algorithmen an die Sperrkonflikte stellen, durch eine abgeschwächte Form von Einschränkungen zu ersetzen, so daß daraus eine größere Anzahl möglicher zulässiger überschneidungen resultiert.

Die Idee läßt sich wie folgt formulieren: Transaktionen sind serialisierbar in der Reihenfolge, in der sie das commit ausführen. Nun kann in einem ersten Schritt als ein Bestandteil des commit-Protokolls einer Transaktion eine Zeitmarke (*timestamp*) von einer logischen Uhr erzeugt werden; danach wird eben diese Zeitmarke den Objekten, die durch die Transaktion beeinflußt wurden, "aufgedrückt".

Die durch die Sperr-Konflikte implizierte partielle Ordnung der Transaktionen wird somit erweitert auf die explizit gegebene totale Ordnung, die aus der Ordnung der Zeitmarken der Transaktionen für die einzelnen Objekte folgt. Dadurch, daß die Serialisierungsordnung explizit gemacht wurde, läßt sich die oben genannte Kommutativitätsbedingung durch eine schwächere Bedingung, die *Abhängigkeit* (*dependency*), ersetzen.

7.2 Das zugrundeliegende Modell

Um den Algorithmus formulieren zu können, muß zuvor ein geeignetes Modell definiert werden. Das hier verwendete Modell besitzt zwei Grundelemente — Transaktionen und Objekte. Jedes Objekt besitzt Operationen, die von den Transaktionen aufgerufen werden können, um den Objektzustand abzufragen oder zu verändern. Diese Operationen sind zugleich die einzige Möglichkeit, wie die Transaktionen auf die Objekte zugreifen können. Die Notation sieht T_i für die Transaktionen und O_i für die Objekte vor.

Das Modell basiert auf vier verschiedenen Elementarereignissen, die jeweils auf Ereignisse an der Schnittstelle von Transaktion und Objekt abheben. Im einzelnen sind dies die Ereignisse

- Aufruf (*invocation*) $\langle inv, O, T\rangle$
  Er tritt ein, wenn eine Transaktion T eine Operation des Objekts O aufruft. Das inv–Feld beinhaltet sowohl den Namen des Ereignisses als auch die Argumente.

- Antwort (*response*) $\langle res, O, T\rangle$
  Sie tritt ein, wenn ein Objekt eine Antwort auf einen vorher abgesetzten Aufruf einer Operation des Objekts O durch Transaktion T zurückliefert.

- Abschluß (commit) $\langle commit(t), O, T\rangle$
  Er tritt ein, wenn ein Objekt O registriert, daß die Transaktion T ein commit zur Zeitmarke t durchgeführt hat. Die Zeitmarken müssen Elemente einer abzählbaren, total geordneten Menge sein (und werden von einer logischen Uhr geliefert).

- Abbruch (abort) $\langle abort, O, T\rangle$
  Er tritt ein, wenn ein Objekt O registriert, daß die Transaktion T abgebrochen wurde.

commit– und abort–Ereignisse werden in ihrer Gesamtheit auch als *Beendigungsereignisse* (*completion events*) bezeichnet. Außerdem spricht man davon, daß ein Ereignis e bei der Formulierung $\langle e, O, T\rangle$ Objekt O und Transaktion T *involviert*.

Wir betrachten nun einige Festlegungen im Zusammenhang mit Ereignissequenzen. Ist H eine Sequenz von Ereignissen und O eine Objektmenge, so wird H|O —"H eingeschränkt auf O" — definiert als die Teilsequenz von H, die nur Ereignisse enthält, die Objekte aus der Menge O involvieren. Eine entsprechende Definition ergibt sich, wenn man anstelle einer Objektmenge eine Transaktionenmenge T betrachtet.

Weiterhin bezeichnet committed(H) die Menge der Transaktionen, für die ein commit–Ereignis innerhalb von H auftritt. Wiederum läßt sich aborted(H) analog definieren. Schließlich bezeichnet completed(H) die Menge der Transaktionen, die in H ein commit oder abort durchgeführt haben.

Bei der Betachtung aller möglichen Ereignisfolgen läßt sich feststellen, daß manche von ihnen keinen sinnvollen Ablauf ergeben. Es ist deshalb unerläßlich, einige zusätzliche Bedingungen an die Ausführung der einzelnen Transaktionen (*well-formedness constraints*, Bed. 1–3) sowie an die Zeitmarken (Bed. 4–6) zu stellen.

1. Eine Transaktion T muß auf die Antwort einer laufenden Anfrage warten, bevor sie die nächste Operation aufruft. Entsprechend darf ein Objekt O eine Antwort für eine Transaktion T nur dann zurückliefern, wenn T noch einen laufenden Aufruf hat.
   Wenn wir H|T für eine Ereignissequenz H und eine Transaktion T betrachten, so bedeutet dies, daß die Teilsequenz aller inv– und res–Ereignisse aus einer alternierenden Folge derselben besteht, wobei das erste Element der Folge ein inv–Ereignis sein muß. Außerdem müssen ein Aufruf und die unmittelbar darauf folgende Antwort dasselbe Objekt involvieren.

2. Eine Transaktion T kann in H ein commit oder abort durchführen, aber nicht beides.

3. Eine Transaktion T kann kein commit durchführen, während sie auf Antwort zu einem noch laufenden Aufruf wartet. Außerdem ist es nicht möglich, daß eine Transaktion nach ihrem commit noch irgendwelche Operationen ausführt.

Durch obige Bedingungen wird zum einen Nebenläufigkeit innerhalb einer Transaktion untersagt, dadurch daß eine Transaktion zu einem Zeitpunkt höchstens einen laufenden Aufruf haben kann. Zum anderen darf eine Transaktion nicht auf einem Teil der involvierten Objekte ein commit, auf anderen ein abort durchführen. Dieses letztere Ergebnis bezeichnet man auch als *atomaren Abschluß* (*atomic commitment*).

Sind die Bedingungen 1–3 erfüllt, so spricht man bei einer sich daraus ergebenden Ereignissequenz von einer Historie.

4. Zwei commit–Ereignisse in H, die von derselben Transaktion ausgehen, haben dieselbe Zeitmarke.

5. Zwei commit–Ereignisse in H, die von verschiedenen Transaktionen ausgehen, haben verschiedene Zeitmarken.

6. Wenn eine Transaktion T_2 eine Operation auf Objekt O ausführt, nachdem eine andere Transaktion T_1 auf O ein commit durchgeführt hat, so muß die Zeitmarke, die T_2 generiert und vergibt, später sein als die von T_1.

Bedingung 4 stellt die Eindeutigkeit der Zeitmarken für die Transaktionen sicher, Bedingung 5 besagt, daß eine Transaktion höchstens eine Zeitmarke vergibt.

Die letzte Bedingung schließlich ist notwendig, um dem Algorithmus die Möglichkeit zu gestatten, Antworten auf Aufrufe 'online' zurückzuliefern. Hybrid atomare Algorithmen wie der vorliegende garantieren, daß die Transaktionen, die ein commit durchgeführt haben, in der Reihenfolge ihrer Zeitmarken serialisierbar sind. Da die Zeitmarken aber generiert werden, wenn die Transaktionen ein commit durchführen, ist es einem Objekt bei der Rückgabe einer Antwort an die aufrufende Transaktion nicht bekannt, welche Zeitmarke von der Transaktion vergeben wird. Um Serialisierbarkeit zu gewährleisten, muß diese letzte Bedingung deshalb gefordert werden.

Bedingung 6 läßt sich noch genauer formulieren. Dazu sind allerdings zuvor noch einige Definitionen zu machen. Ist H eine Ereignissequenz, die ein oder mehrere Objekte involviert, so definieren wir precedes(H) als Relation auf Transaktionen: $(T_1,T_2) \in$ precedes(H) genau dann, wenn es eine Operation gibt, die von T_2 aufgerufen wurde und eine Antwort zurückliefert, nachdem T_1 den commit durchgeführt hat. precedes(H) repräsentiert folglich eine Art potentiellen Informationsfluß zwischen Transaktionen. Wenn nämlich $(T_1,T_2) \in$ precedes(H), dann bedeutet dies, daß es eine von Transaktion T_2 ausgeführte Operation in der Folge H gibt, nachdem T_1 den commit durchgeführt hat. Folglich kann T_2 auf eine von T_1 freigegebene Sperre getroffen sein, was bedeutet, daß T_2 nach T_1 serialisierbar sein muß.

Mit Hilfe von precedes läßt sich weiterhin TS(H) als partielle Ordnung auf Transaktionen definieren: $(T_1,T_2) \in$ TS(H), wenn $T_1,T_2 \in$ commited(H) und wenn für die von T_1 und T_2 vergebenen Zeitmarken $t_{T_1} < t_{T_2}$ gilt. Für die Vergabe der Zeitmarken gilt folgende weitere Bedingung: Die Reihenfolge der Zeitmarken, die beim commit einer Transaktion übergeben wird, muß für jedes Objekt mit dessen durch precedes gegebenen Ordnung konsistent sein.
Informell ausgedrückt heißt dies: Wenn T_2 auf O zugreifen will und T_1 auf O bereits abgeschlossen hat, so muß T_2 eine Zeitmarke vergeben, die größer ist als die von T_1 vergebene, formal läßt sich der Sachverhalt $\forall O$ precedes(H|O) $\subseteq$ TS(H) schreiben.
Möglichkeiten, diese letztgenannte Bedingung zu erfüllen, bieten beispielsweise Algorithmen, die auf Zeitmarken von logischen Uhren basieren oder solche, die Informationen über die vergebene Zeitmarke den Nachrichten des commit–Protokolls mitgeben.

7.3 Atomizität

Der Begriff Atomizität ist definiert im Zusammenhang und mit Hilfe von Objektspezifikationen in dem Sinne, daß Transaktionen dann atomar sind, wenn ihre Ausführung serialisierbar ist und Wiederanlaufbarkeit gewährleistet ist, wenn nur die Objektspezifikationen gegeben sind.

7.3.1 Objekt–Spezifikationen

Serielle Spezifikation

Jedes Objekt besitzt eine serielle Spezifikation, die sein Verhalten beschreibt, wenn keine fehlerbedingten Ausfälle und Nebenläufigkeiten auftreten. Diese Spezifikation besteht aus einer Menge von Operationenfolgen, wobei Operation hier ein Paar, bestehend aus

Aufruf und zugehöriger Antwort, meint. Desweiteren identifiziert eine Operation immer auch das Objekt, auf das sie angewendet wird.

Verhaltensspezifikation

Jedes Objekt besitzt auch eine Verhaltensspezifikation, die sein Verhalten beim Auftreten von Fehlern und Nebenläufigkeiten widerspiegelt. Diese Spezifikation ist eine Menge von Historien, die nur Ereignisse enthalten, die dieses Objekt involvieren.
Eine Implementierung eines Objekts ist korrekt, wenn sie nur Historien zuläßt, die mit der Verhaltensspezifikation des Objekts vereinbar sind. Wenn weiter die Implementierung eines jeden Objekts in einem System korrekt ist, dann ist, falls H eine Historie des Systems darstellt, H|O (für jedes O) mit der Verhaltensspezifikation von O vereinbar.

7.3.2 Globale Atomizität

Die Historie eines Systems ist atomar, wenn die Transaktionen, die ein commit durchgeführt haben, in einer seriellen Ordnung und mit dem identischen Endeffekt ausgeführt werden können. Da die typspezifischen Eigenschaften eingebracht werden sollen, muß Serialisierbarkeit und Atomizität definiert werden im Kontext der seriellen Spezifikation von Objekten.

Eine serielle Spezifikation ist eine Menge von Operationenfolgen, nicht aber eine Menge von Historien. Folglich muß ein Bezug zwischen den Begriffen Historie und Operationenfolge hergestellt werden. Eine Historie ist *seriell*, wenn Ereignisse verschiedener Transaktionen sich nicht zeitlich überlappen und sie ist *fehlerfrei*, wenn keine Transaktion mit abort abbricht, formal also, wenn aborted(H)= $\emptyset$.

Ist nun H eine serielle, fehlerfreie Historie, so ist OpSeq(H), die mit der Historie H korrespondierende Operationenfolge, wie folgt definiert:
Für eine einzelne Transaktion T_i ist OpSeq(H|T_i) die Operationenfolge, die man aus H|T_i erhält, indem man Paare der Aufrufe mit den zugehörigen Antworten bildet und commit–Ereignisse und laufende Aufrufe vernachlässigt.
Betrachtet man die gesamte Historie H, so ist OpSeq(H) definiert als Konkatenation OpSeq(H|T_1) • OpSeq(H|T_2) •$\cdots$• OpSeq(H|T_n).

Liegt als Objekt O beispielsweise eine FIFO–Schlange vor mit den beiden Operationen enqueue, die ein Element am Schwanz der Schlange einfügt und dequeue, die das Kopfelement zurückgibt und gleichzeitig aus der Schlange löscht, so ergibt sich aus der seriellen, fehlerfreien Historie $H_{Schlange}$

$$\begin{gathered}\langle enqueue(x), O, T_1\rangle\\ \langle OK, O, T_1\rangle\\ \langle commit(t_1), O, T_1\rangle\\ \langle dequeue(), O, T_2\rangle\\ \langle x, O, T_2\rangle\\ \langle commit(t_2), O, T_2\rangle\end{gathered}$$

die Operationenfolge OpSeq($H_{Schlange}$)

$$\begin{gathered}O : [enqueue(x), OK]\\ O : [dequeue(), x]\end{gathered}$$

Hierin sind lediglich noch die beiden Aufruf/Antwort–Paare der obigen Historie enthalten, also als Aufruf zunächst das Einfügen des Wertes 'x' in die Schlange (mit Antwort 'OK') sowie als Aufruf das Auslesen/Löschen des ersten Wertes mit dem Wert 'x' als zurückgelieferter Antwort.

Eine fehlerfreie Historie H ist *akzeptabel* bezüglich Objekt O, wenn OpSeq(H|O) in der seriellen Spezifikation von O enthalten ist, wenn also die Operationenfolge in H, die O involviert, von der seriellen Spezifikation zugelassen ist.
Zwei Historien H_1 und H_2 sind *äquivalent*, wenn jede Transaktion T die gleiche Folge von Operationsschritten in beiden Historien ausführt, d.h. wenn für jede Transaktion T gilt: $H_1|T=H_2|T$.
Weiter definiert man, wenn H eine Historie und T_{TO} eine Totalordnung der Transaktionen ist, serial(H,T) als die serielle Historie, die äquivalent ist zu H und in der die Transaktionen in der Totalordnung T_{TO} abgearbeitet werden.
Eine fehlerfreie Historie H bezeichnet man als *serialisierbar in der Ordnung* T_{TO}, wenn serial(H,T) akzeptabel ist. Informell ist H also in der Ordnung T_{TO} serialisierbar, wenn es unter Berücksichtigung der seriellen Spezifikation der Objekte zulässig ist, daß die in H involvierten und in der Ordnung T_{TO} ausgeführten Transaktionen die gleichen Operationsschritte machen wie in der ursprünglichen Historie H.
Eine fehlerfreie Historie H ist *serialisierbar*, wenn sich eine Totalordnung T_{TO} finden läßt, in der H serialisierbar ist.

Definiert man schließlich permanent(H) = H|commited(H), so läßt sich als Atomizitätsbedingung letztendlich angeben: H ist atomar, wenn permanent(H) serialisierbar ist. Wiederanlaufbarkeit ergibt sich also, indem man die Ereignisse einer Historie, die nicht abgeschlossen haben, ignoriert und die Serialisierbarkeit der abgeschlossenen Transaktionen fordert. Das Ignorieren nichtabgeschlossener Transaktionen ist schlüssig, da diese keine Wirkung nach außen haben dürfen.

