Proc. BTW '93, Braunschweig, März 1993

Benutzergesteuerte, flexible Speicherungsstrukturen für komplexe Objekte

Ullrich Keßler, Peter Dadam
Universität Ulm
Fakultät für Informatik
Abt. Datenbanken und Informationssysteme
Oberer Eselsberg, 7900 Ulm
e-mail: {kessler, dadam}@informatik.uni-ulm.de

Zusammenfassung

In der Vergangenheit wurden eine Reihe neuer Datenmodelle entwickelt, in denen komplex strukturierte Objekte unmittelbar dargestellt werden können. In der Regel werden hierbei für die logischen Strukturen eines Datenmodells bereits auch die physischen Speicherungsstrukturen festgelegt. Für die Wahl einer "optimalen" physischen Speicherungsstruktur - bei gegebener logischer Struktur - müßte jedoch die Art, in der auf die Daten später zugegriffen werden soll, mit berücksichtigt werden. Dies ist aber nur in einem System möglich, in dem die Abbildung der logischen Strukturen auf interne Speicherungsstrukturen frei definiert werden kann. Dazu werden in diesem Beitrag geeignete Basiskonstrukte für interne Speicherungsstrukturen sowie wesentliche Elemente und Parameter einer fiktiven Datendefinitionssprache diskutiert. Die hierbei vorgestellten Parameter sind zusammen so mächtig, daß mit ihnen explizit definiert werden kann, wie komplexe Objekte systemintern dargestellt werden sollen. Außerdem wird beschrieben, wie auch die Clusterung der Objekte gesteuert werden kann. Insgesamt wird dadurch eine Flexibilität erreicht, die so groß ist, daß sehr viele der in der Literatur vorgeschlagenen Speicherungsstrukturen nachgebildet werden können. Zusätzlich können eine große Zahl weiterer, noch nicht diskutierter Varianten und Mischformen beschrieben werden.

1. Einleitung und Problemstellung

Die sehr eingeschränkten Möglichkeiten, technisch-wissenschaftliche und sogenannte Non-Standard-Anwendungen mit traditionellen Datenmodellen zu modellieren, haben in der Vergangenheit zu einer Reihe von Vorschlägen für neuartige Datenmodelle, wie etwa Smalltalk-ähnliche, persistente objektorientierte Datenmodelle [Nier89], das Molekül-Atom-Datenmodell [Mits88] oder das NF^2-Datenmodell [ScPi82] bzw. dessen Erweiterung, das eNF^2-Datenmodell [PiAn86], geführt. Darauf aufbauend entstanden unterschiedliche DBMS-Prototypen, wie etwa O_2 [Banc88], AIM-P [Dada86], Prima [HMMS87], Orion [Kim89], XSQL [Lori85], GemStone [MSOP86], DASDBS [Paul87] und COCOON [ScSc90]. Diese Systeme unterstützen im Gegensatz zu herkömmlichen Datenbanksystemen die Speicherung komplex strukturierter Objekte unmittelbar, und zwar sowohl auf der Datenmodellebene als auch in den Anfragesprachen.

Gleichzeitig mit der Entwicklung neuer Datenmodelle und DBMS-Prototypen wurde auch diskutiert, wie komplexe Objekte auf Seiten des Hintergrundspeichers abgebildet werden können. Typische Vorschläge hierzu finden sich unter anderem in [Dada86], [DPS86], [HaOz88], [KFC90], [Kim89], [Lori85], [MSOP86] und [Sike88]. Daß es hierbei keine

"beste" Abbildungsstrategie gibt, die in jedem Einzelfall das beste Ergebnis liefert, liegt auf der Hand. Wird ein komplexes Objekt in nur wenige "große" Speichereinheiten zerlegt, müssen auch bei der Bearbeitung von Anfragen[1], die nur kleine Teile eines komplexen Objektes betreffen, stets diese "großen" Speichereinheiten mit entsprechend vielen Seitenzugriffen gelesen werden. Umgekehrt ist eine Strategie, die ein komplexes Objekt in "viele" kleine Speichereinheiten zerlegt, auch nicht immer die beste Lösung. Beim häufigen Zugriff auf ganze Substrukturen komplexer Objekte müssen dann viele einzelne Speichereinheiten gelesen werden. Hierdurch summieren sich die beim Zugriff auf Speichereinheiten häufig zu beobachtenden langen Pfadlängen zu hohen Kosten. Sind die Speicherelemente zusätzlich nur ungenügend geclustert, kommt es auch noch zu vielen Seitenzugriffen. Eine gute Strategie müßte also häufig als Ganzes zugegriffene Substrukturen eines komplexen Objektes in eine oder wenige Speichereinheiten abbilden und selten gemeinsam zugegriffene Substrukturen separieren. Dieses Ziel kann aber, wie auch das nachfolgende Beispiel einer eNF^2-Roboterrelation zeigt, mit einer anwendungsunabhängigen Abbildungsstrategie, welche die Anwendungsprofile nicht berücksichtigt, nicht oder nur bedingt erreicht werden.

In Abb. 1 ist die stark vereinfachte Ausprägung einer eNF^2-Relation zu sehen, in der sowohl die für die Einsatzplanung von Robotern benötigten konstruktiven Daten als auch die für die Verwaltung benötigten administrativen Daten verschiedener Roboter gespeichert werden. Folgt man den oben genannten Vorschlägen, wird jede Achse eines Roboters auf 4 bis 8 Records verteilt. Werden die Daten der Achsen jedoch überwiegend im Zusammenhang gelesen, wäre eine Lösung, welche alle Daten einer Achse in einem einzigen Record speichert, wie dies in Abb. 1 für die Achse 1 durch den ▭-Rahmen angedeutet ist, vorzuziehen. Hierdurch würde die Zahl der Record-Zugriffe stark reduziert. Umgekehrt werden bei den obigen Heuristiken die Attribute "R_Nr", "Name" und "Beschreibung" meistens gemeinsam in einem Record abgelegt, obwohl sie unter Umständen besser auf mehrere Records verteilt werden sollten. Werden nämlich die Nummer und der Name eines Roboters wesentlich häufiger als dessen Beschreibung benötigt, könnte so vermieden werden, daß die unter Umständen lange Beschreibung jedesmal mit in den Hauptspeicher übertragen wird.

Ein zweites Optimierungspotential ist die Clusterung der Daten. Eine typische Heuristik in diesem Zusammenhang ist, alle Records eines komplexen Objektes auf benachbarten Seiten zu speichern (vgl. [BeDe89], [Dada86], [DPS86], [KFC90], [Kim87], [ScSi89]). Im Fall der Roboterrelation würde zum Beispiel versucht werden, alle Records eines Roboters auf der Platte benachbart zu speichern. Werden nun Teile der Daten häufig unabhängig von ihrer Objektzugehörigkeit gelesen, bewirkt diese Clusterung genau das Gegenteil. Ein Beispiel hierfür könnten die Einsatzdaten sein. Werden diese Werte für Abrechnungszwecke zumeist unabhängig von den übrigen Roboterdaten gelesen, wäre eine Clusterung, die sämtliche Einsatzdaten in einem einzigen objektübergreifenden Cluster speichert - wie dies in Abb. 1 durch die ▒-Schattierung angedeutet ist - einer rein objektbezogenen Clusterung vorzuziehen.

Bei einer festen Abbildung logischer Konstrukte in physische Speicherungsstrukturen lassen sich Probleme dieser Art nur lösen, indem die logische - und damit auch die physische - Struktur der Objekte geändert wird. Dies hat allerdings zur Folge, daß auch

[1] Im folgenden wird der Begriff Anfrage als Synonym für Anfrage oder Datenmanipulationsauftrag verwendet.

{Roboter}										
R_Nr	Name	<Achsen>			{Einsatz}			{Effektoren}		Beschreibung
		Achs_Nr	<Positionsmatrix>		Produkt	{Leistung}		E_Nr	Aufgabe	
			Reihe	<Vektor>		Woche	Kosten			
R_1	Robi	1	1	<20,20,20,20>	Klappe	17	200	E_1	schweißen	Roboter ...
			2	<34,37,56,90>		30	300	E_2	nieten	
			3	<21,34,78,60>	Stange	16	400	E_3	pressen	
			4	<45,56,78,12>		29	100	E_4	kleben	
		2	1	<16,90,30,14>		41	500			
			2	<16,42,45,78>						
			3	<12,79,59,78>						
			4	<23,67,31,67>						
R_2	Bigi	...	...	...	...	...	...	...	...	...

Legende: Mengenbildung: {}, Listenbildung: <>

gewünschte Recordbildung:

gewünschte objektübergreifende Clusterung:

Abb. 1: Ausprägung einer eNF²-Roboterrelation

die - evtl. bereits existierenden - Anwendungsprogramme abgeändert werden müssen. Der bessere Weg, mit solchen Problemen umzugehen, ist eine Trennung zwischen logischer und physischer Struktur. Diesen Weg hat man bereits erfolgreich in einigen kommerziellen relationalen Datenbanksystemen eingeschlagen. In Ingres [Ingr90] kann der Anwender beispielsweise unter verschiedenen Methoden zur Speicherung der Relationen auswählen. In Oracle [Orac90] kann der Anwender die Clusterung der Relationen explizit definieren. Andere Beispiele sind das hierarchische Datenbanksystem IMS [Gee77] und Netzwerkdatenbanken, die eine Speicherbeschreibungssprache anbieten. In diesen Systemen hat der Anwender Möglichkeiten, in einer eigenen Sprache die internen Speicherungsstrukturen seinen Bedürfnissen anzupassen. Im Zusammenhang von komplexen Objekten gibt es jedoch erst wenige Vorschläge, wie die Speicherungsstrukturen und die Clusterungs-Strategien in Abhängigkeit von den Anwendungen definiert werden können. In [Scho92] werden zum Beispiel alternative Strategien zur Implementation von komplexen Objekten unter Verwendung eines Speichermanagers für hierarchische Strukturen diskutiert. In [BeDe89], [Kim87] und [ScSi89] wird gezeigt, wie die Daten in Abhängigkeit von den Anwendungen geclustert werden können.

