Speicherung und Indexierung komplexer Objekte in objektrelationalen Datenbank-Management-Systemen

Dissertation

zur Erlangung des akademischen Grades
Doktor-Ingenieur (Dr.-Ing.)

vorgelegt dem Rat der Fakultät für Mathematik und Informatik
der Friedrich-Schiller-Universität Jena
im Dezember 2005

von
Dipl.-Inf. Steffen Skatulla
geboren am 2. November 1971 in Jena

Gutachter:
1. Prof. Dr.-Ing. Klaus Küspert, Friedrich-Schiller-Universität Jena
2. Prof. Dr.-Ing. Stefan Dessloch, Technische Universität Kaiserslautern
3. Dr.-Ing. Harald Schöning, Software AG, Darmstadt

Tag der letzten Prüfung des Rigorosums: 10. April 2006

Tag der öffentlichen Verteidigung: 28. April 2006

Die im Rahmen der vorliegenden Arbeit durchgeführten praktischen Untersuchungen wurden seit 2002/2003 durch einen *Shared University Research Grant der IBM Deutschland Entwicklung GmbH, Böblingen und des IBM Silicon Valley Lab, San Jose, Kalifornien, USA* unterstützt.

Meiner Familie

Kurzfassung

Die vorliegende Arbeit zur Speicherung und Indexierung komplexer Objekte leistet einen Beitrag zur Verbesserung der Funktionalität und der Datenunabhängigkeit in objektrelationalen Datenbank-Management-Systemen (ORDBMS).

Um die derzeit in ORDBMS vorrangig anzutreffende Abhängigkeit und Verquickung physischer Speicherstrukturen von logischen Datenmodellkonstrukten aufzuheben, wird die Speicherbeschreibungssprache Physical Representation Definition Language (PRDL) eingeführt und damit eine Möglichkeit zur physischen Datenmodellierung und umfassenden Spezifikation von Speicherstrukturen geschaffen. Mit PRDL bietet sich das Potential zur Optimierung physischer Speicherstrukturen, insbesondere für komplexe Objekte mit tiefverschachtelten, strukturierten und kollektionswertigen Attributen, wie sie nun auch SQL:2003 zuläßt, durch die Anpassung an spezielle Datenbestände und Arbeitslasten und somit letztendlich auch zur Verbesserung der Effizienz objektrelationaler DBMS.

Im Verlauf der Arbeit wird nach der Darstellung der Grundlagen objektrelationaler DBMS durch die Untersuchung von Speicherungs- und Indexierungskonzepten aus Literatur, DBMS-Prototypen und -Produkten eine auf dem aktuellen Stand der Forschung aufsetzende Basis für die Speicherbeschreibungssprache PRDL gelegt. Es wird ein konsistentes Konzept zur Speicherung und Indexierung komplexer Objekte in ORDBMS erarbeitet, das in die Definition von PRDL einfließt.

Auf der Grundlage dieser Vorarbeiten werden Syntax und Semantik von PRDL ausführlich vorgestellt und der Einsatz dieser Spezifikationsprache anhand von Beispielen erläutert. PRDL bietet einen umfassenden Satz an physischen Modellierungskonzepten, wie zum Beispiel Möglichkeiten,

- ▷ Objekte, Subobjekte, Attribute und ganze Attributstrukturen zu sogenannten physischen Primär- und Sekundärspeichersätzen zuzuordnen,
- ▷ Angaben zur Art und zum Ort der Speicherung dieser Sätze zu machen,
- ▷ für die Verbindung von Speichersätzen verschiedene Referenzarten festzulegen,
- ▷ lokale und globale Indexstrukturen für Objekt- und Subobjektmengen anzulegen und
- ▷ unterschiedlichste Indexstrukturen über eine Menge von Konstruktoren zu erzeugen.

Diese Speicherkonzepte erlauben es, die physische Repräsentation komplexer Objekte an unterschiedlichste Verarbeitungsanforderungen anzupassen.

In der Arbeit werden ebenfalls weitergehende Themen untersucht, die sich aus der Möglichkeit zur Reorganisation der Objektspeicherung ohne Beeinflussung der logischen Modell- und Anwendungsebene ergeben. Die Diskussion der Verarbeitung komplexer Objekte zeigt, daß heutige relationale DBMS beste Voraussetzungen für eine vollständige Implementation der in dieser Arbeit vorgestellten Konzepte mitbringen und dazu nur wenige Erweiterungen erfordern. Zur Optimierung der Speicherung komplexer Objekte wird weiterhin ein Kostenmodell in Grundzügen vorgestellt und durch die Gegenüberstellung mit Oracle und mit Simulationsergebnissen validiert. Anhand von Simulationsergebnissen wird insbesondere verdeutlicht, welches Potential in der Optimierung der Speicherstrukturen liegt. Die weiterführenden Untersuchungen weisen zudem einen Weg zu einer zukünftigen automatischen Optimierung und Reorganisation der Speicher- und Indexstrukturen für komplexe Objekte in ORDBMS (Schlagwort „self-tuning“).

Durch die Ergebnisse dieser Arbeit wird ein gangbarer Pfad zur Verbesserung der Datenunabhängigkeit, zur Effizienzsteigerung, zur besseren praktischen Benutzbarkeit und damit zur Beseitigung wichtiger Hindernisse beim Einsatz objektrelationaler Datenbank-Management-Systeme aufgezeigt.

Danksagung

Bei der Entstehung dieser Dissertation wurde ich von einigen Menschen ganz wesentlich unterstützt, diesen möchte ich an dieser Stelle herzlich danken.

Der erste Dank gebührt meinem Doktorvater Klaus Küspert für die wertvolle Förderung meiner Arbeit. Er hat mich als mein Professor und mein Chef nicht nur in der Doktorandenzeit begleitet, sondern auch meinen Weg vom Studium über Industrie- und Auslandspraktika bis hin zu meiner jetzigen Tätigkeit entscheidend mitgeprägt.

Mein Dank gilt auch Stefan Dessloch und Harald Schöning für die freundliche Übernahme der Gutachten.

Eine Reihe von Studenten haben durch die Bearbeitung von Teilaspekten in Studien- und Diplomarbeiten zu dieser Arbeit beigetragen. Hier ist an erster Stelle Felix Kissel zu nennen, mit dem die enge Kooperation sehr fruchtbar war und mir viel Freude bereitet hat, sowie Kay Schwarz, Christian Kauhaus, Nico Lachmann, Rene Schmidt, Matthias Walther, Daniel Klan und Roccy Marhold.

Gleichfalls möchte ich meinen Kollegen Jens Lufter, Jan Nowitzky, Christoph Gollmick, Knut Stolze, Thomas Müller, Stefan Dorendorf, Frank Hüsemann, Klaus Friedel und Frank Mäurer für die anregende und angenehme Zusammenarbeit danken.

Ein besonderer Dank gilt Christian und Ivonne Erfurth, deren Freundschaft mich durch die gemeinsamen Studien, Industriepraktika und die Doktorandenzeit begleitet hat und die mir insbesondere in der Schlußphase der Schreibarbeiten große Unterstützung und zeitweilig sogar ‚Schreib-Asyl' gewährt haben.

Eine Anzahl weiterer Menschen hat diese Arbeit und meinen Weg kontinuierlich oder punktuell begleitet, dafür möchte ich ihnen danken. Stellvertretend denke ich hierbei besonders an Peter Pistor, der mich nicht nur seit nunmehr zehn Jahren immer wieder unterstützt, sondern auch meinen Gastaufenthalt bei IBM in Kalifornien mitermöglicht hat.

Auch möchte ich Detlef Hornbostel danken, der als mein Vorgesetzter bei der Ibykus AG durch sein Verständnis und Entgegenkommen bei der Arbeitszeit- und Urlaubsregelung die Fertigstellung der Arbeit ebenfalls mitbefördert hat.

„Last but not least" möchte ich meiner Familie herzlich danken. Meine Eltern Ruth und Lutz Hummel sowie meine Frau Almut und mein Sohn Johann haben mir stets hilfreich, liebe- und verständnisvoll zur Seite gestanden. Ihnen sei diese Arbeit gewidmet.

Mühlhausen, Mai 2006

Steffen Skatulla

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Syntaxregeln

Kapitel 1

Einleitung

Das Thema dieser Dissertationsschrift ist die

Speicherung und Indexierung komplexer Objekte
in objektrelationalen Datenbank-Management-Systemen.

Dieses erste Kapitel soll die Beschäftigung mit dem Thema einleiten, zur Motivation die Herkunft der Fragestellung darlegen und erklären, warum sich eine wissenschaftliche Auseinandersetzung damit lohnt.

1.1 Einführung und Motivation

1.1.1 Entstehung objektrelationaler DBMS

Die ersten Datenbank-Management-Systeme (DBMS) entstanden vor über 35 Jahren aus dem Bedürfnis heraus, Daten über mehrere Anwendungsprogramme hinweg konsistent und dauerhaft zu halten. Dazu mußten die Datenstrukturen unabhängig von den Anwendungen strukturierbar und modellierbar sein. Es entwickelten sich somit parallel zu den Konzepten der Anwendungsprogrammierung eine Reihe von Datenmodellen, die unterschiedlichen Paradigmen folgten.

Die anfänglich wenig strukturierte Ablage von Daten in Dateien der ‚Vor-DBMS-Zeit' wurde von der hierarchischen Datenmodellierung abgelöst, die eine anwendungsunabhängigere Datenrepräsentation ermöglichte und so ein grundlegendes Maß an Datenunabhängigkeit gewährte. Danach beziehungsweise teils parallel erfolgte die Modellierung in Form von vernetzten Strukturen, so daß Verbindungen in komplexeren Datensachverhalten direkter modellierbar wurden. Und die Entwicklung des Entity-Relationship-Modells gestattete zudem den DBMS- und datenmodellunabhängigen Entwurf.

Einen wesentlichen Qualitätssprung bei der Datenunabhängigkeit bildeten die relationalen Datenbanken (RDBMS), deren Datenmodell erstmals mathematisch fundiert war und die eine deskriptive Datenmanipulationssprache (SQL) hervorbrachten. Im folgenden gab es eine Vielzahl von Versuchen, Datenmodelle und Datenbank-Management-Systeme semantisch anzureichern, Techniken aus der künstlichen Intelligenz sowie temporale und aktive Mechanismen zu integrieren und viele andere Ansätze, die jedoch nur begrenzt Eingang in die praktische Anwendung von Datenbanken fanden. Eine große Reihe von Arbeiten etwa in den 1980er Jahren reagierten auf die immer komplexer werdenden Einsatzszenarien von

DBMS und brachten Modelle und Techniken zur Repräsentation komplexer, verschachtelter und teils vernetzter Datenstrukturen, wie das MAD-, das NF^2- und das eNF^2-Modell hervor.

Seit nun schon über zehn Jahren gibt es schließlich Ansätze – auch verstärkt in Produkten –, auf den Siegeszug der Objektorientierung in der Softwareentwicklung zu reagieren und objektorientierte Konzepte in Datenbanken zu etablieren. Es entstanden etliche Objektdatenmodelle und objektorientierte Datenbanksysteme (OODBMS), die sich jedoch insbesondere in geschäftskritischen Anwendungsbereichen nicht durchsetzen konnten. Einen Versuch zur Verschmelzung von objektorientierten und relationalen Techniken stellen seit circa 1995 die objektrelationalen DBMS (ORDBMS) dar.

Allerdings erfüllt die Entwicklung und derzeitige Verbreitung objektrelationaler DBMS nicht die anfänglich großen Erwartungen. Ob das durch die noch immer fehlende vollständige Umsetzung der Konzepte, die weiterhin mangelnde Normkonformität, die Defizite in der Normierung, die nicht großflächige Anwendung oder aber durch den ‚Teufelskreis' aus allen diesen Problematiken bedingt ist, kann wohl nur schwer bestimmt werden. Jedoch scheint sich in den letzten Jahren zumindest eine Nutzung der objektrelationalen Technologie in speziellen Bereichen, wie zum Beispiel bei der Verarbeitung geographischer Informationen und von XML-Daten in DBMS, durchgesetzt zu haben; wenn nicht sogar von einer, allerdings eher langsamen, Verbreitung des Einsatzes von ORDBMS insgesamt gesprochen werden kann.

1.1.2 Komplexe Objekte in DBMS

Die Entwicklung von Anwendungsprogrammen folgt heutzutage weitgehend den Konzepten der Objektorientierung. Software wird dabei in der Form von Objekten (Instanzebene) beziehungsweise Klassen (Typebene) modelliert und programmiert. Verschachtelte komplexe Objekte und vernetzte Objektstrukturen dienen dabei der Modellierung und Repräsentation komplexer Sachverhalte.

Bei DB-Anwendungen gibt es folgende drei grundlegende Ansätze zur Speicherung und Verarbeitung solcher komplexer Objektstrukturen in Datenbanken:

Der am weitesten verbreitete Ansatz ist die Kopplung von objektorientierten Anwendungen mit relationalen DBMS, entweder direkt oder über sogenannte Persistenzschichten. Dabei müssen jedoch die Objektstrukturen auf Relationen und die oft navigierende Anwendungslogik auf SQL-Anfragen abgebildet werden. Durch den damit teils verbundenen Trend, RDBMS als ‚dumme' Datenspeicher zu betrachten, entstehen Load-and-Store-Anwendungsprogramme, die die Fähigkeiten der DBMS zu effizienten Massenoperationen nur bedingt ausnutzen.

Ein zweiter Ansatz ist die Kopplung der objektorientierten Anwendungen mit objektorientierten DBMS. Dabei ist zwar der zu überwindende Graben zwischen Anwendungs- und Datenbankmodell (Impedance Mismatch) sehr gering, aber OODBMS haben sich aus den unterschiedlichsten, in Abschnitt 1.1.1 bereits kurz angesprochenen Gründen nicht durchgesetzt.

Der dritte Ansatz ist die Kopplung von objektorientierten Anwendungen mit objektrelationalen DBMS, die ebenfalls versuchen, den Impedance Mismatch zu verringern. Die Hoffnung dabei ist, daß durch die entsprechenden Objektkonstrukte zumindest die datenintensiven Teile der Anwendungslogik wieder in die Datenbank „zurückgeholt" und dort effizient ausgeführt werden können. Gleichzeitig sollen dadurch auch neue Anwendungsfelder mit „von Natur aus" komplexen Datenstrukturen, wie zum Beispiel Geometriedaten,

effizient von ORDBMS verwaltet und verarbeitet werden können.

Eine umfassende Menge erweiterter Modellierungskonstrukte für ORDBMS wurde unter der Bezeichnung SQL$^+$ in [Luf02b] vorgeschlagen und mittlerweile auch weitgehend in die SQL:2003-Norm aufgenommenen. Durch sie ist es einerseits möglich, Teile des objektorientierten Anwendungsentwurfs mehr oder weniger direkt in eine Datenmodellierung für die Datenbank zu übersetzen. Andererseits wird auch eine eigenständige objektorientierte Datenmodellierung in der Datenbank ermöglicht. Diese gestattet sowohl anwendungsübergreifende Objektmodelle als auch alternative Abbildungen von Anwendungs- auf Datenbankobjekte.

1.1.3 Trennung von logischem und physischem Entwurf in ORDBMS

Die Integration objektorientierter Modellkonstrukte in ORDBMS führte dazu, daß physische Speicherformen in der Regel direkt an logische Datenstrukturen gebunden wurden. Beispielsweise wird in Oracle durch die Verwendung der logischen Kollektionsstruktur Array auch die dichte Speicherung aller Kollektionselemente direkt im physischen Satz impliziert. Eine Auslagerung der Elemente in separate physische Sätze, die dann auch zum Beispiel mittels B*-Baum für einen schnellen Suchzugriff indexiert werden könnten, ist nicht möglich. Dafür ist eine Änderung der logischen Struktur, etwa in eine Nested Table, notwendig. Allerdings ändert sich damit auch der logische Entwurf und dementsprechend auch der Datenzugriff von der Anwendungsebene aus, denn die Elemente einer Nested Table lassen sich zum Beispiel nicht mehr direkt über einen Feldindex ansprechen.

An diesem Beispiel wird deutlich, daß durch die direkte Kopplung von logischen und physischen Datenstrukturen eine mangelnde Datenunabhängigkeit verursacht wird. Zur Verbesserung dieser Situation wird in dieser Arbeit die bewußte Trennung von logischer und physischer Modellierungsebene vorgeschlagen. Diese soll es ermöglichen, physische Speicherstrukturen zu verändern, *ohne* daß dadurch Änderungen auf der logischen Ebene erfordlich werden. Es sollen beispielsweise die Speicherstrukturen von Tabellen anpaßbar sein, ohne daß SQL-Anfragen oder Anwendungsprogramme geändert werden müssen.

Diese Zielstellung erfordert eine Spezifikationsform zur Steuerung der physischen Speicherstrukturen. Dazu gibt es eine Reihe von Möglichkeiten: Einerseits kann SQL mit Annotationen zur Speicherung versehen werden, wie es viele DBMS-Produkte und -Prototypen tun. Oracle erlaubt beispielsweise in der `CREATE-TABLE`-Anweisung Annotationen zum Speicherort, zur Partitionierung, zur Clusterung und so weiter. Andererseits sind auch erweiterte Datenbankkatalogeinträge oder Konfigurationen in Steuerdateien zur direkten Beeinflussung der Speicherstrukturen denkbar. Den geeignetsten Ansatz aus unserer Sicht stellt jedoch der Rückgriff auf das mindestens seit CODASYL-SSDL (CODASYL Storage Structure Definition Language) bekannte Konzept einer eigenen Speicherbeschreibungssprache dar.

Die hier vorgeschlagene Speicherbeschreibungssprache Physical Representation Definition Language, abgekürzt PRDL, erlaubt nicht nur die variable Abbildung logischer Datenstrukturen in für die jeweilige Datenmenge und die darauf auftretende Arbeitslast angepaßte physische Repräsentationsformen, sondern sie gestattet es auch, den Entwurf und den Entwurfsprozeß strikter als bisher möglich in Phasen für die logische Datenmodellierung und die physische Modellierung zu trennen. Die logische Modellierung kann sich dann gezielter um eine ‚saubere' Umsetzung der Anwendungsanforderungen kümmern, das heißt, sich um Verständlichkeit, Wartbarkeit, Erweiterbarkeit und so weiter bemühen. Und erst danach werden von der physischen Modellierung Aspekte der Anfrage- und Ände-

rungsleistung, des Datenwachstums und viele weitere für den praktischen Betrieb wichtige Aspekte behandelt. Es soll hier nicht behauptet werden, daß eine vollständige Trennung beider Ebenen möglich wäre, denn natürlich wird ein ungünstiger logischer Entwurf immer Auswirkungen auf die spätere Leistungsfähigkeit haben. Aber die möglichst saubere Trennung kann dazu beitragen, sich in den beiden Entwurfsphasen auf die ‚eigentlichen' Fragestellungen zu fokussieren, die Modellierungskomplexität zu reduzieren, und so insgesamt zu einfacher erstellbaren, besser wartbaren und leistungsfähigeren Entwürfen führen.

Eine wesentliche zu erwartende Verbesserung ist auch die Möglichkeit von physischen Reorganisationen zur Leistungserhaltung oder -verbesserung, ohne daß das logische Datenmodell oder Datenbankoperationen in Anwendungsprogrammen angepaßt werden müssen.

Eine weitere Verbesserung betrifft die Wiederverwendbarkeit objektrelationaler Entwürfe. Ein objektrelationaler Klassenentwurf kann durch die Trennung von logischer und physischer Ebene an unterschiedliche Einsatzfälle, Datenbestände und Verarbeitungsanforderungen angepaßt werden, ohne daß sich seine Datenmodellierung oder seine Funktionsimplementierung ändern.

1.2 Zielstellung und Lösungsansatz

Zielstellung dieser Arbeit ist es zuerst, Konzepte und Umsetzungsmöglichkeiten für die Speicherung und Indexierung komplexer Objekte in objektrelationalen DBMS zusammenzutragen und zu analysieren. Weiterhin soll durch die Entwicklung der Speicherbeschreibungssprache PRDL zur Spezifikation der physischen Speicherstrukturen die Trennung von logischer und physischer Datenmodellierung ermöglicht werden. Die Sprache soll dazu auf bekannten Ansätzen zur Speicherung und Indexierung komplexer Objekte aufbauen und diese zu einem ausreichend mächtigen, konsistenten und doch erweiterbaren ‚Werkzeugkasten' zusammenfügen und so zu einer Verbesserung der Datenunabhängigkeit beitragen.

Auf diese Weise soll eine Möglichkeit zur Anpassung der physischen Objektstrukturen an unterschiedliche Verarbeitungsanforderungen und -lasten geschaffen werden. Schließlich soll auch eine Grundlage für den sich abzeichnenden nächsten Schritt hin zur automatisierten Speicherstrukturoptimierung geschaffen werden. Erste Produktansätze hierzu sind in den ‚großen' RDBMS-Produkten bereits zu finden.

1.3 Gliederung der Arbeit

Zunächst werden als Grundlage in Kapitel 2 objektrelationale Konzepte erarbeitet.

Da die Spezifikationssprache PRDL auch die bereits bekannten Speicherstrukturen berücksichtigen muß, folgt in Kapitel 3 eine Aufarbeitung der entsprechenden Literatur, DBMS-Prototypen und -produkte. Dabei wird durch die Untersuchung, den Vergleich und die Einordnung der bekannten Ansätze ein möglichst umfassender und konsistenter Satz an relevanten Speicherkonzepten zusammengetragen, der für die Repräsentation und Verarbeitung komplexer Objekte in ORDBMS geeignet ist.

Gleiches gilt auch für die Indexierung komplexer Objekte. Deshalb werden in Kapitel 4 wiederum aus Literatur, Prototypen und Produkten relevante Indexstrukturen und zugehörige Einsatzkonzepte zusammengetragen und untersucht. Das daraus entstehende Indexierungskonzept bildet dann das zweite Fundament für PRDL.

In Kapitel 5 wird die auf den erarbeiteten Speicher- und Indexstrukturen aufbauende Speicherspezifikationssprache PRDL präsentiert. Es werden ihre Entwurfskonzepte, ihre

Syntax und ihre Semantik vorgestellt und an Beispielen verdeutlicht. Ein etwas vollständigeres Beispiel verdeutlicht am Schluß des Kapitels die Mächtigkeit und den Einsatz der Sprache zur Trennung von logischem und physischem Entwurf und damit zur Verbesserung der Datenunabhängigkeit in ORDBMS.

In Kapitel 6 sind weiterführende Überlegungen zusammengefaßt, die auf den Überlegungen zur separaten physischen Modellierung mit PRDL aufbauen. Dort wird es um die Umsetzung von Operationen auf komplexen Objekten auf der Basis der vorhandenen Anfragebearbeitungstechniken in relationalen beziehungsweise objektrelationalen DBMS gehen. Es werden Arbeiten zu einem Kostenmodell für solche Operationen vorgestellt, das zusätzlich zur üblichen Anfrageoptimierung auch eine Optimierung der physischen Speicherstrukturen in bezug auf unterschiedliche Datenbestände und vor allem Arbeitslasten erlaubt. Mit einer Simulationsumgebung werden nicht nur die vorgeschlagenen Speicherungs- und Indexierungskonzepte überprüft, sondern auch die Kosten- und Optimierungsüberlegungen validiert. Alle diese Überlegungen führen auf das abschließend präsentierte Ziel einer automatischen Optimierung der Speicherstrukturen durch das ORDBMS selbst hin.

Kapitel 7 beschließt die Arbeit durch eine Zusammenfassung und einen Ausblick auf weitere Aufgabenstellungen.

1.4 Beispielszenario

An vielen Stellen dieser Arbeit werden Beispiele zur Verdeutlichung von bekannten Sachverhalten und neuen Konzepten gegeben. Um ihre Verständlichkeit zu erhöhen, sollen sie sich in der Regel an einem Beispielszenario orientieren, das hier kurz vorgestellt werden soll.

Dabei handelt es sich, dem Thema dieser Arbeit entsprechend, um eine Miniwelt, die die prinzipiellen Möglichkeiten komplexer, verschachtelter Objektstrukturen anschaulich machen soll, ohne selbst zu komplex und umfangreich und damit unverständlich zu werden.

Abbildung 1.1 bis Abbildung 1.3 stellen dieses Szenario graphisch dar. Im Diagramm werden Klassen als Rechtecke und einfache und strukturierte Attribute als einfache Ellipsen dargestellt. Mengenwertige Attribute stehen in doppelten Ellipsen.