Als Beispiel betrachten wir eine Historie, die als Objekt O nochmals die oben eingeführte FIFO–Schlange involviert:

$$\begin{gathered}
\langle enqueue(x), O, T_1\rangle \\
\langle OK, O, T_1\rangle \\
\langle enqueue(y), O, T_2\rangle \\
\langle OK, O, T_2\rangle \\
\langle enqueue(z), O, T_1\rangle \\
\langle OK, O, T_1\rangle \\
\langle commit(2), O, T_1\rangle \\
\langle commit(1), O, T_2\rangle \\
\langle dequeue(), O, T_3\rangle \\
\langle y, O, T_3\rangle \\
\langle dequeue(), O, T_3\rangle \\
\langle x, O, T_3\rangle \\
\langle commit(5), O, T_3\rangle
\end{gathered}$$

Die Historie enthält nur abgeschlossene Transaktionen und sie ist serialisierbar in der Totalordnung T_2–T_1–T_3. Folglich ist diese Historie atomar. Es ist an diesem Beispiel zu erkennen, daß die Frage der Serialisierbarkeit bzw. Atomizität einer Historie — im Gegensatz zu den meisten anderen Algorithmen, die die Nebenläufigkeitskontrolle behandeln —

hier unabhängig ist von der Reihenfolge, in der die Operationen der verschiedenen Transaktionen in der Historie auftreten. Entscheidend sind hingegen die Zeitmarken, die als Argument den commit-Ereignissen der einzelnen Transaktionen mitgegeben werden.

7.3.3 Lokale Atomizität

Während die oben gegebene Definition von Atomizität global war, also die Historie eines Gesamtsystems zum Gegenstand der Untersuchung machte, ist es für den Aufbau eines modularen, flexiblen Systems notwendig, lokale Eigenschaften der Objekte zu untersuchen, die die gewünschte globale Eigenschaft der Atomizität für das System garantieren können. Eine Eigenschaft, die dies leistet, bezeichnet man als *lokale Atomizitätseigenschaft* (*local atomicity property*).

Eine lokale Atomizitätseigenschaft E ist eine durch Objektspezifikationen formulierte Eigenschaft, so daß gilt: Erfüllt die Spezifikation jedes einzelnen Objekts des zu betrachtenden Gesamtsystems die Eigenschaft E, dann ist jede Historie des Systemverhaltens atomar.
Beim Entwurf einer solchen lokalen Atomizitätsbedingung muß also sichergestellt werden, daß es mindestens eine Serialisierungsreihenfolge für die abgeschlossenen Transaktionen gibt, die mit den Objektspezifikationen vereinbar ist.

Hier soll eine spezielle lokale Atomizitätseigenschaft, die sog. *hybride Atomizität*, definiert werden. Diese Eigenschaft benützt die Zeitmarken, um die lokale Serialisierungsreihenfolge eines jeden Objekts zu erreichen. Die Idee, die dahintersteckt, ist die, daß jedes Objekt sicherstellt, daß die abgeschlossenen Transaktionen in der Ordnung ihrer Zeitmarken serialisierbar sind — oder formaler: Eine Historie H ist *hybrid atomar*, wenn permanent(H) in der durch die Zeitmarken auf committed(H) definierten totalen Ordnung TS(H) serialisierbar ist. Ein Objekt ist hybrid atomar, wenn jede Historie in der Verhaltensspezifikation hybrid atomar ist.

Hybride Atomizität ist eine lokale Atomizitätsbedingung, was durch das folgende Theorem festgehalten wird: Wenn jedes Objekt eines Systems hybrid atomar ist, so ist auch jede Historie im Systemverhalten atomar.

7.3.4 Online-hybride Atomizität

Der im folgenden behandelte Algorithmus erlaubt einer aktiven Transaktion, ein commit durchzuführen, wenn sie zu diesem Zeitpunkt keine Operation tätigt. Diese Form der sog. pessimistischen Synchronisation läßt sich durch die *Online-hybride Atomizität* darstellen. Sei H wieder eine Historie und C eine Transaktionenmenge, so bezeichnet man C als commit–Menge für H, wenn committed(H) $\subseteq$ C und C $\cap$ aborted(H) $= \emptyset$. In C sind also all die Transaktionen enthalten, die bereits ein commit durchgeführt haben und die, die potentiell abschließende Transaktionen sind, weil sie noch kein abort durchgeführt haben. Damit läßt sich die Menge known(H) = precedes(H) $\cup$ TS(H) definieren. Somit erfaßt known(H|O), was dem Objekt O über die Zeitmarkenordnung aller — sowohl der abgeschlossenen als auch der noch aktiven — Transaktionen bekannt ist. Jedes einzelne Objekt muß folglich so beschaffen ein, daß die aktiven Transaktionen Zeitmarken vergeben

können in einer beliebigen Ordnung, die konsistent ist mit dem lokalen Informationsgehalt der Objekte.

Eine Historie ist *online-hybrid atomar* für ein Objekt O, wenn für jede commit–Menge C für die Historie H und für jede Totalordnung T, die mit known(H|O) vereinbar ist, H|C|O in eben dieser Ordnung T serialisierbar ist. H ist insgesamt online-hybrid atomar, wenn H online-hybrid atomar für alle O ist. Es läßt sich schließlich feststellen, daß H auch hybrid atomar ist, wenn H online-hybrid atomar ist.

Zurück zum Beispiel aus Abschnitt 7.3.2. Die Historie des Objekts FIFO– Schlange ist hybrid atomar, denn jedes Präfix ist online-hybrid atomar. Betrachtet man nämlich ein Präfix, in dem weder T_1 noch T_2 abgeschlossen haben, so ist known(H) leer, die Historie jedoch serialisierbar in beliebiger Ordnung, also T_1– T_2 oder T_2–T_1. Zu einem späteren Zeitpunkt, an dem T_1 und T_2 abgeschlossen haben, steht das Transaktionenpaar (T_2,T_1) in known(H). Führt schließlich auch die letzte Transaktion T_3 eine Operation aus, so enthält known(H) desweiteren auch die Paare (T_2,T_3) und (T_1,T_3). Dies wiederum ergibt eine Totalordnung T_2–T_1–T_3 auf den Transaktionen. Damit ein Präfix von H, das eine Operation von R umfaßt, online-hybrid atomar ist, muß es serialisierbar sein in der oben genannten Reihenfolge T_2–T_1–T_3. Dies wurde bereits in Abschnitt 7.3.2 gezeigt.

7.4 Konflikt und Nebenläufigkeit

Um den Algorithmus mit geeigneten Mitteln zur Lösung von Konflikten auszustatten, sind zunächst einmal Kriterien aufzustellen, wann ein Sperr-Konflikt vorliegt.

7.4.1 Eine erste, informelle Beschreibung des Algorithmus'

Wie auch andere typische Sperr-Algorithmen verwendet der Algorithmus folgendes Prinzip: Eine Operation prüft, ob sie in ihrer Ausführung fortfahren kann in Abhängigkeit davon, ob andere aktive Transaktionen Operationen ausführen, die mit den eigenen in Konflikt stehen. Entscheidend ist aber, daß die Formulierung eines Konflikts hier mit weniger Einschränkungen behaftet ist als in bisherigen Ansätzen. Zusätzlich wird hier genau festgelegt, wie mit den commits und aborts der Transaktionen umgegangen wird.

Drei Komponenten sind vom Algorithmus für jedes Objekt vorgesehen:

- Jede Transaktion besitzt eine *Intentionsliste* (*intention list*), die die Operationenfolge[1] enthält, die auf das entsprechende Objekt angewendet werden, wenn die Transaktion abschließt.

- Der *Abschluß–Zustand* (*committed state*) eines Objekts spiegelt die Wirkung der Transaktionen wider, deren commit dem entsprechenden Objekt bekannt ist. Man den Abschluß–Zustand beispielsweise als Intentionsliste der abgeschlossenen Transaktionen betrachten.

- Eine Menge von *Sperren* verbindet jede Operation mit der Menge der aktiven Transaktionen, die diese Operationen ausgeführt haben. Die Sperren stehen über eine symmetrische Konflikt–Relation, die *Abhängigkeitsrelation*, miteinander in Beziehung.

[1] 'Operation' meint hier das bereits früher definierte Ereignis-Paar ⟨inv, res⟩.

Der Ablauf des Algorithmus' läßt sich verbal folgendermaßen beschreiben:

1. Ruft eine Transaktion eine Operation auf, so erstellt sie zunächst eine *Sicht* (view), indem sie ihre Intentionsliste an den Abschluß–Zustand anhängt.

2. Danach wählt sie eine Antwort, die konsistent ist mit der Sicht.

3. Bevor die Transaktion die neue auszuführende Operation an die Intentionsliste anhängt, fordert sie eine Sperre für die Operation an.

4. Wird von einer anderen Transaktion eine damit in Konflikt stehende Sperre gehalten, wird die Sperranforderung zurückgewiesen, die Antwort verworfen und der Aufruf der Operation zu einem späteren Zeitpunkt erneut versucht. Dabei kann die zurückgelieferte Antwort einen anderen Wert haben als beim vormaligen Versuch.

5. Wird der Transaktion hingegen die Sperre für die gewünschte Operation erteilt, so wird die Operation der Intentionsliste der Transaktion hinzugefügt und eine Antwort zurückgeliefert. Für den Fall, daß die verwendete Konflikt–Relation die zurückzuliefernde Antwort nicht mit in Betracht zieht, kann die Sperre auch vor der Auswahl der Antwort angefordert werden.

6. Wenn eine Transaktion ein commit durchführt, wird ihre Intentionsliste mit dem Abschluß–Zustand zusammengefaßt und nach den Zeitmarken geordnet.

7. Führt eine Transaktion ein commit oder abort durch, so werden ihre Sperren aufgehoben und ihre Intentionsliste wird verworfen.

Das Problem des gegenseitigen Sperrens von Operationen durch verschiedene Transaktionen, die sog. *Verklemmung* (*deadlock*), muß hier durch auch bei anderen Algorithmen übliche Methoden wie z.B. Verklemmungserkennung oder Time-out vermieden oder aufgelöst werden.

Betrachten wir als Beispiel erneut die Historie der FIFO–Schlange aus Abschnitt 7.3.2, so erkennt man (wie bereits grundsätzlich festgestellt), daß enqueue–Operationen auf der Schlange keine Konflikte erzeugen. Der Algorithmus im allgemeinen und die notierte Historie im besonderen erlauben also nebenläufig ausgeführte enqueue–Operationen. Die Reihenfolge, in der die zuvor nebenläufig eingefügten Schlangen–Elemente durch die dequeue–Operation ausgelesen/gelöscht werden, wird einzig durch die Zeitmarken der commits bestimmt, die von den nebenläufigen Transaktionen vergeben werden. Hierin liegt der gravierende Unterschied zu Algorithmen, die die Serialisierbarkeit im Kontext der Kommutativitätsbedingung definieren, da die enqueue–Operationen hier nicht kommutativ sind und die Historie durch solche Algorithmen nicht akzeptiert würde, wohl aber durch den oben skizzierten.

7.4.2 Die Abhängigkeitsrelation

Die grundlegende Einschränkung zur Behandlung von Sperrkonflikten ist die Formulierung der *Abhängigkeit*: Zwei Operationen o_1 und o_2 können nicht nebenläufig ausgeführt werden, wenn die eine von der anderen abhängt.
Sei R eine zweistellige Relation auf Operationen, h eine Operationenfolge und SSp die

serielle Spezifikation eines Objekts O. Dann definieren wir: Eine zweistellige Relation R auf Operationen ist eine *Abhängigkeitsrelation* für die serielle Spezifikation SSp, wenn für alle Operationenfolgen os_1, os_2 und alle Operationen o, so daß $os_1 \bullet os_2$ und $os_1 \bullet o$ legal sind und für alle Operationen p in os_2, $(p,o) \notin R$, $os_1 \bullet o \bullet os_2$ legal ist.

Wenn also os_2 nach os_1 legal ist, o nach os_1 legal ist und keine Operation in os_2 von o abhängt, so kann os_2 nach o ausgeführt werden.

Man nennt eine Abhängigkeitsrelation R *minimal*, wenn es keine echte Teilmenge R′ von R gibt, die ebenfalls eine Abhängigkeitsrelation darstellt. Es läßt sich zeigen, daß es zum einen unterschiedliche minimale Abhängigkeitsrelationen für ein Objekt geben kann und — als wichtiges Ergebnis — daß zum anderen der angegebene Algorithmus genau dann korrekt ist, wenn die Konfliktrelation eine symmetrische Abhängigkeitsrelation ist.

Eine Teilfolge g einer Operationenfolge h ist *R-abgeschlossen*, wenn immer dann, wenn g eine Operation q aus h enthält, g auch alle früheren Operationen p aus h enthält, so daß $(q,p) \in R$. Eine Teilfolge g einer Operationenfolge h ist eine *R-Sicht* von h für q, wenn g R-abgeschlossen ist und jedes p in h enthält, so daß $(q,p) \in R$.

An dieser Stelle sind zwei wichtige Eigenschaften von Abhängigkeitsrelationen festzuhalten:

1. Sei R eine Abhängigkeitsrelation für SSp, os eine Operationenfolge und k_1 und k_2 Operationenfolgen, so daß gilt: $os \bullet k_1$ und $os \bullet k_2$ sind legal. Wenn keine Operation in k_1 von einer Operation in k_2 abhängt, so ist auch $os \bullet k_2 \bullet k_1$ legal.

2. Sei R eine Abhängigkeitsrelation für SSp und g und h Operationenfolgen in SSp, so daß g eine R-Sicht von h ist für eine Operation q. Wenn $g \bullet q$ in der seriellen Spezifikation SSp ist, dann ist es auch $h \bullet q$ in SSp.

Obige Definition der Abhängigkeitsrelation ist nicht konstruktiv, d.h. es läßt sich zwar testen, ob eine vorliegende Relation eine Abhängigkeitsrelation ist oder nicht, die Definition liefert aber keine Hinweise zur Herleitung einer solchen Bedingung aus der seriellen Spezifikation der Objekte.

Deshalb soll zunächst eine Methode skizziert werden, wie man für ein gegebenes Objekt eine Abhängigkeitsrelation aufstellt. Wir tun dies, indem wir sagen, eine Operation ist von irgendwelchen früheren Operationen abhängig, die sie invalidieren[2] könnten.

Eine Operation o_1 *invalidiert* eine Operation o_2, wenn es Operationenfolgen f_1 und f_2 gibt, so daß $f_1 \bullet o_1 \bullet f_2$ und $f_1 \bullet f_2 \bullet o_2$ legal sind, nicht aber $f_1 \bullet o_1 \bullet o_2$. Mit Hilfe dieser Definition läßt sich dann eine Relation invalidiert_von definieren, die alle Paare (o_1,o_2) von Operationen umfaßt, so daß die Operation o_2 die Operation o_1 invalidiert.
Es läßt sich zeigen, daß invalidiert_von eine — nicht unbedingt minimale — Abhängigkeitsrelation ist.

Am Ende soll ein Beispiel — unter Einbeziehung der bereits bekannten FIFO–Schlange — die Verwendung der Abhängigkeitsrelation verdeutlichen. Zuvor ist noch anzumerken, daß sehr wohl unterschieden werden muß zwischen Abhängigkeits– und Konfliktrelationen. Bei Abhängigkeitsrelationen ist keine Symmetrie erforderlich, die im Algorithmus verwendeten Konfliktrelationen müssen allerdings symmetrisch sein! üblicherweise kann man eine Konfliktrelation gewinnen, indem man den symmetrischen Abschluß der Abhängigkeitsrelation bildet.

[2] d.h. ungültig machen

Das Objekt FIFO–Schlange besitzt die beiden bereits bekannten Operationen enqueue und dequeue. Die Operation dequeue stoppt bei leerer Schlange ohne Wirkung, d.h. ihre Spezifikation ist partiell.
Für die FIFO–Schlange lassen sich zwei verschiedene minimale Abhängigkeitsrelationen angeben.
Die erste läßt sich im Diagramm wie folgt angeben:

	enqueue(v),OK	dequeue(),v
enqueue(v'),OK		
dequeue(),v'	$v \neq v'$	$v = v'$

Sie stellt die Relation invalidiert_von für FIFO–Schlangen dar: Eine dequeue–Operation, die ein beliebiges gegebenes Datenelement involviert, hängt sowohl von enqueue–Operationen, die andere Datenelemente involvieren als auch von dequeue–Operationen ab, die dasselbe Datenelement involvieren. Folglich kann ein dequeue nicht nebenläufig zu anderen enqueue– oder dequeue–Operationen ausgeführt werden, während man aus dem Diagramm ersehen kann, daß enqueue–Operationen diesbezüglich keine Einschränkungen haben und nebenläufig ablaufen können.