Wie wir im folgenden am Beispiel des eNF²-Datenmodells zeigen werden, gibt es beim Entwurf von Speicherungsstrukturen für komplexe Objekte im wesentlichen zwei orthogonale Freiheitsgrade. Der erste Freiheitsgrad ist die Wahl einer Datenstruktur zur Implementation von Mengen, Listen und Tupeln. Der zweite Freiheitsgrad ist die Entscheidung, ob die Elemente von Mengen und Listen bzw. die Attribute von Tupeln direkt in diesen Datenstrukturen oder in referenzierten Records abgelegt werden. Um dem

Anwender die Kontrolle über diese Freiheitsgrade zu geben, reichen, wie wir ebenfalls zeigen werden, bereits drei einfache, orthogonale Parameter in einer entsprechenden Datendefinitionssprache aus. Mit ihnen kann die Abbildung eines komplexen Objektes auf Records annähernd frei definiert werden. Die Flexibilität, die diese Parameter bieten, ist dabei so groß, daß praktisch alle Vorschläge in den Papieren [Dada86], [DPS86], [HaOz88], [KFC90] und [Lori85] nachgebildet werden können. Darüber hinaus lassen sich viele weitere Varianten definieren. Im Extremfall kann ein vollständiges komplexes Objekt in einem einzigen Record gespeichert werden. Das andere Extrem, jeden atomaren Wert eines komplexen Objektes in einem eigenen Record zu speichern, ist ebenfalls möglich. Die logische Struktur der komplexen Objekte ist hiervon stets unbeeinflußt[2].

Darüber hinaus wird im weiteren Verlauf des Beitrags diskutiert, wie auch die Clusterung der Records gesteuert werden kann. Die diskutierten Techniken erlauben es, annähernd beliebige Records zu Clustern zusammenzufassen. So lassen sich beispielsweise beliebige Substrukturen komplexer Objekte gruppieren. Es ist aber auch möglich, Records unabhängig von ihrer Objektzugehörigkeit objektübergreifend in Clustern zusammenzufassen. Im Falle der Roboterrelation können beispielsweise die Achs- und Effektordaten objektbezogen jeweils in einem Cluster pro Roboter gespeichert werden. Gleichzeitig können die Einsatzdaten aller Roboter unabhängig von ihrer Objektzugehörigkeit in einem einzigen Cluster gespeichert werden.

Obwohl in diesem Beitrag mit dem eNF^2-Datenmodell ein rein disjunktes und nicht rekursives Datenmodell gewählt wurde, beschränkt sich die Anwendbarkeit der hier vorgestellten Verfahren nicht zwangsläufig auf Systeme, die auch an der Benutzerschnittstelle dieses Datenmodell anbieten (wie z.B. AIM-P). Für viele Systeme, die nach außen hin mächtigere Datenmodelle unterstützen, wurde vorgeschlagen, aus Effizienzgründen intern hierarchische Datenstrukturen zur Speicherung der komplexen Objekte zu verwenden. Beispielsweise wird in [BeDe89], [ScSi89] und [Kim87] ausführlich diskutiert, wie in O_2, Prima und in Smalltalk-ähnlichen Systemen, wie GemStone und Orion, deren komplexe Objekte, die nicht unbedingt hierarchisch aufgebaut sind, intern auf hierarchische Datenstrukturen abgebildet werden können. Eine ähnliches Ziel wird mit dem Speicherkernsystem DASDBS verfolgt. Auf einen Speichermanager, der ein hierarchisches Datenmodell anbietet, werden sowohl relationale Systeme [SPS87] als auch objektorientierte Systeme [Scho92] aufgesetzt. Der Grund ist, daß sich hierarchische Strukturen besonders gut zu Clustern zusammenfassen lassen, wodurch wiederum die Performanz eines Systems gesteigert werden kann.

Der Rest des Beitrages gliedert sich entsprechend der Idee, sowohl die Record-Struktur eines komplexen Objektes als auch dessen Clusterung frei definieren zu können. In Abschnitt 2 werden dazu kurz das hier zugrunde gelegte eNF^2-Datenmodell und eine Datendefinitionssprache mit einer stark vereinfachten Syntax eingeführt. In Abschnitt 3, dem Hauptteil dieses Beitrages, werden dann ausführlich die zwei Freiheitsgrade - Wahl einer Datenstruktur zur Implementation von Mengen, Listen und Tupeln und Entscheidung, ob Elemente bzw. Attribute materialisiert oder referenziert gespeichert werden - beim Entwurf von Speicherungsstrukturen für komplexe Objekte und die Kontrolle dieser

[2] Die nachträgliche Änderung der Speicherungsstrukturen eines Objektes erfordert unter Umständen eine interne Reorganisation des Objektes oder eine entsprechende Katalogverwaltung.

Freiheitsgrade durch Parameter einer Datendefinitionssprache diskutiert. In Abschnitt 4 wird beschrieben, wie die Clusterung der Records gesteuert wird. Abschnitt 5 faßt die Diskussion zusammen und gibt einen kurzen Ausblick.

2. Das eNF²-Datenmodell

Als Grundlage für die weiteren Diskussionen wird im folgenden das eNF²-Datenmodell nach [PiAn86] verwendet. An diesem Datenmodell wird exemplarisch gezeigt, wie die systeminterne Repräsentation und Clusterung von komplexen Objekten definiert werden können. Komplexe Objekte werden im eNF²-Datenmodell aus atomaren Werten und darauf rekursiv angewendeten Mengen-, Listen- und Tupelkonstruktoren gebildet. Typische atomare Wertebereiche sind Integer, Real und String. Abb. 2 veranschaulicht diesen Prozeß. Ein komplexes Objekt kann beispielsweise eine Menge von Mengen von Strings sein, aber auch nur ein einzelner Wert oder aber ein wesentlich komplexeres Objekt, wie die Roboter-Relation in Abb. 1. Jede Relation in erster Normalform ist ebenfalls ein Objekt des eNF²-Datenmodells.

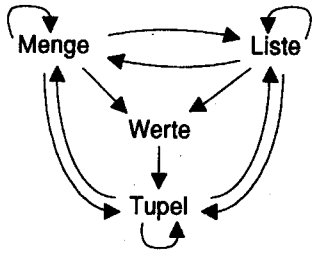


Abb. 2: Graphische Darstellung des eNF²-Datenmodells

Zur Definition des Typs und der Speicherungsstruktur eines komplexen Objektes wird in diesem Beitrag exemplarisch eine Datendefinitionssprache mit einer sehr einfachen Syntax verwendet, da hier die wesentlichen Prinzipien der Definition von Speicherungsstrukturen und nicht die syntaktischen Konstrukte einer solchen Sprache diskutiert werden sollen. Daher verzichten wir auch auf die eigentlich wünschenswerte Trennung zwischen Datendefinitions- und Speicherstruktursprache. Die vollständige Syntax dieser Sprache ist in der Abb. 9 im Anhang gegeben. Beschränkt man sich auf die reine Typdefinition, würde eine typische Mitarbeiterrelation mit den Attributen Pers_Nr., Name, Gehalt und Lebenslauf (s. Abb. 5) wie folgt definiert[3] (die Speicherstrukturbeschreibung wird später an den mit [...] gekennzeichneten Stellen eingetragen):

```
complex_object Mitarbeiter [...]
               set [...] of tuple (Pers_Nr.    [...]: integer,
                                   Name        [...]: fix_string(30),
                                   Gehalt      [...]: real,
                                   Lebenslauf  [...]: var_string)
```

3. Flexible Abbildung logischer Konstrukte in physische Speicherungsstrukturen

Beim Entwurf von physischen Speicherungsstrukturen für komplexe Objekte gibt es zwei wichtige Freiheitsgrade. Der erste Freiheitsgrad ist die Wahl der internen Speicherungsstrukturen zur Implementation von Mengen, Listen und Tupeln. Im folgenden werden

[3] Zum leichteren Verständnis werden in den Beispielen Schlüsselworte stets klein und frei zu wählende Bezeichner stets groß geschrieben.

diese internen Speicherungsstrukturen auch als "Konstruktordatenstruktur" bezeichnet. Zur Implementation einer Menge oder Liste kann zum Beispiel - ähnlich wie in Netzwerkdatenbanken - ein variabel langes Array oder eine verkettete Liste verwendet werden. Die Attribute eines Tupels können zusammenhängend in einem Record oder aufgeteilt auf mehrere gespeichert werden.

Der zweite Freiheitsgrad ist die Entscheidung, ob die Elemente einer Menge oder Liste bzw. die Attribute eines Tupels direkt in der Konstruktordatenstruktur gespeichert oder aber aus ihr heraus referenziert werden. Im folgenden werden diese Fälle auch als "materialisierte" bzw. "referenzierte" Speicherung bezeichnet. Für eine einfache Menge von atomaren Werten, die als Array implementiert ist, bedeutet dies: Bei einer materialisierten Speicherung enthält das Array die atomaren Werte. Bei der referenzierten Speicherung enthält das Array Referenzen auf Records mit den atomaren Werten. Graphisch ist dies in Abb. 3 für die Menge "{a, b, c}" dargestellt.