Folgende Miniwelt liegt dem Szenario zugrunde:

> Ein Unternehmen produziert eine große Anzahl verschiedener Werkstücke, die der heutigen Zeit entsprechend „on demand“ und „just in time“ für Kunden gefertigt werden. Dazu besitzt das Unternehmen eine Menge von Maschinen, die für verschiedene Bearbeitungsvorgänge eingesetzt werden können und unterschiedliche Fertigungskapazitäten besitzen.

Die Maschinen werden durch die Objektklasse `Fertigungsmaschine` modelliert. Eine Fertigungsmaschine wird durch eine `Maschinenidentifikationsnummer` eindeutig identifiziert und besitzt eine textuelle `Maschinenbezeichnung`. Die Menge von Bearbeitungsvorgängen, die von der Maschine durchführbar sind, werden im mengenwertigen Attribut `Bearbeitungsfähigkeiten` festgehalten. Jede Bearbeitungsfähigkeit wird durch den `Identifikationscode des Bearbeitungsvorgangs` bezeichnet und durch eine bestimmte `Fertigungskapazität` charakterisiert. Außerdem werden der `Standort` der Maschine, der `Maschinenstatus` und ein `Maschinenlogbuch` mit einer Liste von Wartungsmaßnahmen gespeichert.

ABBILDUNG 1.1: Beispielszenario: Fertigungsmaschine

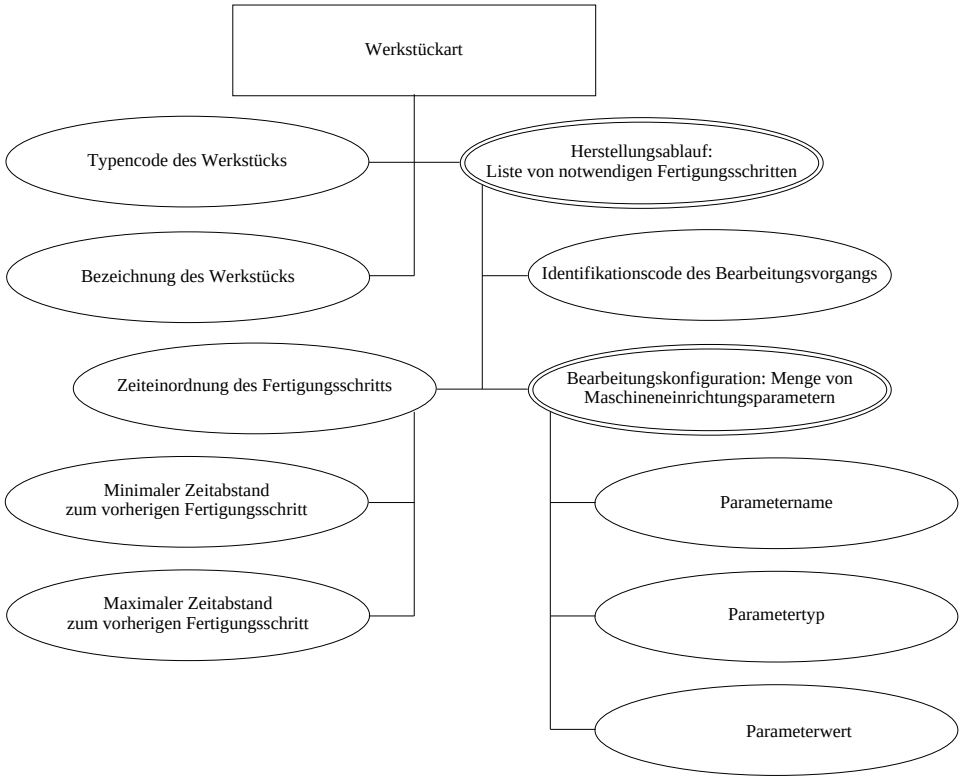


ABBILDUNG 1.2: Beispielszenario: Werkstückart

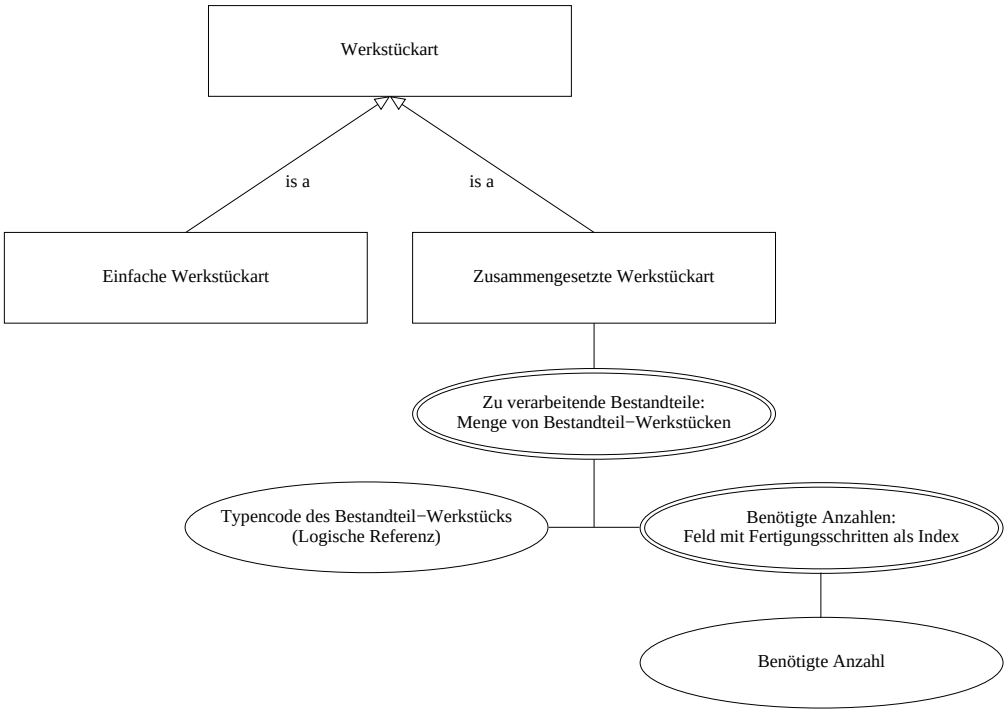


ABBILDUNG 1.3: Beispielszenario: Einfache und zusammengesetzte Werkstückart

Die Objekte der Klasse `Werkstückart` werden durch einen `Typencode` und wieder durch eine textuelle `Bezeichnung` gekennzeichnet. Jede Werkstückart erfordert einen gewissen `Herstellungsablauf`. Das ist eine Liste von notwendigen Fertigungsschritten, die jeweils durch den `Identifikationscode des Bearbeitungsvorgangs`, eine `Zeiteinordnung des Fertigungsschritts` und eine `Bearbeitungskonfiguration` gekennzeichnet sind. Eine Bearbeitungskonfiguration ihrerseits ist eine Menge von Name-Wert-Parametern.

Von der Klasse Werkstückart werden die Klassen `Einfache Werkstückart` und `Zusammengesetzte Werkstückart` abgeleitet. Die einfache Werkstückart besitzt keine zusätzlichen Attribute. Nur die zusammengesetzte Werkstückart fügt das mengenwertige Attribut `Zu verarbeitende Bestandteile` hinzu.

Anhang A listet die Klassen und Attribute im Detail auf und zeigt die Definition entsprechender Typen in SQL$^+$, einer SQL:2003-Erweiterung um vollständige Kollektionsunterstützung, die in [Luf05] definiert wird.

Kapitel 2

Grundlagen objektrelationaler DBMS

Dieses Kapitel soll sich mit den charkteristischen Eigenschaften objektrelationaler Datenbank-Managment-Systeme beschäftigen und so die Grundlagen für die in den nächsten Kapiteln folgenden Untersuchungen zur Speicherung und Indexierung komplexer Objekte legen.

Der große Erfolg relationaler DBMS liegt nicht zuletzt in ihrem einfachen und zugleich sehr mächtigen Ansatz zur Datenmodellierung durch Relationen. Diese Ralationen, die in der Praxis oft auch als Tabellen bezeichnet werden, sind Mengen (in der Praxis auch Multimengen) von Tupeln (oder Datensätzen), die ihrerseits aus benannten Attributen aufgebaut sind (es wird auch von Spalten gesprochen). Diese Attribute haben einfache, skalare Typen, wie zum Beispiel `INTEGER` oder `VARCHAR`, die aus einer Reihe vom RDBMS fest vorgegebener Datentypen gewählt werden können. Über Integritätsbedingungen, die vom Nutzer beziehungsweise Datenbankadministrator definiert werden können, lassen sich gewisse Anforderungen an die Korrektheit der Daten sicherstellen. Gleichzeitig lassen sich auf diesem Wege auch Beziehungen zwischen den Datensätzen unterschiedlicher Tabellen darstellen und überwachen.

Das relationale Datenmodell bietet eine sehr gute Grundlage zur Verwaltung von relaiv einfach (flach) strukturierten Daten, wie sie vorrangig in Verwaltungs- und Betriebswirtschaftsumgebungen auftreten. So lassen sich beispielsweise Stamm-, Buchungs-, Bestands- und andere ähnliche Daten hervorragend speichern, verwalten und mengenorientiert verarbeiten. Das gleiche gilt für die zwischen diesen Daten bestehenden Beziehungen, wie zum Beispiel die Zuordnung von Buchungsdatensätzen zu gewissen Stammdatensätzen.

Allerdings läßt sich seit den 1980er Jahren eine Zunahme an Anwendungen und Szenarien feststellen, die die Speicherung weitaus komplexer strukturierter Daten erfordern. Zunächst kamen solche Anforderungen aus dem Bereich von Ingenieursanwendugen mit dem Wunsch, beispielsweise CAD/CAM-Daten (Computer Aided Design/Manufacturing), das heißt, 3-dimensionale Konstruktionsdaten, in Datenbanken zu verwalten. Mit der starken Verbreitung objektorientierter Softwareentwicklungsmethoden sind jedoch heute komplexe Daten- beziehungsweise Obejektstrukturen fast flächendeckend anzutreffen. Selbst in den bereits angesprochenen Verwaltungsbereichen wurden durch die mehr an der menschlichen Denkweise orientierte objektorientierte Softwareentwicklung die flachen Datenstrukturen von komplexer struturierten Objekten abgelöst.

Durch diese Entwicklung stehen die zu speichernden komplex strukturierten Objekte

aus objektorientierten Anwendungen auf der einen Seite den flachen Strukturen in relationalen Datenbanken auf der anderen Seite gegenüber. Diese Diskrepanz verstärkt ein bereits bekanntes Phänomen, das als Impedance Mismatch bezeichnet wird. Damit sind die konzeptuellen Unterschiede zwischen Anwendungsprogramm und Datenbanksystem in der Datenmodellierung, -repräsentation und -verarbeitung gemeint. Standen sich bisher die satzorientierte und navigierende Arbeitsweise prozeduraler, nicht objektorientierter Anwendungen und die mengenorientierte, deskriptiv gesteuerte Arbeitsweise von RDBMS gegenüber, so verstärkt sich der Impedance Mismatch durch die Diskrepanz zwischen den komplex strukturierten Anwendungsobjekten und den flachen Datenbankstrukturen.

Natürlich können komplexe Objektstrukturen genauso wie bereits vorher vernetzte Anwendungsdatenstrukturen durch Normalisierung beziehungsweise allgemeiner durch eine Abbildung auf relationale Strukturen ‚heruntergebrochen' und damit abgespeichert werden [KLS02]. Allerdings ist eine solche Abbildung mit einer Reihe von Nachteilen verbunden:

- ▷ *Verlust der logischen Zusammengehörigkeit*
  Die logische Zusammengeörigkeit der Daten beziehungsweise Objekte ist auf Datenbankebene nicht mehr vorhanden.

- ▷ *Teure Rekonstruktion*
  Tendenziell ist es sehr teuer, die komplexen Datenstrukturen beziehungsweise Objekte aus den normalisierten Datenbanksätzen zu rekonstruieren.

- ▷ *‚Händische' Abbildung*
  Die Abbildung von komplexen Anwendungsdatenstrukturen beziehungsweise -objektstrukturen auf flache relationale Strukturen ist in der Regel nicht automatisch realisierbar, sondern muß in ‚Handarbeit' entworfen und in entsprechenden Abbildungsroutinen oder -methoden implementiert werden.

- ▷ *Suboptimale DBMS-Nutzung*
  Sehr oft führt die Abbildung von komplexen Anwendungsdatenstrukturen beziehungsweise -objektstrukturen zur suboptimalen Ausnutzung der auf Massendatenbearbeitung hin optimierten Fähigkeiten der DBMS. Indem die Datenbank oft nur als ‚dummer' Ablageort benutzt wird, die Daten beziehungsweise Objekte in der Regel erst in das Programm geladen und dann dort verarbeitet werden, entstehen wenig leistungsfähige Anwendungen, die die sehr guten Fähigkeiten der DBMS zur Massendatenverarbeitung nicht oder schlecht nutzen.

Aus dieser Situation heraus entstand sehr zeitig der Wunsch, komplexe Daten- beziehungsweise Objektstrukturen ‚direkt' in Datenbanken ablegen zu können oder zumindest die Modellierungsmöglichkeiten von DBMS so zu erweitern, daß die Abbildung möglichst einfach erfolgen kann. Im Zuge der Entwicklungen auf dieses Ziel hin entstanden eine Reihe neuer Datenmodelle, DBMS-Prototypen und -Produkte. In Bezug auf komplexe Datenstrukturen wären hier in erster Linie semantische sowie NF^2- und eNF^2-Datenmodelle und DBMS-Prototypen zu nennen [CDF+82, DPS86]. Und in Bezug auf komplexe Objektstrukturen entstanden objektorientierte Datenmodelle, DBMS-Prototypen und auch -Produkte wie Orion, O_2 sowie einige andere [KCB87, KV87, KGBW90, Deu90].

Dennoch konnten sich die wissenschaftlich sehr interessanten und vielversprechenden Ansätze in der Praxis nicht etablieren. Einzig objektorientierte DBMS (OODBMS) erreichten

eine gewisse Verbreitung in speziellen Anwendungsbereichen. Aber auch sie konnten nur sehr selten in die relationale und auch sogar noch vorrelationale Kerndomäne unternehmenskritischer Hochleistungs- und Hochverfügbarkeitsdatenbanken eindringen. Die Gründe hierfür sind vielschichtig:

- *‚Kinderkrankheiten‘*
  Anfänglich hatten neuentwickelte objektorientierte DBMS einen enormen Rückstand in Bezug auf Leistungsfähigkeit, Funktionsumfang und Stabilität gegenüber den seit Jahrzehnten im Einsatz gereiften relationalen (und vorrelationalen) DBMS. Außerdem stellte die zumindenst anfangs fehlende Standardisierung von Schnittstellen und Abfragesprachen ein weiteres Hindernis dar.

- *Fehlende Interoperablität*
  Durch fehlende Interoperablität der neunen objektorientierten DBMS mit den existierenden relationalen (und vorrelationalen) DBMS gibt es größte Schwierigkeiten, die bestehenden riesigen Datenvolumina und vielfältigen Datenbankanwendungen in die neue objektorientierte DBMS-‚Welt‘ einzubinden oder zu migrieren.

- *Mangelnde Verbreitung*
  Die vorgenannten Gründe führten OODBMS in einen gewissen ‚Teufelskreis‘, denn diese Nachteile behinderten einerseits die Verbreitung, und die geringe Verbreitung behindert andererseits das Aufholen des Entwicklungsrückstands im Vergleich zu RDBMS.

Da auf der einen Seite OODBMS sich in der Praxis nicht wirklich etablieren konnten, aber auf der anderen Seite das Problem der Speicherung und Verarbeitung komplexer Objekte in DBMS jedoch bestehen blieb, wurde weiter nach Lösungsmöglichkeiten gesucht. Einen solchen Lösungsvorschlag stellen objektrelationale DBMS dar. Sie versuchen, die neuen objektorientierten Konzepte in relationale DBMS zu integrieren und so durch die Erweiterung bestehender relationaler DBMS zu objektrelationalen DBMS die Vorteile beider Welten miteinander zu verbinden. Diese evolutionäre Weiterentwiklung relationaler DBMS zu objektrelationalen DBMS soll die Nachteile objektorientierter DBMS vermeiden, die ja durch die erfolgten Neuentwicklungen quasi eine ‚revolutionäre‘ Entwicklung darstellen. Diese objektrelationalen Datenbank-Management-Systeme werden in diesem Kapitel näher beleuchtet, um die Grundlagen für die nächsten Kapitel zu legen.

Nach einer kurzen Darstellung der zentralen Grundlagen der Objektorientierung werden die Konzepte objektrelationaler DBMS genauer vorgestellt. Danach wird kurz auf die Integration von ORDBMS in objektorientierte Softwareumgebungen und die damit verbundenen Probleme eingegangen, bevor das altbekannte Thema der Datenunabhängigkeit aufgegriffen und in Bezug auf ORDBMS neu diskutiert wird. An dieser Stelle wird die zentrale Grundüberlegung dieser Arbeit zur Trennung von logischem und physischem Datenbankentwurf eingeführt und erläutert. Abschließend werden die Aspekte der Integration komplexer Objekte in relationale DBMS, das heißt Grundüberlegungen zur Implementierung, besprochen.

2.1 Objektorientierte Konzepte: Objekte und Klassen

Die Entwicklung umfangreicher Softwareprojekte erfordert die Zerlegung des Gesamtsystems in eine Anzahl relativ unabhängiger Einheiten beziehungsweise Module. Ziel ist da-

bei, die Gesamtlösung durch die Lösung überschaubarer Teilprobleme zu erhalten und flexibel neue Einheiten hinzufügen beziehungsweise vorhandene Einheiten modifizieren oder ersetzen zu können (Wiederverwendbarkeit). Der objektorientierte Softwareentwurf geht noch einen Schritt weiter. Prinzipiell will man die Probleme einer real existierenden oder gedachten Welt durch Software modellieren und faßt dazu die Teileinheiten des zu modellierenden Weltaussschnitts als ein System kooperierender Objekte auf.

Ein wichtiges Merkmal dieser Objekte besteht im Vorhandensein einer Identität, die unabhängig vom Zustand beziehungsweise von den Daten dieser Objekte ist. Objekte setzen sich aus einem Methoden- und einem Datenteil (Attribute, Eigenschaften) zusammen. Der Datenteil speichert die zu einem Objekt gehörenden Daten, die zusammengenommen den Zustand des Objekts darstellen. Der Methodenteil definiert das Verhalten eines Objekts und besteht – zumindest bei Objekten, deren Aufgabe primär die Datenhaltung ist – im wesentlichen aus den Methoden zum Verarbeiten und Präsentieren der Attribute des Datenteils.

In beiden Teilen kann unterschieden werden, ob eine Methode beziehungsweise ein Attribut nach außen sichtbar ist oder nur anderen Methoden dieses Objekts zur Verfügung steht. Ein Objekt kapselt demnach seine eigentlichen internen Eigenschaften durch eine Anzahl öffentlicher Methoden oder Attribute ein, die zusammengenommen die Schnittstelle dieses Objekts repräsentieren.

Die Definition von Objekten erfolgt über das Konzept der Klasse: Erstellt man eine Klasse, werden Methoden und Attribute festgelegt. Anschließend können Objekte dieser Klasse erzeugt (instanziert) werden. Klassen können also auch als Objekttyp (auch Object Factory) gesehen werden. Sie definieren die Struktur und das Verhalten eines gewissen Typs von Objekten.

Innerhalb des Objektmodells einer größeren Anwendung kann man nach Jacobson [JCJÖ95] drei Arten von Objekten unterscheiden:

- ▷ *Schnittstellenobjekte*

  Diese Objekte steuern und verwalten die Kommunikation des Systems mit Benutzern und anderen Systemen.

- ▷ *Entitätsobjekte*

  In diesen Objekten werden die persistent abgelegten Informationen der Anwendung gespeichert, weshalb hier auch von Datenobjekten gesprochen werden kann.

- ▷ *Kontrollobjekte*

  Diesen Objekten kommt die Aufgabe einer übergeordneten Steuerung der Anwendung zu (Workflow).

Der Schwerpunkt von Schnittstellen- und Kontrollobjekten liegt in der in ihnen integrierten Programmlogik (Methodenteil), während Entitätsobjekte primär der Repräsentation der zentralen Anwendungsdaten dienen. Somit sind es die Entitätsobjekte, für deren Verwaltung, das heißt Speicherung und Verarbeitung, der Einsatz von Datenbanken sinnvoll ist.

Bei der Abbildung dieser komplexen Daten in erweiterte, objektrelationale Datenbanken ergeben sich von seiten der Softwareentwicklung eine Reihe von konzeptuellen Minimalanforderungen an das Datenbanksystem über die bekannte Funktionalität relationaler DBMS hinaus:

▷ *Benutzerdefinierte Typen*

Es muß die Speicherung komplexer und verschachtelter Datentypen möglich sein. Dies beinhaltet auch eine Unterstützung von mengen- beziehungsweise – als Verallgemeinerung – von kollektionswertigen Attributen.

▷ *Benutzerdefinierte Zugriffs- und Manipulationsfunktionen*

Zu benutzerdefinierten Typen müssen zugehörige benutzerdefinierte Funktionen erstellt werden können.

▷ *Objektidentität und Objektreferenzierung*

Es muß möglich sein, Datenobjekte zu definieren, deren Identität unabhängig von den Werten ihrer Attribute ist. Darüber hinaus müssen Objekte aus anderen Objekten heraus referenzierbar sein.

An Objektidentifikatoren zur eindeutigen Bezeichnung von Objekten sind je nach System und Kontext weitere Anforderungen geknüpft (nach [STS97]):

▷ *Generierung vom System*

Ein Objektidentifikator muß vom System vergeben werden. Diese Bedingung soll verhindern, daß die Erzeugung von Objektidentifikatoren in (beabsichtigter oder unbeabsichtigter) Abhängigkeit der Anwendung geschieht und die Werte der Identifikatoren Semantik enthalten.

▷ *Räumliche Eindeutigkeit*

Hier wird die Frage aufgeworfen, in welchem räumlichen Kontext die Eindeutigkeit der Identifikatoren gewährleistet wird. Mögliche Granulate bei relationalen Datenbanken wären zum Beispiel Relation, Schema, Datenbank(-system) oder sogar ‚mehr'.

▷ *Zeitliche Eindeutigkeit*

Ähnlich wie bei räumlicher Eindeutigkeit geht es hier um den Rahmen, in dem die Identität gültig sein soll. Die schwächste Form der zeitlichen Eindeutigkeit wäre, daß ein Identifikator direkt mit der Lebensdauer des Objekts verknüpft ist. Nach dem Löschen eines Objekts könnte der zugehörige Identifikator wieder bei der Erstellung neuer Objekte eingesetzt werden. Die stärkste Form der Eindeutigkeit wäre, daß ein Identifikator nur ein einziges Mal vergeben wird, unabhängig von der eigentlichen Lebensdauer des Objekts („unique for ever").

▷ *Kapselung*

Der Wert eines Objektidentifikators darf nur dem System bekannt sein. Ähnlich wie der erste Punkt dient diese Forderung der Differenzierung zwischen Identität und Wert eines Objekts. Ein Anwender darf nicht „in Versuchung geraten", dem Identitätswert irgendwelche Semantik zuzusprechen.

Zunächst soll sich hier lediglich auf die Tatsache beschränkt werden, daß in irgendeiner Weise Referenzen auf Objekte, wie sie auch aus Zugriffsstrukturen heraus benutzt werden, möglich sein müssen.

2.2 Konzepte objektrelationaler Datenbanksysteme

Bei der Anbindung relationaler Datenbanksysteme an prozedurale beziehungsweise objektorientierte Programmiersprachen wird ein Entwickler mit zwei prinzipiellen Problemen konfrontiert:

- ▷ *Impedance Mismatch*
  Die Konzepte der Datenmodellierung und -speicherung unterscheiden sich zwischen relationalen Datenbanken einerseits und gängigen Programmiersprachen andererseits. Dies gilt insbesondere mit Hinblick auf den Softwareentwurf und die Softwareentwicklung nach dem vorherrrschenden objektorientierten Paradigma. Auf der einen Seite stehen relationale, mengenorientierte Datenmodelle und Abfragen über eine deklarative Abfragesprache, auf der anderen Seite mit Zeigern vernetzte Datenstrukturen, die navigierend verarbeitet werden.