Die zweite Abhängigkeitsrelation läßt sich durch das Diagramm

	enqueue(v),OK	dequeue(),v
enqueue(v'),OK	$v \neq v'$	
dequeue(),v'		$v = v'$

darstellen. Hier sind enqueue–Operationen, die verschiedene Datenelemente involvieren, voneinander abhängig, ebenso dequeue–Operationen, die dieselben Datenelemente involvieren. Es hängen aber weder die enqueue– von den dequeue–Operationen ab noch umgekehrt. Eine enqueue ausführende Transaktion kann also nebenläufig zu einer dequeue ausführenden Transaktion ablaufen, solange letztere die dequeue–Operation auf Datenelementen ausführen kann, die durch enqueue–Operationen von bereits abgeschlossenen Transaktionen eingefügt wurden.

7.5 Der Hybride Sperr–Algorithmus

Ist die serielle Spezifikation SSp eines Objekts O gegeben, so stellt der folgende Algorithmus sicher, daß alle Historien, die von seiner Implementierung erzeugt werden, hybrid atomar sind.

Für die Beschreibung des Algorithmus werden wir einen Zustandautomaten verwenden. Dieser wird repräsentiert durch eine Menge von Zuständen, eine Menge von Zustandübergängen, einen Startzustand und eine partielle übergangsfunktion, die ein Paar (Zustand,übergang) abbildet auf einen neuen Zustand. Ist die übergangsfunktion für ein gegebenes Paar (z,ü) definiert, so nennen wir ü *definiert in z*. Die übergangsfunktion kann offensichtlich auf Sequenzen von übergängen erweitert werden. Man nennt eine übergangssequenz *akzeptiert* von einem Automaten A, wenn sie in seinem Anfangszustand definiert

ist. Die Menge endlicher übergangssequenzen, die von A akzeptiert werden, nennt man die Sprache des Automaten und bezeichnet sie mit L(A).

Der Zustandsautomat

Der Algorithmus wird durch einen Automaten, den wir SPERRE nennen, repräsentiert, und dessen Sprache aus einer Menge von Ereignissequenzen besteht. Der Automat hat eine partielle, symmetrische Konfliktrelation Konflikt zur Verfügung, die testet, ob zwei Operation in gegenseitigem Konflikt stehen oder nicht.
Es zeigt sich, daß Konfliktrelationen, die von den zuvor besprochenen Abhängigkeitsrelationen abgeleitet werden, notwendig und hinreichend sind für eine garantierte Korrektheit des Algorithmus' in dem Sinne, daß jede Historie in L(SPERRE) hybrid atomar ist.

Die Zustände

Vier verschiedene Komponenten kennzeichnen einen Zustand des Zustandsautomaten SPERRE:

- z.laufend, eine partielle Funktion von Transaktionen nach Aufruf–Ereignissen. z.laufend nimmt laufende Aufrufe von Transaktionen auf. Da im Initialzustand keine Transaktion einen laufenden Aufruf hat, ist z.laufend im Startzustand von SPERRE undefiniert.

- z.intention, eine totale Funktion von Transaktionen nach Operationensequenzen. z.intention nimmt die Folge von Operationen auf, die von den Transaktionen ausgeführt werden. Im Startzustand von SPERRE bildet z.intention jede Transaktion auf die leere Sequenz ab.
  Sperren werden in diesem formalen Algorithmen-Modell nicht explizit gehalten, sie lassen sich jedoch aus der Intentionsliste (z.intention) implizit gewinnen.

- z.committed, eine partielle Funktion von Transaktionen nach Zeitmarken. In z.committed sind diejenigen Transaktionen geführt, die bereits ein commit durchgeführt haben und für jede Transaktion die von ihr vergebene Zeitmarke. Für alle Transaktionen ist z.committed im Startzustand undefiniert.

- z.aborted, eine Menge von Transaktionen. z.aborted nimmt die Transaktionen auf, die ein abort durchgeführt haben. Im Startzustand gilt z.aborted $= \emptyset$.

Sei z ein beliebiger Zustand in SPERRE. Dann ist z.completed definiert als z.aborted $\cup\{T|$z.committed$(T) \neq \perp\}$. z.completed besteht also aus denjenigen Transaktionen, die entweder ein commit oder ein abort durchgeführt haben. Ist eine Transaktion $Q \notin$ z.completed, so definiert man Sicht(Q,z) als Operationenfolge, die man erhält, wenn man die die Intentionslisten aller abgeschlossenen Transaktionen in der Reihenfolge ihrer Zeitmarken konkateniert und daran die Intentionsliste von Q anhängt.

Die Zustandsübergänge

Die Zustandsübergänge des Automaten SPERRE sind die Ereignisse, die das Objekt O involvieren. Die einzelnen Ereignisse sowie ihre Vor– und Nachbedingungen sind unten beschrieben. Dabei wurde zur Vereinfachung unterstellt, daß alle auszuführenden Historien den 'well-formedness'–Bedingungen aus Abschnitt 7.2 genügen. Eine explizite Prüfung

der Einhaltung dieser Bedingungen würde sich in umfangreicherer Formulierung der Zustandskomponenten und Vorbedingungen niederschlagen.

Bei der Notation meint m[a→b], wobei m eine (u.U. partielle) Funktion von A nach B, a ∈ A, b ∈ B darstellt, die Funktion, die identisch ist zu m mit Ausnahme von a, das auf b abgebildet wird. Weiterhin bedeutet bei der Formulierung der Vor– und Nachbedingungen konventionsgemäß eine gestrichene Variable den Zustand *vor* dem angegebenen Ereignis, die ungestrichene Variable den Zustand danach. Zusätzlich gilt stillschweigend die Vereinbarung, daß eine in der Nachbedingung nicht aufgeführte Zustandskomponente durch die Ausführung des Ereignisses nicht verändert wurde.

Die drei Ereignisse Aufruf, Abschluß und Abbruch sind Ereignisse, die von den Transaktionen initiiert werden. Ihre Vorbedingungen sind deshalb TRUE. Aufgrund der definierten Aufgaben ist der Zustandsübergang für jedes dieser drei Ereignisse vergleichsweise einfach, da jedes Ereignis 'lediglich' in den Zustand des Automaten SPERRE aufgenommen wird:

- Aufruf (invocation) $\langle inv, O, Q \rangle$
  Vorbedingung: TRUE
  Nachbedingung: z.laufend = z′.laufend[Q→inv]

- Abschluß (commit) $\langle commit(t), O, Q \rangle$
  Vorbedingung: TRUE
  Nachbedingung: z.committed = z′.committed[Q→t]

- Abbruch (abort) $\langle abort, O, Q \rangle$
  Vorbedingung: TRUE
  Nachbedingung: z.aborted = z′.aborted $\cup\{Q\}$

Das letzte verbleibende Ereignis, die Antwort, geht vom Objekt aus. Die Vor– und Nachbedingungen sind aufgrund der folgenden überlegungen aufwendiger.

1. Um eine Antwort an eine Transaktion zurückliefern zu können, muß diese Transaktion einen laufenden Aufruf haben.

2. Die Transaktion, die die Antwort erhalten soll, darf kein commit und kein abort durchgeführt haben.

3. Die aus einem Paar ⟨Aufruf,Antwort⟩ bestehende Operation muß aus der Sicht der Transaktion legal sein.

4. Kann ein Antwort–Ereignis stattfinden, so muß der noch laufende Aufruf aus dem Zustand entfernt werden.

5. Außerdem muß die Intentionsliste der Transaktion durch Aufnahme der neuen Operation auf den neuesten Stand gebracht werden.

Daraus ergibt sich schließlich formal:

- Antwort (response) $\langle inv, O, Q\rangle$
  Vorbedingung: $z'.\mathsf{laufend}(Q) \neq \perp$
  $Q \notin z'.\mathsf{completed}$
  $q := \langle s'.\mathsf{laufend}(Q), \mathsf{response}\rangle$
  $\mathsf{Sicht}(Q, z') \bullet q \in \mathsf{SSp}$
  Für alle Transaktionen $P \notin z'.\mathsf{completed} \cup \{Q\}$ und
  für alle Operationen p in $z'.\mathsf{intention}(P)$,
  $\langle p, q\rangle \notin \mathsf{Konflikt}$
  Nachbedingung: $z.\mathsf{laufend} = z'.\mathsf{laufend}[Q \rightarrow \perp]$
  $z.\mathsf{intention} = z'.\mathsf{intention}[Q \rightarrow z'.\mathsf{intention}(Q) \bullet q]$

Es ist zu beachten, daß z.intention für alle, also auch für die bereits abgeschlossenen Transaktionen gehalten wird. Der Abschluß–Zustand ist also gleich der Intentionsliste in der Reihenfolge der Zeitmarken. Diese Vorgehensweise mag nicht die günstigste sein, gestattet es aber, den Algorithmus in einfacher und allgemeiner Form zu notieren[3]. Alle anderen Methoden des Wiederanlaufs sind quasi nur Spezialfälle der hier eingeführten Intentionsliste in dem Sinne, daß keine andere Methode mehr Informationen über die vergangenen Operationen im Zustand abspeichert.

An dem folgenden Ablauf soll die Wirkungsweise des Algorithmus nochmals verdeutlicht werden:

- Es sei Konflikt eine Abhängigkeitsrelation.

- Eine Transaktion T_1 führt eine Operationenfolge os_1 aus und danach ein commit. Aus den obigen Vorbedingungen für das response–Ereignis ergibt sich, daß os_1 legal ist, d.h. die Historie bis zu diesem Zeitpunkt ist hybrid atomar.

- Nun führen zwei Transaktionen T_2 und T_3 die Operationenfolgen os_2 und os_3 aus. $os_1 \bullet os_2$ und $os_1 \bullet os_3$ müssen dazu beide legal sein. Weiterhin ergibt sich wieder aus den Vorbedingungen von response, daß keine Operation in os_2 mit einer Operation in os_3 in Konflikt stehen kann.

- Wenn T_2 nun ein commit ausführt, so muß die vergebene Zeitmarke größer sein als die von T_1.

- Da $os_1 \bullet os_2$ legal ist, ist die Historie noch immer hybrid atomar.

- Wenn schließlich auch T_3 ein commit ausführt, so ist zu zeigen, daß entweder $os_1 \bullet os_2 \bullet os_3$ oder $os_1 \bullet os_3 \bullet os_2$ legal ist.

- Dies hängt davon ab, ob die von T_3 vergebene Zeitmarke kleiner oder größer ist als die von T_2 vergebene. Da keine Operation in os_2 von einer Operation in os_3 abhängig ist und umgekehrt, folgt aus den in Abschnitt 7.4.2 angegebenen Eigenschaften der Abhängigkeitsrelationen das gewünschte Ergebnis.

[3] Eine kompaktere und effizientere Möglichkeit der Formulierung ist in [HW91] zu finden.

7.6 Zusammenfassung

Mechanismen, die auf Zwei-Phasen–Sperren beruhen, sichern, daß die Transaktionen in der durch precedes gegebenen Totalordnung serialisierbar sind. Diese Eigenschaft bezeichnet man auch als 'dynamische Atomizität'. Im Kontrast hierzu gibt es auch die 'statische Atomizität', die beispielsweise in Multiversionsalgorithmen zum Tragen kommt.

Eine noch weiter verallgemeinerte Form des obigen Algorithmus' behandelte lesende und schreibende Transaktionen unterschiedlich: Schreibende Transaktionen vergeben die Zeitmarke bereits zu Beginn des Lesezugriffs, während die Zeitmarken für alle anderen Transaktionen erst beim commit vergeben werden. In dieser Vorgehensweise liegt auch der Ursprung der begrifflichen Bezeichnung 'hybrid', was man im Kontext so auffassen kann, daß der Algorithmus die Methode der Zeitmarkenvergabe 'gemischt', also nicht für alle Arten von Transaktionen einheitlich, behandelt.

Die meisten der entwickelten Algorithmen, die die Nebenläufigkeitskontrolle im Zusammenhang mit Konflikten behandeln, machen die bereits erwähnte Kommutativitätseigenschaft zur Grundlage der Entscheidung.

Der oben formalisierte Algorithmus löst Konflikte durch Forderung schwächerer Eigenschaften, indem die Bedingung der Kommutiativität durch die Abhängigkeitsrelation zwischen Operationen ersetzt wird. Dadurch wird ein größeres Maß an Nebenläufigkeit als bei den 'herkömmlichen' Verfahren zugelassen. Ein weiterer Vorteil liegt in der Handhabbarkeit von partiellen und nicht-deterministischen Operationen.

Schließlich läßt sich festhalten, daß der Algorithmus *optimal* ist in dem Sinne, daß es kein hybrid atomares Sperrverfahren gibt, das eine höhere Nebenläufigkeit gestatten kann.

Teil IV

Erweiterte Funktionalität

Kapitel 8

Evolution in objektorientierten DBMS(*Christoph Toussaint*)

8.1 Evolution in OTGen

Datenbanken werden oft mit abstrakten Sprachen beschrieben, bevor sie in der Implementierungsumgebung codiert werden. Aus den abstrakten Definitionen kann man meistens auch schon große Teile des Codes automatisch generieren. Werden die abstrakten Definitionen und damit der Aufbau der Datenbank gendert, steht man vor dem Problem, daß man auch die Daten mitändern muß, da man diese im Regelfall weiterverwenden will.

OTGen (Object Transformer Generator) ist ein System, mit dem man beschreiben kann, was mit bestehenden Daten geschehen soll, wenn die Struktur einer objektorientierten Datenbank geändert wird. Damit kann dann Code erzeugt werden, mit dem dieser Prozeß automatisch ausgeführt werden kann.

OTGen ist eine Weiterentwicklung von TransformGen, einem entsprechenden System für Datenbanksysteme mit Baumstruktur [SKG88]. Bei beiden Systemen werden Transformationstabellen generiert, die der Anwender dann bearbeiten kann, um weitergehende Transformationen auszuführen, wie zum Beispiel die Verlagerung von Daten aus einem Objekt in ein anderes. Die TransformGen-Umgebung wird noch heute an vielen Stellen benutzt, um Datenbanken zu definieren und gegebenenfalls zu ändern. OTGen wurde im Oktober 1990 in [LH90] zum erstenmal vorgestellt, um gleiches auch für objektorientierte Datenbanken anzubieten.

8.1.1 Das objektorientierte Datenmodell

Zum besseren Verständnis des Textes soll ein einfaches objektorientiertes Datenmodell eingeführt werden. Dieses Datenmodell entspricht in etwa dem der meisten objektorientierten Datenbanken und Programmiersprachen.

Man kann beliebig viele Klassen definieren. Eine Klasse besteht aus Variablen und Methoden. Objekte einer Klasse werden dynamisch erzeugt. Ein Objekt hat einen Typ, der durch die korrespondierende Klasse definiert wird. Jede Klasse hat mindestens eine Oberklasse[1] außer der speziellen Klasse ANY, die an der Wurzel der Klassenhierarchie

[1] In den meisten Texten wird hier von der Basisklasse (base class) gesprochen. Da es hierzu jedoch kein Gegenstück zur Beschreibung der umgekehrten Beziehung gibt, wird das Wortpaar Oberklasse/Unterklasse (super class/sub class) benutzt.

steht. Eine Klasse erbt die Variablen und Methoden ihrer Oberklassen und kann neue hinzudefinieren oder die ererbten überschreiben, indem in der Klasse lokal eine Methode oder Variable mit dem gleichen Namen definiert wird. Mit den folgenden Forderungen soll dies nun etwas präziser gefaßt werden. Die Namen der Forderungen wurden absichtlich nicht übersetzt, da sich diese nur schwer in der deutschen Sprache mit einer ähnlichen Prägnanz und Kürze ausdrücken lassen.

Class Lattice Invariant Durch die Ober-/Unterklassenbeziehung wird eine Hierarchie gebildet, deren Wurzel die spezielle Klasse ANY ist. Die Hierarchie darf insbesondere keine Zyklen haben.

Unique Name Invariant Jede Klasse muß einen eindeutigen Namen haben. Jede Variable und Methode, die in einer Klasse definiert wurde, muß einen eindeutigen Namen haben[2].

Full Inheritance Invariant Eine Klasse erbt alle Variablen und Methoden ihrer Oberklassen, es sei denn in der Klasse wird eine Variable oder Methode mit dem gleichen Namen definiert. Wenn mehr als eine Oberklasse eine Methode oder Variable mit dem gleichen Namen definiert hat, wird diejenige geerbt, die von der Klasse definiert wurde, die in der Liste der zu erbenden Oberklassen in der Definition der Klasse zuerst steht[3].