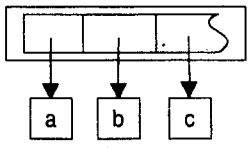


Abb. 3.1: Referenzierte Speicherung

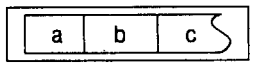


Abb. 3.2: Materialisierte Speicherung

Abb. 3: Zwei Implementationen der Menge {a, b, c}

Sind die Elemente einer Menge oder Liste bzw. die Attribute eines Tupels selbst komplex strukturiert, so ergeben sich weitere Möglichkeiten, das Objekt auf Records aufzuteilen. Dazu betrachte man beispielsweise die Menge

{{String_1, String_2}, {String_3, String_4}}.

In dieser Menge sind die Elemente "{String_1, String_2}" und "{String_3, String_4}" der äußeren Menge selbst wieder Mengen. Nun können sowohl die Elemente der äußeren als auch die Elemente der inneren Mengen referenziert oder materialisiert gespeichert werden. Die daraus resultierenden vier Speicherungsstrukturen[4] sind in der Abb. 4 graphisch dargestellt.

Abb. 4.1 zeigt die Speicherungsstruktur, in der sowohl die Elemente der äußeren als auch die Elemente der inneren Mengen referenziert gespeichert werden. Die Konstruktordatenstruktur der äußeren Menge enthält Referenzen auf Records mit den Konstruktordatenstrukturen der inneren Mengen. Diese wiederum enthalten Referenzen auf Records mit den Werten "String_1", ..., "String_4". In Abb. 4.2 wird angenommen, daß die Konstruktordatenstrukturen der Mengen "{String_1, String_2}" und "{String_3, String_4}" (= materialisierte Subobjekte) in demselben Record wie die Konstruktordatenstruktur der äußeren Menge gespeichert werden. Die Elemente "String_1", ..., "String_4" werden jedoch weiterhin in referenzierten Records abgelegt. Dieser Fall ist ein typisches Beispiel dafür, daß sich die Aussage: "ein komplexes Subobjekt wird materialisiert gespeichert" zunächst einmal nur auf die Speicherung der Konstruktordatenstruktur des betroffenen Subobjektes, nicht aber auf die Speicherung seiner Elemente bezieht. Abb. 4.3 zeigt den genau umgekehrten Fall: Die Elemente der

[4] Hierbei werden variabel lange Arrays als Konstruktordatenstrukturen angenommen. Werden zusätzlich auch verkettete Listen als Konstruktordatenstrukturen betrachtet, erhält man insgesamt 16 Varianten.

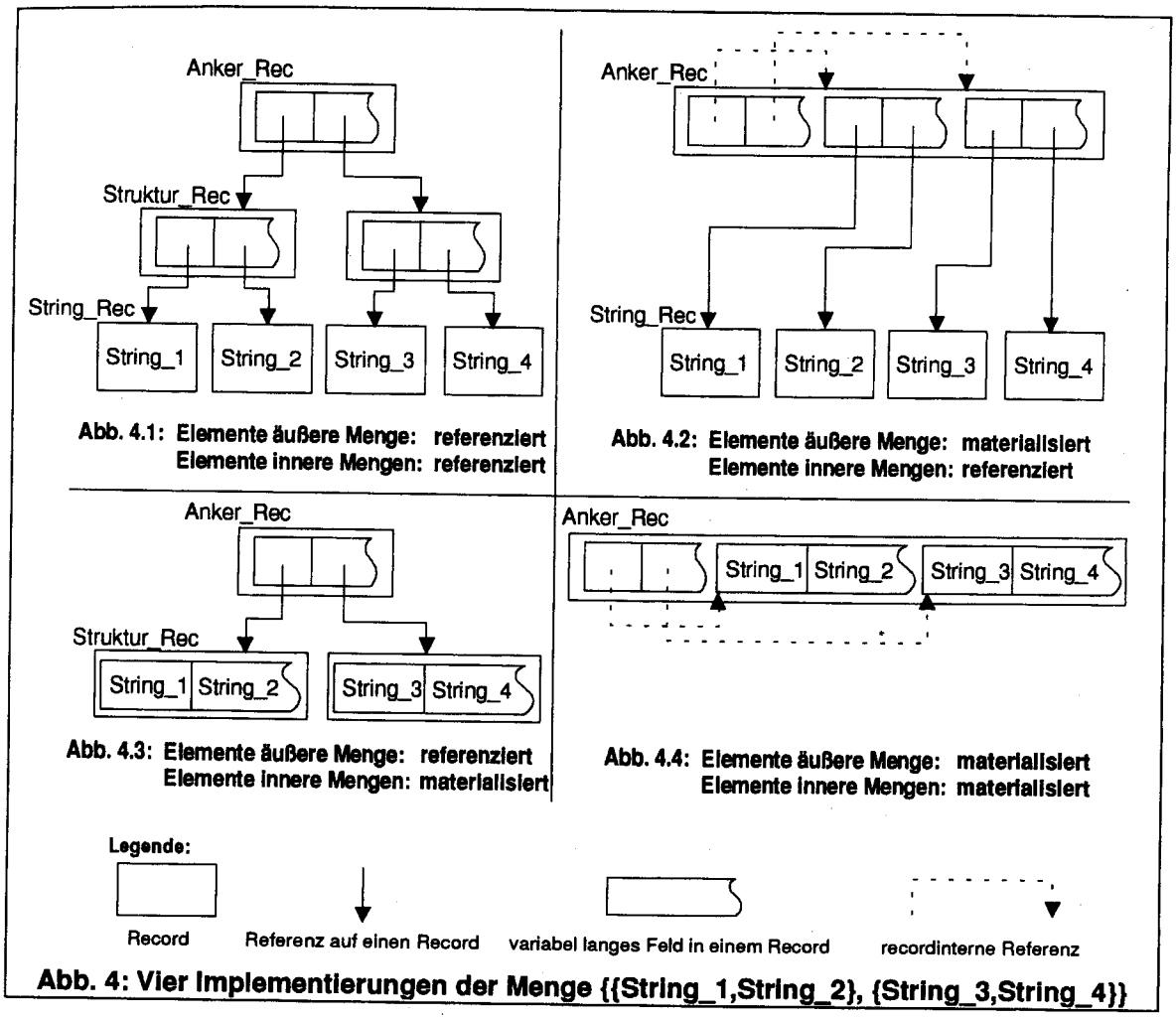


Abb. 4.1: Elemente äußere Menge: referenziert
Elemente innere Mengen: referenziert

Abb. 4.2: Elemente äußere Menge: materialisiert
Elemente innere Mengen: referenziert

Abb. 4.3: Elemente äußere Menge: referenziert
Elemente innere Mengen: materialisiert

Abb. 4.4: Elemente äußere Menge: materialisiert
Elemente innere Mengen: materialisiert

Abb. 4: Vier Implementierungen der Menge {{String_1,String_2}, {String_3,String_4}}

äußeren Menge werden referenziert, die Elemente der inneren Menge materialisiert. Daher enthält die Konstruktordatenstruktur der äußeren Menge Referenzen auf Records. Diese speichern die Konstruktordatenstrukturen der inneren Mengen zusammen mit den materialisierten Strings. Abb. 4.4 zeigt schließlich den letzten Fall, in dem sowohl die Elemente der äußeren als auch die Elemente der inneren Mengen materialisiert werden. Das vollständige Objekt wird daher in einem einzigen Record gespeichert.

Im folgenden werden nun orthogonale Parameter herausgearbeitet, mit denen diese Freiheitsgrade - Wahl einer geeigneten Konstruktordatenstruktur, Entscheidung zwischen referenzierter und materialisierter Speicherung - sowohl bei der Mengen- und Listenbildung als auch bei der Tupelbildung kontrolliert werden können. Mit diesen Parametern können unterschiedlichste Abbildungen der logischen Konstrukte des eNF²-Datenmodells auf physische Speicherungsstrukturen beschrieben werden, so daß die Speicherungsstrukturen der Objekte auf die jeweiligen Anwendungen abgestimmt werden können. Dazu werden zunächst die Begriffe "Record" und "Record-Typname" etwas genauer eingeführt. Anschließend wird diskutiert, wie die Speicherungsstrukturen von Mengen und Listen definiert werden können. Hierbei werden unter anderem auch die 4 Speicherungsstrukturen aus Abb. 4 vollständig definiert. Danach wird die Tupelbildung betrachtet und diskutiert, wie sich dort die Speicherungsstrukturen beschreiben lassen.

3.1 Records und Record-Typnamen

Ein "Record" ist - so wie der Begriff in diesem Beitrag verwendet wird - ein logisches Speicherobjekt, das aus einer variablen Anzahl von Bytes besteht. Zur Identifizierung besitzt jeder Record einen eindeutigen Identifier. Je nach Größe der Records können sowohl mehrere Records in einer Seite des Hintergrundspeichers abgelegt als auch ein Record über viele Seiten verteilt werden.

Zur Speicherung eines komplexen Objektes wird mindestens ein Record benötigt. Dieser wird im folgenden auch als "Anker-Record" bezeichnet. Ist das komplexe Objekt lediglich ein atomarer Wert, so enthält der Anker-Record genau diesen Wert. Ist das komplexe Objekt hingegen eine Menge, Liste oder ein Tupel, so enthält der Anker-Record mindestens die gewählte Konstruktordatenstruktur. Über diesen Anker-Record können durch Verfolgung von Referenzen alle Subobjekte erreicht werden.