- ▷ *Keine transparente Persistenz*
  Die persistente Speicherung erfolgt im allgemeinen nicht transparent, sondern muß explizit erfolgen. Dafür werden Daten aus der Datenbank in Variablen geladen und umgekehrt aus Variablen in die Datenbank zurückgespeichert.

In der Weiterentwicklung der relationalen Systeme zu objektrelationalen Systemen wird der Impedance Mismatch zwar nicht beseitigt, aber zumindest etwas verringert. Von seiten der Datenbank erfolgt also eine Annäherung an die Anforderungen und Konzepte der objektorientierten Softwareentwicklung. Da der Graben aber nicht vollständig geschlossen werden kann, müssen bei der Softwareentwicklung auch beim Einsatz objektrelationaler Systeme Kompromisse gemacht werden. Besonders deutlich wird dies bei einem Blick auf die Abfrageschnittstelle und die mengenorientierte Verarbeitung, wobei sich andererseits gerade diese Fähigkeiten als Hauptvorteile für den Einsatz objektrelationaler Systeme zeigen werden.

Im folgenden werden nun die konzeptuellen ‚Annäherungsversuche' objektrelationaler Datenbanksysteme an objektorientierte Systeme erläutert.

2.2.1 Opaque Types

Im Rahmen der Weiterentwicklung relationaler Datenbanksysteme hin zu objektrelationalen Datenbanksystemen wurde zunächst mit dem Konzept der Opaque Types eine Möglichkeit geschaffen, komplexe Objekte mit relativ einfachen Mitteln in vorhandene relationale Datenbank-Management-Systeme zu integrieren.

Bei Opaque Types wird das DBMS nur als Speichermöglichkeit für die binäre Repräsentation eines komplexen Typs genutzt. Das Datenbanksystem hat keine Informationen über die innere Struktur der Objekte und greift auf diese nur über deren Objektmethoden zu.

2.2.2 Benutzerdefinierte Funktionen und Prädikate

Eine nächste Erweiterung besteht in der Möglichkeit, im Datenbanksystem UDFs (User Defined Functions) zu definieren. Diese Funktionen sind in einer üblichen Programmiersprache (oder dem systemspezifischen, erweiterten SQL-Dialekt) formuliert und ermöglichen beliebige anwendungsspezifische Berechnungen. Auf diese Weise sind zum Beispiel auch Selektionsabfragen möglich, die sich auf berechnete Eigenschaften von Objekten beziehen.

UDFs werden somit also auch als Prädikate (UDP – User-Defined Predicate) eingesetzt. [Ska99a]

2.2.3 Benutzerdefinierte Indexe

Speziell zur Unterstützung von Selektionsabfragen mit benutzerdefinierten Prädikaten (UDP) können anwendungsspezifische benutzerdefinierte Indexe (UDI), vom Nutzer implementierte Indexstrukturen, in ORDBMS ‚eingeklinkt' werden.

Bei benutzerdefinierten Indexen werden im kommerziellen Umfeld je nach Hersteller zwei unterschiedliche Ansätze verfolgt. In dem einfacheren Ansatz stellt das Datenbanksystem Vorlagen für Indexstrukturen zur Verfügung, wie zum Beispiel B*-Bäume im Fall von DB2. Der Benutzer hat eine Transformationsfunktion zu implementieren, die ein Objekt auf einen eingebauten Datentyp abbildet, der als Indexschlüssel genutzt werden kann.

In einem zweiten, flexibleren Ansatz, wie etwa bei Oracle oder Informix, wird es dem Benutzer ermöglicht, vollständige Indexstrukturen zu implementieren. Dazu muß eine bestimmte Anzahl von Funktionen implementiert werden: eine Funktion zur Erstellung eines Indexes, eine zum Einfügen von Indexeinträgen, eine zum Löschen von Indexeinträgen, eine Funktion, mit der Anfragen auf dem Index ausgeführt werden können, und so weiter.

Eine Abgrenzung zu den in dieser Arbeit behandelten anwendungsunabhängigen Indexstrukturen findet sich in Abschnitt 4.2.1.3.

2.2.4 Benutzerdefinierte Typen

Benutzt man das Konzept der Opaque Types bei der Realisierung von komplexen, zum Beispiel kollektionswertigen Attributen, wird die Organisation der Teildaten der Objekte innerhalb des Binärdatenfeldes vollständig über benutzerdefinierte Zugriffs- und Änderungsfunktionen bestimmt und liegt damit in der Verantwortung des Softwareentwicklers. Innerhalb des Binärfeldes wird demnach eine Art eigenes kleines Speichersystem entworfen, dessen interne Struktur allerdings dem Datenbanksystem völlig unbekannt ist.

Zwar ist über die Implementierung geeigneter Zugriffsstrukturen eine gewisse Optimierung hin zu einem effizienten Zugriff möglich, jedoch stößt man in der optimalen Anpassung der Speicherung an spezielle Zugriffs- und Änderungsprofile schnell an Grenzen.

Wenn die Definition nutzerdefinierter Typen (User-Defined Types) vom Datenbanksystem selbst angeboten wird [FDC+99], ergeben sich deutlich flexiblere Möglichkeiten bei der Speicherung von Objektdaten, die zur Optimierung genutzt werden können. Darüber hinaus vereinfacht sich die Datenmodellierung erheblich, da in den meisten Fällen eine komplizierte Implementierung von benutzerdefinierten Prädikaten und Zugriffsfunktionen hinfällig wird. Natürlich sind diese Vorteile nur Begleiterscheinungen bei der Einführung eines erweiterbaren Datentypsystems in DBMS, wenngleich sehr willkommene. Die Hauptziele nutzerdefinierter Typen liegen in Verbesserungen bei der Daten- und Anwendungsmodellierung und -entwicklung.

Im weiteren ist die Problematik von Bedeutung, welche konzeptuellen Erweiterungen an dem vorhandenen Typsystem relationaler Systeme nötig sind, um eine Integration komplexer Objekte zu ermöglichen.

Grundsätzlich muß ein Benutzer in der Lage sein, eigene neue Typen zu definieren. Bei den Mechanismen, die diese Fähigkeit zur Verfügung stellen, gibt es verschiedene Abstufungen.

2.2.4.1 Distinct Types

Mit Distinct Types kann der Benutzer neue Typen aus vorhandenen Datentypen ableiten. Der neu erzeugte Typ übernimmt die Operationen und die interne Repräsentation von dem bereits vorhandenen. Beispielsweise können aus dem vorhandenen Typ `INTEGER` neue Typen abgeleitet werden:

```
CREATE DISTINCT TYPE Etage_t AS INTEGER;
CREATE DISTINCT TYPE Hausnummer_t AS INTEGER;
```

Diese Variante benutzerdefinierter Typen ist in dieser Form bereits in SQL99 festgelegt [ISO99] und insbesondere dann wichtig, wenn strenge Typisierung erforderlich ist. Für die obigen Beispiele hätte das zur Folge, daß Vergleiche von Variablen oder Ausdrücken vom Typ `Etage_t` mit solchen vom Typ `Hausnummer_t` unzulässig wären.

2.2.4.2 Komplexe Typen und Typgeneratoren

Die eigentlichen komplexen Typen, die Strukturen und Kollektionen, wie sie etwa in SQL$^+$ [Luf02a] definiert sind, werden mit Hilfe sogenannter Typgeneratoren erzeugt. Das Konzept der Typgeneratoren ermöglicht es, durch Angabe von Parametern aus Typvorlagen konkrete Typen zu definieren. Mit einem bestimmten Typgenerator erzeugte Typen gleichen sich bezüglich ihrer Schnittstelle, also der Operatoren, Funktionen und Methoden, mit denen auf sie zugegriffen wird beziehungsweise mit denen sie bearbeitet werden. Eine Ausnahme bilden in gewisser Weise strukturierte Typen, da die Anzahl beziehungsweise Bezeichner ihrer Attribute implizit die Zugriffsschnittstelle definieren (Punktnotation). Ein Beispiel für einen Typgenerator ist `SET(<type>)`, der die Generierung beliebiger Mengentypen wie zum Beispiel `SET(integer)` oder `SET(float)` ermöglicht.

Typgeneratoren können entweder ad hoc, das heißt direkt bei der Definition eines Attributs, oder im Zusammenhang mit der `CREATE-TYPE`-Anweisung eingesetzt werden. Letzteres ermöglicht – wie auch bei Distinct Types – strenge Typisierung.

In dem erweiterten Typsystem sind drei Arten von beliebig verschachtelten Typgeneratoren vorgesehen:

- ▷ Definition strukturierter Typen
- ▷ `SET`, `MULTISET`, `ARRAY` und `LIST` für kollektionswertige Typen
- ▷ `REF` zur Definition von Referenztypen

1. Strukturierte Typen und typisierte Tabellen

Strukturen stellen eine Konkatenation benannter oder unbenannter Typen dar. Der entsprechende Generator akzeptiert mindestens einen bis beliebig viele Parameter, die sich aus den Namen von bereits definierten Typen oder Ad-hoc-Typdefinitionen mit vorangestellten Attributnamen zusammensetzen. Des weiteren sind für jeden Parameter nachgestellte Optionen wie `NOT NULL` oder `PRIMARY KEY` erlaubt:

```
CREATE TYPE Werkstückart_t AS (
    Typencode                   VARCHAR(25) NOT NULL PRIMARY KEY,
    Bezeichnung                 VARCHAR(100),
    ...                                                        );
```

TABELLE 2.1: Charakteristische Eigenschaften der verschiedenen Kollektionsarten

Kollektion	**Elementeigenschaft**	**typische Operationen**
SET	Ungeordnet und duplikatfrei	Enthaltensein, Vergleich, Mengenoperationen
MULTISET	Ungeordnet, Duplikate erlaubt	Enthaltensein, Duplikate entfernen, Elementzahl, Mengenoperationen
LIST	Geordnet, Duplikate erlaubt	Elemente einfügen oder anhängen, Sublisten, Verkettung
ARRAY	Geordnet, Duplikate erlaubt, Maximallänge	Indexzugriff auf einzelne Elemente oder Teilfelder

Mit diesem Beispiel wird ein strukturierter Typ Werkstückart_t definiert, mit den zwei Attributen Typencode und Bezeichnung. Ein so erstellter Datentyp kann nun beispielsweise als Parameter bei der Definition weiterer Typen verwendet werden.

Für strukturierte Typen können des weiteren Methoden definiert werden, um so das dynamische Verhalten beziehungsweise die Schnittstelle dieses Typs zu spezifizieren.

Benutzerdefinierte Typen können auf viererlei Weise verwendet werden:

- ▷ als Parameter einer Methode beziehungsweise Funktion
- ▷ als Parameter eines Typgenerators
- ▷ als Spaltentyp (Column Type) bei der Definition einer Relation
- ▷ als Row Type zur Definition einer sogenannten typisierten Tabelle (Typed Table)

Typisierte Tabellen heben sich unter anderem insofern von konventionellen Tabellen ab, als daß sie implizit ein zusätzliches Attribut besitzen. Dieses ist für einen Benutzer im Normalfall nicht sichtbar und enthält einen Identifikator (Objekt-ID), der mindestens innerhalb der entsprechenden Relation eindeutig ist. Über Referenzattribute, auf die weiter unten eingegangen wird, können Tupel einer typisierten Tabelle referenziert werden.

Optional kann bereits bei der Definition eines strukturierten Typs ein Primärschlüssel als Vorgabe für die spätere Verwendung festgelegt werden. Dieser besteht, analog zur Definition von Relationen, aus einem oder mehreren Attributen. Diese Definition schränkt die Verwendung jedoch nicht ein und kann bei der Verwendung überschrieben werden.

2. Kollektionen

In Kollektionen werden mehrere Instanzen (Elemente) eines Typs zusammengefaßt. In [Luf02a, Luf05] wird die konsistente Erweiterung von SQL um ein vollständiges System zur Kollektionsunterstützung beschrieben. Es werden Syntax und Semantik der Datendefinition, von Abfrage- und Änderungsoperationen sowie von Prädikaten und Integritätsbedingungen für Kollektionen dargestellt. Wie auch bei strukturierten Typen erfolgt bei diesem Ansatz die Definition von Kollektionen über Typgeneratoren, die beliebig schachtelbar sind. Jeder Kollektionsgenerator bringt einen Satz vordefinierter Operationen und Prädikate mit. TABELLE 2.1 beschreibt die vier grundlegenden Arten mit ihren jeweils charakteristischen Eigenschaften.

Die Syntax für diese Kollektionstypen soll an folgenden Verwendungsbeispielen verdeutlicht werden:

- ▷ *Kollektionstyp Menge*
  - ◇ *Typgenerator*
    `SET(INT)` erzeugt den Typ „Menge von Zahlen“
  - ◇ *Objektkonstruktor*
    `SET(1,2,3)` erzeugt die Menge der Zahlen 1, 2 und 3

- ▷ *Kollektionstyp Multimenge*
  - ◇ *Typgenerator*
    `MULTISET(INT)` erzeugt den Typ „Multimenge von Zahlen“
  - ◇ *Objektkonstruktor*
    `MULTISET(2,1,2)` erzeugt die Multimenge der Zahlen 1 und 2, wobei 2 zweimal vorkommt

- ▷ *Kollektionstyp Liste*
  - ◇ *Typgenerator*
    `LIST(INT)` erzeugt den Typ „Liste von Zahlen“
  - ◇ *Objektkonstruktor*
    `LIST(1,2,1)` erzeugt die geordnete Liste der Zahlen 1, 2 und 1 in genau dieser Reihenfolge

- ▷ *Kollektionstyp Feld*
  - ◇ *Typgenerator*
    `ARRAY(INT, 3)` erzeugt den Typ „Feld von drei Zahlen“ mit dem Indexbereich von 1 bis 3
  - ◇ *Objektkonstruktor*
    `ARRAY(1,2,1)` erzeugt das Feld aus den Zahlen 1 an Position 1, 2 an Position 2 und 1 an Position 3

Die Verwendung der Typgeneratoren soll nachfolgendes Beispiel illustrieren. Es wird der Typ `Werkstückart_t`, der aus Abschnitt 1.4 bekannt ist, mit Hilfe der Typgeneratoren `SET` und `ROW` definiert:

```
CREATE TYPE Fertigungsschritt_t AS (
    ...
    Bearbeitungskonfiguration       SET( ROW(
        Parametername                    VARCHAR(50),
        Parametertyp                     VARCHAR(20),
        ParameterwertF                   FLOAT,
        ParameterwertV                   VARCHAR(100)
                                    ) )                          );
```

Zu beachten ist, daß die Definition des Elementtyps – wie hier gezeigt wird – auch ad hoc erfolgen kann.

Und die Verwendung der Objektkonstruktoren verdeutlicht dieser Beispielausdruck, der in einer Tabelle `Fertigungsschritt` vom Typ `Fertigungsschritt_t` einen Bearbeitungsparameter ergänzt:

```
UPDATE Fertigungsschritt
    SET Bearbeitungskonfiguration = Bearbeitungskonfiguration
            UNION SET( ROW('Zielgewicht in kg', 'FLOAT', 0.25) );
```

3. *Referenztypen*

Objektorientierte Modellierung setzt voraus, daß Verweise zwischen Objekten möglich sind. Das Typsystem wird dafür um einen weiteren Typgenerator `REF` ergänzt, der Referenztypen erzeugt. Parameter dieses Generators ist der Typ, dessen Tupel referenziert werden sollen. Für jeden Referenztyp werden implizit Operationen zur Dereferenzierung, Zuweisung und zum Vergleich angelegt.

Konkrete Referenzen sind, wie schon weiter oben erläutert, letztendlich Referenzen auf Tupel einer typisierten Tabelle. Da der Referenztyp jedoch in bezug auf den die Tabelle erzeugenden Typ und nicht auf die Tabelle selbst definiert ist, ist es konzeptuell möglich, daß mit dem Referenztyp Tupel unterschiedlicher Tabellen referenziert werden.

Dieser zusätzliche Freiheitsgrad könnte für manche Fälle vielleicht von Vorteil sein, birgt jedoch unter Umständen die Gefahr einer unsauberen Datenmodellierung (siehe auch weiter unten: Abbildung von Klassen und Objekten). Es ist daher sinnvoll, Möglichkeiten zur Einschränkung der referenzierbaren Tabellen anzubieten. In konkreten Systemen erfolgt dies mit dem Schlüsselwort `SCOPE` und ist in den meisten Fällen zwingend.

Im folgenden werden die zwei Tabellen Bearbeitungsvorgang und Fertigungsmaschine erstellt. Aus der Menge Bearbeitungsfähigkeiten in der Tabelle Fertigungsmaschine werden über ein Referenzattribut Tupel aus der Tabelle Bearbeitungsvorgang referenziert[1].

```
CREATE TABLE Bearbeitungsvorgang AS (
    Identifikationscode VARCHAR(20) NOT NULL PRIMARY KEY,
    ...                                                    );

CREATE TABLE Fertigungsmaschine AS (
    ...
    Bearbeitungsfähigkeiten     SET( ROW(
        Bearbeitungsvorgang     REF(Bearbeitungsvorgang),
    ...                             ) )                    );
```

2.3 Einbindung objektrelationaler Datenbanken in objektorientierten Softwareumgebungen

Der Einsatz objektrelationaler Datenbanksysteme macht es trotz der vielen Erweiterungen notwendig, bei der objektorientierten Softwareentwicklung Kompromisse einzugehen. Eine erste prinzipielle Feststellung ist, daß man durch den Einsatz dieses speziellen Typs von Datenbanksystemen nach wie vor auf eine gewisse Art festgelegt wird, wie Datenobjekte zu speichern sind – nämlich in Relationen, also Tabellen. Auch wenn diese Feststellung für objektrelationale Datenbanksysteme zutrifft, sei wenigstens erwähnt, daß unter anderem in der Forschung zum eNF^2-Datenmodell und entsprechenden Systementwicklungen Abweichungen davon vorgeschlagen werden [AB84, DKA$^+$86, Bat86b, PA86, PSS$^+$87, LKD$^+$88, CDV88, GGH$^+$92].

[1] Die SQL-Norm erlaubte in der Fassung von 1999 eine Einschränkung des Zielbereiches von Referenzen mit der `SCOPE`-Klausel, die jedoch aus der aktuellen Norm SQL:2003 wieder gestrichen wurde.

2.3.1 Abbildung von Klassen und Objekten

Objekte werden zu Klassen zusammengefaßt. Zu einer Klasse gehört demnach eine Menge von Objekten. Um diese 1:n-Beziehung in einer objektrelationalen Datenbank zu repräsentieren, bietet sich zunächst nur ein mögliches Speicherkonzept: Eine Relation wird als eine Klasse betrachtet und deren Tupel als Instanzen der Klasse, also als die eigentlichen Objekte.

Da Objekte im Regelfall auch referenzierbar sein sollen, bietet sich für die Definition objektspeichernder Relationen das Konzept der typisierten Tabelle an, was ja auch genau zu diesem Zweck eingeführt wurde.

Zu beachten ist, daß durch die Möglichkeit, mit einem Typ mehrere Tabellen anzulegen, pro Klassendefinition potentiell mehrere Extensionen angelegt werden können. Im Vergleich zu herkömmlichen objektorientierten Umgebungen stellt dies eine Besonderheit dar.

Um referenzierbare Objekte in einem objektrelationalen Datenbanksystem zu erhalten, sind diese also zwingend als Tupel einer typisierten Tabelle zu speichern. Sie erhalten auf diesem Wege eine Identität und sind referenzierbar.

Setzt man strukturierte Typen bei der Definition von Tabellenattributen ein (Column Types), wird das einfache Typsystem des relationalen Modells um die Möglichkeit erweitert, komplex strukturierte Daten zu speichern. Sieht man ein strukturiertes Attribut als skalaren Typ mit klar definierter Schnittstelle an, verletzt diese Erweiterung im übrigen nicht die Forderungen der ersten Normalform.

2.3.2 Mengenorientierter Objektzugriff

Der Zugriff auf Objekte kann prinzipiell auf zweierlei Weise erfolgen:

- ▷ Anhand der Identität der Objekte (Objektzugriff)
- ▷ Über die Abfrage von Eigenschaften der Objekte (Wertezugriff)

Der Objektzugriff kommt insbesondere in solchen Fällen zum Einsatz, in denen eine zwischen Anwendungsprogramm und Datenbank liegende Persistenzschicht die Transformation von Anwendungsobjekten in Datenbankobjekte (und umgekehrt) vornimmt. Diese Schnittstelle ist auf einen navigierenden Objektzugriff ausgelegt, wobei einzelne Objekte oder kleinere Objektmengen aus der Datenbank geladen, bearbeitet und wieder gespeichert werden. Gerade für die Bearbeitung oder Abfrage umfangreicher Datenmengen ist ein navigierender Zugriff jedoch unpraktisch und ineffizient, da erstens das Durcharbeiten der einzelnen Objekte explizit ausformuliert werden muß und es zweitens in vielen Fällen zu unnötig vielen Objektmaterialisierungen im Hauptspeicher kommt.

Durch den Einsatz objektrelationaler Datenbanken erschließt sich die Möglichkeit, auf größere Datenmengen zuzugreifen, ohne diese einzeln in das Anwendungsprogramm transferieren zu müssen. Die Adressierung der Objekte erfolgt sowohl mittels Wertezugriff als auch – bei Anfragen an durch Referenzen verknüpfte Objekte – mittels Identitätszugriff und ermöglicht zum Beispiel eine effiziente Bearbeitung folgender Szenarien (nach [Luf00]):

- ▷ *Auswertung von Massendaten*

  Dadurch, daß die Persistenzschicht umgangen wird, müssen keine unnötigen Objektmaterialisierungen vorgenommen werden. Die Auswertungen können „direkt in der Datenbank“ erfolgen.

▷ *Datenaggregation mit Datenbankfunktionen*

Auch hier kann durch den Einsatz von in der Datenbank integrierter Funktionalität die Auswertung von Daten in die Datenbank verlegt werden.

▷ *Ad-hoc-Datenzugriff*

Durch die Datenbankschnittstelle wird es möglich, ungeplante Anfragen auf Daten von Objekten zu formulieren und auszuführen.

Der mengenorientierte Zugriff ist ein wesentliches Argument für den Einsatz von Datenbanksystemen. Neben Leistungsvorteilen vereinfacht sich der Programmieraufwand zusätzlich für viele Anwendungen in einem erheblichen Maße: Was bisher Teil der Anwendungslogik war (beziehungsweise zwangsläufig sein mußte), verlagert sich in ein zentrales Datenbanksystem.

2.4 Datenunabhängigkeit in ORDBMS: Trennung von logischen und physischen Aspekten

Ein wesentlicher Aspekt bei der Entwicklung von Datenbankanwendungen ist die Unterstützung der Datenunabhängigkeit durch das DBMS [HS00]. Datenunabhängigkeit hat das Ziel, oft langlebige Datenbanken und Anwendungen hinsichtlich notwendiger Änderungen weitgehend zu entkoppeln. Speziell die Implementierungsunabhängigkeit bedeutet, daß die konzeptuelle Sicht auf eine Datenbank weiter besteht, unabhängig von der jeweils für die Daten gewählten Speicherstruktur und ihren möglichen Änderungen.

Da heutige DBMS das konzeptuelle Modell relativ statisch und unflexibel in eine bestimmte Art der physischen Speicherung abbilden und konzeptuelle mit internen (physischen) Aspekten in der DDL stark vermischen – beispielsweise bei der Erweiterung der `CREATE-TABLE`-Anweisung durch unzählige Angaben zur Speicherung –, orientiert sich der Entwurf eines Datenmodells häufig zu sehr an den Leistungsbedürfnissen der konkreten Anwendungen. Eine Unabhängigkeit der Datenbank durch ein allgemeines Datenmodell ist dann nicht mehr gegeben.

Daher soll für den Datenbankentwurf in dieser Arbeit folgender zweistufiger Ansatz propagiert werden:

1. Schritt: *Objektrelationaler Entwurf mit SQL*

   In einem ersten Schritt findet eine Abbildung der relevanten Klassen des objektorientierten Klassenmodells in ein objektrelationales Modell statt. Die Abbildung soll möglichst einfach und intuitiv sein. Denkbar wäre auch eine (teil-)automatische Abbildung. Wichtig ist, daß zu diesem Zeitpunkt der Entwurf nicht auf Anwendungs- und Workload-Anforderungen ausgerichtet ist und auch nicht sein muß. Die Sprache, in der das Modell in die Datenbank gebracht wird, ist SQL.