Type Compatibility Invariant Wird in einer Klasse t eine Variable überschrieben, die sie andernfalls von der Oberklasse s geerbt hätte, dann muß der Typ der Variablen in der Klasse t eine Unterklasse des Typs der Variablen der Klasse s sein. Variablen können nur durch Variablen überschrieben werden, Methoden nur durch Methoden[4].

Typed Variable Invariant Der Typ jeder Variablen muß eine korrespondierende Klasse in der Klassenhierarchie haben.

8.1.2 Mögliche Änderungen in der Klassenhierarchie

Während bei einem Datenmodell ohne Vererbung Änderungen einer Klasse nur die Klasse selbst betreffen, kann dies bei einem Datenmodell mit Vererbung Auswirkungen auf alle Unterklassen haben. Damit lassen sich die Auswirkungen einer Änderung auch wesentlich schlechter voraussehen und der Einsatz eines Systems wie OTGen wird unausweichlich.

Es werden nun alle möglichen Änderungen in dem objektorientierten Datenmodell aufgelistet und dann überlegt, welche Auswirkungen diese Änderungen haben können.

- Hinzufügen einer Variablen in einer Klasse
- Löschen einer Variablen in einer Klasse
- Umbenennen einer Variablen in einer Klasse

[2] Dadurch werden die Probleme des Überladens umgangen.

[3] Genauso könnte man wie etwa in Eiffel [Mey92] die Möglichkeit bieten, explizit zu wählen, woher geerbt werden soll. Die Art und Weise hingegen wie dies in C++ [Str91] geschieht, erfüllt die Forderung nicht ganz korrekt.

[4] Die Probleme der Typenkompatibilität von Parametern der Methoden aufgrund der Polymorphie sind hier nicht interessant und werden deswegen weggelassen.

- Ändern des Typs einer Variablen in einer Klasse
- Hinzufügen einer Klasse in die Liste der Oberklassen einer Klasse
- Löschen einer Klasse in der Liste der Oberklassen einer Klasse
- Hinzufügen einer neuen Klasse
- Löschen einer Klasse
- Umbenennen einer Klasse

Es gibt Änderungen, die vom System als Fehler zurückgewiesen werden müssen, weil sie die Forderungen auf Seite 140 verletzen würden. Ein solcher Konflikt wäre z.B. das Ändern der Vererbungsstruktur derart, daß in der Klassenhierarchie Zyklen entständen. Diese Konflikte werden dem Anwender vom OTGen mitgeteilt, und der Anwender kann sie dann beseitigen. Sind alle Konflikte ausgeräumt und die Forderungen erfüllt, kann die Standardtransformationstabelle erzeugt werden. Der Anwender kann die Tabelle edieren und darüberhinausgehende Transformationen angeben.

Bei der Durchführung der Datentransformation selbst können nur noch Typinkompatibilitäten bei den Zuweisungen auftreten. Diese kann OTGen durch Analyse der Transformationstabellen im voraus erkennen und den Anwender darauf hinweisen. Der Anwender kann die Warnungen jedoch ignorieren, etwa weil er weiß, daß in den Datenbeständen diese fehlererzeugenden Konstellationen nicht auftreten. Durch diese Analyse wird sichergestellt, daß Fehler in der Transformation nicht erst beim eigentlichen Prozeß der Transformation erkannt werden, sondern gleich im Vorfeld. Andernfalls könnte es nämlich durchaus vorkommen, daß der Systembetreuer Transformationen plant, die mit bestimmten Datenbeständen, die einige Zeit später transformiert werden sollen, nicht vereinbar sind.

8.1.3 Forderungen an die Transformation

Der eigentliche Transformationsprozeß muß folgende drei Forderungen erfüllen, damit er sinnvoll die Daten der alten Definition in die neue überführt.

Vollständigkeit *Jedes* Objekt muß in seine neue Version transformiert worden sein, bevor man darauf zugreifen kann.

Korrektheit Nach der Transformation muß jedes Objekt zu einer Klassendefinition in der neuen Version *passen*. Inbesondere muß der Wert einer Variablen zu dem Typ der Variablen in der entsprechenden Klassendefinition *passen*.

Erhaltung gemeinsamer Daten Wenn zwei Variablen vor der Transformation auf ein *gemeinsames* Objekt zeigten, und wenn

- keine der Variablen gelöscht wurde,
- keine der Standardtransformationen der Variablen überschrieben wurde,
- und der Typ des transformierten gemeinsamen Objekts zu den Variablen paßt,

dann müssen sie auch nach der Transformation auf ein *gemeinsames* Objekt zeigen.

8.1.4 Die Standardtransformationen

Anhand der Forderungen an die Transformation kann man jetzt die Standardtransformationen definieren, die bei den einzelnen Änderungen in Abschnitt 8.1.2 ausgeführt werden sollen. Diese Standardtransformationen sollten die Datenbestände soweit wie möglich unberührt lassen und nur dort etwas ändern, wo es durch Änderungen der Definitionen unbedingt nötig ist. Ändert sich in der Klasse nichts, wird man also die Objekte dieser Klasse einfach in die neue Version der Datenbank hinüberkopieren. Für weitere Erläuterungen soll nun ein kleines Beispiel eingeführt werden.

Class C subclass of ANY is
$v1$: T1
$v2$: T2

Die Standardtransformationstabelle sähe in diesem Fall folgendermaßen aus (Zu beachten ist, daß in der Transformationstabelle sowohl die lokal definierten wie auch die ererbten Variablen aufgeführt werden):

Class C:
new self: C
$v1$: Transform $v1$
$v2$: Transform $v2$

Die alten Werte der Variablen $v1$ und $v2$ werden rekursiv transformiert. Die Transformation muß rekursiv sein, da für die Typen dieser Variablen ebenso Transformationen angegeben sind, die durchgeführt werden müssen. Passen die jeweiligen Ergebnisse zu den neuen Typen der Variablen, werden sie den Variablen zugewiesen, ansonsten bekommen die Variablen den speziellen Wert $Void$.

Wird hingegen eine neue Variable $v3$ hinzugefügt, sieht die Standardtransformation so aus:

Class C:
new self: C
$v1$: Transform $v1$
$v2$: Transform $v2$
$v3$: $Void$

Im nächsten Abschnitt werden die zusätzlichen Möglichkeiten vorgestellt, die Datenbank zu ändern.

8.1.5 Die zusätzlichen Transformationen

Initialisieren von Variablen

Will der Anwender die Variable mit einem Wert, etwa 0, initialisieren, ändert er die Tabelle folgendermaßen:

Class C:
new self: C
$v1$: Transform $v1$
$v2$: Transform $v2$
$v3$: 0

Kontextabhängige Änderungen

Es besteht die Möglichkeit, vor jeden Teil der Tabelle eine boolsche „Weiche“ zu setzen. Damit lassen sich abhängig vom Kontext unterschiedliche Transformationen durchführen. Angenommen, man fügt eine Klasse D als Unterklasse der Klasse C in dem Beispiel wie folgt ein:

```
Class D subclass of C is
    v2: T4
```

Will man nun die Objekte der Klasse C, bei denen der Variablenwert von $v2$ vom Typ T4 ist, in Objekte des Typs D umwandeln, gibt man folgende Transformationstabelle an:

```
Class C:
    if TypeCheck (Transform(v2), T4)
        new self: D
            v1: Transform v1
            v2: Transform v2
    else
        new self: C
            v1: Transform v1
            v2: Transform v2
```

Die Funktion TypeCheck liefert *true*, wenn der Typ der Objekts der gleiche oder der einer Unterklasse der gegebenen Klasse ist. Der Aufruf mit dem Argument „Transform($v2$)“ ist nötig, da man den Typ des transformierten Objekts untersuchen will und nicht den des Originals. Mit Hilfe eines Verfahrens, das in Abschnitt 8.1.6 vorgestellt wird, kann sichergestellt werden, daß die Transformation nur einmal vorgenommen wird.

Bewegen von Daten

Häufig will man Daten von einem Objekt in ein anderes bewegen. Auch dies ist problemlos möglich, wie das folgende Beispiel zeigt:

```
Class OUTER subclass of ANY is
    o1: T1
    o2: INNER

Class INNER subclass of ANY is
    i1: T2
    i2: T3
```

wird geändert in

```
Class OUTER subclass of ANY is
    o1: T1
    o2: INNER
    i2: T3

Class INNER subclass of ANY is
    i1: T2
```

Die Standardtransformationstabelle würde hier dann so aussehen:

Class OUTER:
    new self: OUTER
        $o1$: Transform $o1$
        $o2$: Transform $o2$
        $i2$: $Void$

Class INNER:
    new self: INNER
        $i1$: Transform $i1$

Will man den alten Wert $i2$ nicht verlieren, ändert man sie folgendermaßen:

Class OUTER:
    new self: OUTER
        $o1$: Transform $o1$
        $o2$: Transform $o2$
        $i2$: Transform $o2.i2$

Class INNER:
    new self: INNER
        $i1$: Transform $i1$

Erzeugen neuer Objekte

Die Erzeugung neuer Objekte in der Datenbank wird durch das folgende Beispiel erläutert, bei dem der Wert der Variablen $v1$ in eine neue Klasse gepackt werden soll:

Class C subclass of ANY is
    $v1$: T1
    $v2$: T2

wird geändert in

Class C subclass of ANY is
    $v1$: WRAPPER
    $v2$: T2

Class WRAPPER subclass of ANY is
    $w1$: T1

Die Standardtransformation wäre hier wenig sinnvoll, denn die Zuweisung schlägt immer fehl, da der Typ WRAPPER und C nicht kompatibel sind. Es würde also immer $Void$ zugewiesen werden. Die gewünschte Wirkung wird folgendermaßen erreicht:

Class C:
    new self: C
        $v1$: Create WRAPPER
            $w1$: Transform $v1$
        $v2$: Transform $v2$

Gemeinsame Objekte

In Abschnitt 8.1.3 wird verlangt, daß gemeinsame Objekte vor der Transformation auch nach der Transformation gemeinsame Objekte bleiben. Dies wird automatisch von dem System sichergestellt. Will man jedoch neue gemeinsame Objekte erzeugen, braucht man einen zusätzlichen Mechanismus. Hierzu halte man sich noch einmal das Beispiel aus dem vorhergehenden Absatz vor Augen. Zeigen die Variablen $v1$ von mehreren Objekten der Klasse C auf ein gemeinsames Objekt der Klasse T1, dann geht dies beim Erzeugen der Objekte der Klasse WRAPPER verloren. Die Lösung des Problems läuft über folgendes Konstrukt:

Class C:
    new self: C
        $v1$: Share $NewWrap(v1)$
        $v2$: Transform $v2$

Shared expression $NewWrap(OldObj)$
    Create WRAPPER
        $w1$: Transform $OldObj$

Ein gemeinsamer Ausdruck ist eine Funktion mit Parametern, die für jede Kombination von Werten der Argumente genau einmal ausgewertet wird. Das Ergebnis wird, indiziert nach den Werten der Argumente, in einer Tabelle gespeichert. Wenn die Funktion wieder aufgerufen wird, kann anhand dieser Tabelle festgestellt werden, ob der Funktionswert für diese Werte der Argumente schon berechnet wurde. In diesem Falle das Ergebnis aus der Tabelle abgelesen und zurückgeliefert werden.

8.1.6 Technische Aspekte der Transformation

In diesem Abschnitt sollen noch einige technische Aspekte des Transformationsprozesses angesprochen werden. Änderungen in der Datenbank müssen nicht sofort ausgeführt werden. Vielmehr kann man warten, bis auf das Objekt zugegriffen wird.

Implementiert wurde diese Möglichkeit mit Hilfe von Versionsnummern, die zusätzlich in jedem Objekt abgespeichert werden. Erfolgt ein Zugriff auf ein Objekt, wird die Versionsnummer in dem Objekt mit der aktuellen Versionsnummer verglichen. Ist das Objekt noch von einer älteren Version, wird der Transformationsprozeß für dieses und alle erreichbaren Objekte aufgerufen. Hierbei ist es sogar möglich, Objekte zu transformieren, deren Versionsnummern weiter zurückliegen. Es werden dazu die einzelnen Transformationen nacheinander aufgerufen. Dadurch brauchen nicht bei jeder neuen Version alle Daten sofort umgestellt werden, sondern es müssen nur die Datenbankserver neu gestartet werden.

Das Einhalten der Forderung „Erhaltung gemeinsamer Daten“ in Abschnitt 8.1.3 wird über eindeutige Kennummern (UID, Unique Identifiers) garantiert. Jedes Objekt hat eine solche UID. Auch wenn ein Objekt transformiert wird, behält es in der neuen Datenbank die gleiche UID. Man braucht also vor jeder Transformation nur in der neuen Datenbank nachzuschauen, ob bereits ein Objekt mit dieser UID existiert. Ähnlich funktioniert dies bei „Shared expressions“. Hier wird eine Tabelle angelegt, die nach den UID der Werte der Argumente indiziert ist. Als Tabellenwert wird die UID des Objektes abgespeichert, das als Funktionswert zurückgegeben wurde. Bei jedem Funktionsaufruf kann nun ebenfalls

über diese Tabelle festgestellt werden, ob die Funktion mit diesen Werten der Argumente bereits evaluiert wurde.

8.2 Evolution in ObjectStore

8.2.1 Der Aufbau von ObjectStore

ObjectStore [Obj93] ist eine Datenbank, die die Möglichkeit der Persistenz von Objekten in C++ bietet. Während normale Datenbanken nur Funktionen zum Zugriff auf die Daten in der Datenbank bieten, geht ObjectStore hier einen anderen Weg. Man spricht die Objekte über Zeiger im Hauptspeicher an. Dabei sind die Zeiger allerdings virtuelle Zeiger, wie sie auch beim virtuellen Speichermanagement verwendet werden. Der Wert eines Zeigers ist entweder eine gültige Hauptspeicheradresse oder ein Zeiger in die Datenbank. ObjectStore schaltet sich direkt hinter den virtuellen Speichermanager. Wenn der Zeiger keine gültige Hauptspeicheradresse enthält und deswegen ein Adressierungsfehler ausgelöst wird, schaut ObjectStore in einer Tabelle nach, ob es sich um einen gültigen Zeiger auf die Datenbank handelt. In der gleichen Tabelle steht auch, ob das entsprechende Objekt bereits in den Hauptspeicher geladen wurde, oder ob dies noch getan werden muß. In beiden Fällen wird der Zeiger in den entsprechenden Hauptspeicherzeiger geändert. Steht der Zeiger nicht in der Tabelle, war er ungültig, und der Adressierungsfehler wird weitergereicht. Auf diese Weise wird erreicht, daß die Datenbank wesentlich einfacher zu bedienen ist und vor allem schneller ist. Denn nachdem das Objekt in den Hauptspeicher geladen wurde, kann es genauso schnell angesprochen werden wie jedes normale Hauptspeicherobjekt. Am Ende des Programms muß das System dann nur noch dafür Sorge tragen, daß alle Objekte wieder in der Datenbank abgespeichert werden. Der zweite große Vorteil ist, daß bestehende Programme ganz leicht umgeschrieben werden können, sodaß sie Persistenz unterstüzten, da es keine äußerlichen Unterschiede zwischen den alten nicht-persistenten und den neuen persistenten Objekten gibt.

Da ObjectStore deswegen keine Datenbank im klassischen Sinne ist, wird die Struktur der Datenbank auch nicht über eine eigene Definition angegeben, sondern durch die im C++-Programm definierten Klassen. Es wird eine spezielle Klasse PERSISTENT gestellt, die ererbt wird. Alle Klassen, die diese Klasse erben, sind dann persistent. Durch einige Makros wird erreicht, daß die Vererbungsstruktur und der interne Aufbau der Klassen ObjectStore mitgeteilt werden und in der Datenbank abgepeichert werden können.

Wird nun das C++-Programm geändert, kann ObjectStore sofort feststellen, daß die Struktur der Datenbank nicht mehr aktuell ist. In diesem Fall müßte der Programmierer dann die Evolution durchführen.

8.2.2 Die Phasen der Evolution

Die Evolution gliedert sich in folgende Phasen:

Strukturänderung Hier wird die Struktur der Datenbank geändert. Dies geschieht in ObjectStore über das neue C++-Programm, in dem die neue Klassenstruktur definiert ist. Die sich daraus ergebenden Änderungen erkennt ObjectStore dann selbst.