Um bei der Definition von Speicherungsstrukturen und insbesondere bei der Definition der Clusterung (s. Abschnitt 4) symbolisch auf Records Bezug nehmen zu können, werden Records mit semantisch äquivalentem Inhalt zu Record-Typen zusammengefaßt. Jeder Record-Typ erhält dazu bei der Definition der Speicherungsstrukturen einen eindeutigen, frei wählbaren "Record-Typnamen". Beispielsweise werden bei der Definition der Speicherungsstruktur in Abb. 4.1 die Record-Typnamen "Anker_Rec", "Struktur_Rec" und "String_Rec" vergeben. Dabei wird in der hier verwendeten Syntax der Record-Typname des Anker-Records in dem Parameter "anchor_record_type" vergeben (s. Anhang, Abb. 9 [1]). Die beiden anderen Record-Typnamen werden in den Definitionen der Mengen festgelegt.

3.2 Speicherungsstrukturen für Mengen - und Listenkonstruktoren

Entsprechend der vorangegangenen Diskussion gibt es bei der Implementation von Mengen und Listen zwei voneinander unabhängige Freiheitsgrade. Der erste Freiheitsgrad ist die Wahl der Konstruktordatenstruktur. Dies kann zum Beispiel ein variabel langes Array oder eine verkettete Liste sein; andere Implementationen sind aber auch denkbar. Der zweite Freiheitsgrad ist die Entscheidung, ob die Elemente direkt in der Konstruktordatenstruktur gespeichert werden oder ob diese nur Zeiger auf Records mit den Elementen enthält. Sollen beide Freiheitsgrade bei der Mengen- und Listenbildung unabhängig spezifiziert werden können, so werden hierfür in einer Datendefinitionssprache zwei entsprechende Parameter benötigt. In der hier verwendeten Syntax werden dazu in den Term zur Objektdefinition die Parameter "implementation" und "element_placement" integriert (vgl. Anhang Abb. 9, [2]-[7]):

```
object_type = ...
              /* Definition einer Menge. */
              set  [implementation    = implementation_type,
                    element_placement = placement_type] of object_type |
              /* Definition einer Liste. */
              list [implementation    = implementation_type,
                    element_placement = placement_type] of object_type |...
```

In dem Parameter "implementation" wird die gewünschte Implementation der Menge oder Liste ausgewählt. Kann in einem System eine Menge oder Liste durch ein Array oder eine verkettete Liste implementiert werden, so kann dieser Parameter die gültigen Werte "array" und "linked_list" annehmen (vgl. Abb. 9, [11]):

```
implementation_type = array | linked_list
```

Mit dem zweiten Parameter, hier "element_placement" genannt, wird bestimmt, ob die Elemente in der gewählten Konstruktordatenstruktur materialisiert oder aus ihr heraus referenziert werden. Als gültige Werte werden im folgenden verwendet (vgl. Abb. 9, [10]):

```
placement_type = inplace | referenced (record_type_name)
```

Wird in dem Parameter "element_placement " der Wert "inplace" angegeben, so werden die Elemente direkt in der Konstruktordatenstruktur gespeichert. Wird hingegen "referenced" verwendet, so wird für jedes Element der Menge oder Liste ein eigener Record vom Typ "record_type_name" angelegt. Beispielsweise werden die Elemente "String_1", ..., "String_4" der inneren Mengen der Menge "{{String_1, String_2}, {String_3, String_4}}" im Fall 1 entsprechend der nachfolgenden Definition in referenzierten Records mit dem frei gewählten Record-Typnamen "String_Rec" gespeichert (vgl. Abb. 4.1). Der gewählte Typname muß dabei innerhalb eines komplexen Objektes eindeutig sein. In die Konstruktordatenstruktur werden dann nur noch die Identifier dieser Records eingetragen.

Die Verwendung dieser Parameter sei am Beispiel der vier Speicherungsstrukturen für die Menge "{{String_1, String_2}, {String_3, String_4}}" aus Abb. 4 näher erläutert. Abb. 4.1 zeigt den Fall, daß sowohl die Elemente der äußeren als auch die Elemente der inneren Mengen referenziert gespeichert werden. Die vollständige Definition der Speicherungsstruktur lautet:

```
complex_object Menge_von_Mengen_von_Strings [anchor_record_type=Anker_Rec] 1
 set [implementation=array, element_placement=referenced (Struktur_Rec)A] of 2
   set [implementation=array, element_placement=referenced (String_Rec)B] of 3
       var_string.                                                          4
```

In Zeile 1 dieser Definition werden der Name "Menge_von_Mengen_von_Strings" des komplexen Objektes und der Record-Typname "Anker_Rec" des Anker-Records festgelegt. Zeile 2 besagt, daß das komplexe Objekt eine Menge ist. Zu ihrer Implementation wird ein Array verwendet. Die Elemente der Menge werden in referenzierten Records mit dem frei gewählten Record-Typnamen "Struktur_Rec" gespeichert. In Zeile 3 wird definiert, daß die Elemente der äußeren Menge selbst wieder Mengen sind. Auch diese inneren Mengen werden als Arrays implementiert. Die Elemente der inneren Mengen werden wiederum referenziert. Dazu werden sie in Records mit dem Typnamen "String_Rec" gespeichert. Zeile 4 besagt schließlich, daß die Elemente der inneren Mengen Strings variabler Länge sind.

Die Speicherungsstrukturen der Abb. 4.2, 4.3 und 4.4 können aus der Definition der Speicherungsstruktur der Abb. 4.1 durch Variation der Parameter "A" und "B" abgeleitet werden. Wird der Parameter "A" auf den Wert "element_placement = inplace" gesetzt, ergibt sich die Struktur der Abb. 4.2, in der die Elemente der äußeren Menge materialisiert, die der inneren aber referenziert werden. Umgekehrt ergibt sich die Struktur in Abb. 4.3, in der die Elemente der inneren Mengen materialisiert werden, nicht aber die Elemente der äußeren Menge, indem der Parameter "B" auf "element_placement = inplace" gesetzt wird. Schließlich erhält man die Struktur in Abb. 4.4, in der das gesamte komplexe Objekt in einem einzigen Record gespeichert wird, indem beide Parameter "A" und "B" auf "element_placement = inplace" gesetzt werden.

Bereits an diesem einfachen Beispiel sieht man den hohen Grad der Flexibilität, der durch den Parameter "element_placement" erreicht wird. Variiert man noch den

Parameter "implementation" von "array" nach "linked_list", kommen weitere zwölf Varianten zur Implementation der Menge von Mengen von Strings hinzu.

3.3 Speicherungsstrukturen für Tupelkonstruktoren

Nachdem im vorigen Abschnitt die Repräsentation von Mengen und Listen diskutiert wurde, wird nun dargestellt, wie sich die internen Speicherungsstrukturen von Tupeln beschreiben lassen. Prinzipiell existieren hierbei die gleichen Freiheitsgrade - Wahl einer Konstruktordatenstruktur, Entscheidung, ob Attribute referenziert oder materialisiert gespeichert werden - wie bei der Mengen- und Listenbildung. Im Falle der Tupelbildung entspricht der erste Freiheitsgrad der Entscheidung, ob ein Tupel - oder genauer gesagt seine Konstruktordatenstruktur - in einem Record oder auf mehrere Records verteilt gespeichert wird. Der zweite Freiheitsgrad ist die Entscheidung, ob die Attributwerte in der Konstruktordatenstruktur materialisiert gespeichert oder aus ihr heraus referenziert werden. Um diese beiden Freiheitsgrade ebenfalls unabhängig voneinander kontrollieren zu können, werden wiederum zwei Parameter gebraucht. Exemplarisch werden hierzu der neue Parameter "location" und der bereits bekannte Parameter "element_placement" in die Attributdefinition aufgenommen (vgl. Abb. 9, [8]-[9]):

```
attribute_description = attribute_name [location = location_type,
                                        element_placement=placement_type]: object_type
```

Mit dem Parameter "element_placement" wird wie bei Mengen und Listen definiert, ob ein Attributwert bzw. - wenn das Attribut eine Menge, Liste oder ein Tupel ist - dessen Konstruktordatenstruktur direkt in der Konstruktordatenstruktur des Tupels gespeichert oder aus ihr heraus referenziert wird. Dazu wird hier angenommen, daß ein Tupel durch eine Datenstruktur implementiert wird, die einem Record in einer Pascal-ähnlichen Programmiersprache gleicht. In dieser Datenstruktur wird für jedes Attribut ein Feld vorgesehen. Abhängig von dem Parameter "element_placement" enthält dieses Feld entweder den Attributwert oder eine Referenz auf einen Record mit dem jeweiligen Attributwert. Da der Parameter "element_placement" an die Attributdefinition gebunden ist, kann unabhängig für jedes Attribut entschieden werden, ob es materialisiert oder referenziert gespeichert wird.