2. Schritt: *Physischer Entwurf mit PRDL*

   Erst in einem zweiten Schritt wird die konkrete physische Abbildung des logischen Modells in der Datenbank definiert. Zu diesem Zweck wurde von der Jenaer Datenbankgruppe die Speicherbeschreibungssprache PRDL [Kis02] entworfen.

Die Einführung einer weiteren Abstraktionsschicht ist im Grunde nicht nur eine direkte Folge der angestrebten Datenunabhängigkeit, sondern auch des objektorientierten Aspekts

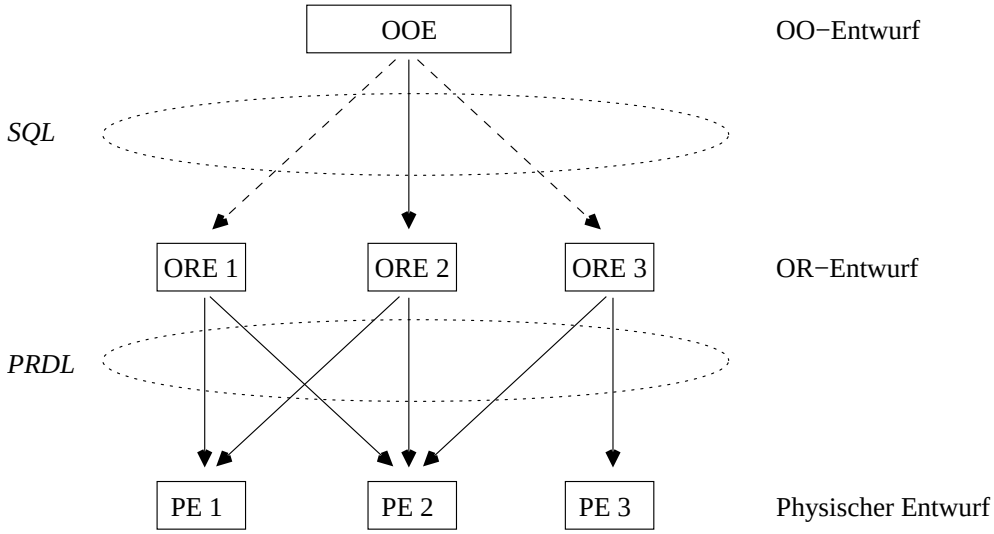


ABBILDUNG 2.1: Trennung von logischem und physischem Entwurf

der Wiederverwendung. Gleiche logische Entwürfe können an verschiedenen Stellen des Objektmodells in unterschiedlichen Kontexten eingesetzt werden. Je nach Kontext findet eine unterschiedliche Speicherung und Verarbeitung dieser – vom logischen Aufbau her – gleichen Objekte statt. Nur durch die Möglichkeit, gleiche logische Objekte unterschiedlich physisch abzubilden, kann der wichtige Faktor der Wiederverwendung von Entwürfen in das DBMS mit hineingebracht werden.

Das in ABBILDUNG 2.1 gezeigte Schema verdeutlicht diesen zweistufigen Prozeß und dessen Konsequenzen: Für ein und denselben objektorientierten Entwurf kann es neben einer vielleicht bevorzugten ‚kanonischen' Abbildung auf einen objektrelationalen Entwurf potenziell mehrere weitere logische Abbildungen geben. Diese können durch den Einsatz von PRDL in unterschiedlichen Varianten physisch abgespeichert werden. Durch den zweistufigen Ansatz ist es insbesondere möglich, daß unterschiedliche OR-Entwürfe den selben physischen Entwurf besitzen.

Ähnliche Überlegungen zur Einführung einer Speicherbeschreibungssprache gab es schon vor circa 30 Jahren bei den CODASYL-DBMS [COD70]. Allerdings haben sich diese sehr ineressanten Ansätze nicht wirklich durchsetzen können und sind insbesondere bei relationalen, objektorientierten und auch bei objektrelationalen DBMS nicht weiterverfolgt worden. Genau an diesem Punkt setzt der Gedankengang dieser Arbeit an, denn die eben beschriebenen Effekte der Trennung von logischem und physischem Entwurf versprechen große Vorteile und Effektivierungspotentiale gerade bei ORDBMS.

An diesem wichtigen inhaltlichen Punkt der Arbeit wollen wir mit der Nennung des Begriffs *Baum* für die Aufmerksamkeit des geneigten Lesers danken. Was er damit anfangen kann, soll später an geeigneter Stelle enthüllt werden.

2.5 Integration komplexer Objekte in relationale Datenbanksysteme

Für eine optimale Ausnutzung der Konzepte relationaler Datenbanken bei der Integration objektorientierter Erweiterungen ist es notwendig, einen genaueren Blick auf die gängige

Architektur dieser Systeme zu werfen. Die Ausführungen werden sich dabei auf Merkmale beschränken, die für die Überlegungen der nachfolgenden Kapitel relevant sind. Eine detailliertere Darstellung findet sich in wissenschaftlicher Literatur [HR01, SH99] oder auch in den Dokumentationen konkreter Datenbanksysteme [Ora01c, Ora01d, Ora01a, Ora01b, IBM00a, IBM00b, IBM03d, IBM03a].

Heutige kommerzielle relationale DBMS basieren zum größten Teil auf der Architektur des IBM Prototypen System R aus den 1970er Jahren [ABC+76]. Diese beschreibt eine Schichtenarchitektur, bestehend aus den fünf Schichten Datensystem, Zugriffssystem, Speichersystem, Pufferverwaltung und Betriebssystem (ABBILDUNG 2.2 nach [SH99]).

Das Datensystem realisiert die Schnittstelle nach außen zu Anwendungsprogrammen über die sogenannte mengenorientierte Schnittstelle. Es übersetzt und optimiert Anfragen und übernimmt die Benutzerauthentifizierung. Abfrageergebnisse werden immer als Menge beziehungsweise bei realen DBMS als Multimenge präsentiert. Die Kommunikation zur nächsttieferen Schicht, dem Zugriffssystem, erfolgt satzorientiert.

Innerhalb des Zugriffssystems werden die Definitionen der Relationen und der zugehörigen Zugriffspfade verwaltet (Data Dictionary beziehungsweise Datenbankkatalog). Dazu gehört auch die Umwandlung der logischen Tupel in das Format, in dem sie als Bytestruktur in den internen Sätzen gespeichert werden.

Interne Sätze sind die wesentlichen Adressierungseinheiten der Schnittstelle, welche das Zugriffssystem mit dem Speichersystem verbindet. Letzteres sorgt dafür, daß die Sätze in Speicherseiten abgelegt und dort möglichst optimal organisiert werden.

Die Pufferverwaltung stellt dem Speichersystem zu diesem Zweck Seiten zur Verfügung. Hauptaufgabe ist die Organisation eines Puffers, in dem nach bestimmten Strategien Speicherseiten gecached werden.

Für diese Arbeit relevant sind Aufbau und Konzepte des Zugriffs- und Speichersystems. Primäre Aufgabe des Zugriffssystems ist das Umsetzen der typisierten Tupel der satzorientierten Schnittstelle in untypisierte Sätze der internen Satzschnittstelle. Bei den typisierten Sätzen handelt es sich um Tupel einer Relation. Mit Hilfe der Informationen des Datenbankkatalogs, in dem das Format und der Aufbau aller Tabellen verwaltet werden, transformiert das Zugriffssystem die typisierten Daten in Datenobjekte, welche als sogenannte interne Sätze dem Speichersystem übergeben werden.

Das Speichersystem präsentiert also dem Zugriffssystem auf der internen Satzschnittstelle die Möglichkeit, Daten in adressierbaren internen Sätzen zu speichern. Die Speicherung von Sätzen erfolgt innerhalb von Speicherseiten, wobei in einer Seite im allgemeinen immer nur Sätze eines Typs, also einer Relation, abgelegt werden. Eine Ausnahme bilden sogenannte Cluster, auf die später im Text näher eingegangen wird.

Neben ‚normalen' Sätzen bietet das Speichersystem in der Regel eine weitere Speicherform für Daten an: sogenannte LOBs (Large Objects) ermöglichen das Speichern besonders großer Datenobjekte und finden besonders im Zusammenhang mit Multimediadaten Anwendung.

Datenseiten werden eindeutig einer Relation zugeordnet und damit in bestimmte Typen von Seiten unterteilt. Ein Seitentyp wird in diesem Zusamenhang also über die Zugehörigkeit zu einer Relation definiert. Man erhält auf Seitenebene dann auch eine hohe Dichte von Sätzen gleichen Typs, das heißt von Sätzen, die zu einer Realtion gehören und deshalb eine gleiche Struktur besitzen.

Um innerhalb des zur Verfügung stehenden Speicherraums (Tablespace) eine hohe Dichte auch über Seitengrenzen hinaus zu erreichen, werden Seiten gleichen Typs zu sogenanten Extents gruppiert. Das heißt, der zur Verfügung stehende Speicherraum ist in Extents

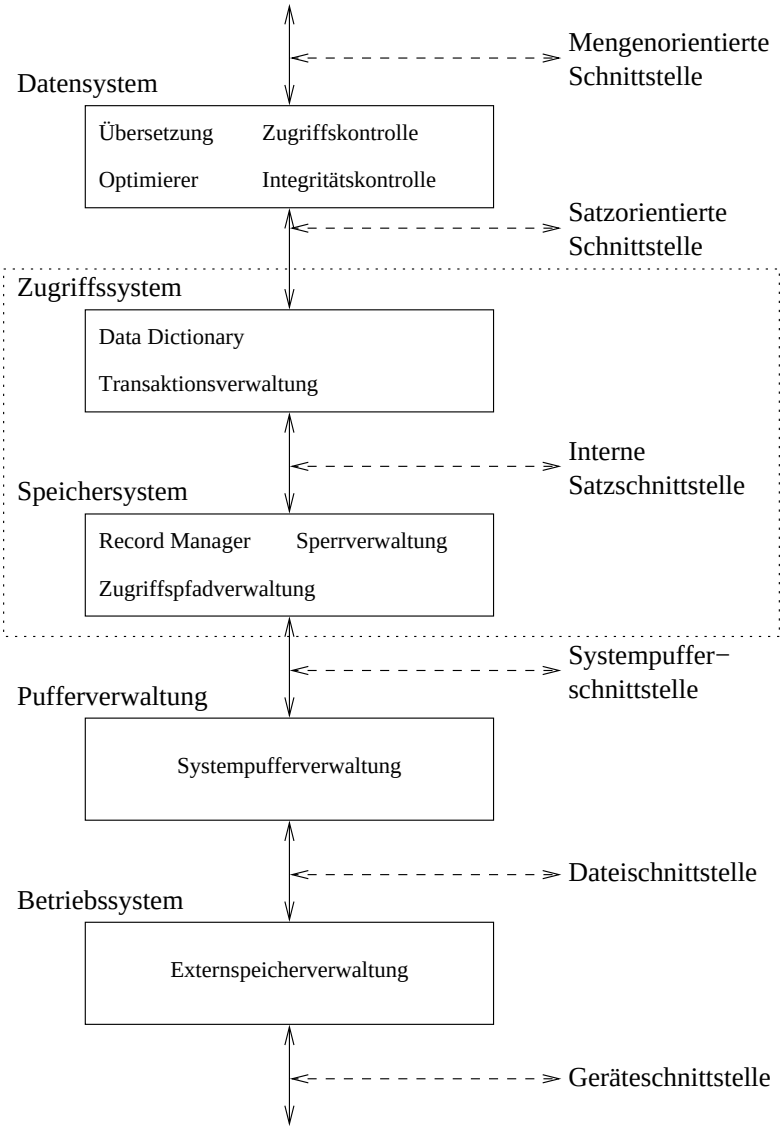


ABBILDUNG 2.2: Schichtenmodell zur Implementierung eines datenunabhängigen DBMS

aufgeteilt, wobei jedes Extent einer bestimmten Relation zugeordnet ist und aus einer Menge von Speicherseiten besteht. Die Menge aller Extents mit einer bestimmten Funktion (Datensegment, Indexsegment, BLOB-Segment etc.) nennt man üblicherweise Segmente, wobei Extents eines Segments zwar in der Regel, aber nicht zwingend, physisch dicht gespeichert werden. ABBILDUNG 2.3 skizziert das Verhältnis zwischen den drei Einheiten.

Es handelt sich hier um das wohl gängigste und am weitesten verbreitete Modell. Zu beachten ist jedoch, daß sich (natürlich) sowohl in der Literatur als auch in Produkten verschiedenste Varianten dieser Konzepte finden.

Wird auf Daten einer Datenbank zugegriffen, ist es aus Sicht des Optimierers im Datensystem immer oberstes Ziel, die Anzahl der Externspeicherzugriffe zu minimieren. Der wahlfreie Zugriff auf große Datenmengen verbunden mit vergleichsweise langen Zugriffszeiten (oft im Millisekunden-Bereich) führt dazu, daß die mit Abstand dominierenden Kosten bei einer Anfragebearbeitung durch die blockweisen Ein-/Ausgabeoperationen auf

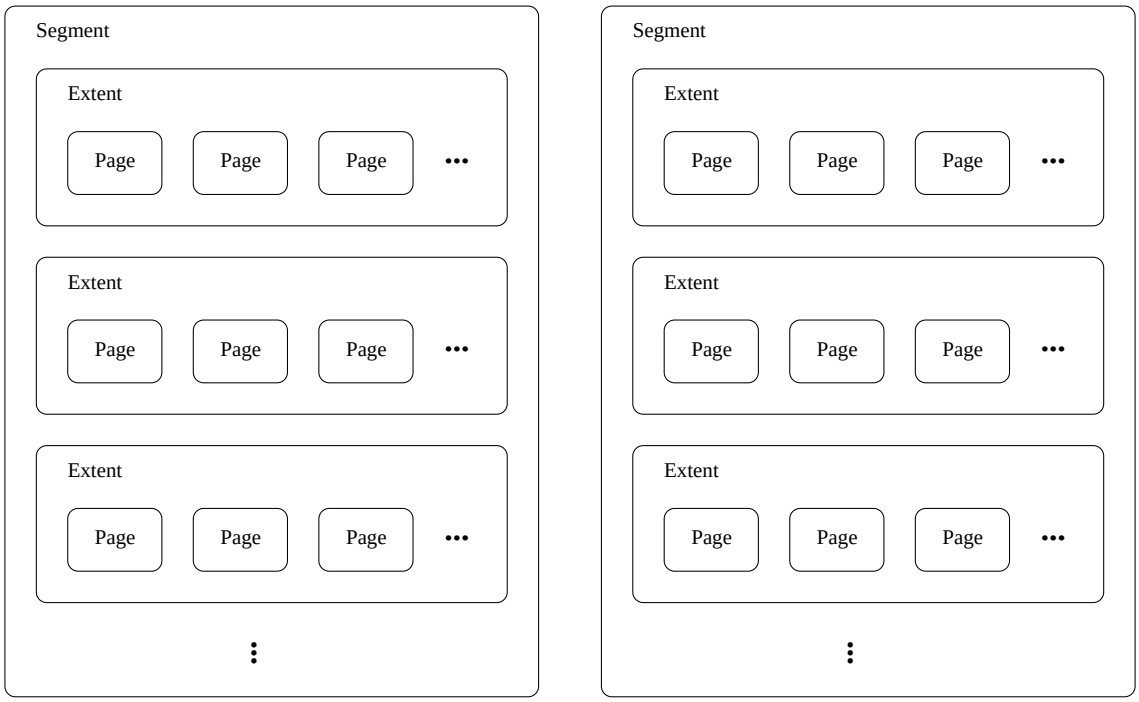


ABBILDUNG 2.3: Die Beziehung zwischen Segment, Extent und Speicherseite

Externspeichermedien entstehen. Das Vorhandensein einer Pufferverwaltung bedeutet, daß zur Minimierung der Externspeicherzugriffe letztendlich eine Minimierung der Anzahl der physischen Seitenzugriffe anzustreben ist.

2.5.1 Implementierung der Komplexobjektunterstützung in den oberen DBMS Schichten

Objektrelationale Datenbank-Management-Systeme stellen den Versuch dar, existierende relationale DBMS um Objektunterstützung zu erweitern. Dabei muss jedoch die Verwaltung traditioneller relational repräsentierter einfacher Daten in flachen Tabellen mit der Speicherung und Verarbeitung komplexer und geschachtelter Datenobjekte koexistieren. Zentrale Anforderung dabei ist, dass die Integration neuer objektrelationaler Konstrukte und die neugewonnene Erweiterbarkeit des DBMS die gute Performance bei der Verarbeitung der klassischen flachen Daten nicht beeinträchtigt.

Bei der Frage nach geeigneten Implementierungsstrategien hat sich herausgestellt, daß dieses Ziel am ehesten durch die Beschränkung der Erweiterungen und Änderungen auf die oberen Architekturschichten Datensystem und Zugriffssystem und insbesondere auf die Anfrageübersetzung erreichbar ist [HR01, CCN+99, Ska02]. Dadurch können insbesondere die Puffer-, Seiten- und physische Satzverwaltung weitgehend unverändert bleiben. Das erhebt allerdings die Frage, wie die komplexen und verschachtelten Objekte der logischen Ebene auf die angebotenen physischen Datensätze abgebildet werden können. Dabei gibt es verschiedenste Möglichkeiten, die im nächsten Absatz diskutiert und klassifiziert werden sollen.

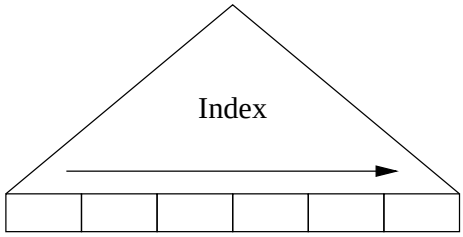


ABBILDUNG 2.4: Indexorganisierte Speicherung

2.5.2 Basistechniken zur Unterstützung komplexer Objekte

Zur Unterstützung komplexer Objekte in (objekt-)relationalen DBMS spielen neben den in den nachfolgenden Kapiteln detailliert diskutierten Speicher- und Zugriffsuntertützungskonzepten eine Anzahl bereits aus RBMS bekannter grundlegender Basistechniken eine Rolle [Ska02]. Diese sollen gewissermaßen als Einstieg in die Thematik in den nächsten Abschnitten erläutert werden.

2.5.2.1 Techniken zur Clusterbildung

Da Segmente beziehungsweise Extents nur einer Tabelle zugeordnet werden, entsteht eine Clusterung nach Relationen. Clusterung, also physisch dichte Speicherung bestimmter Elemente, dient der Verringerung der teuren, verteilten Lesezugriffe auf Externspeichermedien. Sind Daten, auf die aus logischer Sicht häufig gemeinsam zugegriffen wird, physisch dicht gespeichert, verringert sich in tieferen Schichten des Datenbanksystems die Anzahl der Lesezugriffe, was zu signifikanten Leistungssteigerungen führt (Ausnutzung des Lokalitätsprinzips).

Zusätzlich zur Clusterung auf der Ebene von Segmenten beziehungsweise Extents kann auch auf interner Satzebene geclustert werden. Ziel bei dieser Art von Clusterung ist es, logisch zusammenhängende interne Sätze möglichst auf ein und dieselbe Speicherseite zu plazieren. Es gibt zu diesem Zweck zwei gängige Konzepte: die indexorganisierten Tabellen und die tabellenübergreifenden Cluster.

1. Indexorganisierte Speicherung

Indexorganisierte Tabellen (Index-Organized Tables, IOT) basieren auf einem Index über der entsprechenden Relation, die indexorganisiert gespeichert werden soll. Tupel, die den selben oder nebeneinanderliegenden Indexeinträgen zugeordnet sind, werden nach Möglichkeit zusammen auf einer Seite abgelegt. Auf diese Weise befinden sich auf einer Speicherseite der entsprechenden Relation nur Tupel, die bezogen auf eine Ordnung über ein oder mehrere Attribute nah beieinanderliegen. Der Hauptvorteil indexorganisierter Speicherung liegt in der effizienten Durchführung von Index Scans.

Realisiert wird diese Speicherungsart, indem ein (B*-)Baum-Index definiert wird, in dessen Blättern anstelle der sonst üblichen Referenzen auf interne Sätze die Sätze der Relation selbst abgespeichert sind. Sowohl die Knoten als auch die Blätter des Baumes sind dabei wie üblich ganze Speicherseiten (ABBILDUNG 2.4).

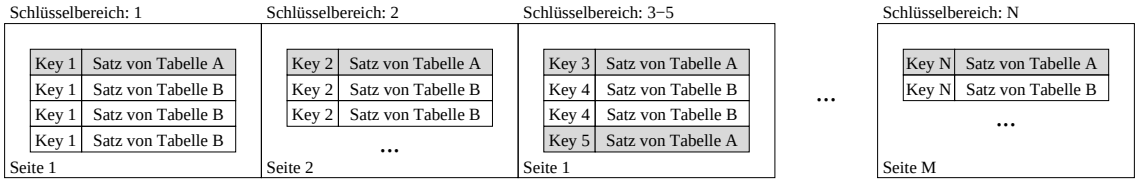


ABBILDUNG 2.5: Tabellenübergreifende Clusterung

Es ist Aufgabe des Speichersystems, die entsprechenden Speicherstrukturen zur Verfügung zu stellen. Ähnliche Strukturen werden auch für die Erstellung konventioneller Indexe benötigt. Wie an späterer Stelle noch erläutert wird, unterscheidet sich aus Gründen der Leistungssteigerung die Adressierung von internen Sätzen bei indexorganisierten Tabellen in der Regel von der bei konventionellen Tabellen.

2. Tabellenübergreifende Clusterung

Die zweite Art der Clusterung ist die tabellenübergreifende Clusterung. Theoretisch können an der Clusterung beliebig viele Relationen beteiligt sein. Voraussetzung ist, daß alle Relationen mindestens ein Attribut des gleichen oder verträglichen Typs besitzen. Tupel, die den selben Wert in den entsprechenden Attributen haben, werden geclustert – also nach Möglichkeit auf einer Speicherseite – abgelegt (ABBILDUNG 2.5).

In einem ersten Schritt wird der Cluster angelegt und es wird definiert, über wieviele Attribute ein Cluster identifiziert wird und welchen Typ die einzelnen Attribute haben sollen. Alle Werte der identifizierenden Felder zusammen ergeben den sogenannten Clusterschlüssel. Um den werteabhängigen Speicherort für Tabellentupel zu lokalisieren, ist ein Index oder eine Hash-Funktion nötig, die einem Clusterschlüssel eine Speicherseite zuordnet.

Bei der Definition von Tabellen kann als Speicherort nun der bereits erstellte Cluster angegeben werden. Zusätzlich müssen den Attributen des Clusterschlüssels Attribute der Relation zugeordnet werden.

Folgendes Beispiel soll das Prinzip dieser Art der Clusterung erläutern. Die Syntax entspricht der des Datenbanksystems Oracle. In einem ersten Schritt wird der Cluster als solches sowie ein zugehöriger Clusterindex erzeugt:

```
CREATE CLUSTER BearbeitungsvorgangsCluster (Bearbeitungsvorgang
VARCHAR(20));

CREATE INDEX BearbeitungsvorgangsCluster_Idx
    ON CLUSTER BearbeitungsvorgangsCluster;
```

Der Clusterschlüssel dieses Clusters besteht also aus einer Zeichenkette. Als nächstes können Tabellen angelegt werden:

```
CREATE TABLE Bearbeitungsvorgang AS Bearbeitungsvorgang_t
    CLUSTER BearbeitungsvorgangsCluster (Identifikationscode);

CREATE TABLE Fertigungsschritt AS Fertigungsschritt_t
    CLUSTER BearbeitungsvorgangsCluster
            (IdentifikationscodeDesBearbeitungsvorgangs);
```

Nach diesen Definitionen können Tupel in die Tabellen eingefügt werden. Dabei werden alle Bearbeitungsvorgänge zusammen mit dem jeweiligen Fertigungsschritten geclustert gespeichert.

Bei dieser Syntax ist das Schlüsselwort `CLUSTER` jedoch unglücklich gewählt und in gewisser Weise irreführend. Cluster steht hier für das Datenbankobjekt, das durch eine `CREATE-CLUSTER`-Anweisung entstanden ist und als Speicherort bei der Definition von Tabellen angegeben werden kann. Andererseits wird eine konkrete Gruppe von Tupeln mit einem gemeinsamen Clusterschlüssel beziehungsweise der konkrete Speicherort für Tupel mit einem bestimmten Clusterschlüssel als Cluster bezeichnet. Für ersteren Fall sollte man daher eher von Clustertyp sprechen.