Datenübertragung Hier werden die Daten aus der alten Datenbank in die neue übertragen. Dies geschieht in zwei Phasen:

Dateninitialisierung Hier werden die neuen Objekte mit Standardwerten initialisiert. Dies geschieht über die C++-Standardkonstruktoren. Ist der neue Typ einer Klassenvariablen zuweisungskompatibel zum alten Typ, wird der alte Wert zugewiesen. Der Schritt ist in etwa vergleichbar mit den Standardtransformationen, die OTGen vorschlägt.

Datentransformation Hier kann der Anwender eigene Transformationen angeben. Dieser Schritt ist vergleichbar mit den anwenderdefinierten Transformationen in OTGen. Anders als in OTGen finden die zwei Phasen jedoch hintereinander statt.

Die erste Phase ist weitestgehend automatisiert. Der Anwender baut eine neue Klassenstruktur auf und die Konsistenz der Struktur ist durch den C++-Sprachstandard gewährleistet. ObjectStore braucht demnach nicht die Einhaltung von Forderungen wie die aus dem Abschnitt 8.1.1 zu überprüfen. Der Strukturänderungsvorgang ist also ein rein technisches Problem, das den Anwender nicht zu interessieren braucht.

Die zweite Phase ist schon wesentlich schwieriger. Da der Anwender hier von ObjectStore so gut wie gar nicht unterstützt wird, ist die Einhaltung der Forderungen an die Transformation aus dem Abschnitt 8.1.3 schwer zu überprüfen. Während in OTGen alle Transformationen in der Transformationstabelle angegeben werden, sind diese Informationen in ObjectStore über mehrere Stellen verteilt.

Zuerst hat der Anwender wie auch in OTGen die Möglichkeit, das Objekt zu reklassifizieren. In OTGen geschieht dies anhand der Klasse, die nach dem Schlüsselwort self aufgeführt ist, in ObjectStore über eine Reklassifizierungsfunktion.

8.2.3 Der Aufbau einer Reklassifizierungsfunktion

Eine solche Funktion hat folgenden Aufbau:

```
char* reclassify(os_typed_pointer_void &the_old_obj)
```

Für jede Klasse kann eine solche Funktion angegeben werden. Die Funktion liefert als Rückgabewert den Namen der neuen Klasse als String oder, wenn die alte Klasse beibehalten werden soll, den Wert 0. Im Gegensatz zu OTGen ist es jedoch nur möglich, eine Klasse in eine Unterklasse zu reklassifizieren. Als Argument wird ein spezieller Zeiger auf das alte Objekt an die Funktion übergeben. Anhand dieses Zeigers lassen sich der Typ des alten Objekts ermitteln und Werte von Klassenvariablen erfragen. Da die alten Klassen jedoch in dem neuen C++-Programm nicht mehr vorhanden sind, können Werte der alten Klassenvariablen nur über spezielle Zugriffsfunktionen ermittelt werden. Diese liefern den Wert einer Variablen, wenn man den oben genannten Zeiger und den Namen der Variablen als String übergibt. Dadurch entstehen jedoch leicht Fehlerquellen, die erst bei Ablauf des Programms auftreten, nämlich dann, wenn man sich bei Variablennamen geirrt hat und es keine solche Variable gibt.

8.2.4 Der Aufbau einer Transformationsfunktion

Eine Transformationsfunktion, die für jede Klasse angegeben werden kann, hat folgenden Aufbau:

```
void transform(void *the_new_obj)
```

Anders als bei der Reklassifizierungsfunktion wird hier schon das neue Objekt übergeben. Es stehen dann Funktionen zur Verfügung, um auf das dazugehörige alte Objekt zuzugreifen. Da das neue Objekt bereits ein ganz normales persistentes C++-Objekt ist, kann auf die Variablenwerte ganz normal zugegriffen werden. Nur für „const"- und Referenzvariablen braucht man spezielle Funktionen. Das Objekt, dessen Zeiger an die Funktion übergeben wird, ist schon nach den oben genannten Regeln initialisiert worden. In der Transformationsfunktion kann man alle Operationen durchführen, die im C++-Sprachumfang erlaubt sind.

Will man eine Klasse WRAPPER wie in Abschnitt 8.1.5 einführen, bei der gemeinsame Objekte erhalten bleiben, ist man in ObjectStore auf sich selbst gestellt, denn das System bietet nichts vergleichbares.

Bei der Initialisierung der Objekte, die der Anwender ja nicht beinflussen kann, gibt es die Möglichkeit, sich vor der eigentlichen Transformation eine Liste der angewendeten Initialisierungen ausgeben zu lassen. Damit kann der Anwender kontrollieren, ob diese in seinem Sinne sind.

Die Möglichkeit, wie bei OTGen die Durchführung der Datentransformation erst beim Zugriff auf die Daten durchzuführen, fehlt bei ObjectStore. Ebensowenig ist es möglich, automatisch mehrere Transformationen hintereinander durchzuführen, wenn die Daten älter als eine Generation sind.

8.3 Zusammenfassung

Wenn man beide Systeme miteinander vergleicht, sieht man sofort, daß das erste wesentlich angenehmer zu handhaben ist als das zweite. Die Unterstützung des Anwenders bei dem ersten System ist optimal, während sich bei dem zweiten der Anwender um fast alles selbst kümmern muß. ObjectStore kann deswegen auf Dauer wohl nur als Hintergrundsystem für eine intelligentere Umgebung dienen, die als Ausgabe dann C++-Quelltexte hat.

OTGen unterstützt eine beliebige Reklassifizierung der Klassen, während in ObjectStore nur in Unterklassen einer bestehenden Klasse reklassifiziert werden kann. ObjectStore hat dagegen den Vorteil, daß die Transformation über ein normales C++-Programm läuft, man also mehr Möglichkeiten hat, Daten zu verändern.

Ein wesentliches Problem der Evolution ist jedoch bei beiden Ansätzen noch nicht gelöst. Es wird immer nur die Evolution der Datenbanken und der Daten behandelt, jedoch sind die Veränderungen in Routinen, die diese Daten ansprechen, oft wesentlich. Der Programmierer hat sich beim Erstellen der Routinen meist auf eine gewisse Struktur der Klassen verlassen, die nun geändert wurde. Wenn die Routine nach der Änderung syntaktisch noch korrekt ist, wird der Fehler kaum in Erscheinung treten. Um das zu vermeiden, müßte man Invarianten, Vorbedingungen und Nachbedingungen zur jeder Routine angeben. Sobald diese verletzt würden, müßte ein Fehler angezeigt werden.

Kapitel 9

Aktive Datenbanken (*Joachim Feist*)

9.1 Aktive Datenbanken

9.1.1 Motivation

Das klassische Datenbankkonzept ist es, Möglichkeiten zu bieten, Daten dauerhaft abzuspeichern und wieder abzurufen. Sämtliche Aktivität — also das Einbringen oder Auslesen von Daten — geht von Programmen oder vom Nutzer und somit von außerhalb der Datenbasis aus. Für Anwendungen, bei denen es sich um im wesentlichen gleichbleibende Datenbestände handelt, reicht diese Funktionalität auch aus. Es gibt aber Fälle, in denen sich ändernde Daten auch sofort Aktionen nach sich ziehen müssen. Dies ist in folgenden Bereichen vorstellbar: Patientenüberwachung, Börsendaten, Datenbanken in der Industrie beim rechnerintegrierten Engineering. Hier ist es angebracht, daß die Datenbank bei sich ändernden Daten selbst eingreift. Die Datenbank ist dazu deshalb am Besten geeignet, da ihr alle Änderungen bekannt sind. Sie kann datenflußgesteuert die Aktivitäten ausführen.

Bei der Patientenüberwachung müssen bei sich ändernden kritischen Daten Ärzte benachrichtigt werden. Änderungen von Börsenkursen haben An- oder Verkäufe aus Wertpapierdepots zur Folge. In der Fertigunsleittechnik ziehen neue oder modifizierte Aufträge Veränderungen bei der Maschinenbelegung nach sich. Bei Datenbanken mit Anzeigefunktion auf dem Bildschirm, kann durch die aktiven Möglichkeiten auch beim Löschen oder Einbringen von Daten, sofort deren Darstellung auf dem Bildschirm angepaßt werden [DJWaQ94].

Außer in diesen Bereichen, in denen sich zeitabhängige Änderungen ergeben, können aktive Datenbanken auch noch vorteilhaft zur Konsistenzsicherung eingesetzt werden. Bei der Änderung von Daten wird dann überprüft, inwieweit diese komplexeren Bedingungen genügen. Bei einem Verstoß kann dann eine Aktion eingeleitet werden, die die Konsistenz sicherstellt oder die zur Inkonsistenz führende Änderung wird zurückgewiesen und der Benutzer informiert. Ein Beispiel wäre das Überwachen der Bedingung, daß der Ehepartner ein anderes Geschlecht haben muß.

Zur Durchführung von komplexeren Transaktionsmodellen können aktive Regeln auch beitragen. Mit Regeln werden hierbei Abhängigkeiten zwischen einzelnen Transaktionen modelliert. So muß etwa in einem Reisebüro die Transaktion, die den Flug bucht mit der Transaktion, die die Hotelreservierung durchführt, gekoppelt werden. Ein Scheitern der Hotelreservierung zieht eine Kompensationstransaktion nach sich, die die Flugbuchung rückgängig macht.

Im relationalen Modell können aktive Datenbanken auch dazu genutzt werden, Sichten zu materialisieren. Unter einer Sicht versteht man abgeleitete Relationen, die normalerweise bei jeder Anfrage neu ermittelt werden müssen. Speichert man diese abgeleiteten Relationen explizit, hat man das Problem, daß Änderungen an den ursprünglichen Daten zu Inkonsistenzen führen oder die komplette Neuberechnung aller Sichten erfolgen muß. Mittels aktiver Datenbanken ist es nun möglich, dies doch zu verwirklichen: Die Datenbank wird aktiv und sorgt nach Änderungen an den anfänglichen Daten für die notwendigen Änderungen an den abgeleiteten. Näheres zu den zuletzt genannten Aspekten findet sich in [vB94].

Auf diese Weise wäre es auch denkbar, aktive Regeln zum Abgleich verschiedenartiger Datenbanken einzusetzen, um etwa die Lieferdaten des Zulieferers mit den Lagerhaltungsdaten der Unternehmung bei Änderungen sofort anzupassen.

Eine Anwendung im Bereich CSCW (Computer Supported Cooperative Work) zeigt [CKNT93] auf.

9.1.2 Erweiterung des Datenmodells

Regeln

Die obenangeführte zusätzliche Funktionalität wird durch die Einführung von Regeln erreicht. Diese haben vereinfacht das folgende Aussehen:

On <**Ereignis**> if <**Bedingung**> do <**Aktion**>

Die Regel wird dabei dann geprüft, wenn das Ereignis eintritt, die Aktion ausgeführt, wenn die Bedingung wahr ist. Von diesem Konstrukt leitet sich auch der englische Begriff für diese Regeln ab: ECA (Event, Condition, Action) Rules.

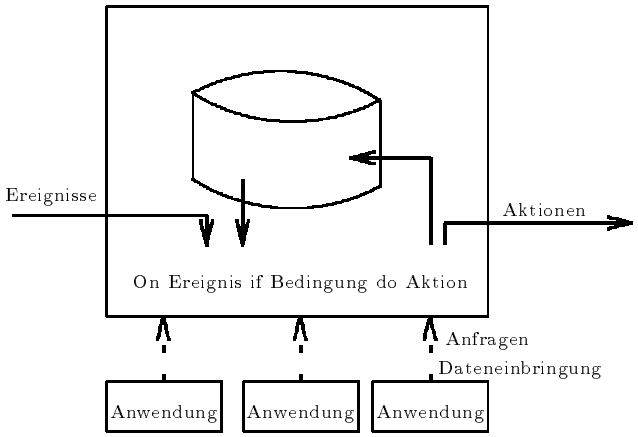


Abbildung 9.1: Aktive Datenbank

Ereignisse sind dabei Änderungen des Datenbankzustandes; aber auch Ereignisse externer Natur, wie z.B. Zeitereignisse, sind denkbar. Dabei unterscheidet man *einfache* und *zusammengesetzte* Ereignisse. Ein einfaches Ereignis wäre die Änderung eines Datenwertes in der Datenbank. Zusammengesetzte Ereignisse ergeben sich durch die Kombination mittels AND, OR und weiteren Operatoren etwa SEQ, der verlangt, daß das erste Ereignis vor dem zweiten eintritt. Die **Bedingung** kann durchaus selbst eine komplizierte Anfrage

sein. Die **Aktion** kann sowohl Änderungen an der Datenbasis durchführen, als auch eine sonstige Funktion haben. Schematisch stellt sich dies wie in Abbildung 9.1 dar.

In der Börsendatenbank wäre ein Ereignis etwa die <Änderung eines Aktienwertes>, die Bedingung <Wenn Wert unter Limit> die Aktion <Kaufe Aktie>. In der Industrie könnte eine Regel „On <Maschinenausfall> if <heute schon der vierte> do <informiere Wartungsabteilung>“ lauten. Als auf höherer Ebene angesiedelte Konsistenzregel wäre denkbar, daß man verhindern will, daß ein Angestellter mehr verdient als sein Chef. Mittels „On <setze_Gehalt_Angestellte> OR <setze_Gehalt_Manager> if <Gehaltsbedingung verletzt> do <Melde Konsistenzverletzung>“ könnte man solche Verstöße erkennen. Hier kam ein zusammengesetztes Ereignis zum Einsatz, da ja sowohl die Erniedrigung des Lohn des Chefs als auch die Erhöhung des Angestelltenlohn zu einer Konsistenzverletzung führen könnte.

Für Regeln gibt es mehrere **Kopplungsarten**. Die Regeln können *unmittelbar* (immediate mode) im Anschluß an das auslösende Ereignis ausgeführt werden. Die zweite Kopplungsart ist *verschoben* (deferred). Hier wird die Aktion zum Ende der Transaktion ausgeführt. Dies ist etwa bei Regeln sinnvoll, die die Konsistenz sicherstellen: erst am Ende einer Transaktion muß Konsistenz gelten. Die dritte Variante erlaubt es Aktionen als *separate* Transaktionen durchzuführen (detached mode).

Ereignisse

Bei Ereignissen konzentrieren wir uns im weiteren auf Datenbankereignisse. Als primitive Ereignisse kommen dabei für jede Operation, die die Datenbank manipulieren, zwei mögliche Ereignisse in Betracht. Das Ereignis *vor* und das Ereignis *nach* Ausführung der Operation. Zum Ereignis dazu gehören auch noch etwaige Parameter — in der Regel eben die in der Datenbasis geänderten Werte. Im objektorientierten Fall entspricht dies den Parametern der das Ereignis auslösenden Methode. Löst z. B. die Methode Setze_SAP ein Ereignis aus, so ist natürlich auch von Interesse, welchen neuen Kurswert die SAP-Aktie bekommen hat. Oder bei der Regel mit der Gehaltserhöhung bzw. -erniedrigung muß natürlich als Parameter des Ereignisses der neue Gehaltswert übermittelt werden. Bei zusammengesetzten Ereignissen stellt sich dann natürlich auch die Frage nach der Speicherung der Parameter, solange noch weitere Ereignisse erwartet werden.

Aktionen

Die Ausführung der Aktion kann wiederum zur Erzeugung von Ereignissen und somit zum Aktivieren von Regeln führen. Hier gibt es Freiheitsgrade in punkto Ausführungsreihenfolge, so kann nach Breitenstrategie zuerst die Aktion der ersten Regel zuendegeführt werden oder nach Tiefenstrategie die durch die Aktion ausgelöste Aktion begonnen werden.

Desweiteren kann es vorkommen, daß durch die eingetretenen Ereignisse mehrere Regeln auf einmal greifen. Auch hier gibt es verschiedene Strategien, um die Abarbeitung zu reglementieren, etwa über Prioritäten der Regeln. In diesen Fällen und in Mehrbenutzerumgebungen kann es vorkommen, daß sich Aktionen gegenseitig behindern, wenn sie auf gleiche Datenelemente zugreifen. Hier sind geeignete Konfliktauflösungsstrategien gefragt.

9.1.3 Architekturvarianten

Um aktive Fähigkeiten anzubieten, gibt es drei Modellierungsmöglichkeiten.