Aus Optimierungsgründen, wenn zum Beispiel einige Attribute eines Tupels nur sehr selten zugegriffen werden, kann es nützlich sein, die Konstruktordatenstruktur eines Tupels auf mehrere Records aufzuteilen. Dazu wird sie im folgenden in einen Primärblock und optional mehrere Sekundärblöcke aufgeteilt. Sowohl dem Primär- als auch den Sekundärblöcken können hierbei mehrere Attribute zugeordnet werden. Jeder Sekundärblock wird in einem eigenen Record gespeichert. Die Referenzen auf diese Records werden in dem Primärblock gespeichert. Ob für den Primärblock ebenfalls ein Record angelegt wird, hängt davon ab, ob das Tupel selbst referenziert oder materialisiert gespeichert wird. Mit dem oben eingeführten Parameter "location" wird für jedes Attribut festgelegt, ob das zugehörige Feld in dem Primärblock oder in einem Sekundärblock lokalisiert wird. Der Parameter erhält dazu zwei zulässige Werte (Abb. 9, [12]):

```
location_type = primary | secondary (record_type_name)
```

Wird für ein Attribut "primary" angegeben, so wird das zugehörige Feld in dem Primärblock angelegt. Hat der Parameter hingegen den Wert "secondary (record_type_name)", so wird das Feld in einem Sekundärblock angelegt. Der Sekundärblock wird in einem Record vom Typ "record_type_name" gespeichert. Sollen mehrere Attribute in dem

Mitarbeiter			
Pers_Nr.	Name	Gehalt	Lebenslauf
77234	Maier	4000	Frau Bettina Maier ist am ...
77235	Schmidt	4400	Herr Fritz Schmidt ist am ...

Abb. 5: Ausprägung einer Mitarbeiterrelation

gleichen Sekundärblock gespeichert werden, so ist in "record_type_name" jeweils derselbe Record-Typname anzugeben.

Das Zusammenspiel der beiden Parameter "location" und "element_placement" soll nun an dem Beispiel der Mitarbeiterrelation in Abb. 5 verdeutlicht werden. Dabei nehme man an, daß die Relation sehr häufig verwendet wird, um aus dem Mitarbeiternamen die Personalnummer abzuleiten und umgekehrt. Auf das Gehalt und den Lebenslauf werde selten zugegriffen. Daher sollen der Name und die Personalnummer gemeinsam in dem Primärblock materialisiert gespeichert werden, das Gehalt und der Lebenslauf sollen hingegen in einem gemeinsamen Sekundärblock ausgelagert werden. Der unter Umständen lange Lebenslauf wird referenziert gespeichert. Um die Tupel der Relation zu verbinden, wird eine verkettete Liste verwendet. Eine Definition der Mitarbeiterrelation, die diese Eigenschaften hat, lautet dann:

```
complex_object Mitarbeiter [anchor_record_type=Link_Rec]
 set [implementation=linked_list, element_placement=referenced (Prim_Rec)]
  of tuple
  (Pers_Nr.    [location=primary, element_placement=inplace]: integer,
   Name        [location=primary, element_placement=inplace]: fix_string(30),
   Gehalt      [location=secondary (Sec_Rec),element_placement=inplace]: real,
   Lebenslauf [location=secondary (Sec_Rec),
               element_placement=referenced(Lebenslauf_Rec)]: var_string).
```

Die sich daraus ergebende Speicherungsstruktur ist in Abb. 6.1 dargestellt.

An diesem Beispiel ist gut zu erkennen, daß die Parameter "location" und "element_placement" beide benötigt werden und nicht redundant sind. Würde auf einen von beiden verzichtet, so könnte nicht ausgedrückt werden, daß die Referenz auf das Lebenslauf-Record zusammen mit dem Gehalt in einem Sekundärblock zu speichern ist. Es wäre nur noch möglich, für den Lebenslauf einen eigenen Sekundärblock anzulegen.

Ein Nachteil der dargestellten Speicherungsstruktur sind die vielen kleinen Link-Records, die jeweils nur einen Zeiger auf das nächste Tupel und den referenzierten Primärblock eines Tupels enthalten. Dieses Problem läßt sich aber leicht lösen, indem die Primärblöcke in den Link-Records materialisiert gespeichert werden. Dazu ist in der zweiten Zeile der Definition nur der Parameter "element_placement" von "referenced (Prim_Rec)" nach "inplace" umzusetzen:

```
complex_object Mitarbeiter [anchor_record_type=Link_Rec]
 set [implementation=linked_list, element_placement=inplace] of ...
```

Dies bewirkt, daß die Primärblöcke mit den Referenzen auf die Sekundär-Records und den Feldern für die Attribute Pers_Nr. und Name in den Link-Records materialisiert werden. Die sich jetzt ergebende Speicherungsstruktur ist in Abb. 6.2 dargestellt.

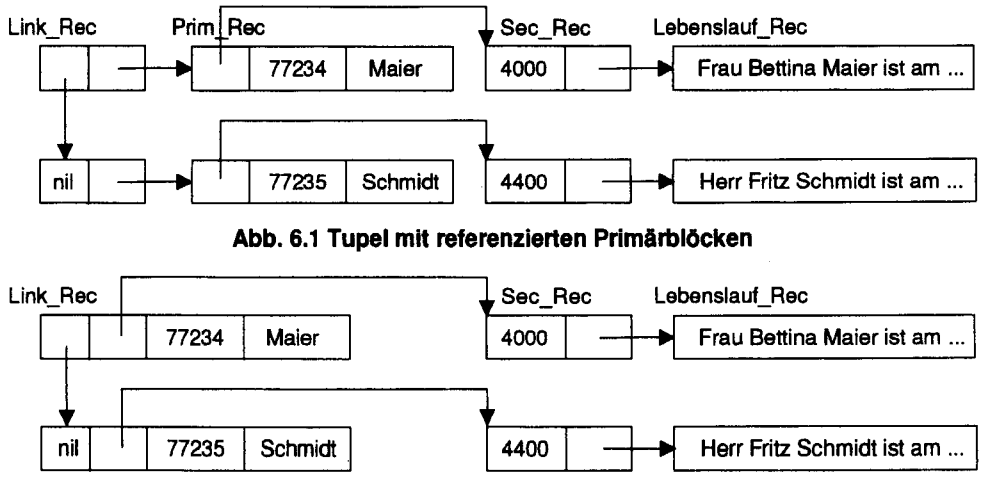


Abb. 6.1 Tupel mit referenzierten Primärblöcken

Abb. 6.2 Tupel mit materialisierten Primärblöcken

Abb. 6: Zwei mögliche Speicherungsstrukturen für die Mitarbeiterrelation

3.4 Vollständiges Beispiel einer Speicherungsstruktur für die Roboterrelation

Nachdem die Parameter "element_placement", "implementation" und "location" ausführlich diskutiert wurden, soll ihre Mächtigkeit noch einmal an dem Beispiel der eNF²-Roboterrelation aus Abb. 1 demonstriert werden. Dazu ist im Anhang in Abb. 8 eine vollständige Definition des Typs und einer Speicherungsstruktur der Roboterrelation gegeben. Die Speicherungsstruktur wurde dabei entsprechend den Annahmen in der Einleitung entworfen. Häufig gemeinsam benötigte Daten wurden zusammengefaßt und selten gemeinsam benötigte Daten separiert. Die sich ergebende Record-Struktur ist im Anhang in Abb. 10 graphisch dargestellt.

Die folgenden Details sind an dieser Struktur besonders interessant. Die Konstruktordatenstruktur für die Robotertupel wurde auf einen Primär- und einen Sekundärblock aufgeteilt. Dies ist in [1], [2], [3], [13], [16] und [18] der Abb. 8 zu erkennen. So können die Konstruktionsdaten eines Roboters, wie Achsen und Effektoren, objektbezogen zu Clustern zusammengefaßt werden. Die hiervon separierten Einsatzdaten können hingegen objektübergreifend gespeichert werden. Nähere Einzelheiten dazu werden in Abschnitt 4 beschrieben werden. Die Daten einer Achse werden jeweils in einem "Achs-Record" zusammengefaßt ([5] - [12] in Abb. 8). Als Konstruktordatenstruktur für die Liste der Achsen wird ein Array ([4] in Abb. 8) verwendet. Dieses wird in dem "Sec_Rec"-Record materialisiert ([3] in Abb. 8). Diese Implementation einer Liste entspricht den Vorschlägen in DASDBS (vgl. [DPS86]), komplexe Objekte zu implementieren. Im Gegensatz dazu wird zur Implementation der Menge der Einsatzdaten eine Speicherungsstruktur gewählt, die der Strategie in AIM-P (vgl. [Dada86]) entspricht. Aus dem "Prim_Rec"-Record heraus wird ein Zeiger-Record referenziert ([13] in Abb. 8). Dieser enthält Referenzen auf Daten-Records mit den Namen der bearbeiteten Produkte und auf Records mit den Einsatzwochen und Kosten ([14], [15] in Abb. 8). Die Implementation der Menge der Effektoren entspricht schließlich den Vorschlägen zur Implementation von XSQL (vgl. [Lori85]). Die Elemente der Mengen werden durch Zwillingspointer verkettet ([17] in Abb. 8). Die umfangreichen Bedienungsanleitungen der Roboter werden in eigenen Records gespeichert werden ([18] in Abb. 8).

4. Benutzergesteuerte Clusterung komplexer Objekte

Mit den in Abschnitt 3 entworfenen Methoden können komplexe Objekte entsprechend den Anforderungen einer Anwendung auf Records aufgeteilt werden. Der zweite Schritt in einer optimierten Speicherung komplexer Objekte ist, häufig gemeinsam zugegriffene Records auf den gleichen oder benachbarten Seiten zu speichern. So können die Zahl der Seitenzugriffe und damit die Kosten, eine Anfrage auszuwerten, weiter reduziert werden. Auch hier gilt, daß eine feste Heuristik, wie zum Beispiel alle Records eines komplexen Objektes in einem Cluster zusammenzufassen (vgl. [BeDe89], [Dada86], [DPS86], [KFC90], [Kim87], [ScSi89]), in vielen, wenn nicht sogar in den meisten Fällen gut sein kann, aber auch manchmal versagt. Dann ist wieder die Kontrolle des Anwenders notwendig. Dazu führen wir nachfolgend die Begriffe "Segment" und "Cluster" ein und zeigen, wie der Anwender selbst bestimmen kann, welche Records in welchen Segmenten und Clustern gespeichert werden. Dabei wird hier eine Variante verfolgt, bei der für jeden Record-Typ festgelegt wird, in welchem Cluster seine Records abgelegt werden. Hierdurch wird unter anderem die Optimierung von Anfragen erleichtert, da bereits zur Übersetzungszeit die Clusterung der Records bekannt ist.