Von Vorteil und prädestiniert ist diese Art der geclusterten Speicherung vor allem dann, wenn – wie im obigen Beispiel – zwischen den Clusterschlüsseln der beteiligten Relationen eine Fremdschlüsselbeziehung besteht. Man spricht in diesem Fall auch oft von einer hierarchischen Clusterung. Im Fall von Verbundabfragen ist gewährleistet, daß zusammengehörende Tupel physisch dicht gespeichert sind und die Ein-/Ausgabekosten entsprechend gering gehalten werden. Es liegt quasi ein ‚vorberechneter' Verbund vor. Aber auch bei Abfragen auf nur einer Tabelle, in der (unter anderem) über Clusterschlüsselattribute selektiert wird, können sich geclusterte Speicherungen – ähnlich wie die indexorganisierte Speicherung – positiv auf die Verarbeitungsleistung auswirken.

2.5.2.2 Adressierungskonzepte für interne Sätze

Die Sätze nicht indexorganisiert gespeicherter Relationen werden ungeordnet in ihrem Segment abgespeichert (Prinzip der Halde). Um dennoch einen schnellen Zugriff auf Daten zu gewährleisten, werden Indexe angelegt, die unter Angabe eines oder mehrerer Attributwerte dazu genutzt werden können, die Adresse der betroffenen Sätze möglichst direkt aufzufinden.

Es ist also eine Adressierungstechnik nötig, mit der Sätze aus Indexen heraus adressiert werden können. Diese hat folgenden Kriterien zu genügen (nach [HR01]):

- ▷ Schneller und möglichst direkter Zugriff auf einen Satz
- ▷ Stabil gegenüber notwendigen Satzverschiebungen, so daß es zu keinen lawinenartigen Folgeänderungen kommt
- ▷ Reorganisationsbedarf so gering wie möglich

Für die Adressierung der Sätze ergeben sich intuitiv zunächst zwei Möglichkeiten: die direkte oder die indirekte Adressierung.

1. Direkte versus indirekte Adressierung

Die direkte Adressierung würde bedeuten, daß eine Satzadresse direkt auf der relativen Byteadresse innerhalb eines Tablespaces basiert. Dies würde zwar einen sehr schnellen Zugriff gewährleisten, allerdings nicht die Möglichkeit von Satzverschiebungen zulassen. Da dies im allgemeinen im Datenbankbetrieb jedoch unvermeidlich ist, da Änderungsoperationen sehr häufig auch eine Änderung der Größe von Datensätzen und damit Verschiebungen in der physischen Speicherung bewirken, scheidet die direkte Adressierung als Option aus.

Die indirekte Adressierung zeichnet sich dadurch aus, daß die nach außen sichtbare Adresse konstant bleiben kann und gleichzeitig eine Verschiebung des Zielobjekts innerhalb gewisser Grenzen möglich ist. Im Bereich der Satzadressierung bei Datenbanken werden zwei Abstufungen der indirekten Adressierung unterschieden: die physische Adressierung verbunden mit dem TID-Konzept und die logische Adressierung.

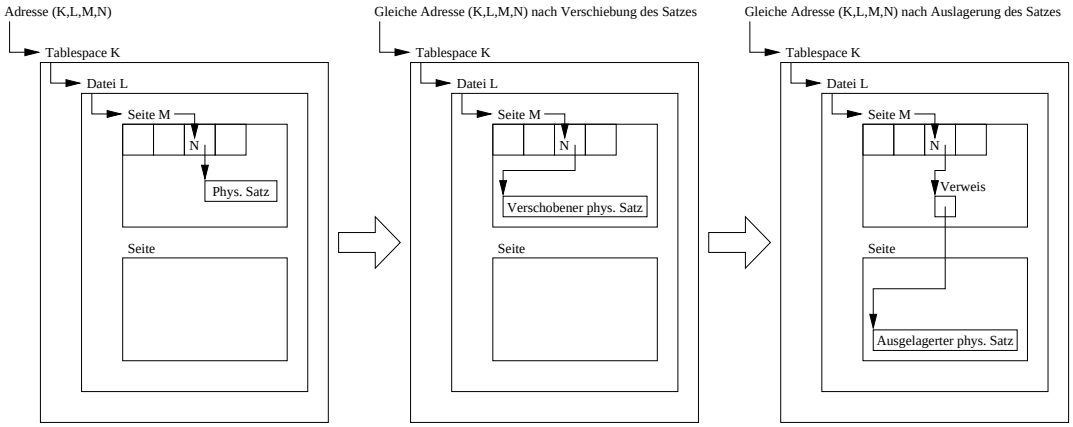


ABBILDUNG 2.6: Physische Adressierung mit TID-Konzept

2. *Physische Adressierung mit TID-Konzept*

Bei der physischen Adressierung setzt sich die Adresse aus Identifikatoren zusammen, die sich aus dem realen Speicherort des Satzes ableiten. Die folgenden Erläuterungen sind an dem Adressierungskonzept von Oracle orientiert. Physische Adressen eines internen Satzes setzen sich dort aus vier Teilen zusammen (siehe ABBILDUNG 2.6). Der erste Teil identifiziert den Tablespace, der zweite Teil die physische Datei für die Fälle, in denen ein Tablespace auf mehreren Dateien verteilt ist. Der dritte Teil zeigt auf die Speicherseite innerhalb der Datei (Seitennummer).

Eine Speicherseite besteht aus einem Seitenkopf und einem Datenteil. Während im Datenteil die eigentlichen internen Sätze abgelegt werden, besteht der Kopf einer Seite im wesentlichen aus einem kleinen Feldindex, dessen Einträge auf die einzelnen Sätze zeigen, die auf dieser Seite vorhanden sind.

Der vierte Teil einer physische Adresse besteht aus einem Verweis auf den Satzindex. Durch dieses Vorgehen wird ermöglicht, daß sich die Position eines Satzes innerhalb einer Seite ändern kann, ohne daß dies Auswirkungen auf die nach außen sichtbare Adresse hat. Es genügt eine Änderung im Satzindex der Seite.

Verschiebungen von Sätzen können immer dann nötig sein, wenn sich durch Änderungsoperationen auf variabel langen Feldern die Größe eines internen Satzes ändert. Sie stellen, wie eben festgestellt wurde, kein Problem dar. Schwierigkeiten ergeben sich, wenn der auf der Seite vorhandene Platz zu klein und eine Verschiebung auf eine andere Seite nötig wird.

Bei seitenübergreifender Verschiebung sind die beiden letzten Teile der physischen Adresse eines Satzes betroffen. Um im Fall einer seitenübergreifenden Verschiebung dennoch eine konstante Adresse zu gewährleisten, wurde das sogenannte Tupel-ID-Konzept (TID oder auch Row-ID beziehungsweise RID) eingeführt. Dabei wird die neue Adresse unter der ursprünglichen Adresse des Satzes (Stellvertreter-ID) gespeichert. Kommt es erneut zu einer Verschiebung auf eine weitere, dritte Seite, wird die Adresse wiederum nur auf der ersten, ursprünglichen Seite geändert. Auf diese Art und Weise bleibt die Referenzierungskette maximal zweistufig, eine Veränderung der Referenzen innerhalb anderer Objekte – wie zum Beispiel Indexe oder Verweise mit einem Hinweis auf die physische Speicherposition, auf die später noch eingegangen wird – ist nicht notwendig.

Geht man beispielsweise davon aus, daß 10% der internen Sätze einer Relation auf eine andere Speicherseite ausgelagert wurden, kommt man zu einem Tupel mit durchschnittlich 1.1 Seitenzugriffen.

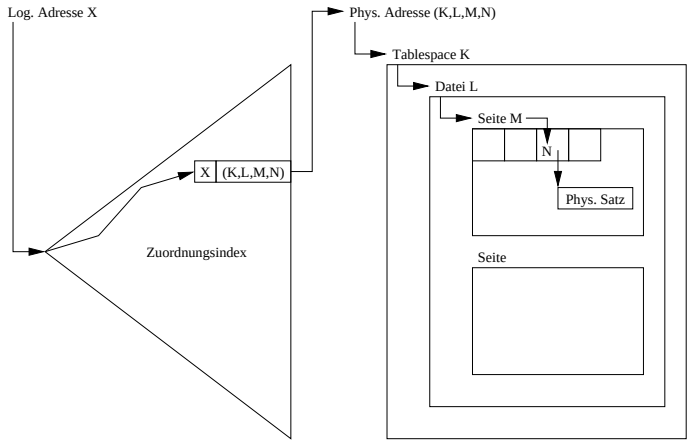


ABBILDUNG 2.7: Logische Adressierung mit Zuordnungsindex

Die Vorteile der physischen Adressierung liegen darin, daß Indirektionen

- ▷ Nur in den wirklich benötigten Fällen angelegt werden
- ▷ Maximal zweistufig sind
- ▷ Lokal abgespeichert werden und dadurch die Anzahl der Seitenzugriffe exakt abschätzbar ist

Kommt es zu häufigen Verschiebungen von Sätzen, ist physische Adressierung im Prinzip nicht mehr brauchbar. Dies ist zum Beispiel der Fall bei indexorganisierten Tabellen, die beinahe ‚regelmäßige' physische Verschiebungen von Datensätzen durch Änderungen von Clusterschlüsselwerten, Baum-Splits und so weiter erfordern. Abgesehen vom Speicher-Overhead durch eine große Zahl von Verweissätzen, die durch solche Verschiebungen entstehen würden, müßte vor allem ab einem schnell erreichten bestimmten Degenerierungsgrad der größte Teil aller Satzzugriffe über zwei Seitenzugriffe erfolgen.

3. Logische Adressierung

Bei der zweiten Art der indirekten Adressierung, der logischen Adressierung, wird die Indirektion explizit innerhalb eines Zuordnungsindex verwaltet (siehe ABBILDUNG 2.7). Referenzen auf Sätze bestehen aus einem logischen Schlüssel. Dieser kann dabei entweder ein künstlicher Identifikator oder der Primärschlüssel des Tupels sein.

Letztere Möglichkeit bringt allerdings verschiedene Nachteile mit sich. Eine Werteänderung des Primärschlüssels hätte zur Folge, daß sämtliche Referenzen in Indexen geändert werden müßten.

Primärschlüssel sind unter Umständen relativ lang und führen somit zu einem hohen Speicheraufwand in Indexen. Alternativ ist es möglich, basierend auf dem Primärschlüssel über eine Hashfunktion einen künstlichen, eindeutigen Schlüssel zu erzeugen.

Der Zuordnungsindex bei der Verwendung von Identifikatoren, die auf dem Primärschlüssel basieren, kann im allgemeinen nur mit Hilfe eines Baum- oder Hashindexes implementiert werden. Grund dafür ist die variable Länge beziehungsweise sind die potentiell zur Verfügung stehenden unterschiedlichen Typen für die Primärschlüsselattribute. Aufgrund dieser spezifischen Indextypen ergibt sich ein entsprechend großer Zugriffsfaktor für die Transformation des logischen Schlüssels in die physische Adresse.

4. Künstliche Schlüssel bei der logischen Adressierung

Setzt man hingegen vom Datenbanksystem vergebene logische Schlüssel ein, kann man unter bestimmten Voraussetzungen den Transformationsaufwand drastisch reduzieren. Werden als logische Schlüssel fortlaufende ganze Zahlen eingesetzt, kann der Transformationsindex als seitenübergreifender Feldindex mit Feldern gleicher Länge realisiert werden. Der logische Schlüssel entspricht dann der Position eines Feldes, im Feld selber findet sich die physische Adresse des Satzes. Bei dieser Art des logischen Schlüssels erreicht man einen konstanten Zugriff mit zwei Lesezugriffen. Bekannt ist dieses Konzept unter der Bezeichnung Database Key [COD78].

Probleme bereitet diese Variante allerdings, wenn es um Fragen der Reorganisation beziehungsweise Wiederverwertbarkeit von Identifikatoren geht. Soll – wie eigentlich für Objektidentifikatoren gefordert – eine über die Lebenszeit des Objekts andauernde Eindeutigkeit gewährleistet sein, kann diese Alternative überhaupt nur dann eingesetzt werden, wenn innerhalb der Anwendung relativ selten Objekte gelöscht werden und insgesamt eine abgrenzbare Menge neuer Objekte hinzukommt. Letzteres ist wegen der endlichen Anzahl der zur Verfügung stehenden Identifikatoren notwendig.

Zwar ist der Reorganisationsaufwand bei logischen Referenzen deutlich geringer als bei physischen Referenzen, jedoch ist hier jedem Satzzugriff ein Indexzugriff vorangestellt, der die physische Adresse des internen Satzes ermittelt. Aus diesem Grund kann eine Kombination aus beiden Konzepten genutzt werden [HR01].

5. Hybrider Ansatz: logische Adressierung und PPP

Eine Referenz besteht hierbei aus zwei Teilen: der erste Teil ist ein logischer Satzidentifikator, der innerhalb eines Index zu der gültigen Adresse des Satzes führt. Der zweite Teil beinhaltet eine TID mit der wahrscheinlichen physischen Adresse des Satzes (PPP – Probable Page Pointer, auch „physical guess“). Bei einem Satzzugriff über einen Index wird zunächst versucht, den Satz unter der PPP Adresse zu finden. Wurde dieser zwischenzeitlich verschoben, muß die logische Adresse in die gültige TID des Satzes transformiert werden.

Vorteil dieser Methode ist, daß, wie bei rein logischen Referenzen, seitenübergreifende Satzverschiebungen nur Änderungen in der Transformationstabelle zur Folge haben. Auf der anderen Seite sind die durchschnittlichen Seitenzugriffskosten beim PPP annähernd so gering wie bei der rein physischen Referenzierung. Allerdings muß für diese Vorteile auch ein erhöhter Speicherbedarf für Referenzen in Kauf genommen werden, da ja eine logische und eine physische Adresse gespeichert werden müssen. Auch kann der Hinweis auf die physische Adresse nur dann Zugriffskosten sparen, wenn er hinreichend oft richtig ist. Anderenfalls würden die Zugriffskosten sogar steigen, da ja stets ein zusätzlicher vergeblicher Zugriff erfolgen würde.

2.5.3 Anforderungen an ein erweitertes Zugriffs- und Speichersystem

Ebenso wie bei der Definition konventioneller Relationen mit einfachen Attributen, ergeben sich bei der physischen Speicherung erweiterter Typen unmittelbar eine Reihe von Minimalanforderungen an die Wahlmöglichkeiten bei der Speicherung.

Da mit der Einführung komplexer Objekte nun endgültig die Analogie „Tupel einer Relation $\Leftrightarrow$ ein interner Satz“ hinfällig wird, besteht eine der wichtigsten Forderungen an

das Speichersystem darin, daß Sätze verschiedenen Typs zusammen auf einer Seite gespeichert werden können. Einige Datenbanksysteme, wie zum Beispiel Oracle, unterstützen dies bereits mit dem Konzept der Cluster.

Oberstes Ziel bei der Speicherung komplexer Objekte ist es, je nach Anforderungsprofil eine optimale Speicherdichte zu erhalten, wobei insbesondere bei häufigen Änderungsoperationen eine optimale Dichte nicht automatisch mit einer möglichst hohen Dichte gleichzusetzen ist. Gerade im Zusammenhang mit parallelem Zugriff ist oftmals sogar eine gewisse verteilte Speicherung optimal.

Ein weiterer wichtiger Parameter wird durch die Frage bestimmt, welche Teile eines Objekts beziehungsweise einer Klasse dicht gespeichert und welche Teile (im allgemeinen selten benötigte Attribute) in einen anderen Bereich des Speicherraums ausgelagert werden sollen. Es geht also um die Anwendbarkeit von Clusterungs- und Partitionierungsstrategien.

Je nach gewählter Speichervariante können Größenänderungen an Attributen eines Objekts einen mehr oder weniger hohen Reorganisationsaufwand der gewählten Speicherstruktur zur Folge haben. Auch hier ist die pauschale Definition eines Optimums unzulässig; vielmehr müssen für jedes spezielle Anforderungsprofil neu die Vor- und Nachteile erwogen werden.

Objektrelationale DBMS sollten also verschiedene Varianten und Optionen zur Speicherung komplexer Objekte anbieten. Der Benutzer beziehungsweise der Administrator wählt darunter die für seine Anwendung passende Speicherart aus.

Längerfristig wären an dieser Stelle allerdings auch automatische beziehungsweise teilautomatische Mechanismen denkbar, die auf der Basis anwendungsspezifischer Zugriffsanalysen bei der Auswahl der physischen Variante helfen beziehungsweise Vorschläge für sinnvolle Reorganisationen geben.

2.6 Zusammenfassung des Kapitels

Ziel dieses Kapitels war es, die Grundkonzepte objektrelationaler DBMS zusammen zu fassen und so eine Basis für die Beschäftigung mit der Speicherung und Indexierung komplexer Objekte zu schaffen. Dazu wurde kurz die Entstehung objektrelationaler DBMS als Weiterentwicklung relationaler DBMS zur Verringerung des Impedance Mismatch angerissen.

Nach der Diskussion allgemeiner Paradigmen der Objektorientierung wurde auf die neuen Konzepte eingegangen, die durch die Verschmelzung der objektorientierten und der relationalen Modelle in objektrelationalen DBMS entstanden sind. Insbesondere wurde auf das reichhaltige Typsystem in ORDBMS eingegangen, das die Konstruktion verschachtelter, komplexer Objekte mit einfachen, strukturierten und kollektionswertigen Attributen ermöglicht und so die Datenbankmodellwelt an die Objektmodellwelt in Programmiersprachen annähert.

Ergänzt wurde die Darstellung der Konzepte von ORDBMS um die Untersuchung des Zusammenspiels zwischen objektorientierten Anwendungen und objektrelationalen DBMS.

Das wesentliche Ziel der in den ersten drei Abschnitten des Kapitels vorgestellten Konzepte ist, wie schon angesprochen, die Reduzierung des Impedance Mismach zwischen Datenbanksystem und Anwendung. Allerdings sollte eine Verringerung der Kluft zwischen DBMS- und Anwendungsmodellwelt nicht zu Lasten der Datenunabhängigkeit führen.

Bei der Betrachtung der Datenunabhängigkeit in ORDBMS stellt sich jedoch heraus, daß gerade durch die Annäherung zwischen Anwendungsmodell und Datenmodell die Datenunabhängigkeit indirekt beeinträchtigt wird. Die objektrelationalen Konstrukte ermöglichen

nämlich eine weitgehend direkte, kanonische 1:1-Abbildung der Anwendungsobjekte in DB-Objekte. Diese auf den ersten Blick zu begrüßende, „natürliche“ Abbildung vereinfacht und verbessert zwar die Datenmodellierung, aber auf den zweiten Blick ist festzustellen, daß nun Änderungen der Speicherstrukturen zur Erhaltung oder Verbesserung der DB-Leistung oft bis auf die Anwendungsebene ‚durchschlagen‘. Dies liegt insbesondere an der fehlenden Trennung von logischen und physischen Aspekten im Datenbankentwurf bei ORDBMS-Prototypen und Produkten.

Aus diesen Überlegungen ergibt sich der Grundansatz dieser Arbeit: Durch die Einführung einer Zwischenschicht sollen logische und physische Datenmodellierung getrennt werden und so zu einer erhöhten Datenunabhängigkeit bei objektrelationalen DBMS führen. Die Beschreibung der physischen Strukturen zu den per SQL definerten logischen Datenstrukturen soll über eine Speicherspezifikationssprache erfolgen.

Um eine solche Trennung von logischen und physischen Modellierungsaspekten erreichen zu können, widmete sich der letze Abschnitt dieses Kapitels der Untersuchung der Integration komplexer Objekte in RDBMS. Es wurden die Techniken der Implementierung der Unterstützung komplexer Objekte untersucht, die sich, wie festzustellen ist, weitgehend auf die ‚oberen‘ DBMS-Architekturschichten Datensystem und Zugriffssystem beschränken. Die wenigen an das Speichjersystem zu stellenden Anforderungen werden zum guten Teil schon von RDBMS erfüllt.

Ausgehend von den in diesem Kapitel gelegten Grundlagen zu objektrelationalen DBMS können nun in den folgenden beiden Kapiteln 3 und 4 die Speicherungs- und Indexierungskonzepte für komplexe Objekte untersucht werden, auf deren Grundlage dann in Kapitel 5 die Speicherspezifikationssprache PRDL zur vom logischen Entwurf getrennten Steuerung physischer Speicherstrukturen entworfen wird.

Kapitel 3

Speicherstrukturen für komplexe Objekte

Dieses Kapitel beschäftigt sich mit der physischen Speicherung komplexer Objekte. Viele unterschiedliche Speicherformen sind für komplexe Objekte denkbar. Als zwei entgegengesetzte Vertreter dieser Variantenvielfalt seien hier die Speicherung als ein kompaktes Binärdatenobjekt und die normalisierte Speicherung in Relationen genannt. Nun wird die Verarbeitungsleistung bei Operationen auf diesen komplexen Objekten natürlich direkt von dieser Speicherungsform positiv oder negativ beeinflußt. So ist es unmittelbar einsichtig, daß eine kompakte Speicherung den Zugriff auf das komplette Objekt besser ermöglicht als die normalisierte Speicherung, die die Objektdaten über mehrere physische Sätze verteilt. Auf der anderen Seite bietet diese verteilte Speicherung möglicherweise einen Leistungsvorteil, wenn nur auf ein spezielles Subobjekt oder Attribut zugegriffen werden soll.

Dieses kleine Beispiel zeigt, das eine Variantenvielfalt von Speicherstrukturen für komplexe Objekte wünschenswert ist, da sie es ermöglicht, die Verarbeitung komplexer Objekte in objektrelationalen DBMS zu beschleunigen. Leider bieten heutige objektrelationale DBMS nur eine zu geringe Auswahlmöglichkeit bei der Objektspeicherung. Allerdings sind in der Literatur eine ansehnliche Reihe von Vorschlägen für alternative Respräsentationsformen für komplexe Objekte und Datenstrukturen zu finden.

Dieses Kapitel präsentiert also in seinem ersten Teil Speicherstrukturen aus der Literatur, aus DBMS-Prototypen und -Produkten. Durch den Vergleich dieser wird ein Spektrum von relevanten Ablagekonzepten erarbeitet, das auf seine Eignung für objektrelationale DBMS hin untersucht wird. Darauf aufbauend wird im zweiten Teil des Kapitels ein umfassender Satz von Speicherkonzepten und -alternativen vorgeschlagen und diskutiert. Er soll es objektrelationalen DBMS ermöglichen, eine hinreichende Unterstützung von verschiedensten Verarbeitungsanforderungen für komplexe Objekte anzubieten, und stellt so eine Grundlage zur Implementierung verbesserter Komplexobjektunterstützung in ORDBMS dar. Außerdem bildet er die Grundlage für die Speicherbeschreibungssprache PRDL, die die Steuerung und Auswahl der Speicherform für Objekte ermöglicht und in Kapitel 5 beschrieben wird.

3.1 Speicherstrukturen in Literatur und Produkten

In diesem Abschnitt sollen verschiedene Speichertechniken für komplexe Objekte, wie sie in der Literatur und DBMS-Produkten anzutreffen sind, vorgestellt werden. Die Betrachtun-

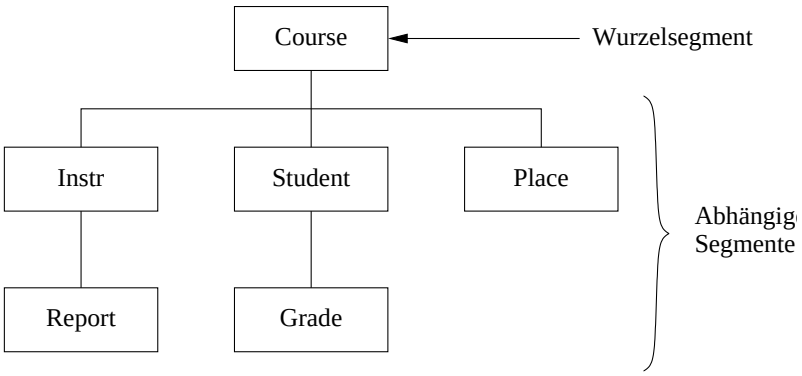


ABBILDUNG 3.1: Segmenttypen in einem IMS-Satz (IMS-Record)

gen beschränken sich nicht nur auf relationale und objektrelationale Systeme, da bereits in vorrelationalen System Implementierungstechniken für Datenstrukturen zu finden sind, die Gemeinsamkeiten mit komplexen Objekten besitzen und somit Grundlagen und interessante Ideen für neue Speicherstrukturansätze liefern können.