Anwendungsbezogen

Auf diese Weise wurden die geschilderten Problemstellungen bisher gelöst. Es ist dabei Aufgabe der Anwendung, interessierende Änderungen auf den Daten zu erkennen und dann Maßnahmen einzuleiten. Dazu müssen in regelmäßigen Abständen Anfragen an die konventionelle Datenbank gerichtet werden (polling). In Systemen in denen nur wenige Programme die Datenbasis verändern, ist es auch möglich, direkt bei den entsprechenden Änderungen an der Datenbank (also Ereignissen) zusätzlichen Code vorzusehen, der Bedingungen testet und daraufhin Aktionen einleiten kann.

Dieser Ansatz birgt jedoch einige Nachteile: Das ständige Abfragen verursacht zusätzlichen Aufwand oder kann zu langsam sein, so daß Änderungen unbemerkt bleiben oder zu spät reagiert wird. Das Einfügen von speziellen Codesegmenten zur Überprüfung der Regeln verletzt die Modularität, verkompliziert Erweiterungen und erlaubt nicht das schnelle Ändern oder die Wiederverwendung von Regeln.

Zusatzschicht

Beim geschichteten Ansatz wird der passiven Datenbank eine Schicht vorgelagert, die für die Überwachung der Regeln zuständig ist. Bei dieser Schicht werden die Regeln angemeldet. Entweder laufen über diese Schicht auch die Änderungswünsche an die Datenbasis oder mittels regelmäßiger Anfragen wird überprüft, ob die Ereignisse eingetreten sind. Im Gegensatz zum vorherigen Ansatz gibt es nur einen Ort für mehrere Applikationen, an dem die Bedingungen überprüft werden. Je nachdem wie weit diese zusätzliche Schicht auf interne Vorgänge in der Datenbasis Einfluß nehmen kann, können die vollen oder nur eingeschränkte aktive Möglichkeiten angeboten werden.

Trotz dieses Nachteils hat auch dieser Ansatz seine Berechtigung, da er es erlaubt, bisher existierende und im Betrieb befindliche Datenbanken ebenfalls mit aktiven Möglichkeiten auszustatten.

Integriert

Der Ansatz, in dem die aktiven Möglichkeiten mit dem restlichen Datenbanksystem verflochten sind, ist der mächtigste. Dieser erlaubt es am effizientesten, bei maximalen Möglichkeiten aber minimalem Aufwand für den Nutzer, aktive Fähigkeiten anzubieten. Dadurch daß sämtliche Statusänderungen der Datenbank bekannt werden, kann dann reagiert werden, wenn das Ereignis eintritt. Die Aktion kann sofort ausgeführt, andere Datenbankaktivitäten dabei zurückgestellt werden. Im folgenden wird auf diesen Ansatz eingegangen.

9.2 Objektorientierte aktive Datenbanken

9.2.1 Unterschiede zum relationalen Fall

Beim relationalen Modell gibt es nur eine geringe Zahl von primitiven Ereignissen, da nur sehr wenige, für alle Relationen gleiche Operationen (etwa UPDATE, INSERT) vorhanden sind. Beim objektorientierten Modell ist jede Methode einer jeden Klasse ein potentielles Ereignis. Im relationalen Modell gibt es keine Kapselung der Daten. Der Zugriff ist global

möglich, also gelten auch Regeln global. Als weiterer Gesichtspunkt kommt beim objektorientierten Modell die Vererbung hinzu, die sich bei konsistenter Einbettung der aktiven Fähigkeiten auch auf die Regeln erstreckt.

Durch die Komplexität des objektorientierten Umfelds treten als weiterer Gesichtspunkt Geschwindigkeitsoptimierungen in den Vordergrund, um mit der großen Zahl von möglichen Ereignissen, der flexiblen Handhabung der Regeln und der konsistenten Einbindung der neuen Möglichkeiten auch noch vernünftig arbeiten zu können.

9.2.2 Realisierung von Ereignissen

Es stellt sich nun die Frage, wie die Erweiterung des Datenmodells bei der oben angeführten erweiterten Problemstellung gegenüber dem relationalen Modell, am Besten durchführbar ist. Als Modellierungsmöglichkeit für Ereignisse bieten sich drei Varianten.

Ereignisse als Ausdrücke in der Klassendefinition

Bei diesem Ansatz werden die Ereignisse bereits in der Klassendefinition festgelegt. Dies erlaubt dann aber nicht mehr zur Laufzeit Ereignisse ein- oder auszuschalten, zu löschen oder hinzuzufügen. Diese Modellierung birgt also eine gewisse Inkonsistenz zum objektorientierten Ansatz, da Ereignisse nicht wie andere Objekte behandelt werden. Dies zeigt sich auch darin, daß die Persistenz der Ereignisse mit der Existenz von anderen Objekten gekoppelt ist. Die Speicherung von Parametern des Ereignisses ist problematisch, da dem Ereignis kein eigener Speicherplatz zusteht. Ein gravierender Nachteil ist, daß Ereignisse, die mehrere Klassen überspannen, nicht möglich sind. So wäre das zusammengesetzte Ereignis, das es erlauben würde einen Aktienkauf an die Höhe des DAX zu koppeln, in folgender Regel „On <setze_DAX(Wert1)> and <setze_SAP_Aktie(Wert2)> if Wert1>2000 and Wert2<2500 do ...“ nicht modellierbar. Der große Vorteil dieser Modellierung ist, daß die Probleme bei der Ereignissgenerierung schon weitgehend zur Übersetzungszeit gelöst werden können, da sämtliche Stellen an denen Ereignisse generiert werden, bekannt sind. Dieser Ansatz wird unter anderem von Ode [GJ91, GJS92] verfolgt.

Ereignisse als Attribute der Regeln

Beim Erzeugen der Regeln ist hier vorgesehen, daß Ereignisse Teil des Regelobjekts sind. So ist es möglich dynamisch Ereignisse hinzuzufügen, zu löschen oder zu modifizieren. Diese Modellierung bietet auch Konsistenz zum objektorientierten Ansatz, da man Ereignisse durchaus als Bestandteil von Regeln betrachten kann. Die anderen oben aufgeführten Nachteile (Abhängigkeit von anderen Objekten, keine klassenübergreifenden Ereignisse) behalten jedoch ihre Gültigkeit.

Ereignisse als selbständige Objekte

Dies ist die natürlichste Realisierungsart: Denn Ereignisse haben Struktur, einen Zustand und ein Verhalten was sie als Objekte identifiziert. Die Struktur ist dabei durch den Aufbau eines zusammengesetzten Ereignisses aus einfachen Ereignissen gegeben. Der Zustand erklärt sich aus den schon mehrfach erwähnten Parametern, die ein Ereignis mit

sich bringt. Dazu kommen noch weitere Werte (etwa ein Zeitstempel), die für die Einmaligkeit des Objekts sorgen. Als Ereignisverhalten kann man die zwei Zeitpunkte ansehen, zu denen das Ereignis ausgelöst werden kann.

Diese Modellierung bringt sämtliche Vorteile des objektorientierten Ansatzes mit sich: Ereignisse können jetzt wie alle anderen Objekte erzeugt, gelöscht oder modifiziert werden; Vererbung wird möglich. Die Ereignisausdrücke werden mächtiger, da auch klassenüberspannende Ereinisse möglich werden. Nachteilig ist dabei, daß durch die erreichte Dynamik, zur Laufzeit zusätzlicher Aufwand entsteht.

Dieser Ansatz wird von ADAM [DPG91] und Sentinel [AMC93] verfolgt.

9.2.3 Realisierung zusammengesetzter Ereignisse

Die Modellierung von zusammengesetzten Ereignissen ist nicht trivial. Je nach dem welche Funktionalität man anbieten will, ergeben sich Schwierigkeiten bei der Zwischenspeicherung der Parameter und der Aufrechterhaltung der Historie von Ereignissen zu früheren Zeitpunkten. So unterscheidet man auch verschiedene **Parameterkontexte**. Hierbei geht es um die Problematik, was mit einem Ereignis geschieht, das zweimal hintereinander eintrifft, bevor es weiterverarbeitet wird. Man kann entweder sämtliche Auftreten und deren Parameter speichern (*kumulativ*) oder vereinbaren, daß nur die Parameter des neuesten Ereignisses gespeichert werden (*zuletzt*). Desweiteren gibt es noch die Kontexte *kontinuierlich* und *chronologisch*; deren Semantik findet sich z.B. in [vB94].

Bei der Einstufung welcher Parameterkontext sinnvoll ist, kommt es sehr darauf an, um was für eine Art von zusammengesetztem Ereignis es sich handelt. Ist das Primitivereignis ein Maschinenausfall, so sollte gewiß jedes Ereignis gespeichert werden, handelt es sich um eine zweite Erhöhung eines Aktienwertes, so genügt es das letzte Ereignis mit dem aktuellen Wert zu speichern.

Endlicher Automat

Hier lassen sich zusammengesetzte Ereignisse mit regulären Ausdrücken beschreiben. Ein eingehendes Ereignis verursacht im Automaten einen Zustandsübergang. Das zusammengesetze Ereignis wird erkannt und die Aktion ausgeführt, sobald der Automat in einen akzeptierenden Zustand übergegangen ist. Um auch die Parameter der Ereignisse mitführen zu können, wird der endliche Automat um Mengen ergänzt, die diese speichern. Dieser Ansatz wird von Ode verfolgt.

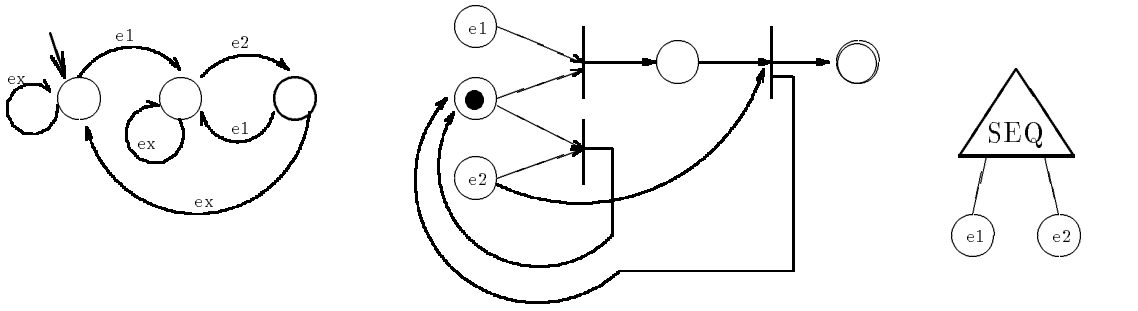
Petrinetz

Bestimmte Teil-Petrinetze modellieren hier die möglichen Verküpfungen wie ODER und SEQ. Es gibt Eingangsknoten, die dann gesetzt werden, wenn das zugehörige Ereignis eingetreten ist. Nach der üblichen Semantik werden die Markierungen dann durch das Petri-Netz geschaltet. Feuert die Ausgangstransaktion, so wird die Aktion ausgelöst. SAMOS [GD93] verwendet Petri-Netze.

Operatorbaum

Beim Operatorbaum unterscheidet man verschiedene Knotentypen, die die Operatoren darstellen. So wären UND, ODER, SEQ denkbare Operatoren. Sind die Söhne eines Ope-

rators entsprechend seiner Spezifikation besetzt, so wird das Ereignis nach oben im Baum weitergemeldet. Diese Vorgehensweise wurde in Sentinel verwendet.



a) endlicher Automat b) Petrinetz c) Operatorbaum

Abbildung 9.2: Modellierungsvarianten von zusammengesetzten Ereignissen

Abbildung 9.2 zeigt die drei verschiedenen Modellierungen mit dem zusammengestzten Ereignis e1 SEQ e2, das dann eintritt, wenn e1 vor e2 auftritt.

9.2.4 Realisierung von Regeln

Auch für Regeln haben sich verschiedene Modellierungsalternativen herausgebildet. Wie bei den Ereignissen ist das Hauptunterscheidungskriterium, ob die Regeln zu bestehenden Klassen und Objekten hinzugenommen oder als eigenständige Objekte realisiert werden.

Deklaration innerhalb der Klasse

Regeln, die bereits zur Übersetzungszeit bekannt sind, können so umgesetzt werden, daß deren Inhalt an jede Stelle eingefügt wird, an der sie potentiell ausgelöst werden können. Regeln werden hier als Teil der Klasse gesehen, über deren Methoden sie Aussagen machen.

Der Vorteil dieses Ansatzes liegt darin, daß bereits zur Übersetzungszeit die Modellierung durch das Einfügen der entsprechenden Anweisungen an der richtigen Stelle im erzeugten Code erledigt wird. Dieser Ansatz unterstützt auch schon die Vererbung von Regeln. Auch die Nachteile sind von der entsprechenden Modellierung von Ereignissen bekannt: Die Regeln werden hier nicht als Objekte behandelt, ihre Existenz hängt an der Existenz einer Klasse. Zur Laufzeit lassen sich an der so vorgezeichneten Regelmenge kaum Änderungen durchführen. Wollte man nachträglich als zusätzliche Eigenschaft von Regeln Prioritäten für deren Abarbeitungsreihenfolge einführen, müßten sämtliche Klassen neu übersetzt werden. Dies wird auch notwendig, wenn man neue Regeln hinzufügen, alte ändern oder löschen will.

Zusammengesetzte Regeln, wie zum Beispiel die Regel die sicherstellt, daß Angestellte weniger verdienen als ihre Chefs, müssen dann auf beide Klassen (hier die Angestellten- und die Managerklasse) aufgeteilt werden, da sowohl eine Erhöhung in der einen, als auch eine Erniedrigung in der anderen Klasse zur Konsistenzverletzung führen könnte.

Regeln als Datenteil anderer Objekte

Man kann Regeln auch als Typ (etwa eine verkette Liste von Regeln mit den gewünschten Komponenten: Ereignis, Bedingung, Aktion, Priorität, ...) modellieren, der dann zur normalen Klassendefinition der bestehenden Objekte als Datenteil hinzukommt. Dessen Struktur läßt sich auch an andere Klassen weitervererben. Ererbt wird aber nur die Struktur des Regelaufbaus, nicht aber deren Inhalte, so daß ein Vererben von Regeln nicht möglich ist. Auch hier ist die Existenz der Regel an die Existenz der Klasse gebunden.

Vorteilhaft ist, daß man Änderungen an der Struktur der Regeln einfach ausführen kann, indem man den Typ — etwa um das Feld Priorität — erweitert. Problemlos ist das Hinzufügen von Regeln zu einzelnen Objekten, das Ändern oder Löschen von Regeln.

Regeln als eigenständige Objekte

Dieser Ansatz bietet wieder am meisten Freiheiten. Der Objektstatus erlaubt das Erzeugen, Ändern und Löschen von Regeln zur Laufzeit. Die Erweiterbarkeit der Regeln ist gewährleistet, da Änderungen nur ein erneutes Übersetzen der Regelklasse erfordern. In der Datenbank können Regeln wie alle anderen Objekte auch gespeichert und somit dauerhaft gemacht werden. Der Wiederverwendung von Regeln steht nichts im Wege.

Wie wichtig gerade die Möglichkeit zur Erzeugung von neuen Regeln oder Änderungsmöglichkeiten von bestehenden Regeln zur Laufzeit ist, wird klar, sobald man sich bei den Beispielen bewußt macht, daß viele Regeln eben nicht schon bei der Erzeugung z.B. der Aktien- bzw. Patientenklassen bekannt sind. So ist bei den Aktienregeln ständiges Anpassen der Regeln notwendig, da neue Kaufregeln aufgestellt werden. Im Verlauf einer Patientenüberwachung kann sich der Name des Arztes ändern, der bei kritischen Vorgängen informiert werden soll. Bei der Überwachung von Produktionsprozessen kann das Management frühzeitigeres Eingreifen der Wartung bei vermehrten Maschinenausfällen beantragen.

Regeln sollten sowohl auf Klassenebene als auch auf Instanzenebene definierbar sein. Auf Klassenebene sind Konsistenzregeln denkbar, die gemeinsame Randbedingungen für alle Objekte überwachen. Regeln müssen aber auch auf Instanzenebene zur Verfügung gestellt werden. So gibt es bei der Klasse Depot in der Börsendatenbank für jedes Objekt (z.B. das Depot von Herrn Meier und das von Herrn Müller) verschiedene Kauf- und Verkaufsregeln.