4.1 Segmente und Cluster

Als ein "Segment" wird im folgenden eine logisch zusammenhängende Einheit von Seiten auf dem Hintergrundspeicher bezeichnet. In einem dateigestützten System würde ein Segment auf eine oder mehrere Direktzugriffsdateien abgebildet werden. Zur Identifikation sollen Segmente eindeutige Namen haben. In dem folgenden Vorschlag wird jedes Segment in "Cluster" aufgeteilt. Jeder Cluster kann beliebig viele Records enthalten. Einem Cluster sollen dazu eine oder mehrere, nach Möglichkeit benachbarte Seiten eines Segmentes zugeordnet werden. Die Zahl der Cluster innerhalb eines Segmentes kann sich mit der gleichen Dynamik ändern, mit der Daten in die Datenbank eingefügt oder aus ihr herausgelöscht werden. Diese Aufteilung eines Segmentes in Cluster ähnelt sehr dem Verfahren zur Clusterung von Tupeln in dem kommerziellen Datenbanksystem Oracle [Orac90].

4.2 Objektbezogene und objektübergreifende Cluster

Bereits in der Einleitung wurde angedeutet, daß man zwei Arten von Clustern unterscheiden kann. Die hier als "objektbezogen" bezeichneten Cluster entsprechen im wesentlichen den "klassischen" Clustern. Sie werden verwendet, um Records entsprechend ihrer Objekt- oder Subobjektzugehörigkeit zusammenzufassen. Dazu wird für jede Ausprägung eines Objekt- oder Subobjekttyps ein (oder mehrere) Cluster eines (oder mehrerer) Cluster-Typs angelegt. In diesen Clustern werden die Records des Objektes oder Subobjektes gespeichert. Beispielsweise werden nachfolgend für jeden Roboter je ein "Sec_Rec_Cluster-i" und ein "Prim_Rec_Cluster-j" der Cluster-Typen "Sec_Rec_Cluster" und "Prim_Rec_Cluster" angelegt. In diesen werden die Records der Typen "Sec_Rec", "Achs_Rec", "Eff_Rec" und "Bes_Rec" entsprechend ihrer Objektzugehörigkeit eingefügt (vgl. ▒▒-Schattierung in Abb. 10). Im Gegensatz dazu werden bei der als "objektübergreifende" Clusterung bezeichneten Methode die Records eines oder mehrerer Record-Typen unabhängig von ihrer Objektzugehörigkeit gemeinsam in einem Cluster gespeichert. Die ▒▒-Schattierung in Abb. 10 deutet beispielsweise an, daß alle "Anker_Rec"- und "Prim_Rec"-Records in dem "Anker-Cluster" und alle "Pointer_Rec"-, "Produkt_Rec"- und "Leist_Rec"-Records in dem "Costs-Cluster" gespeichert werden.

Kombiniert man beide Methoden wie in Abb. 10, so ist es möglich, abhängig von der Anwendung gewisse Teile eines Objektes objektbezogen, andere objektübergreifend zu clustern. Zur Definition der Clusterung wird im folgenden, wie schon bei der Definition der Speicherungsstrukturen, wieder nur eine einfache Syntax verwendet. Sie besteht im wesentlichen aus einem Term mit zwei Alternativen (siehe Abb. 7).

4.2.1 Objektbezogene Cluster

Bei der objektbezogenen Clusterung werden Records entsprechend ihrer Objekt- oder Subobjektzugehörigkeit zusammengefaßt. Für jede Ausprägung eines Objekt- oder Subobjekttyps wird ein Cluster angelegt. Um diesen Zusammenhang zwischen Objekten und Clustern syntaktisch auf der Record-Ebene ausdrücken zu können, verwenden wir sogenannte "objektbezogene Cluster-Typen". Jeder objektbezogene Cluster ist eine Ausprägung eines solchen Typs. Zur Identifizierung der einzelnen Ausprägungen verwenden wir sogenannte "identifizierende" Records (identifying records). Die Idee ist dabei, daß alle Records eines Clusters direkt oder indirekt von dem "Identifying"-Record des Clusters abhängen müssen. Um einen objektbezogenen Cluster-Typ zu definieren, wird die erste Variante der Cluster-Definition (siehe Abb. 7) verwendet:

```
cluster_definition = object_cluster_type (cluster_type_name    = cluster_type_name,
                                          segment              = segment_name,
                                          identifying_records  = record_type_name,
                                          member_records       = list_of_record_types)
```

Durch diesen Term wird ein Cluster-Typ mit dem Namen "cluster_type_name" definiert. Die Ausprägungen (= objektbezogene Cluster) des Typs werden in dem Segment "segment_name" angelegt. Diese Cluster werden durch Records identifiziert, deren Typ im Parameter "identifying_records" gegeben ist. Die Member-Klausel gibt an, welche Records welcher Record-Typen in diesen Clustern gespeichert werden. Diese Typen müssen dabei in einer direkten oder indirekten Eltern-Kind-Beziehung zu den "Identifying"-Records stehen. Außerdem können die "Identifying"-Records selbst Member-Records sein[5].

Das Verfahren der objektbezogenen Clusterung sei am Beispiel der Roboterrelation verdeutlicht. Entsprechend der Einleitung sollen die Achs- und Effektordaten jeweils in einem Cluster pro Roboter zusammengefaßt werden. Betroffen hiervon sind die Record-Typen "Sec_Rec", "Achs_Rec" und "Eff_Rec". Die entsprechende Definition des Cluster-Typs lautet:

```
object_cluster_type (cluster_type_name    = Sec_Rec_Cluster
                     segment              = Main,
                     identifying_records  = Sec_Rec,
                     member_records       = (Sec_Rec, Achs_Rec, Eff_Rec))
```

Für jeden Record vom Typ "Sec_Rec" wird hierdurch im Segment "Main" ein Cluster angelegt. In Abb. 10 sind diese Cluster mit "Sec_Rec_Cluster-1" und "Sec_Rec_Cluster-2" bezeichnet. In diesen Clustern werden die Records der Typen "Sec_Rec", "Achs_Rec" und "Eff_Rec" entsprechend ihrer hierarchischen Abhängigkeit von den beiden "Sec_Rec"-Records gespeichert.

---

[5] Da ein "Identifying"-Record zunächst einmal nur als Identifier (über eine systeminterne Referenz) dient, ist es nicht zwingend notwendig, daß die "Identifying"-Records selbst Member-Records sind.

```
cluster_definition = /* Definition eines objektorientierten Cluster-Typs.   */
                     object_cluster_type (cluster_type_name   = cluster_type_name,
                                          segment             = segment_name,
                                          identifying_records = record_type_name,
                                          member_records      = list_of_record_types)|

                     /* Definition eines objektübergreifenden Clusters.     */
                     segment_cluster     (cluster_name        = cluster_name,
                                          segment             = segment_name,
                                          member_records      = list_of_record_types)

                       /* Liste von "member"-Record-Typen eines Clusters.    */
list_of_record_types = (record_type_name {, record_type_name}*)

cluster_type_name    = string /* Name eines objektorientierten Cluster-Typs. */

cluster_name         = string /* Name eines objektübergreifenden Clusters.   */

segment_name         = string /* Name eines Segmentes.                       */
```

Abb. 7: Term zur Zuordnung von Record-Typen zu Clustern

Bei der objektbezogenen Clusterung können Records einer beliebigen Hierarchiestufe als "Identifying"-Records verwendet werden. Damit können Records beliebiger Subobjekte geclustert werden. Außerdem ist es nicht notwendig, daß die "Identifying"-Records selbst in den Clustern (s. Fußnote 5), die sie identifizieren, gespeichert werden. Dies sei ebenfalls an einem Beispiel verdeutlicht:

```
object_cluster_type (cluster_type_name   = Prim_Rec_Cluster
                     segment             = Secondary,
                     identifying_records = Prim_Rec,
                     member_records      = (Bes_Rec))
```

Hier wird festgelegt, daß die "Bes_Rec"-Records in Clustern, die durch die "Prim-Rec"-Records identifiziert werden, im Segment "Secondary" gespeichert werden (vgl. Abb. 10 "Prim_Rec_Cluster-1" und "-2"). Die "Prim-Rec"-Records selbst werden aber, wie im folgenden definiert, in einem objektübergreifenden Cluster gespeichert werden.

4.2.2 Objektübergreifende Cluster

Objektübergreifende Cluster werden verwendet, um Records unabhängig von ihrer Objekt- und Subobjektzugehörigkeit in Clustern zusammenzufassen. Im Gegensatz zu objektbezogenen Clustern gibt es dabei stets nur eine Ausprägung eines objektübergreifenden Cluster(-Typs). Daher lassen sich diese Cluster auch durch eindeutige Namen identifizieren. Da es in einem Segment keine zwei objektübergreifenden Cluster mit demselben Namen gibt, werden sie im folgenden auch als segmentbezogene Cluster bezeichnet (vgl. Abb. 7). Ihre Verwendung sei ebenfalls am Beispiel der Roboterrelation näher erläutert.