3.1.1 IMS

IMS ist ein hierarchisches DBMS von IBM. Anders als in relationalen Datenbanksystemen werden die Daten in IMS in einer Hierarchiestruktur gespeichert. Dabei werden die Daten in eine Serie von IMS-Sätzen gruppiert. Jeder dieser IMS-Sätze ist selbst wieder in kleinere Gruppen von Daten aufgeteilt. Diese werden IMS-Segmente genannt und stellen die kleinste Einheit dar, in der IMS Daten speichert (siehe ABBILDUNG 3.2). Jedes IMS-Segment ist aus einem oder mehreren Feldern aufgebaut. Vergleicht man diese Elemente mit den Speicherelementen eines relationelen Datenbanksystems, kann man Segmente von IMS mit Tupeln oder Sätzen vergleichen. Die Felder eines IMS-Segments entsprächen dann den Attributen eines Tupels.

Soll im folgenden zwischen dem Typ eines IMS-Segments und dessen Instanz oder Ausprägung unterschieden werden, so werden die Begriffe IMS-Segmenttyp und IMS-Segmentinstanz benutzt.

Zur Veranschaulichung ist in ABBILDUNG 3.1 ein Beispiel einer Segmenttyphierarchie aus [IBM04] dargestellt, welche die Daten einer Schule speichern soll. Die IMS-Segmenttypen modellieren dabei folgende Informationen:

- *Course*

  Der Kurs mit seinem Namen.
- *Instr*

  Der Lehrer des Kurses mit seinem Namen.
- *Report*

  Ein Bericht, der vom Lehrer am Ende des Kurses angefertigt wird.
- *Student*

  Die Schüler in dem Kurs jeweils mit Namen.
- *Grade*

  Der Abschluß, den ein Schüler in dem Kurs erlangt hat.
- *Place*

  Der Ort, an dem der Kurs abgehalten wird.

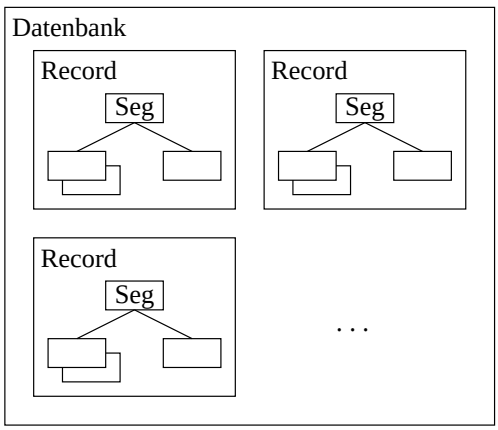
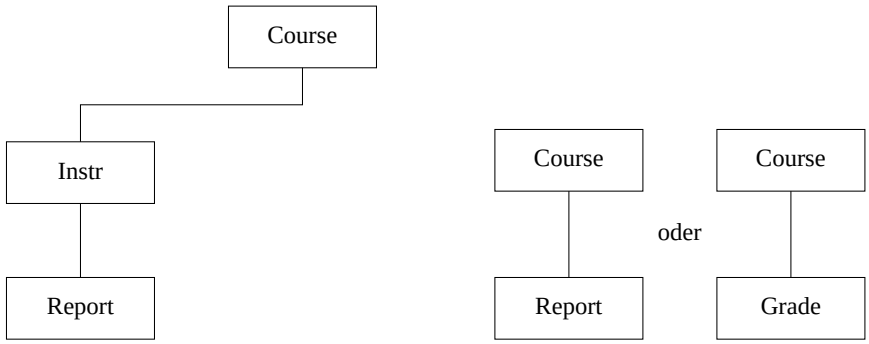


ABBILDUNG 3.2: Logische Sicht auf die IMS-Datenbank



(a) Gültig (b) Ungültig

ABBILDUNG 3.3: IMS-Satz-Ausprägungen

Diese Segmente stehen miteinander in Beziehung. Sie bilden eine Hierarchie beziehungsweise einen Baum mit dem Wurzelsegment (Root Segment) Course.

Sollen nun Daten in der IMS-Datenbank gespeichert werden, so muß für jeden IMS-Satz genau eine Instanz des Wurzelsegmenttyps vorhanden sein. Dessen untergeordnete IMS-Segmentinstanzen werden abhängige IMS-Segmente (Dependent Segments) genannt (Instr, Report, Student, Grade und Place sind abhängige IMS-Segmente von Course). Von dessen Existenz (Existenz der entsprechenden Instanzen) ist die Existenz der IMS-Satzinstanz jedoch nicht abhängig. Es sind somit natürlich auch unvollständig ausgeprägte Hierarchien möglich (siehe ABBILDUNG 3.3(a)).

Für untergeordnete IMS-Segmente besteht jedoch eine Abhängigkeit. Ein solches IMS-Segment kann nur existieren, wenn dessen zugehöriges Vatersegment (Parent Segment) existiert (siehe ABBILDUNG 3.3(b)). Dabei ist jedes IMS-Segment ein Vatersegment, wenn es ein abhängiges IMS-Segment in einer tieferen Hierarchiestufe besitzt. Entsprechend werden IMS-Segmente, die von anderen IMS-Segmenten abhängig sind, Kindersegmente (Child Segments) genannt. Im Beispiel ist Course das Wurzelsegment und Instr, Student und Place sind dessen Kindersegmente. Instr ist zugleich aber auch Vatersegment von Report, wie auch Student Vatersegment von Grade ist.

Neben den bereits erwähnten Beziehungen zwischen IMS-Segmenten, existiert noch eine

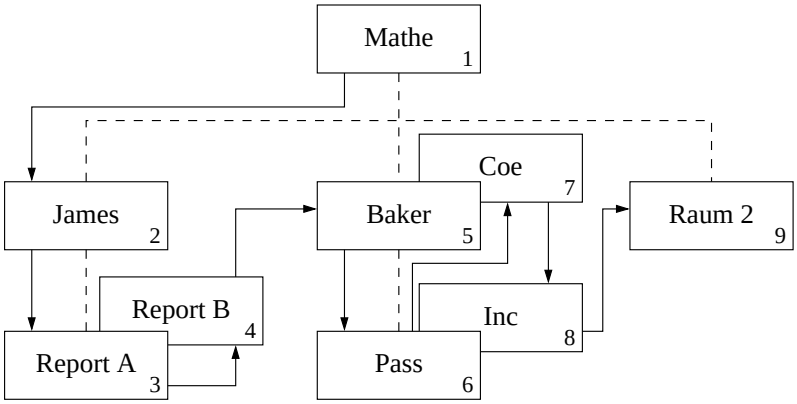


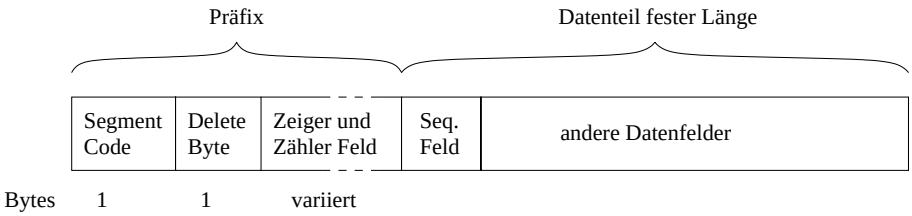
ABBILDUNG 3.4: Speicherreihenfolge der Segmente innerhalb eines IMS-Satzes (IMS-Record)

weitere Beziehung in IMS. Gehören zwei oder mehr Instanzen eines IMS-Segmenttyps zur gleichen Vatersegmentinstanz (Instanz des übergeordneten Segmenttyps), so werden diese als Zwillingssegmente (Twin Segments) bezeichnet.

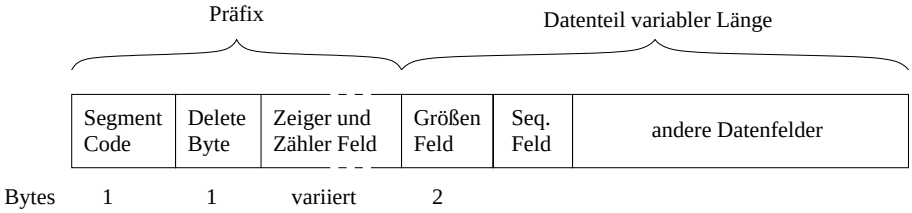
IMS-Sätze werden gespeichert, in dem die zugehörigen Segmente in der Präorder-Reihenfolge ihres Hierarchiebaumes abgelegt werden, wie es ABBILDUNG 3.4 verdeutlicht.

Ist einer Datenbank ein bestimmter Satztyp (bestimmt durch die Typen der IMS-Segmente und des Hierarchiebaums) zugeordnet, so ist es nicht möglich, Sätze anderen Typs in der Datenbank zu speichern. Hierfür muß jeweils eine separate Datenbank angelegt werden. Damit entspricht eine IMS-Datenbank also in etwa einem Segment oder einem Tablespace in relationalen Systemen. Und IMS-Segmente entsprechen am ehesten den Datensätzen in RDBMS und stellen die kleinste Speichereinheit dar. Ein IMS-Segment ist einem Segmenttyp zugeordnet und besteht wie ein ‚relationaler' Datensatz aus einzelnen Feldern (Attributen). Dabei schränkt IMS die Anzahl der IMS-Segmenttypen pro Datenbank auf 256 ein, und deren Attributanzahl ist auf 1000 beschränkt. Zur Speicherung variabel langer Attribute werden IMS-Segmente fester und variabler Länge unterschieden. In ABBILDUNG 3.5 werden beide IMS-Segmentformate veranschaulicht. Damit IMS den Typ der gespeicherten Daten bestimmen kann, wird jedem IMS-Segmenttyp eine eindeutige Kennung zugewiesen. In IMS ist dies ein Integerwert von 1 bis 255, der im ersten Byte jedes IMS-Segments gespeichert wird (Segment Code). Im Zeigerbereich (Pointer Area) werden die Adressen der Segmente gespeichert, auf die das Segment zeigt (nicht in allen Datenbanktypen vorhanden; zu Datenbanktypen siehe nächsten Absatz). Im Datenbereich stehen die Daten der Attribute (Felder) des Segments. Dabei kann ein Feld des Segments als Sequenzfeld definiert werden. Die Daten in diesem Feld werden dann Schlüssel (Key) genannt und können als eindeutig oder nicht eindeutig definiert werden. Ein Sequenzfeld erlaubt es, die Instanzen eines IMS-Segmenttyps in einer gewissen Reihenfolge abzuspeichern. Wird zum Beispiel ein Zeichenkettenfeld als Sequenzfeld definiert, so speichert IMS die Instanzen des IMS-Segments alphabetisch. Ist das IMS-Segment von variabler Länge, so steht im Datenteil zusätzlich noch ein Längenfeld, welches die Länge des IMS-Segments enthält.

IMS kann seine Daten auf unterschiedliche Weise speichern. Daher gibt es für jede Speicherungsart einen eigenen Datenbanktyp. Im folgenden sollen die wesentlichsten kurz aufgelistet werden:



(a) IMS-Segment fester Länge



(b) IMS-Segment variabler Länge

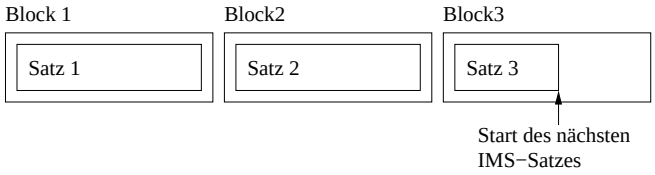
ABBILDUNG 3.5: Speicherformat von IMS-Segmenten

▷ *Sequentielle Speichermethoden*
Hierbei werden die IMS-Segmente eines IMS-Satzes (IMS-Record) physisch benachbart gespeichert. Dabei sind sie durch die physische Hintereinanderreihung miteinander verbunden.

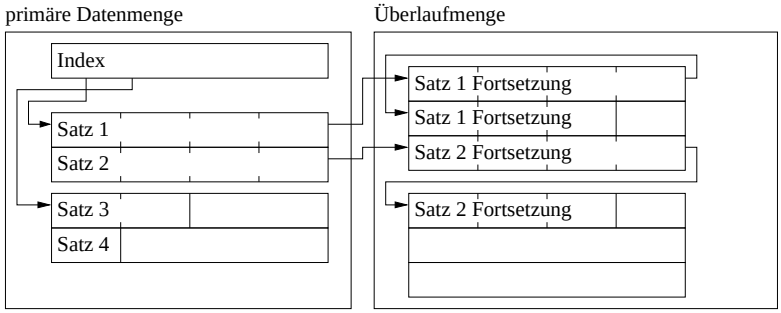
Folgende Speichervarianten werden von IMS angeboten:

◇ *HSAM: Hierarchical Sequential Access Method*
Die Segmente eines IMS-Satzes werden sequentiell in hintereinanderliegende Seiten, hier Blöcke genannt, (geclustert) gespeichert, wie es ABBILDUNG 3.6(a) zeigt. Paßt dabei ein IMS-Segment nicht mehr in die angefangene Seite, so wird die darauffolgende Seite genutzt (keine seitenübergreifenden Sätze). Ein folgender IMS-Satz kann dabei eine von seinem Vorgängersatz belegte Seite auffüllen. Auf die IMS-Sätze sowie deren IMS-Segmente (variable Länge nicht erlaubt) kann nur sequentiell zugegriffen werden.

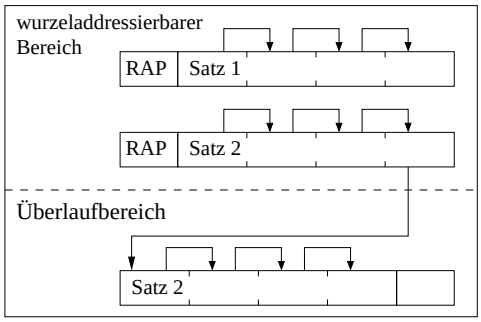
◇ *HISAM: Hierarchical Index-Sequential Access Method*
Auf die IMS-Sätze wird mittels INDEX zugegriffen. Auf die IMS-Segmente aber nur sequentiell. Dabei wird der Speicherbereich für die IMS-Segmente geteilt (siehe ABBILDUNG 3.6(b)). In der primären Datenmenge werden alle Wurzelsegmente gespeichert. Jedes davon belegt eine Seite. Ist darin noch genügend Platz für einige abhängige IMS-Segmente, so werden diese in der Seite mit gespeichert. Alle übrigen abhängigen IMS-Segmente werden in der Überlaufspeichermenge gespeichert. Eine dort belegte Seite enthält nur Daten eines IMS-Saztes. Eine geclusterte Speicherung in DB-Segmente ist nicht möglich.



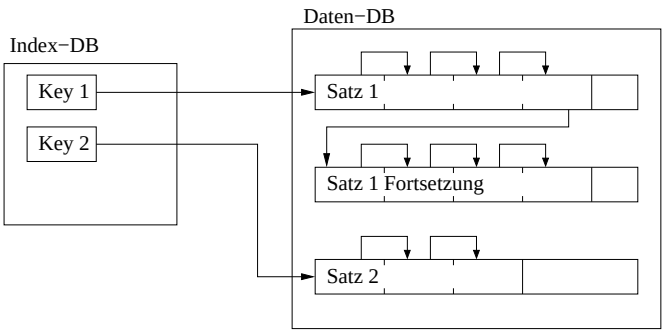
(a) HSAM-Datenbank



(b) HISAM-Datenbank



(c) HDAM/PHDAM-Datenbank



(d) HIDAM/PHIDAM-Datenbank

ABBILDUNG 3.6: Speicherungsmethoden verschiedener IMS-Datenbanktypen

▷ *Direkte Speichermethoden*

Hierbei werden auf die IMS-Segmente direkt mittels Zeigern zugegriffen. Diese Zeiger stehen in jedem Präfix eines IMS-Segments. Dabei existieren folgende drei Arten von Zeigern:

◇ *Hierarchische Zeiger,*
welche von einem IMS-Segment zum anderen IMS-Segment zeigen. Die Reihenfolge der IMS-Segmente ist durch die Hierarchie vorgegeben, das heißt, sie werden in Pre-Order gespeichert (siehe ABBILDUNG 3.4).

◇ *Zeiger auf Kindersegmente,*
welche vom Vatersegment auf das erste Kindersegment oder auf das erste und letzte Kindersegment zeigen.

◇ *Zeiger auf Zwillingssegmente,*
welche auf das nächste Zwillingssegment zeigen.

Für alle genannten Zeiger können in IMS auch Rückwärtszeiger gespeichert werden.
Folgende direkte Speichervarianten unterstützt IMS:

◇ *(P)HDAM: (Partitioned) Hierarchical Direct Access Method*
Die Datenbanken der Typen HDAM und PHDAM bestehen aus zwei Teilen: ein wurzeladdressierbarer Bereich und ein Überlaufbereich (siehe ABBILDUNG 3.6(c)). Der wurzeladdressierbare Bereich enthält die Wurzelsegmente und stellt den primären Speicherbereich für die abhängigen IMS-Segmente dar. In ihm wird auf die Wurzelsegmente der IMS-Sätze zugegriffen, indem in jedem Satz die relativen Byteaddressen der darin enthaltenen Wurzelsegmente gespeichert sind. Diese Adressen werden in IMS Root Anchor Points (RAP) genannt. Passen IMS-Segmente nicht in diesen Bereich, so werden sie in dem Überlaufbereich gespeichert. Die entsprechende Größe des wurzeladdressierbaren Bereichs muß angegeben werden. Außerdem muß festgelegt werden, wieviel Platz (in Byte) ein IMS-Satz im wurzeladdressierbaren Bereich einnehmen darf.

◇ *(P)HIDAM: (Partitioned) Hierarchical Indexed Direct Access Method*
Im Gegensatz zu HDAM/PHDAM haben die Typen HIDAM und PHIDAM keinen wurzeladdressierbaren Bereich und keinen Überlaufbereich (siehe ABBILDUNG 3.6(d)). Beide Typen werden jedoch in zwei Datenbanken getrennt: eine Datenbank für die Daten und eine für die Indexe auf den Wurzelsätzen.

Sollen nun komplexe Objekte in IMS gespeichert werden, so bietet sich eine Hierarchie der IMS-Segmente an, um die möglicherweise hierarchisch aufgebauten Objekte direkt abzubilden. Jedes komplexe Objekt wird genau in einem IMS-Satz (Database Record) in der Datenbank gespeichert. Hat ein komplexes Objekt ein mengenwertiges Attribut, so kann für den Elementtyp der Menge ein IMS-Segmenttyp definiert werden. Jede Instanz dieses IMS-Segments ist dann ein Element der Menge.

Für IMS läßt sich die Nutzung folgender Konzepte zur Speicherung komplexer Datenstukturen zusammenfassen:

▷ Auslagerung von Kollektionselementen.

▷ Clusterung.

- ▷ Kollektionen als (einfach und doppelt) verkettete Liste (xDAM).
- ▷ Kollektionen als sequentielle Reihung von IMS-Segmenten (HSAM).
- ▷ Beschränkung der IMS-Segmentgröße (Größe der Tupel/Sätze) durch die physische Seitengröße.
- ▷ Nutzung von seitenübergreifenden Sätzen. IMS-Sätze können über mehrere Seiten verteilt werden.
- ▷ Nutzung von Indexen bei (HISAM, HIDAM, PHIDAM).

Die genannten Konzepte sind jedoch abhängig vom jeweils gewählten Datenbanktyp. So ist zum Beispiel in HSAM-Datenbanken die Clusterung direkt integriert, in HISAM wiederum nicht. Auch sind Kollektionen unterschiedlich implementiert. Sind diese in xDAM als einfach oder doppelt verkettete Liste implementiert, so sind sie in HSAM dagegen einfach nur als hintereinander plazierte IMS-Segmente abgelegt.

3.1.2 MAD und PRIMA

PRIMA ist ein DBMS-Prototyp, der eine Implementierung des Molekül-Atom-Datenmodells (MAD) darstellt [HMWMS87]. Beim MAD-Modell werden komplexe Objekte als molekulare Objekte oder kurz Moleküle bezeichnet. Jedes Molekül besteht aus einfacheren Molekülen und gehört zu einem Molekültyp. Die einfachsten Moleküle werden Atome genannt. Jedes Atom ist aus Attributen mit verschiedenen Typen zusammengesetzt. Zusätzlich besitzen Atome einen Identifikator und werden einem Atomtyp zugeordnet.

Neben den üblichen primitiven Typen gibt es noch die zusätzlichen Typen Record, Array, Set und List. Somit sind komplexe Objekte mit mengenwertigen Attributen möglich.

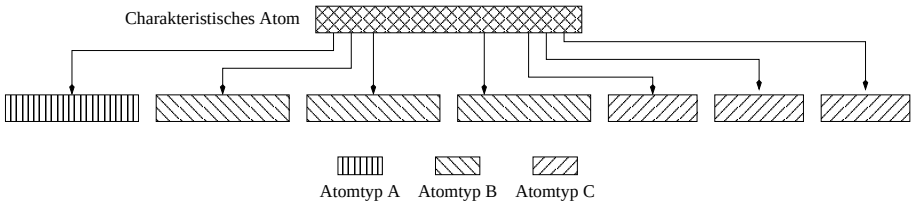
Um Atome untereinander verbinden und damit hierarchisch strukturierte Objekte definieren zu können, werden zwei spezielle Attributtypen eingeführt: Identifier und Reference. Der Identifier-Typ stellt Surrogate bereit, welche jedes Atom eindeutig identifizieren. Diese nutzen die Referenzen (Reference-Typ) zum Verbinden mit anderen Atomen.

Referenziert nun ein Atom ein anderes, so ist dies nicht Teil des Atoms, sondern nur die Referenz darauf. Dies ist für komplexe Objekte ungünstig, da dort die referenzierten Objekte zum Objekt dazugehören sollen. Hierfür können im MAD-Modell Molekültypen definiert werden, welche aus den definierten Atomtypen oder Molekültypen zusammengesetzt sind.

Werden in PRIMA nun komplexe Objekte (Moleküle) geschrieben beziehungsweise gelesen, so werden die Moleküle vom Datensystem in die entsprechenden Atome zerlegt. Diese werden dann weiter an das Zugriffssystem gegeben, wo die Atome in physische Sätze umgeformt und in Segmente/Seitenmengen gepackt werden. Von dort aus geht es weiter an das Speichersystem, welches die Seiten physisch ablegt.

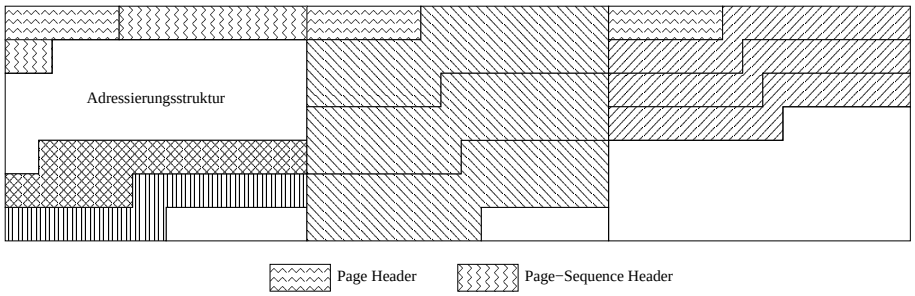
Jedes Atom hat also im allgemeinen seinen eigenen Satz, in dem seine Attribute gespeichert werden. Es gibt jedoch auch die Möglichkeit, daß sich mehrere Atome einen Satz teilen, das heißt zusammen in einem Satz liegen. Diese Sätze enthalten dann sogenannte Atom-Cluster. Dadurch ist eine physisch enge Speicherung von zusammenhängenden Atomen möglich, wie zum Beispiel aller Atome eines bestimmten komplexen Objekts (Moleküls). Implementiert wird dies durch Anlegen eines sogenannten charakteristischen Atoms. Dies enthält Referenzen auf alle Atome, welche zum Atom-Cluster gehören. In Abbildung 3.7 wird dies verdeutlicht.

Physische Sätze können nicht nur einzelne oder mehrere Atome beinhalten, sondern auch Teile von Atomen, welche Partitionen genannt werden. Diese sind Ergebnis von Projektionen auf Atomen und werden redundant zu den eigentlichen Atomen gespeichert. Letztendlich sind physische Sätze nur Bytefolgen variabler Länge.