9.2.5 Übersicht

Die Tabelle zeigt eine Übersicht von realisierten Forschungssystemen, die unterschieden werden nach Implementierungsentscheidungen. Laut [EN89] finden sich auch bereits in den meisten kommerziellen Datenbanken aktive Ansätze, allerdings von eingeschränktem Umfang.

9.3 Fallbeispiel Sentinel

9.3.1 Umfeld

Die neueste Version von Sentinel basiert auf dem Open OODB Toolkit, einer passiven objektorientierten Datenbank deren Implementierung zur Verfügung steht. Dieses Toolkit

	Ereignis-überwachung	Ereignis-reichweite	Erkennung zus. Ereignisse	Ereignisse als Objekte	Regel-definition	Regeln als Objekte
Ode	intern	innerhalb einer Klasse	Endlicher Automat	Nein	auf Klassen, auf Instanzen	Nein
ADAM	intern	innerhalb einer Klasse	–	Ja	auf Klassen	Ja
Sentinel	intern u. extern	innerhalb und zw. Klassen	Operator-graph	Ja	auf Klassen, auf Instanzen	Ja

Tabelle 9.1: Vergleich aktiver objektorientierter Datenbanken nach Implementierungsgesichtspunkten

verwendet EXODUS als Speichermanager, der die Persistenz der Datenobjekte sicherstellt und Anfragen parallel bearbeiten kann. Anwendungen für diese Datenbank werden in C++ formuliert. Die Umsetzung der notwendigen Erweiterungen geschieht zum Teil mittels eines C++-Vorübersetzers (siehe auch beim Beispiel). Die bestehenden Klassen der Datenbank wurden erweitert.

9.3.2 Ereignisse

Ereignisse werden als eigenständige Objekte modelliert. Um auch Laufzeitanforderungen gerecht zu werden, wird zum Zeitpunkt der Klassendefinition bereits festgelegt, welche Methoden zu welchem Zeitpunkt (vor oder nach Ausführung der Methode) einfache Ereignisse erzeugen. Diese Festlegung bildet die **Ereignisschnittstelle** der Klasse, sie macht ihr Verhalten auf Ereignisebene nach außen transparent. In Abbildung 9.3 ist dies durch die zusätzliche Leiste, die angibt, ob vor oder nach Methodenausführung ein Ereignis erzeugt wird, angedeutet. Hier werden nur einfache Ereignisse spezifiziert, da sich nur diese eindeutig einer Klasse zuordnen lassen. Jede Klasse, deren Instanzen auf diese Weise als Ereignisgeneratoren tätig werden, wird als *reaktiv* bezeichnet. Die Modellierung realisiert dies als Oberklasse, von allen ereigniserzeugenden Klassen. Die Klasse hat sich auf diese Weise zum Versender von Ereignissen gemacht. Über den lokalen Ereigniserkenner (s.u.) gelangt ein hier erzeugtes Ereignis an das zugehörige Ereignisobjekt. Dieses hat eine Liste *Konsumenten* in der sich die davon abhängigen Operatoren und Regeln mittels der Methode *Abonnieren* eintragen. Aus diesem Grunde müssen einfache Ereignisse, zusammengesetzte Ereignisse und Regeln eine gemeinsame Oberklasse haben, um in der gleichen Liste stehen zu können. Diese gemeinsame Oberklasse ist die Klasse *Ereignis*.

9.3.3 Zusammengesetzte Ereignisse

Die Modellierung geschieht mit Hilfe der in Abschnitt 9.2.3 eingeführten Operatorbäume. Die Struktur dieser Operatorbäume wird dabei durch die im vorherigen Abschnitt erklärten Konsumentenlisten erreicht. Ein zusammengesetztes Ereignis abonniert die Teilereignisse, die es zur Entscheidung braucht. Einem Blatt eines solchen Baumes entspricht ein einfaches oder aber ein externes Ereignis (von einer anderen Applikation, siehe auch

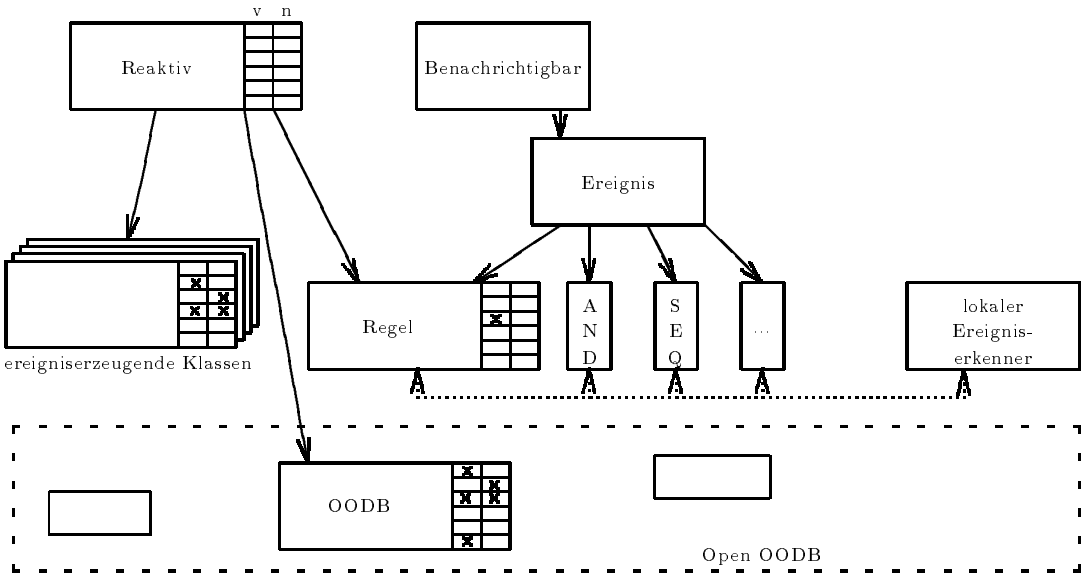


Abbildung 9.3: Erweiterte Klassenhierarchie

globaler Ereigniserkenner). Wie einfache Ereignisse, haben auch die zusammengesetzten eine Liste, in der Abonnenten stehen. Wurzel eines solchen Baumes ist dabei immer eine Regel oder der Verweis, daß das Ereignis für eine Regel gebraucht wird, die mehrere Anwendungen überwacht.

9.3.4 Lokaler Ereigniserkenner

Jede Regel hat unter sich also einen Baum der zur Auslösung benötigten Ereignissen. Nun kommt es aber häufig vor, daß Teilausdrücke von mehreren Regeln gleichzeitig überwacht werden. Dies läßt sich dadurch ausnutzen, daß man alle Operatorbäume zu einem Graphen zusammenfaßt, in dem gleiche Ereignisse in einem Knoten realisiert sind. Diese Graphenstruktur ist schon durch die jeweiligen Einträge in den Abonnentenlisten der einzelnen Ereignisse gegeben. Ein solcher Operatorgraph repräsentiert dann sämtliche interessierenden Ereignisse und die daran gekoppelten Regeln für eine Anwendung. Verwaltet wird dieser Graph von einem lokalen Ereigniserkenner, der die Benachrichtigungen über einfache Ereignisse erhält und dann entscheidet, welche anderen Knoten daraufhin benachrichtigt werden müssen, je nach Semantik des Knotens. Die Verantwortung des lokalen Ereigniserkenners erstreckt sich auch auf die Ausführung der Regeln, die im nächsten Abschnitt beschrieben wird. Auf diese Weise wird mit geringstem Aufwand das Weiterschalten der Ereignisse erreicht. Die Vorgehensweise erinnert dabei an Datenflußgraphen: Aktionen werden nur eingeleitet, wenn sie auch notwendig, d.h. die nötigen Daten vorhanden sind.

Die Parameter werden dabei lokal bei den jeweiligen Knoten in Form von Listen gehalten. Dadurch wird kein Kopieren von Parametern notwendig, es muß nur jeweils die Referenz auf die richtigen Parameter gesetzt werden.

Jeder Knoten weiß auch in welchem Parameterkontext (siehe Abschnitt 9.2.3) er die Parameter sammeln muß. Dabei wurde auch Wert auf große Flexibilität gelegt, ist es doch möglich für verschiedene Regeln verschiedene Kontexte zu vereinbaren. Jeder Knoten hat

für jede mögliche Art der Parameterspeicherung einen Zähler, der ihm mitteilt, ob auf diese Art die Werte gesammelt werden müssen.

Ereignisse werden normalerweise nach Vollendung einer Transaktion gelöscht. Ein „Säubern“ des Ereignisgraphen wird auch notwendig, wenn die Transaktion zurückgesetzt wird. Dies wird konsistent mit dem Regelmechanismus so gelöst, daß es eine Regel gibt, die beim Ereignis „Transaktionsende“ oder „Abbruch“ die Löschung der Ereignisse übernimmt. Durch diese Vorgehensweise ist es auch möglich, falls benötigt, auch Ereignisse zu erlauben, die das Transaktionsende überdauern, in dem einfach die entsprechende Regel deaktiviert wird.

9.3.5 Regeln

Auch Regeln werden als eigene Objekte realisiert. Sie werden als *benachrichtigbar* (notifiable) bezeichnet. Dies findet sich ebenfalls in der Modellierung als eigene Klasse wieder, von der sämtliche Regeln abgeleitet sind. Unterhalb der Klasse *reaktiv* haben wir also die Produzenten, unterhalb der Klasse *benachrichtigbar* die Konsumenten von Ereignissen untergebracht. Regeln sind auch reaktiv, da sie selbst wieder Ereignisse generieren können.

Für die Bedingung und die Aktion wird dabei jeweils die Adresse einer Funktion angegeben. Eingestellt werden kann auch der Parameterkontext, die Kopplungsart und die Priorität. Da Regeln zur Laufzeit an- und ausgeschaltet werden können, erlaubt ein weiteres Attribut anzugeben, ob bei einer Regel, die gerade aktiviert wurde, auch Ereignisse verwendet werden sollen, die vor dem Aktivierungszeitpunkt lagen. Die Formulierung von Regeln findet sich im abschließenden Beispiel.

Realisierung der Kopplungsarten

Wie in Kapitel 9.1.2 erklärt, können Regeln in verschiedenen Kopplungsarten realisiert werden. Bei Kopplungsart *unmittelbar* wird, sobald die Regel aufgrund der eingetretenen Ereignisse ausgelöst wird, für die Bedingung und die Aktion ein Prozeß erzeugt. Dies geschieht gleichzeitig für alle Regeln, die in diesem Moment ausgelöst wurden. Bei den Prozessen handelt es sich dabei um „threads“, die vom lokalen Ereigniserkenner nach der Priorität der Regeln gestartet werden.

Die Modellierung der Regeln, die in Kopplungsart *verschoben* ausgeführt werden sollen, geschieht durch Rückführung auf den eben diskutierten Fall. Dazu wird zu jeder solchen Regel als zusätzliches Ereignis das Commit-Ereignis erwartet. Die Ereignisse, die die Regel eigentlich ausmachen, werden also noch mit einem weiteren komplexen Ereignis verknüpft, das ein Commit-Ereignis erwartet, was durch eine zusätzliche UND Verknüpfung erreicht wird (im Beispiel in Abb.9.5 als gestrichelte Linie abgehoben). Auf diese Weise werden solche Regeln am Ende der Transaktion in Kopplungsart *unmittelbar* ausgeführt. Nötig ist dazu, daß auch Datenbankaktionen wie `Begin_Transaction` und `End_Transaction` zur Generierung eines Ereignisses führen. Die entsprechende Klasse der Datenbank ist also auch Unterklasse von *reaktiv* (siehe Abbildung 9.3).

Die Kopplungsart *separat* wurde noch nicht implementiert, die Einbindung aber schon eingeplant: Die Aktion einer solchen Regel soll als eigene Anwendung gestartet werden, deren Anweisungen dann die Regelausführung als separate Transaktion bewerkstelligen. Soll der Zusammenhang zur auslösenden Transaktion erhalten bleiben (also ein Abbruch

der ausgelösten Transaktion auch die Ursprungstransaktion abbrechen), so sind Regeln und Ereignisse notwendig, die anwendungsübergreifend tätig werden.

9.3.6 Globaler Ereigniserkenner

Notwendig ist ein Ereigniserkenner, der nicht nur Ereignisse und Regeln verwaltet, die eine Anwendung betreffen. Dieser globale Ereigniserkenner bekommt Teilereignisse bereits von den Ereigniserkennern der einzelnen Applikationen zugeschickt und verwaltet diese ebenfalls in einem Operatorgraphen. Problematisch ist dabei der Austausch zwischen den verschiedenen Applikationen, da kein gemeinsamer Adreßraum zur Verfügung steht. Dieser Teil ist noch nicht implementiert.

9.3.7 Geschachtelte Transaktionen

Um Regelabarbeitung von mehreren aktivierten Regeln parallel zu erlauben, wird das gewöhnliche flache Transaktionsmodell erweitert. Dies geschieht durch Einführung von geschachtelten Transaktionen, einer Hierarchie von Transaktionen. Diese entsteht dadurch, daß jede Aktivierung einer Regel eine Untertransaktion zur aktuellen Transaktion startet. Falls es in dieser Transaktion wiederum zu einer Regelausführung kommt, pflanzt sich die Bildung einer Subtransaktion auf der nächsten Ebene fort. Abbildung 9.4 zeigt eine Haupttranskation, bei der zu einem Zeitpunkt zwei Regeln gleichzeitig ausgelöst werden. Die zweite Regelausführung führt wieder zu einer Subtransaktion.

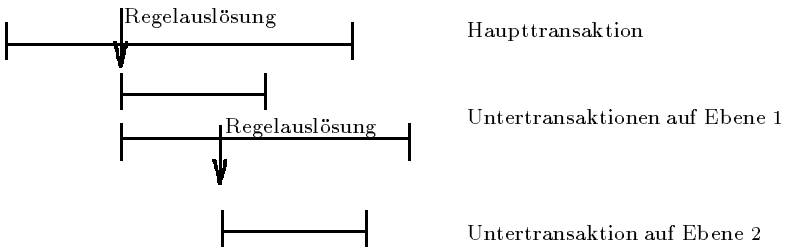


Abbildung 9.4: Geschachtelte Transaktionen

Für jede Subtransaktion gelten dabei alle Bedingungen einer herkömmlichen Transaktion, außer der der Dauerhaftigkeit, denn eine Subtransaktion ist davon abhängig, ob die übergeordnete Transaktion erfolgreich beendet werden konnte oder nicht. In unserem Fall heißt dies: Wird die Transaktion, die zur Ausführung unserer Regel und deren Aktion führte, abgebrochen und zurückgesetzt, so müssen auch die Änderungen der Regelaktion wieder zurückgesetzt werden. Eine ausführliche Behandlung von Transaktionskonzepten findet sich in [vB94].

Bei der Implementierung dieses Transaktionskonzepts gab es Einschränkungen aufgrund der Datenbank und des Speichermanagers, da diese das verschachtelte Konzept nicht unterstützen und die zur Auflösung von Schreib- und Lesekonflikten notwendigen Sperren vom Speichermanager selbsttätig gesetzt werden und nicht als Schnittstellenoperationen zur Verfügung stehen. Die angewandte Implementierungsstrategie findet sich in [CKTB94].

9.3.8 Beispiel

Um die obigen Mechanismen besser zu verstehen, wird im folgenden ein Beispiel in Anlehnung an [CKTB94] durchgegangen. Die dabei aufgestellten Regeln haben keine besondere Bedeutung, sondern dienen nur dazu, die verschiedenen Möglichkeiten aufzuzeigen.