In der Einleitung wurde angenommen, daß die Einsatzdaten häufig unabhängig von den Robotern zugegriffen werden. Es erscheint daher sinnvoll, alle Records der Typen "Pointer_Rec", "Produkt_Rec" und "Leist_Rec" unabhängig von ihrer Objektzugehörigkeit in einem Cluster zusammenzufassen. Dies wird durch den folgenden Term ausgedrückt. Alle Records der drei Typen werden hierdurch im Segment "Main" im Cluster "Costs_Cluster" (vgl. Abb. 10) gespeichert.

```
segment_cluster (cluster_name   = Costs_Cluster,
                 segment        = Main,
                 member_records = (Pointer_Rec, Produkt_Rec, Leist_Rec))
```

Ein anderes Beispiel sind die Records der Typen "Anker_Rec" und "Prim_Rec". Jeder Zugriff auf ein Objekt führt über diese Records. Daher sollen diese Records in einem - hier "Anker_Cluster" genannten (vgl. Abb. 10) - Cluster zusammengefaßt werden:

```
segment_cluster (cluster_name   = Anker_Cluster,
                 segment        = Main,
                 member_records = (Anker_Rec, Prim_Rec))
```

Mit dieser letzten Definition ist die Clusterung der Roboterrelation nunmehr vollständig beschrieben. Die Einsatzdaten der Roboter werden objektübergreifend in einem Cluster zusammengefaßt. Im Gegensatz dazu werden die Achs- und Effektordaten jeweils roboterbezogen geclustert. Hierdurch unterscheidet sich die Speicherungsstruktur von der sonst oft nur angebotenen (und auch hier möglichen) rein objektbezogenen Clusterung.

5. Zusammenfassung und Ausblick

In diesem Beitrag wurden zwei Freiheitsgrade - Wahl einer Konstruktordatenstruktur zur Implementation von Mengen, Listen und Tupeln, Entscheidung, ob Elemente bzw. Attribute materialisiert oder referenziert gespeichert werden - beim Entwurf von Speicherungsstrukturen für komplexe Objekte herausgearbeitet. Darauf aufsetzend wurden drei orthogonale Parameter entwickelt, die geeignet sind, diese Freiheitsgrade und damit die systeminterne Repräsentation komplexer Objekte zu definieren. Mit diesen Parametern kann die Aufteilung eines komplexen Objektes auf Records explizit kontrolliert werden. Es ist möglich, häufig gemeinsam zugegriffene Substrukturen komplexer Objekte in einem einzigen Record und damit in einer Zugriffseinheit zusammenzufassen. Umgekehrt können selten gemeinsam benötigte Substrukturen separat gespeichert werden. Die logische Struktur der Objekte ist hiervon stets unbeeinflußt. Anfragen können unabhängig von der gewählten Speicherungsstruktur formuliert werden. Die Wahl einer geeigneten oder ungeeigneten Speicherungsstruktur hat nur Auswirkungen auf die Performanz des Systems.

Für eine gute Performanz eines Systems ist auch eine sinnvolle Clusterung der Records notwendig. Sonst ist im ungünstigsten Fall für jeden Zugriff auf einen Record ein Plattenzugriff erforderlich. Im zweiten Teil dieses Beitrages wurde daher vorgestellt, wie die Clusterung der Records explizit definiert werden kann. Records können dabei objekt- bzw. subobjektbezogen, aber auch objektübergreifend zu Clustern zusammengefaßt werden. So können häufig gemeinsam zugegriffene Records wahlweise abhängig oder unabhängig von ihrer Objektzugehörigkeit gemeinsam in den gleichen Seiten des Hintergrundspeichers abgelegt werden. Die sonst übliche rein objektbezogene Clusterung wird damit durchbrochen.

Eine geeignete Speicherung komplexer Objekte ist allein jedoch noch kein Garant für eine gute Performanz. Zusätzlich muß auch die Optimierung von Anfragen auf die Objekt- und Speicherungsstrukturen abgestimmt sein. Dazu haben wir in [KeDa91] unterschiedliche Methoden zur Auswertung von Anfragen an komplex strukturierte Objekte mit Indexen diskutiert. Derzeit entwickeln wir Kostenformeln, um die hierbei generierten alternativen Anfragepläne in Abhängigkeit von der Speicherungsstruktur der Objekte bewerten zu können.

6. Literatur

Banc88 F. Bancilhon, G. Barbedette, V. Benzaken, C. Delobel, S. Gamerman, C.Lécluse, P. Pfeffer, P. Richard, F. Velez: *The Design and Implementation of O_2, an Object-Oriented Database System.* K.R. Dittrich (Ed.), Advances in Object-Oriented Database Systems, Proc. 2nd Int. Workshop on Object-Oriented Database Systems, Bad Münster, Lecture Notes in Computer Science 334, Springer-Verlag, pp. 1-22, 1988.

BeDe89 V. Benzaken, C. Delobel: *Dynamic Clustering Strategies in the O_2 Object-Oriented Database System. Altair*, BP105, 78153 Le Chesnay Cedex, France, pp. 1-27, 1989.

Dada86 P. Dadam, K. Kuespert, F. Andersen, H. Blanken, R. Erbe, J. Guenauer, V. Lum, P. Pistor, G. Walch: *A DBMS Prototype to Support Extended NF^2 Relations: An Integrated View on Flat Tables and Hierarchies.* ACM-SIGMOD, Proc. Int. Conf. on Management of Data Washington, D.C., pp. 356-367, 1986.

DPS86 U. Deppisch, H.-B. Paul, H.-J. Schek: *A Storage System for Complex Objects.* K. Dittrich, U. Dayal (Eds.), Proc. Int. Workshop on Object-Oriented Database Systems, Pacific Grove, pp. 183 - 195, 1986.

Gee77 W.C. McGee: *The information management system IMS/VS: Data base facilities.* IBM Systems Journal, Vol. 16, No. 2, pp. 96-123, 1977.

HaOz88 A. Hafez, G. Ozsoyoglu: *Storage Structures for Nested Relations.* IEEE Data Engineering, Vol 11, No. 3, Special Issue on Nested Relations, pp. 31 - 38, 1988.

HMMS87 T. Härder, K. Meyer-Wegener, B. Mitschang, A. Sikeler: *PRIMA - a DBMS Prototype Supporting Engineering Applications.* Proc. 13th Int. Conf. on Very Large Data Bases, Brighton, pp. 433 - 442, 1987.

Ingr90 *INGRES/Database Administrator's Guide.* Release 6.3, 1990.

KeDa91 U. Keßler, P. Dadam: *Auswertung komplexer Anfragen an hierarchisch strukturierte Objekte mit Pfadindexen.* H.-J. Appelrath (Ed), Proc. Datenbanksysteme in Büro, Technik und Wissenschaft, GI-Fachtagung, Springer-Verlag, Informatik-Fachberichte 270, pp. 218-237, 1991.

KFC90 S. Khoshafian, M.J. Franklin, M.J. Carey: *Storage Management for Persistent Complex Objects.* Information Systems, Vol. 15, No. 3, pp. 303-320, 1990.

Kim87 W. Kim, J. Banerjee, H.-T. Chou, J.F. Garza, D. Woelk: *Composite Object Support in an Object-Oriented Database System.* Proc. Int. Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), pp. 118-125, 1987.

Kim89 W. Kim, N. Ballou, H.-T. Chou, J.R. Garza, D. Woelk: *Features of the ORION Object-Oriented Database System.* W. Kim, F.H. Lochovsky (Eds.), Object-Oriented Concepts, Databases, and Applications, ACM Press, Frontier Series, pp. 251-282, 1989.

Lori85 R. Lorie, W. Kim, D. McNabb, W. Plouffe, A.Meier: *Supporting Complex Objects in a Relational System for Engineering Databases.* W. Kim, D.S. Reiner, D.S. Batory (Eds.), Query Processing in Database Systems, Topics in Information Systems, Springer-Verlag, pp. 145-155, 1985.

Mits88 B. Mitschang: *The Molecule-Atom Data Model.* T. Härder (Ed.), The PRIMA Project Design and Implementation of a Non-Standard Database System, University Kaiserslautern, Report No. 26/88, Erwin-Schrödinger-Straße, 6750 Kaiserslautern, Germany, pp. 13-36, 1988.

MSOP86 D. Maier, J. Stein, A. Otis, A. Purdy: *Development on an Object-Oriented DBMS.* Proc. Int. Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), pp. 472-482, 1986.

Nier89 O. Nierstrasz: *A Survey of Object-Oriented Concepts.* W. Kim, F.H. Lochovsky (Eds.), Object-Oriented Concepts, Databases, and Applications, ACM Press, pp. 3-21, 1989.

Orac90 *Oracle RDBMS Database Administrator's Guide*, Version 6.0, 1990.

Paul87 H.-B. Paul, H.-J. Schek, M. H. Scholl, G. Weikum, U. Deppisch: *Architecture and Implementation of the Darmstadt Database Kernel System.* ACM-SIGMOD, Proc. Int. Conf. on Management of Data, San Francisco, USA, pp. 196-207, 1987.

PiAn86 P. Pistor, F. Andersen: *Designing a Generalized NF2 Model with an SQL-Type Language Interface.* Proc. 12th Int. Conf. on Very Large Data Bases, Kyoto, Japan, pp. 278-285, 1986.

Scho92 M. H. Scholl: *Physical Database Design for an Object-Oriented Database System.* J.-C. Freytag, G. Vossen, D.E. Maier (Eds.), Query Processing for Advanced Database Applications, Morgan Kaufmann, to appear, 1993.