(a) Logische Sicht

(b) In einem physischen Satz



(c) In einer Seitenmenge

ABBILDUNG 3.7: Atom-Cluster

Damit Sätze in Seiten und Segmenten gespeichert werden können, müssen diese Konstrukte vom Speichersystem angeboten werden. Hierbei unterstützt das Speichersystem von PRIMA Seiten unterschiedlicher Größe. Für jedes Segment kann als Seitengröße 0.5, 1, 2, 4 oder 8 Kilobyte gewählt werden. Ist die Einschränkung auf eine dieser Größen zu stark, so ist es auch möglich, Seitensequenzen zu definieren. Diese werden dann als Einheit behandelt. Eine dieser Seiten wird als Kopfseite (Header Page) bezeichnet und enthält neben dem üblichen Seitenkopf einen Seitensequenzkopf (Page-Sequence Header). Dieser enthält eine Liste aller Seiten der Sequenz. Diese Speichermöglichkeit ist für Atom-Cluster gut geeignet.

Komplexe Objekte können in PRIMA also als Ganzes in Form von Atom-Clustern gespeichert werden, oder es können alle Komponentenobjekte separat abgelegt werden. Atom-Cluster können jodoch auch zur Clusterung von Atomen genutzt werden, die anderweitig zusammengehören.

Die in PRIMA anzutreffenden Konzepte zur Speicherung komplexer Objekte können in folgender Auflistung zusammengefaßt werden:

- ▷ *Auslagerung von Kollektionselementen, falls diese Objekte sind und Speicherung solcher Kollektionen als Zeigerfeld (Pointer Array)*

  Nur die Referenzen werden im Atom gespeichert.

- ▷ *Inline-Speicherung von Kollektionselementen, falls diese primitiv sind*

{ Wissenschaftler }					
Name	**{ Bildung }**			**Alter**	**{ Mitglied }**
	Grad	**Jahr**	**Uni**		**Organisation**
Doe	Master	76	UT	35	ACM
	PhD	79	MIT		IEEE
	...				...
Jones	PhD	85	UT	26	
	...				...
...					

ABBILDUNG 3.8: NF^2-Beispieltabelle mit komplexen Objekten

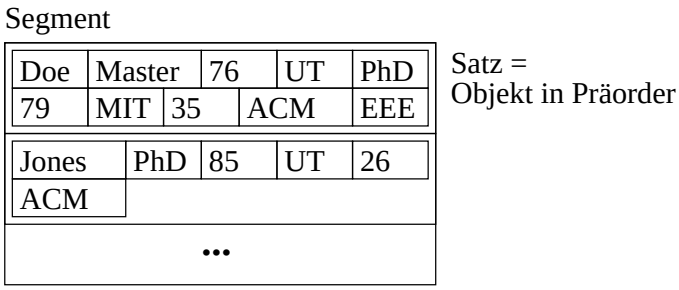


ABBILDUNG 3.9: Direct Storage Model

- ▷ *Clusterung*
  Atom-Cluster speichern Atome physisch dicht.
- ▷ *Seitenübergreifende Sätze*
  Große Sätze können aufgeteilt auf hintereinanderliegende Seiten (Seitensequenzen) gespeichert werden.

3.1.3 Speichermodelle nach Valduriez, Khoshafian und Copeland

In [VKC86] werden normalisierte und sogenannte direkte Speichermodelle für komplexe Objekte behandelt.

Dabei sollen die Speichermodelle durch das in ABBILDUNG 3.8 dargestellte Beispiel einer NF^2-Tabelle verdeutlicht werden. Darin werden Wissenschaftler und deren Eigenschaften gespeichert. Zusätzlich zu den dargestellten Spalten erhält jedes Tupel einer Tabelle/Untertabelle eine eindeutige Kennung (Surrogat), die die Spalte eindeutig identifiziert. Als Untertabellen sind hier Bildung und Mitglied modelliert.

1. Direct Storage Model

In diesem Modell werden die komplexen Objekte so gespeichert, wie sie im konzeptuellen Schema definiert wurden, das heißt als Ganzes (ABBILDUNG 3.9). Dabei werden Tabellen (hier als Mengenobjekte bezeichnet) in eigene Segmente gespeichert. Die Segmente wiederum sind in Sätze eingeteilt. Jedes Tupel der Tabelle stellt ein Objekt dar. Diese können strukturiert sein, das heißt mengenwertige Attribute und darin enthaltene Unterobjekte besitzen. Ein Tupel belegt dabei einen Satz des Segments, das heißt ein Satz pro Objekt. Für die Art und Weise, wie die Komponenten eines Objekts in einen Satz gespeichert werden,

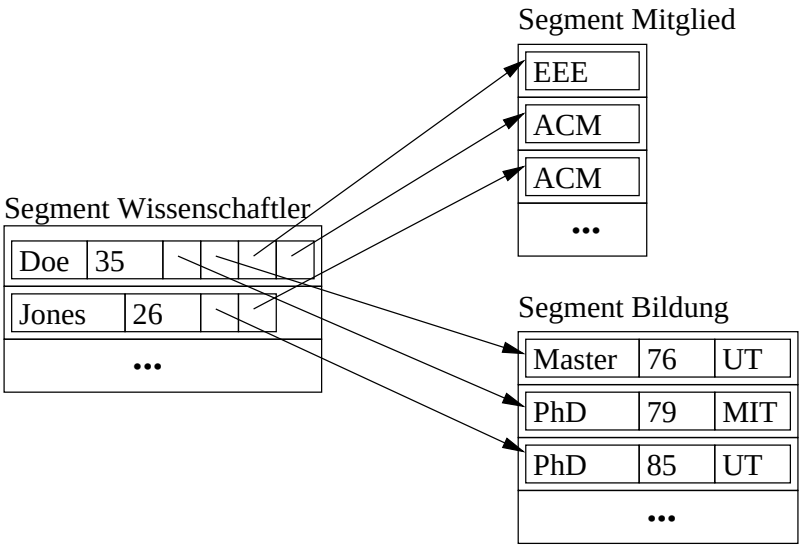


ABBILDUNG 3.10: Normalized Storage Model

bestehen mehrere Möglichkeiten. Hier wird jedoch die einfache Pre-Order-Variante angegeben. Es sind jedoch auch komplexere Techniken möglich, die die Komponenten in einem Objekt nach gewissen Kriterien clustern (zum Beispiel alle Elemente eines mengenwertigen Attributs nach einem Attribut des Elementtyps).

Eine Clusterung zwischen Komponenten verschiedener Objekte ist durch die kompakte Speicherung nicht möglich. Es besteht aber die Möglichkeit, die Objekte untereinander zu clustern. Das heißt, Objekte mit gewissen Eigenschaften werden benachbart in das Segment gespeichert. Diese Eigenschaften sind jedoch nur den atomaren Attributen des Wurzelobjekts zu entnehmen. Im Beispiel kämen nur Name und Alter in Frage. Es können jedoch auch aus mehreren Attributen hergeleitete Eigenschaften zur Clusterung verwendet werden. Die nicht sichtbaren Surrogate können auch zur Clusterung hinzugezogen werden.

Vorteile dieser Speichermodelle sind:

- ▷ Effiziente Anfragen an gesamte Objekte
- ▷ Kein struktureller Unterschied zwischen konzeptuellen Objekten und internen Objekten

Gleichzeitig kann man folgende Nachteile feststellen:

- ▷ Objekte durch Seitengröße beschränkt, falls keine seitenübergreifenden Sätze genutzt werden
- ▷ Zugriff auf Unterobjekte ist ineffizient, da stets ganze Objekte gelesen und geschreiben werden müssen

2. Normalized Storage Model

In diesem Modell werden die Objekte nicht direkt als Ganzes gespeichert, sondern in Mengen von Tupeln atomarer Werte aufgeteilt (ABBILDUNG 3.10). Im Beispiel wird die Tabelle mit den Objekten vom Typ Wissenschaftler in drei einzelne Tabellen aufgeteilt (Wissenschaftler, Bildung und Mitglied). Die Verbindungen unter den zusammengehörigen Tupeln wird durch Surrogate erreicht, das heißt systemweit eindeutige Kennungen. Die neuen Tabellen werden dann auf mehrere Segmente verteilt.

Vorteile bei diesem Modell sind:

- ▷ Effiziente Anfragen an Teile von Objekten
- ▷ Geringere Größeneinschränkung (durch Seitengröße)

Als Nachteile stehen zu Buche:

- ▷ Zugriffe auf vollständige Objekte erfordern mehrere Satzzugriffe und Verbundoperationen, sie sind deshalb kostspieliger
- ▷ Aufwendigere Abbildung vom Objekt in seine Speicherform als beim Direct Storage Model

Für eine Aufteilung der Tabellen auf mehrere Segmente werden sogenannte horizontale und vertikale Partitionierungsfunktionen verwendet. Die vertikale Aufteilung bestimmt, welche Attribute in das gleiche Segment gehören. Die Horizontale Aufteilung bestimmt die Anordnung der Tupel innerhalb des Segments (Clusterung).

Es gibt verschiedene mögliche Partitionierungen:

- ▷ *NSM (N-ary Storage Model)*[1]

  Hierbei wird jede Tabelle (im Beispiel drei) in ein Segment gespeichert. Selektionen sind hier nur dann effizient, wenn nach dem geclusterten Attribut selektiert wird. Am besten werden Projektionen unterstützt.

- ▷ *DSM (Decomposition Storage Model)*[1]

  Jedes Attribut (inklusive Surrogate) kommt potentiell in ein separates Segment. Dies ist die beste Aufteilung für Selektionen und Projektionen auf wenige Attribute.

- ▷ *P-DSM (Partial DSM)*

  Dies ist eine Mischform der beiden oberen. Zusammengehörige Attribute (zum Beispiel häufig zusammen abgefragt) werden zusammen in ein Segment gespeichert.

Um Verbundanfragen (Join Queries) effizienter zu unterstützen, können weiterhin Verbundindexe erstellt werden. Dies sind vorberechnete Verbunde von vorerst zwei Tabellen. Jedoch werden nur die Surrogate beider Tabellen gespeichert. Damit ergibt sich eine geringe Größe, so daß ein Verbundindex im Hauptspeicher Platz finden kann. Bei mehr als zwei Tabellen werden erst Verbundindexe für Paare erstellt und auf diese anschließend ein weiterer übergeordneter Verbundindex. Auf diese Weise werden die Teile eines Objekts miteinander verbunden, falls diese auf mehrere Tabellen beziehungsweise Segmente aufgeteilt wurden. Im Beispiel werden also Verbundindexe zwischen den drei Tabellen Wissenschaftler, Bildung und Mitglied erstellt, die die Objekte der ursprünglichen Wissenschaftlertabelle anhand der gemeinsamen Surrogate verbinden.

Zusammenfassend finden sich in dieser Arbeit also folgende Speicherkonzepte:

- ▷ Kollektionselemente werden inline gespeichert (direkte Speicherung)
- ▷ Auslagerung von Kollektionselementen (normalisierte Speicherung)
- ▷ Kollektionen als ausgelagerte Tabellen (normalisierte Speicherung)
- ▷ Clusterung
- ▷ Sätze durch Seitengröße beschränkt (keine seitenübergreifenden Sätze)
- ▷ Verteilung auf mehrere Segmente anhand der Objektstruktur (normalisierte Speicherung)
- ▷ Nutzung von Indexen

[1] Es sei auf folgende Verwechslungsgefahr hingeweisen:
NSM heißt *N-ary Storage Model* und nicht Normalized Storage Model.
DSM heißt *Decomposition Storage Model* und nicht Direct Storage Modell.
Beide sind Spezialfälle des Normalized Storage Model.

3.1.4 Wisconsin Storage System

Die in [KCB87] beschriebene Implementierung von komplexen Objekten wurde auf dem prototypischen relationalen Speichersubsystem WiSS (Wisconsin Storage System) aufgesetzt. Dabei standen folgende Anforderungen für die Speicherung komplexer Objekte im Vordergrund:

- ▷ Komplexe Objekte sollten physisch geclustert gespeichert werden können. Dadurch wird bei Anfragen auf das gesamte Objekt eine Verringerung der Externspeicherzugriffskosten erreicht.

- ▷ Komplexe Objekte sollten sehr variabel in ihrer Größe sein dürfen. Dies wird für Attribute variabler Länge, wie zum Beispiel Zeichenketten, sowie für mengenwertige Attribute benötigt.

- ▷ Eine Veränderung der Größe eines komplexen Objekts sollte ohne Reorganisation des gesamten Objekts möglich sein.

Um die obigen Anforderungen zur Speicherung komplexer Objekte erfüllen zu können, sollten folgende Merkmale vom Speichersubsystem unterstützt werden. Laut [KCB87] stellen diese sogar fundamentale Voraussetzungen dar.

- ▷ *Heterogene Sätze*
  Es soll möglich sein, daß Seiten des Speichersystems Sätze verschiedenen Typs (das heißt Sätze verschiedener Relationen) beinhalten können.

- ▷ *Clusterung*
  Werden neue Sätze eingefügt, so soll ein Zielspeicherort dafür angegeben werden können. Neue Sätze werden dann möglichst dicht an diesem Zielort gespeichert. Dadurch ist eine physisch enge Speicherung zusammengehörender Sätze möglich.

- ▷ *Sätze variabler Länge*
  Um bei Speicherung großer, variabel langer, komplex strukturierter oder mengenwertiger Attribute nicht durch auf Seitengröße beschränkte Datensätze behindert zu werden, sollte das Speichersystem seitenübergreifende Sätze (Spanned Records) unterstützen.

- ▷ *Nutzung von Indexen*
  Zur assoziativen Suche auf Objekten sollte das Speichersystem eine Reihe von gängigen Indexstrukturen unterstützen und die Definition neuer allgemein einsetzbarer sowie domänenspezifischer Indexstrukturen zulassen.

Bei der Unterstützung von komplexen Objekten liegt der Schwerpunkt auf den Attributen, deren Größe nicht bei jedem Objekt gleich ist. Dies sind Attribute variabler Länge sowie mengenwertige Attribute. Hierfür wurden in der Arbeit jeweils unterschiedliche Speichermethoden angegeben, welche nachfolgend vorgestellt werden.

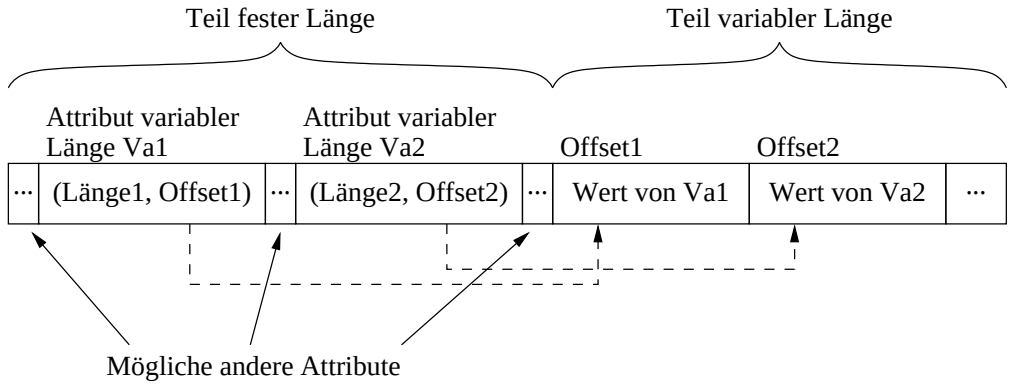


ABBILDUNG 3.11: Struktur eines Attributs variabler Länge

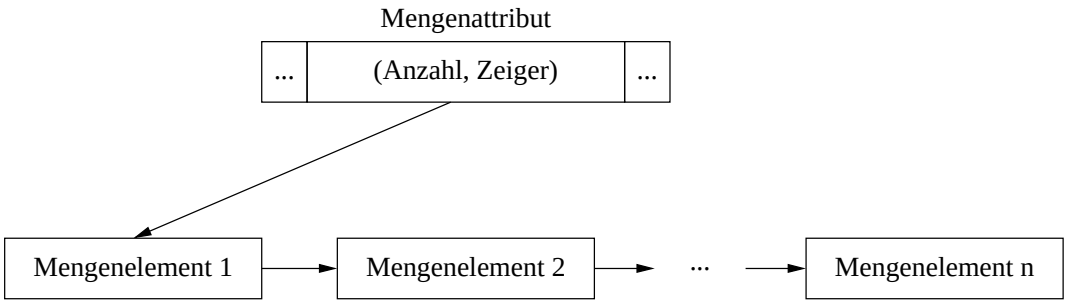


ABBILDUNG 3.12: Struktur eines mengenwertigen Attributs

1. Attribute variabler Länge

ABBILDUNG 3.11 zeigt den Aufbau eines Satzes, welcher Attribute variabler Länge enthält. Dieser besteht aus einem Teil fester Länge und einem Teil veränderbarer Länge. In ersterem werden die Werte von Attributen fester Länge gespeichert sowie die Deskriptoren für die Attribute variabler Länge. Diese Deskriptoren haben die Form (Länge, Offset), wobei ‚Länge' die Länge des Attributs angibt und ‚Offset' den Startpunkt des Attributs im Satzteil veränderbarer Länge. Diese Technik sichert einen schnellen Zugriff, kann aber bei Änderungsoperationen eine Reorganisation im Teil variabler Länge verursachen.

2. Mengenwertige Attribute

Da Elemente von mengenwertigen Attributen selbst wieder variabel in der Länge sein können, kann die entstehende Struktur sehr komplex werden. Daher wurde in WiSS als Speicherstruktur die einfach verkettete Liste mehrerer Sätze, die jeweils ein Element der Menge enthalten, gewählt, wie in ABBILDUNG 3.12 dargestellt ist. Mengenwertige Attribute stehen im Satz des Objekts als Paar (Anzahl, Zeiger), wobei ‚Anzahl' die Kardinalität der Menge darstellt und ‚Zeiger' auf den Satz mit dem ersten Element der Menge zeigt.

Komplexe Objekte umfassen in der Regel also mehr als einen Satz und können somit als Sammlungen von Sätzen betrachtet werden. Einer dieser Sätze ist als Wurzelsatz (Root Record) des komplexen Objekts ausgezeichnet und enthält folgende drei Informationen:

- ▷ Kontrollinformation über das komplexe Objekt
- ▷ Werte der einfachen Attribute (fester sowie variabler Länge)
- ▷ Informationen über mengenwertige Attribute

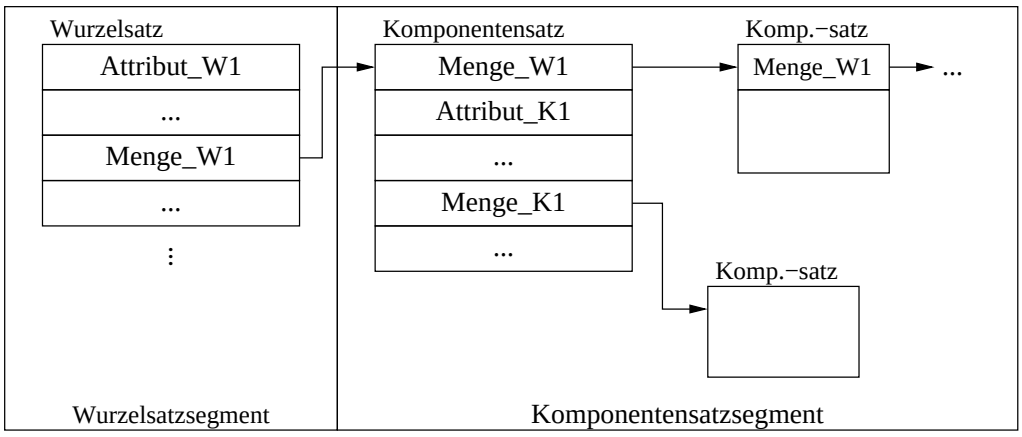


ABBILDUNG 3.13: Getrennte Speicherung der Satztypen

Die übrigen von einem komplexen Objekt belegten Sätze werden Komponentensätze genannt. Solch ein Komponentensatz enthält ein Element eines mengenwertigen Attributs und zeigt auf das nächste Elemente der Menge. Die entstehende Verkettungsstruktur kann weiter geschachtelt werden, da die Elemente eines mengenwertigen Attributs selber mengenwertig sein können. Komponentensätze haben folgenden Inhalt:

- ▷ Zeiger auf das nächste Element
- ▷ Werte der einfachen Attribute (fester sowie variabler Länge)
- ▷ Informationen über mengenwertige Attribute (hierdurch wird Schachtelung möglich)

Die Unterscheidung zwischen Wurzel- und Komponentensätzen ist auch für den jeweiligen Speicherort wichtig. Dabei werden alle Wurzelsätze von komplexen Objekten gleichen Typs zusammen in einem Segment gespeichert, getrennt von den Komponentensätzen, welche ebenfalls zusammen in einem Segment gespeichert werden (siehe ABBILDUNG 3.13). Hierdurch soll eine bessere assoziative Suche auf komplexen Objekten erreicht werden. Ebenfalls ist eine geclusterte Speicherung der Komponentensätze eines komplexen Objekts innerhalb des Komponentensatzsegments möglich.

Zusammenfassend läßt sich die Nutzung folgender Speicherkonzepte feststellen:

- ▷ Auslagerung von Kollektionselementen
- ▷ Kollektionen als verkettete Liste (Linked List)
- ▷ Clusterung
- ▷ Sätze durch Seitengröße beschränkt (keine seitenübergreifenden Sätze)
- ▷ Verteilung auf mehrere Segmente anhand der Objektstruktur
- ▷ Nutzung von Indexen

Dabei werden in [KCB87] zur Speicherung komplexer Objekte Kollektionselemente (mengenwertige Elemente) ausgelagert und in Form einer verketteten Liste gespeichert. Das gesamte Objekt wird dabei anhand der Objektstruktur auf mehrere Segmente verteilt, in denen Clusterung möglich ist.

3.1.5 DB2

Die hier betrachtete Version von IBMs DB2 UDB ist die derzeit aktuelle Version 8.2 [IBM02b]. Ein hier implementiertes objektrelationales Merkmal ist die Unterstützung von strukturierten Typen. Ein solcher Typ kann mehrere Attribute haben, sowie selbst Attribut

eines anderen strukturierten Typs sein. Die dabei entstehenden komplexen Typen können jedoch keine mengenwertigen Attribute haben.

Ein auf diese Weise benutzerdefinierter Typ kann zur Erstellung einer typisierten Tabelle (Typed Table) herangezogen werden. Dabei stellt jede Zeile dieser Tabelle eine Instanz des strukturierten Typs dar. Des weiteren kann jedes Attribut einer beliebigen Tabelle von einem strukturierten Typ sein.

Die physische Speicherung einer Instanz verläuft nach folgendem Schema. Die Instanz wird zusammen mit den restlichen Attributen der Zeile der Tabelle gespeichert (Inline-Speicherung), das heißt, die Daten der Instanz kommen in den selben physischen Satz einer Seite. Dies geschieht jedoch nur bis zu einer bestimmten Größe der Instanz. Wird diese überschritten, so wird eine ähnliche Speicherungsmethode wie bei Large Objects (LOB) angewendet. Anstelle der Daten des strukturierten Attributs wird ein Zeiger gespeichert, der auf den ausgelagerten Speicherort zeigt. Diese Schwellgröße wird Inline Length genannt und kann in Create-Table- und Create-Type-Anweisungen angegeben werden. Eine nachträgliche Änderung ist nicht möglich. Außerdem unterstützt DB2 die Clusterung von Daten in zwei Varianten:

- *Eindimensional*
  Auf die zu clusternde Tabelle wird ein Clusterindex erstellt. Der Indexschlüssel kann aus einem oder mehreren Spalten bestehen. Dieser stellt die eine Dimension der Clusterung dar.

- *Mehrdimensional*
  Für eine Tabelle können mehrere Clusterungsschlüssel angegeben werden. Jede Kombination dieser Schlüsselwerte bildet eine Zelle, welche physisch in Blöcke von Seiten aufgeteilt ist. Diese Seiten liegen möglichst aufeinanderfolgend auf dem Externspeicher, so daß ein Zugriff auf entsprechende Schlüsselwerte möglichst schnell geht. Solch organisierte Tabellen werden Multidimensional Clustering Tables (MDC) genannt.

Weiterhin existiert in DB2 die Möglichkeit, sogenannte Range-Clustered Tables (RCT) anzulegen. Dabei wird für die entsprechende Tabelle vorab eine feste Menge an Speicher reserviert. Werden Tupel in die Tabelle eingefügt, so erhalten sie eine feste Satz-ID (Record-ID – RID), mittels derer die Speicherstelle des Tupels festgelegt ist. Dies macht es erforderlich, daß die eingefügten Tupel in monoton aufsteigender Schlüsselreihenfolge eingefügt werden. Dies entspricht nicht dem Begriff der Clusterung, welcher hier verwendet wird. Eine RCT dient dem schnellen direkten Zugriff und nicht der Gruppierung der Daten.