Ereignisschnittstelle und Klassenregeln

Die Klasse Wertpapier besteht aus drei Methoden: `verkaufe_aktie`, `setze_preis` und `hole_preis`. Die ersten beiden werden in der Klassenschnittstelle zu ereigniserzeugenden Methoden deklariert. Das Ereignis e1 soll nach Ausführung von `verkaufe_aktie` erzeugt werden. Das Ereignis e2 wird vor, e3 nach der Ausführung einer Preisänderung generiert. Als klassenweites zusammengesetztes Ereignis fungiert e4, das als Und-Verknüpfung von e1 und e2 definiert ist. Regel R1 bedeutet dementsprechend „On <nach:verkaufe_aktie> and <vor:setze_Preis> if <Bedingung1> do <Aktion1>“. Diese Regel wird erst am Ende einer Transaktion ausgeführt, alle auftretenden Ereignisse werden gespeichert.

```
class WERTPAPIER : public REAKTIV
{
   private: ...
   public: ...
     /* verkaufe_aktie generiert Ereignis nach Methodenabarbeitung */
     event nach(e1) INT verkaufe_aktie (INT anzahl);
     /* setze_preis generiert Ereignis vor und nach Methodenabarbeitung */
     event vor(e2) && nach (e3) VOID setze_preis (FLOAT wert);
     /* hole_preis generiert kein Ereignis */
     INT hole_preis ();
     /* Ereignis vier ist ein zusammengesetztes Ereignis */
     event e4 = e1 ^ e2;  /* UND-Operator */
     /* On e4 if bedingung1 do aktion1 */
     rule R1 [e4,bedingung1,aktion1,KUMULATIV,VERSCHOBEN]; /* Klassenregel */
}
```

Dies wird nun vorverarbeitet. Dazu werden die ursprünglichen Routinen umbenannt und an ihrer statt kommen Anweisungen, die die Speicherung der Parameter und das Verschicken der Ereignisse zum richtigen Zeitpunkt sicherstellen.

```
class WERTPAPIER: public REAKTIV
{
   private: ...
     /* alte Methoden umbenannt */
     INT alt_verkaufe_aktie (INT anzahl);
     VOID alt_setze_preis(FLOAT wert);
   public: ...
     INT verkaufe_aktie (INT anzahl);
     VOID setze_preis (FLOAT wert);
     INT hole_preis ();
}

INT WERTPAPIER::verkaufe_aktie (INT anzahl)
{
   INT ret_value
   /* Parameter in einer Liste speichern */
   PARA_LISTE *verkaufe_aktie_liste = new PARA_LISTE ();
```

```
   verkaufe_aktien_liste-> insert ("anzahl",INT,anzahl);
   /* rufe eigentliche Routine auf */
   ret_value = alt_verkaufe_aktie(anzahl);
   /* Schicke an lokalen Ereigniserkenner das Endeereignis */
   Benachrichtige(this,"AKTIE","INT verkaufe_aktie(INT anzahl)",
                  "NACH",verkaufe_aktie_liste);
}

VOID WERTPAPIER::setze_Preis(FLOAT Preis)
{
   PARA_LISTE *setze_preis_liste = new PARA_LISTE ();
   setze_Preis_Liste-> insert ("preis",FLOAT,preis);
   /* Verschicke Ereignis vor Methodenabarbeitung
   Benachrichtige (this,"AKTIE","VOID setze_preis(FLOAT preis)",
                   "VOR",setze_preis_liste);
   alt_setze_Preis(Preis); /* eigentliche Routine */
   Benachrichtige (this,"AKTIE","VOID setze_preis(FLOAT preis)",
                   "NACH",setze_preis_liste);
}
```

Laufzeitereignisse und -regeln

Soweit wurde lediglich die Ereignisschnittstelle der Klasse umgesetzt. Wie werden Ereignisse, die zur Laufzeit generiert werden, behandelt? Betrachten wir die weiteren Definitionen in folgendem Hauptprogramm:

```
STOCK SAP, Siemens, Apple;
main()
{
   ...
   /* Primitivereignis, das nur von Objekt SAP erzeugt wird */
   event e5 ("e5",SAP,"VOR","VOID setze_preis(FLOAT Preis);
   event seq_Ereignis = AKTIE_e4 >> e5; /* SEQUENZ Operator */

   /* Erzeugt Regel, die sowohl ein Klassen-
      als auch ein Instanzenereignis beinhaltet */
   rule R2[seq_event, cond2, action2, SOFORT]
   ...
}
```

Dies wird so vorverarbeitet, daß zuerst für die neue Anwendung ein eigener Ereigniserkenner erzeugt wird. Die deklarierten Ereignisse werden in Ereignisobjekte umgewandelt, die Regeln werden zu einem Regelobjekt. Es fällt auf, daß auch die Ereignisse und Regeln, die auf Klassenebene AKTIE definiert wurden, hier in entsprechende Objekte umgewandelt werden. Es werden also alle Arten von Regeln und Ereignissen, ob zu Laufzeit auf Instanzen definiert oder vorher auf ganze Klassen, gleich behandelt.

```
main()
{
   ...
   /* Erzeuge lokalen Ereignisgenerator */
   Ereignis_detektor = new LOKALER_EREIGNIS_ERKENNER();

   /* Erzeuge einfache Ereignisse
```

```
   EVENT *AKTIE_e1 = new PRIMITIV("AKTIE_e1","AKTIE","NACH",
                                  "INT verkaufe_aktie(INT Anzahl)");
   EVENT *AKTIE_e2 = new PRIMITIV("AKTIE_e2","AKTIE","VOR",
                                  "VOID setze_Preis(FLOAT Preis)");
   EVENT *AKTIE_e3 = new PRIMITIV("AKTIE_e3","AKTIE","NACH",
                                  "VOID setze_Preis(FLOAT Preis)");
   EVENT *AKTIE_e4 = new AND ("AKTIE_e1","AKTIE_e2",);/* zusammengesetzt */
   /* Regel R1: Klassenregel */
   RULE *R1 = new RULE("R1",AKTIE_e4,Bedingung1,Aktion1,KUMULATIV];
   R1->setze_Kopplungsart(VERSCHOBEN);
   /* Instanzen-Ereignis */
   EVENT *e5 = new PRIMITIV("e5",SAP,"NACH","VOID setze_Preis(FLOAT Preis)");
   /* zusammengesetztes Ereignis mit SEQUENZ Operator /*
   EVENT *seq_ereignis = new SEQ(AKTIE_e4, e5);
   /* Regel R2: */
   RULE *R2 = new RULE("R2",seq_ereignis,Bedingung2,Aktion2,ZULETZT];
}
```

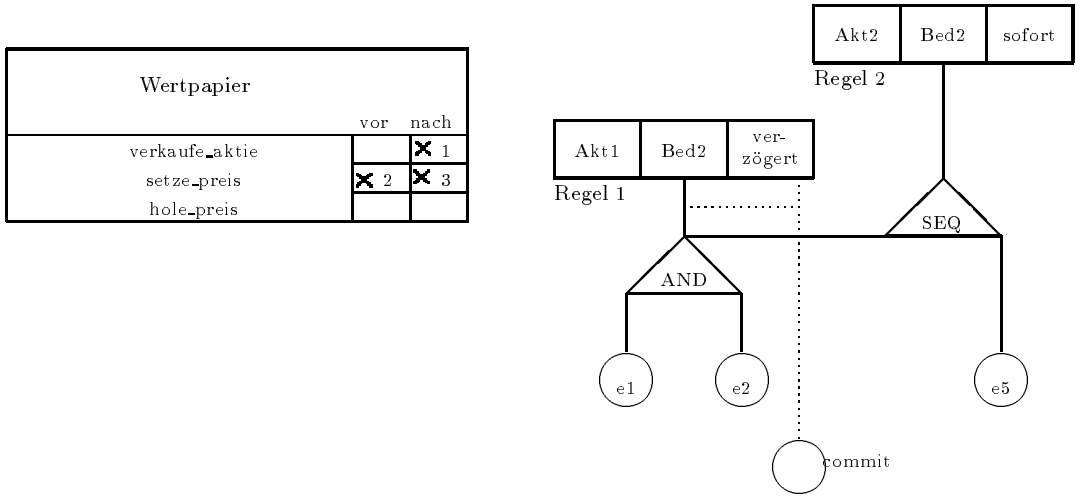


Abbildung 9.5: Klasse und Operatorgraph des Beispiels

Abbildung 9.5 zeigt die so erzeugte Klasse samt Ereignisschnittstelle und die Struktur des Operatorgraphen.

Beispieltransaktion

Betrachten wir nun folgenden Anweisungsteil unserer Anwendung. Es wird eine Transaktion gestartet und darin werden bestimmte Änderungen in die Datenbank eingebracht.

```
   OpenOOBD-> beginTransaction();
      SAP.setze_preis(2500);
      Apple.setze_preis(64);
      Siemens.verkaufe_aktie(200);
      Apple.hole_preis();
      SAP.setze_preis(2520);
   OpenOODB->commit();
```

Abbildung 9.6 zeigt genau wann die einzelnen Ereignisse generiert werden. Betrachtet man die Folge, in der die Regeln aktiv werden, so kommt zuerst Regel zwei zum Zuge, da sie standardmäßig in der Kopplungsart *unmittelbar* ausgeführt wird. Da die Regel insgesamt aus drei Ereignissen besteht, fallen drei Parametertupel an: {Apple,Preis,FLOAT,64}, {Siemens,Anzahl,INT,200}, {SAP,Preis,FLOAT,2520}. Wie man sieht löst erst die letzte Anweisung die Regel aus. Regel R1 wird am Ende der Transaktion ausgeführt, da sie *verschoben* ausgeführt wird. Als Parameter fallen an {SAP,Preis,FLOAT,2500}, {Apple,Preis,FLOAT,64}, {Siemens,Anzahl,INT,200}. Dabei ist das Parameterpaar SAP und Apple vorhanden, weil als Parameterkontext *kumulativ* gewählt wurde.

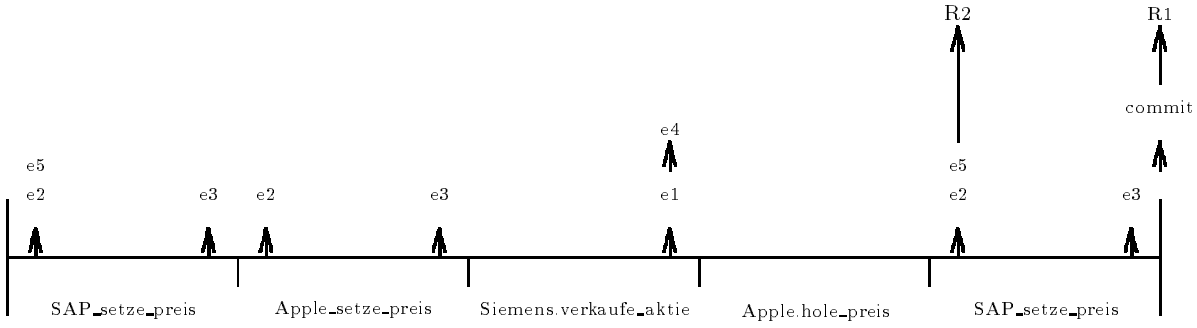


Abbildung 9.6: Auftreten der Ereignisse und Regelauslösungen

Literaturverzeichnis

[ABD+89] M. Atkinson, F. Bancilhon, D. J. DeWitt, K. R. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In *Proc. Int. Conf on Deductive and Object-Oriented Databases*, pages 40–57, Kyoto, Japan, Dec 1989.

[AMC93] E. Anwar, L. Maugis, and S. Chakravarthy. A new perspective on rule support for object-oriented databases. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 99–108, 1993.

[Bil92] A. Biliris. The performance of three database storage structures for managing large objects. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 276–287, San Diego, CA, June 1992.

[CDMB89] R. C. H. Connor, A. Dearle, R. Morrison, and A. L. Brown. An object addressing mechanism for statically typed languages with multiple inheritance. In *Proc. of the ACM Conf on Object-Oriented Programming Systems and Languages (OOPSLA)*, pages 279–285, 1989.

[CDN93] M. Carey, D. DeWitt, and J. Naughton. The OO7 benchmark. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, page ???, Washington, Mai 1993.

[CDRS86] M. J. Carey, D. J. DeWitt, J. E. Richardson, and E. J. Shekita. Object and file management in the EXODUS extensible database system. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 91–100, Kyoto. Japan, Aug 1986.

[CK85] G. Copeland and S. Khoshafian. A decomposition storage model. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 268–279, Austin, TX, May 1985.

[CKNT93] S. Charkravarthy, K. Karlapalem, S. B. Navathe, and A. Tanaka. Database supported cooperative problem solving. *International Journal of Intelligent and Cooperative Systems*, 2(3):249–287, 1993.

[CKTB94] S. Chakravarthy, V. Krishnaprasad, Z. Tamizuddin, and R. H. Badani. Eca rule integration into an oodbms: Architecture and implementation. Technical report, University of Florida, Feb. 1994.

[CRR91] P.K. Chrysanthis, S. Raghuram, and K. Ramamritham. Extracting concurrency from objects: A methodology. In J. Clifford and R. King, editors,

Proc. of the ACM SIGMOD Intl. Conf. on Management of Data, pages 108–117, Denver, Colorado, May 1991.

[DJWaQ94] O. Diaz, A. Jaime, N. W.Paton, and G. al Qaimari. Supporting dynamic displays using active rules. *SIGMOD RECORD*, 23(1):21–26, Mar 1994.

[DPG91] O. Diaz, N. Paton, and P. Gray. Rule management in object oriented databases: A uniform approach. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, Barcelona, Spain, 1991.

[EN89] E. Elmasri and S. Navathe. *Fundamentals of Database Systems.* Benjamin Cummings, Redwood City, CA, 1989.

[GD93] S. Gatziu and K. R. Dittrich. Events in an active object-oriented database system. In *Proc. of 1st Intl. Workshop on Rules in Database Systems*, Edinburgh, UK, 1993.

[GJ91] N. H. Gehani and H. V. Jagadish. Ode as an active database: Constraints and triggers. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, 1991.

[GJS92] N.H. Gehani, H.V. Jagadish, and O. Shmueli. Event specification in an active object-oriented database. In M. Stonebraker, editor, *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 81–90, San Diego, California, June 1992.

[HH91] T. Hadzilacos and V. Hadzilacos. Transaction synchronisation in object bases. *Journal of Computer and System Sciences*, 43(1):2–24, Aug. 1991.

[HW91] M. P. Herlihy and W. E. Weihl. Hybrid concurrency control for abstract data types. *Journal of Computer and System Sciences*, 43(1):25–61, Aug 1991.

[KKM+92] A. Kemper, C. Kilger, G. Moerkotte, H.-D. Walter, and A. Zachmann. Das GOM-Handbuch — Objektorientierte Programmierung und Datenmodellierung. Interner Bericht 25/92, Universität Karlsruhe, Fakultät für Informatik, Dec 1992.

[LH90] B. Staudt Lerner and A. Nico Habermann. Beyond schema evolution to database reorganization. In *Proc. of the ACM Conf on Object-Oriented Programming Systems and Languages (OOPSLA)*, pages 67–76, Oct. 1990.

[LL89] T. Lehman and B. Lindsay. The Starburst long field manager. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 375–383, Amsterdam, NL, Aug 1989.

[LLOW91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The Objectstore database system. *Communications of the ACM*, 34(10):50–63, Oct 1991.

[Mey92] B. Meyer. *Eiffel - The language.* Prentice Hall, 1992.

[Mos90] J. Eliot B. Moss. Working with persistent objects: To swizzle or not to swizzle. Technical Report 90-38, COINS Object-Oriented Systems Laboratory, University of Massachusetts at Amherst, May 1990.

[Obj90] Object Design. ObjectStore User Guide, Release 1.0. Technical report, Object Design, Inc., Burlington, MA, Oct 1990.

[Obj93] Object Design, Inc. *ObjectStore User Guide, Chapter 9 Schema Evolution*, 1993. Release 2.0.

[SKG88] B. Staudt, C. Krueger, and D. Garlan. Automating the maintenance of structure-oriented environments. Technical Report Technical Report CMU-CS-88-186, Carnegie Mellon University, 1988.

[SS84] P. M. Schwarz and A. Z. Spector. Synchronizing abstract types. *ACM Trans. Computer Systems*, 2(3):223–250, 1984.

[Str91] B. Stroustrup. *The C++ Programming Language*. Addison Wesley, Reading, MA, 2nd edition, 1991.

[vB94] G. v. Bültzingsloewen. Aktive Datenbanken im rechnerintegrierten Engineering. Folienkopien zur Vorlesung, Universität Karlsruhe, SS 1994.

[VKC86] P. Valduriez, S. Khoshafian, and G. Copeland. Implementation techniques of complex objects. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 101–110, Kyoto, Japan, Aug 1986.

[Wil91] M. Willshire. How spacey can they get? space overhead for storage and indexing with object-oriented databases. In *Proc. IEEE Conference on Data Engineering*, pages 14–22, 1991.