ScPi82 H.-J. Schek, P. Pistor: *Data Structures for an Integrated Data Base Management and Information Retrieval System.* Proc. Int. Conf. on Very Large Data Bases, Mexico City, pp. 197-207, 1982.

ScSc90 M. H. Scholl, H.-J. Schek: *A relational object model.* Proc. Int. Conf. on Database Theory (ICDT), Paris, Springer-Verlag, Lecture Notes in Computer Science 470, pp. 89-105, 1990.

ScSi89 H. Schöning, A. Sikeler: *Cluster Mechanisms Supporting the Dynamic Construction of Complex Objects.* Proc. 3rd Int. Conf. on Foundations of Data Organization and Algorithms (FODO), Paris, Springer-Verlag, Lecture Notes in Computer Science 367, pp. 31-46, 1989.

Sike88 A. Sikeler: *Key Concepts of the PRIMA Access System.* T. Härder (Ed.), The PRIMA Project Design and Implementation of a Non-Standard Database System, University Kaiserslautern, Rep.-No. 26/88, Erwin-Schrödinger-Straße, 6750 Kaiserslautern, Germany, pp. 69-99, 1988.

SPS87 M. H. Scholl, H.-B. Paul, H.-J. Schek: *Supporting Flat Relations by a Nested Relational Kernel.* Proc. 13th Int. Conf. on Very Large Data Bases, Brighton, England, pp. 137-146, 1987.

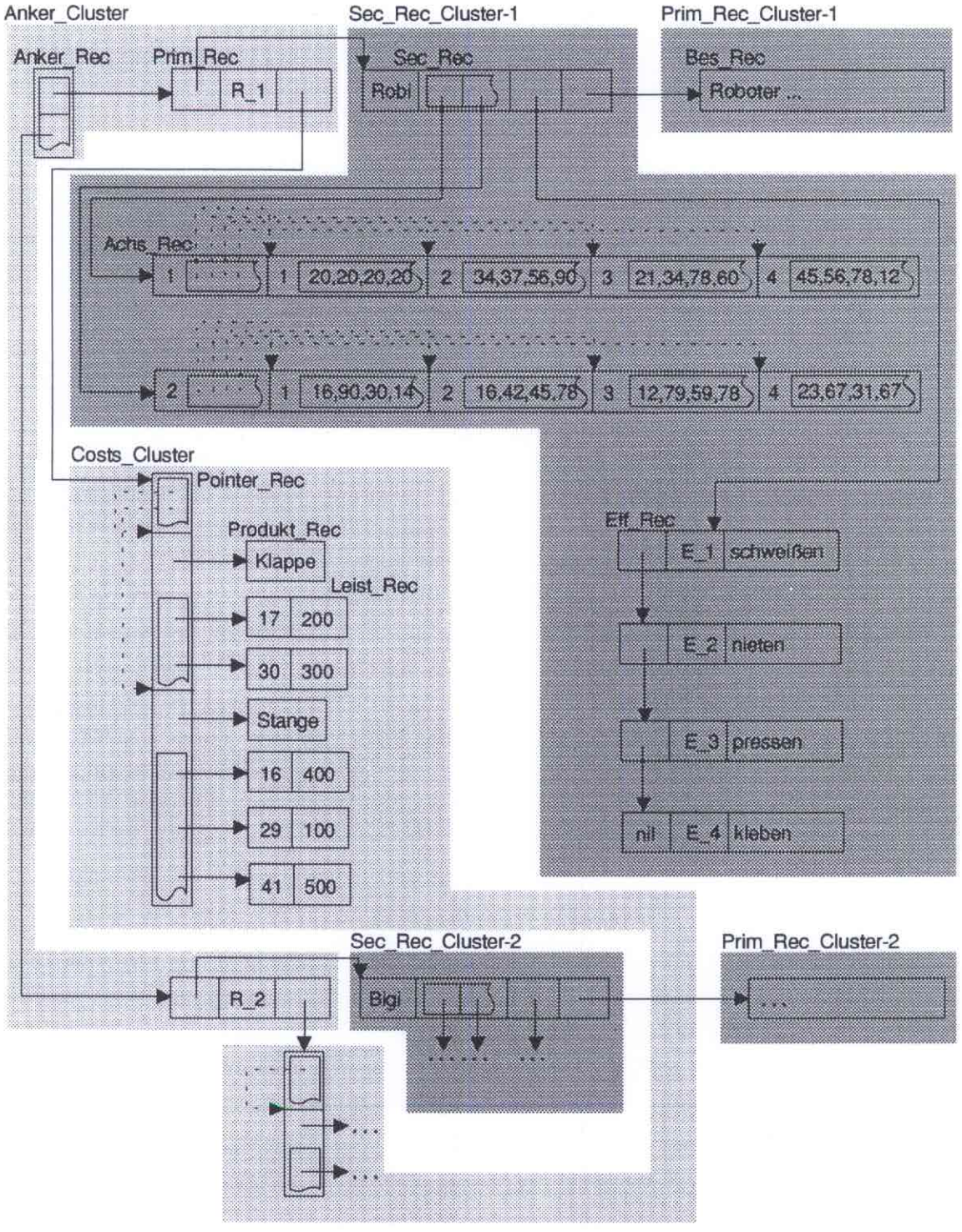
Anhang: Abbildungen

```
complex_object Roboter [anchor_record_type=Anker_Rec]
 set [implementation=array, element_placement=referenced(Prim_Rec)] of
  tuple(
   R_Nr    [location=primary, element_placement=inplace]: integer,                    [1]

   Name    [location=secondary(Sec_Rec), element_placement=inplace]: fix_string(30),  [2]

   Achsen [location=secondary(Sec_Rec), element_placement=inplace]:                   [3]
     list [implementation=array, element_placement=referenced (Achs_Rec)] of          [4]
          tuple (                                                                     [5]
           Achs_Nr          [location=primary, element_placement=inplace]: integer, [6]
           Positionsmatrix [location=primary, element_placement=inplace]:           [7]
            list [implementation=array, element_placement=inplace] of               [8]
                 tuple (                                                             [9]
                  Reihe  [location=primary, element_placement=inplace]: integer,     [10]
                  Vektor [location=primary, element_placement=inplace]:              [11]
                   list [implementation=array,element_placement=inplace] of integer)), [12]

   Einsatz [location=primary, element_placement=referenced (Pointer_Rec)]:           [13]
    set [implementation=array, element_placement=inplace] of
        tuple (
         Produkt  [location=primary, element_placement=referenced (Produkt_Rec)]: [14]
                   fix_string(30),
         Leistung [location= primary, element_placement=inplace]:
           set [implementation=array, element_placement=referenced (Leist_Rec)] of [15]
               tuple (
                Woche  [location=primary, element_placement=inplace]: integer,
                Kosten [location=primary, element_placement=inplace]: integer)),

   Effektoren [location=secondary(Sec_Rec), element_placement=referenced (Eff_Rec)]: [16]
    set [implementation=linked_list, element_placement=inplace] of                   [17]
        tuple (
         E_Nr    [location= primary, element_placement=inplace]: fix_string(30),
         Aufgabe [location= primary, element_placement=inplace]: fix_string(30)),

   Beschreibung [location=secondary(Sec_Rec), element_placement=referenced (Bes_Rec)]: [18]
                 var_string))).
```

Abb. 8: Typ und Speicherungsstruktur-Definition der eNF²-Roboterrelation

```
                    /* Definition eines komplexen Objektes. */
complex_object db_object_name [anchor_record_type = record_type_name] object_type [1]

                    /* Name eines komplexen Objektes. */
db_object_name      = string

                    /* Typname für Records mit semantisch äquivalentem Inhalt. */
record_type_name    = string

                    /* Rekursiver Aufbau eines komplexen Objekttyps. */
object_type         = /* Beispiel für atomare Wertebereiche. */                    [2]
                      integer | real | var_string | fix_string(length) |

                      /* Definition einer Menge. */
                      set  [implementation     = implementation_type,             [3]
                            element_placement = placement_type] of object_type |  [4]

                      /* Definition einer Liste. */
                      list [implementation     = implementation_type,             [5]
                            element_placement = placement_type] of object_type |  [6]

                      /* Definition eines Tupels mit einer Liste von Attributen. */
                      tuple (attribute_description {,attribute_description}*)    [7]

                    /* Länge eines Strings mit fester Länge. */
length              = integer

                    /* Definition eines Attributs eines Tupels. */
attribute_description = attribute_name [location         = location_type,        [8]
                                        element_placement = placement_type]: object_type [9]

                    /* Name eines Attributs. */
attribute_name      = string


    /* Parameter zur Definition der Speicherungsstruktur komplexer Objekte */

                    /* Angabe, ob ein Element einer Menge oder Liste bzw. ein */
                    /* Attribut eines Tupels referenziert oder materialisiert */
                    /* gespeichert wird. */
placement_type      = inplace | referenced (record_type_name)                    [10]

                    /* Implementierungstechniken für Mengen und Listen.*/
implementation_type = array | linked_list                                       [11]

                    /* Angabe, ob ein Attribut in dem Primär- oder einem */
                    /* Sekundärblock gespeichert bzw. daraus referenziert wird. */
location_type       = primary | secondary (record_type_name)                    [12]
```

Abb. 9: Vereinfachte Syntax zur Beschreibung komplexer Objekte



Legende: objektbezogene Clusterung = 1 Cluster pro Objekt

objektübergreifende Clusterung = 1 Cluster für alle Records, unabhängig von ihrer Objektzugehörigkeit

Abb. 10: Eine mögliche Speicherungsstruktur für die eNF²-Roboterrelation