Die in DB2 verwendeten Speicherkonzepte für komplexe Objekte können folgendermaßen zusammengefaßt werden:

- Keine Kollektionen.
- Clusterung.
- Sätze durch Seitengröße beschränkt (keine seitenübergreifenden Sätze). Bei Erreichen der Seitengröße wird in LOB-Überlauf gespeichert.
- Nutzung von Indexen.

3.1.6 Informix

Informix [IBM03c, IBM03b, IBM03d] ist ein Datenbanksystem von Informix Software, das von IBM aufgekauft wurde, und besitzt vergleichsweise viele objektrelationale Merkmale.

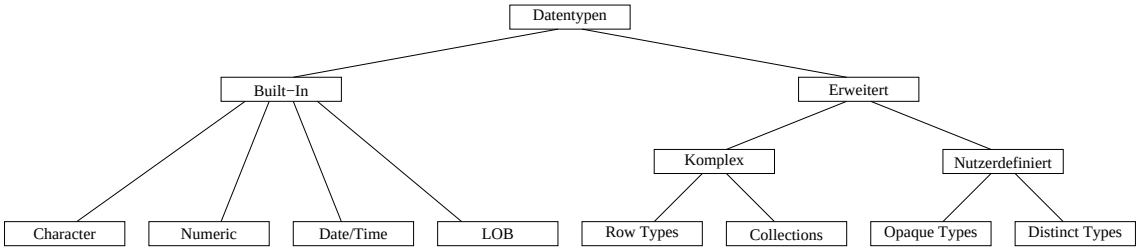


ABBILDUNG 3.14: Datentypen im Informix Dynamic Server

Neben den gebräuchlichen eingebauten Datentypen (Integer, Character, ...) ist es in Informix möglich, erweiterte Datentypen zu definieren (jedoch nur im Dynamic Server). In ABBILDUNG 3.14 ist die Typklassifikation von Informix dargestellt. Erweiterte Datentypen werden in komplexe Datentypen und nutzerdefinierte Datentypen eingeteilt. Zur letzteren Kategorie gehören die Opaque Types und die Distinct Types. Komplexe Datentypen stellen Kollektionen und Zeilentypen (Row Types) dar. Im folgenden werden die erweiterten Datentypen beschrieben:

- ▷ *Nutzerdefinierte Datentypen*
  - ◇ *Distinct Types*

    Ein Distinct Type wird von einem anderen bereits definierten Typ abgeleitet. Damit erhält er dieselbe interne Repräsentation und physische Speicherung. Außerdem erbt er die Struktur seines Quelltyps. Als Quelltypen können vordefinierte Typen, benannte Tabellentypen, Opaque Types und Distinct Types dienen. Wird ein Distinct Type angelegt, so werden vom System Cast-Funktionen zwischen diesem und dessen Quelltyp angelegt. Diese dienen dem Vergleich zwischen beiden Typen, da kein direkter Vergleich möglich ist.

    Die Definition eines Distinct Types erfolgt mit folgender Anweisung:

    ```
    CREATE DISTINCT TYPE
    ```

  - ◇ *Opaque Types*

    Diese Typen sind Datentypen, die völlig vom System gekapselt sind. Dem System ist damit keine interne Struktur des Typs bekannt. Somit muß der Nutzer dem System mitteilen, wie die interne Struktur aussieht und wie die Daten auf dem Externspeicher gespeichert werden sollen. Zusätzlich müssen dazu jegliche Funktionen für den Typ angegeben werden. Dies schließt Funktionen ein, die für die Umwandlung des Typs zwischen Externspeicherformat und Speicherformat zuständig sind.

    Mit folgender Anweisung können Opaque Types definiert werden:

    ```
    CREATE OPAQUE TYPE
    ```

- ▷ *Komplexe Datentypen*
  - ◇ *Benannte Zeilentypen*

    Daten vom Typ Row (Named Row Type) stellen eine Zusammenfassung einzelner Attribute dar, die unter einem einzigen Bezeichner angesprochen werden können.

Sie stellen somit einen Verbundtyp dar und sind zu vergleichen mit dem Typ Struct der Programmiersprache C/C++ oder Record aus Pascal. Ein solcher Typ kann zur Definition einer ganzen Tabelle dienen (Typed Table) oder für eine Spalte einer Tabelle. Zeilenattribute werden direkt im Tupel (eingelagert/inline) gespeichert. Die Definition von Benannten Zeilen Typen erfolgt mit:

```
CREATE ROW TYPE
```

◇ *Unbenannte Zeilentypen*

Der wichtigste Unterschied zwischen benannten und unbenannten Row-Typen (Unnamed Row Types) ist, daß eine Tabelle nicht von einem unbenannten Row-Typ angelegt werden kann. Er dient dazu, Spalten einer Tabelle zu definieren.

◇ *Kollektionstypen*

Mit den Kollektionstypen (Collection Types) ist es möglich, Sammlungen von Daten innerhalb eines Attributs einer Tabelle zu speichern. Die Elemente können fast jeden beliebigen Typ haben (außer Text, Byte und Serial). In Informix existieren drei verschiedene Arten von Kollektionen, die direkt in den jeweiligen Satz des Tupels (inline) gespeichert werden.

- *Set (Menge)*

  Kollektionen vom Typ Set stellen eine ungeordnete Sammlung von Elementen dar, die untereinander eindeutig sind. Für die Definition einer Menge lautet der Typkonstruktor in SQL:

  ```
  SET (element type)
  ```

- *Multiset (Multimenge)*

  Im Gegensatz zu Set dürfen in Multimengen Duplikate vorkommen. Eine Reihenfolge der Elemente gibt es auch hier nicht. Sollen Attribute vom Typ Multiset angelegt werden, so steht hierfür die folgende Anweisung bereit:

  ```
  MULTISET (element type)
  ```

- *List (Liste)*

  Die Elemente eines List-Typs besitzen eine Ordnung. Diese wird von der Einfügereihenfolge bestimmt. Standardmäßig fügt das DBMS die Daten am Ende der Liste ein. Es kann jedoch auch eine bestimmte Position in der Liste angegeben werden. Folgende Anweisung dient zur Definition eines Listenattributs:

  ```
  LIST(element type)
  ```

Bei der physischen Speicherung der so definierbaren potentiell komplexen Objekte werden seitenübergreifende Datensätze (Spanned Records) unterstützt. Tritt der Fall ein, daß die Größe einer Seite nicht ausreicht, so werden Folgeseiten genutzt. Auf diese Weise können Seiten neben ganzen Sätzen auch Teile von Sätzen beinhalten. Füllt ein Teilsatz eines seitenübergreifenden Satzes (insbesondere der letzte Teilsatz) seine Seite nicht vollständig aus, so kann der restliche Platz für andere Sätze genutzt werden.

Das DBMS reserviert den Speicherplatz für Tabellen extentweise, das heißt, für Tabellen werden stets feste Mengen an Seiten angefordert und belegt. Deshalb ist zu vermuten, daß Informix auch keine heterognen Satzypen in einer Seite gestattet.

Informix erlaubt es jedoch, Attribute mit den Typen Text, Byte, BLOB und CLOB in andere Seiten in speziellen Segmenten auszulagern. Anstelle der Daten stehen im Satz dann entsprechende Referenzen beziehungsweise Deskriptoren.

Informix unterstützt bei der Speicherung von Kollektionen folgende zwei Varianten:

- ▷ Kollektionen können serialisiert direkt im Datensatz des übergeordneten Objekts gespeichert werden (Inline Array).
- ▷ Kollektionen können aber auch als LOB ausgelagert gespeichert werden.

Weiterhin kennt Informix neben den üblichen Indexierungstechniken für ‚normale' Tabellen auch indexorganisierte Tabellen.

Bei Informix läßt sich zusammenfassend die Nutzung folgender Speicherkonzepte feststellen:

- ▷ Seitenübergreifende Sätze.
- ▷ Eingelagerte Speicherung von Kollektionen (Inline Array).
- ▷ Auslagerung von Kollektionen.
- ▷ Indexbasierte Clusterung.

3.1.7 Oracle

Damit objektorientierte Programme ihre Objekte in einer Datenbank speichern können, bietet Oracle die Möglichkeit, Objekttypen anzulegen [Ora01b]. Instanzen eines Objekttyps (Objekte) werden in Tabellen gespeichert. Dabei unterscheidet Oracle zwischen sogenannten Column Objects und Row Objects.

Column Objects sind Objekte, die in einer Spalte einer gewöhnlichen Tabelle gespeichert werden. Wenn es also einen Objekttyp Angestellter gibt, so kann man eine Tabelle mit einer Spalte vom Typ Angestellter anlegen. Hierbei ergibt sich, daß ein Objekt nicht die gesamte Zeile der Tabelle einnimmt. Bei den Row Objects werden die Objekte hingegen in Objekttabellen gespeichert, bei denen es sich um typisierte Tabellen handelt (Tabellen des entsprechenden Objekttyps). Ein Objekt entspricht hier einer gesamten Zeile der Tabelle. Diese wird von Oracle als zu bevorzugende Speicherform angegeben. Haben zwei Objekte das gleiche Unterobjekt, so kann dies nämlich einfach mittels des Referenzdatentyps (Ref) referenziert werden, welcher auf das entsprechende Row Object zeigt. Anders als bei Fremdschlüsseln kann der Anwender das referenzierte Teilobjekt über Pfadausdrücke so benutzen, als ob es sich direkt im Objekt befände. Objekttabellen unterscheiden sich zwar allgemein kaum von normalen relationalen Tabellen, aber ein wichtiger spezieller Unterschied ist, daß diese Tabellen eine zusätzliche, in der Regel indexierte aber für den Nutzer nicht sichtbare Spalte für den Objektidentifikator (OID) besitzen. Die Objekt-ID wird von Oracle generiert und wird vom Referenzdatentyp genutzt, um die Objekte zu erreichen.

Haben Objekte mengenwertige Attribute, so unterstützt hierfür Oracle die zwei unterschiedlichen Kollektionstypen VArray und Nested Table:

- ▷ *VArray*

  Attribute von diesem Typ stellen eine Sammlung von geordneten Elementen des selben Typs dar (ausgenommen LOBs). Wie bei einem Feld üblich, hat jedes Element eine Indexnummer, über die der Zugriff erfolgt. Wird ein Attribut vom Typ VArray deklariert, so muß dessen maximale Elementanzahl vorab angegeben werden. Wie

viel Speicherplatz ein solches Attribut belegt, wird nicht von der maximalen Größe bestimmt, sondern von der momentanen Größe des Feldes. Somit wird eine variable Größe des Feldes erlaubt – daher der Name V(ariable)Array. Die maximale Größe eines VArray-Attributs kann per `ALTER TABLE` nachträglich geändert werden. Oracle speichert VArrays als Opaque Objects, das heißt als BLOB oder als Raw-Feld. Welche Speicherform genau gewählt wird, hängt von der aktuellen Feldgröße ab. Überschreitet ein VArray-Attribut eine gewisse Größe nicht, so wird es gemeinsam mit den übrigen Attributen direkt im Datensatz als Raw-Feld gespeichert. Andernfalls erfolgt eine Auslagerung als BLOB. Änderungen an einem VArray-Attribut können nie nur auf Teilen davon ausgeführt werden. Wird eine Änderungsoperation ausgeführt, so wird der gesamte alte Attributwert durch den neuen ersetzt.

- ▷ *Nested Table*

  Dieser Kollektionstyp stellt eine unsortierte Menge von Datenelementen gleichen Datentyps dar. Eine Nested Table besitzt nur eine Spalte, deren Typ entweder ein eingebauter Datentyp oder ein Objekttyp sein kann. Nested Tables sind von sich aus keine Tabellen, sondern Typen und belegen somit nicht von selbst Speicher. Diese Typen können als Datentypen in relationalen Tabellen verwendet werden oder als Attribut von Objekttypen. Wird eine relationale Tabelle mit einer Spalte vom Typ Nested Table angelegt, so speichert Oracle die Daten dieser Spalte für jede Zeile der Tabelle in der selben Speichertabelle. Gleichermaßen geschieht dies bei Objekttabellen, deren Objekttyp ein Attribut vom Typ Nested Table besitzt. Die Daten dieses Attributs werden für jede Instanz in der selben Speichertabelle abgelegt. Damit erhält jede Nested-Table-Spalte bzw. jedes Nested-Table-Attribut seine eigene Speichertabelle. Zur Veranschaulichung sei auf die Abbildung 3.15 verwiesen. Um eine Verbindung zwischen den Zeilen eines Nested-Table-Attributs und den zugehörigen Daten in der Speichertabelle herzustellen, wird vom System für jede Zeile eine 16 Byte große Nested-Table-ID erzeugt. Diese IDs stehen in der Nested-Table-Spalte sowie in einer zusätzlichen Spalte in der Speichertabelle. Gehören also gewisse Zeilen in einer Speichertabelle zu einer Nested-Table-Zeile, so sind die Nested-Table-IDs gleich. Da nicht direkt auf die Speichertabellen zugegriffen werden kann, sind die Nested-Table-IDs auch nicht für den Nutzer sichtbar.

Neben den beiden angegebenen Kollektionstypen unterstützt Oracle zusätzlich noch die Verschachtelung beider. Damit können beispielsweise Mengen von Mengen angelegt werden. Oracle nennt diese Typen Multilevel Collection Types. Diese können folgende Formen haben:

- ▷ Nested Table eines Nested-Table-Typs
- ▷ Nested Table eines VArray-Typs
- ▷ VArray eines Nested-Table-Typs
- ▷ VArray eines VArray-Typs
- ▷ Nested Table oder VArray eines benutzerdefinierten Typs, welcher ein Nested-Table- oder VArray-Attribut besitzt.

Weitergehende Schachtelungen sind ebenfalls möglich, da zum Beispiel eine Nested Table eines VArray-Typs ein Nested-Table-Typ ist.

Multilevel Collection Types haben den gleichen Einsatzbereich wie einfache Kollektionen, das heißt in relationalen Tabellen und in Objekttabellen. Die Speicherung erfolgt jedoch uneinheitlich und ist abhängig von der Art der Verschachtelung. Wird eine Nested Table

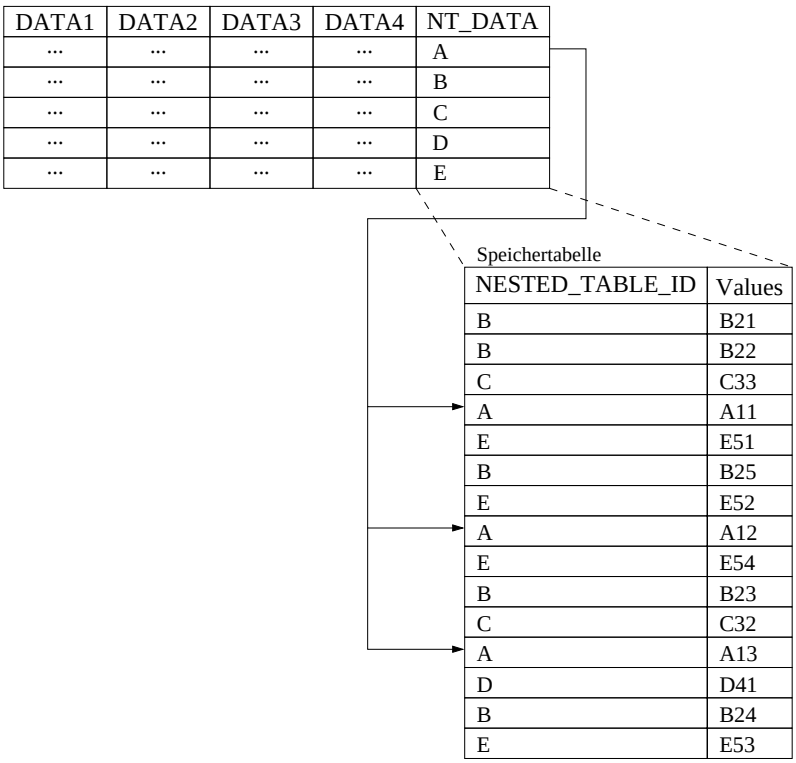


ABBILDUNG 3.15: Speicherung einer Nested Table

eines Nested-Table-Typs erstellt, so wird eine Speichertabelle für die innere Menge sowie für die äußere Menge angelegt. Bei Nested Tables von VArrays wird eine Speichertabelle für die Nested Table angelegt, und darin werden die VArray-Daten direkt (inline) bzw. ausgelagert als LOB gespeichert (siehe VArray). VArrays von VArrays werden genauso wie einfache VArrays gespeichert. Alle Daten werden direkt (inline) gespeichert, das heißt in der jeweiligen Zeile. Werden 4000 Bytes überschritten, so erfolgt eine Speicherung als BLOB. Bei VArrays von Nested Tables wird das gesamte VArray als LOB gespeichert. In den jeweiligen Zeilen werden nur die LOB-Lokatoren gespeichert. Damit wird in diesem Fall keine Speichertabelle für die Nested Table angelegt, sondern deren Daten werden innerhalb des VArrays gespeichert.

Desweiteren unterstützt Oracle das Prinzip der Clusterung. Dadurch können die Daten eines kollektionswertigen Attributs physisch dicht gespeichert werden, was in bestimmten Fällen eine beschleunigte Anfrageabarbeitung zur Folge hat.

Die in Oracle für die Implementierung komplexer Objekte genutzten Konzepte lassen sich zusammenfassend auflisten:

- ▷ Auslagerung von Kollektionselementen.
- ▷ Clusterung.
- ▷ Kollektionen mittels logischer Rückwärtsreferenzen.
- ▷ Seitenübergreifende Sätze. In Oracle werden Seiten Blöcke genannt. Dabei dürfen die Zeilen einer Tabelle (Sätze) die Blockgröße überschreiten [Ora03].
- ▷ Einsatz von Indexen.

{ Departments }								
DNo	MgrNo	{ Projects }				Budget	{ Equip }	
		PNo	PName	{ Members }			Qu	Type
				EmpNo	Function			
314	56194	17	CGA	39582	Leader	320000	2	3278
				56019	Consultant		3	PC/AT
				69011	Secretary		1	PC
		23	HPAR	58912	Staff			
				90011	Leader			
				78218	Secretary			
				98902	Staff			
				...	...			
218	71349	25	LEXI	72227	Staff	440000	2	3278
				89211	Staff		2	PC/AT
				92100	Leader		1	3179
				89921	Consultant		1	PC/GA
				99025	Secretary			
				44512	Consultant			
417	91093	37	NDBS	87710	Secretary	360000	1	4361
				81193	Leader		1	PC/XT
				75913	Staff		1	PC/AT
				96001	Staff		2	3278
							1	3270
							1	3179
							1	PC/GA

ABBILDUNG 3.16: Beispiel einer NF^2-Tabelle

3.1.8 AIM-P

In [DKA+86] wird die Speicherung von komplexen Objekten in eNF^2-Tabellen beschrieben. Dies wurde innerhalb des Advanced Information Management Projekts (AIM-P) der IBM am Wissenschaftlichen Zentrum Heidelberg in einem DBMS Prototypen implementiert. eNF^2-Tabellen sind eine Erweiterung von NF^2-Tabellen. NF^2-Tabellen stellen eine Verallgemeinerung des relationalen Datenmodells dar. In NF^2-Tabellen können Relationen als Attributwerte von Tupeln vorkommen. Das Modell entsricht damit nicht mehr der ersten Normalform (1NF) und wird daher als Non First Normal Form (NF^2) bezeichnet. Das eNF^2-Modell (Extended NF^2) erweitert das NF^2-Modell unter anderem durch die Unterstützung von geordneten Tabellen, welche als Listen interpretiert werden können. Die somit definierbaren komplexen Objekte können hierarchisch strukturiert sein, sowie mengen- und listenwertige Attribute besitzen.

Zum besseren Verständnis wird das Beispiel aus [DKA+86] übernommen. Darin wird ein komplexer Objekttyp namens Department definiert (siehe ABBILDUNG 3.16), welcher die Attribute DNo (Department Number), MgrNo (Manager Number), Projects (Projects of Department), Budget und Equip (Equipment of Department) besitzt. Dabei stellen Equip und Projects mengenwertige Attribute dar. Elemente von Equip haben Qu (Quantity) und Type als Attribute, Elemente von Projects haben PNo (Project Nummer), PName (Project Name) und Members (Project Members) als Attribute. Bei Members handelt es sich um ein mengenwertiges Attribut. Dessen Elemente haben EmpNo (Employee Number) und Function (Function of Employee) als Attribute. Der somit definierte Objekttyp besitzt also unter anderem ein mengenwertiges Attribut mit einem weiteren eingeschachtelten mengenwertigen Attribut. Eine entsprechende mit Beispieldaten gefüllte NF^2-Tabelle ist in ABBILDUNG 3.16 abgebildet.

Zur Implementierung der Speicherung solcher komplexer Objekte werden folgende Forderungen aufgelistet:

▷ *Datenclusterung*

Datenclusterung sollte unterstützt werden. Werden also komplexe Objekte in einer möglichst kleinen Menge von Seiten möglichst dicht beieinander gespeichert, kann das Arbeiten an dem Objekt beschleunigt werden.

▷ *Trennung von Daten und Struktur*

Daten und Strukturinformationen (zum Beispiel Zeigerlisten) sollten voneinander getrennt sein. Dadurch sollen sowohl die Navigation auf Objekten als auch die die Auswertung gewisser Anfragen möglich werden, ohne auf die Daten zugreifen zu müssen.

▷ *Schneller Teilobjektzugriff*

Schnelles Arbeiten sollte nicht nur auf dem gesamten Objekt (Complex Object) möglich sein, sondern auch auf beliebigen Teilen davon (Complex Subobject).

Zur Beschreibung der Speichertechnik führen die Autoren folgende Unterscheidungen im Datenmodell ein:

▷ *Tabelle (NF^2 oder 1NF)*

größte logische Modellierungseinheit, ist kein Teil einer anderen Tabelle

▷ *Objekt (komplex oder flach)*

Zeile (Tupel) einer Tabelle

▷ *Untertabelle (NF^2 oder 1NF)*

logische Speichereinheit, die Elemente von mengenwertigen Attributen aufnimmt

▷ *Unterobjekt (komplex oder flach)*

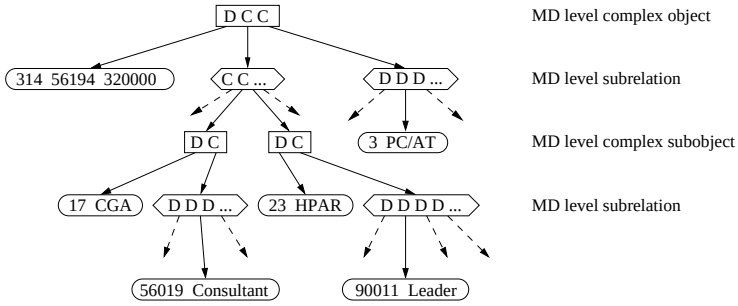
Zeile (Tupel) einer Untertabelle

Im Beispiel ist demzufolge ‚Department' eine NF^2 Tabelle, „Department 314" ein komplexes Objekt, ‚Projects' und ‚Equip' Untertabellen (komplex und flach), „Projekt 17" ein Unterobjekt und ‚Members' eine Untertabelle.

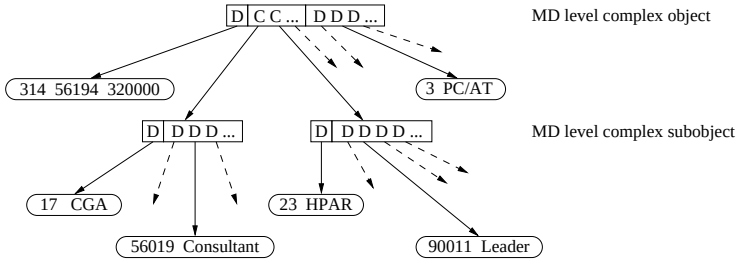
Zur Implementierung von komplexen Objekten und zur Trennung von Daten und strukturellen Informationen wird in AIM-P für jedes komplexe Objekt ein sogenanntes Mini Directory (MD) implementiert. Dabei handelt es sich um einen Baum, der die strukturellen Informationen eines komplexen Objekts enthält, nicht aber dessen Daten.

Ein Mini Directory (MD) ist aus MD-Subtupeln (Knoten des MD-Baums) zusammengesetzt, welche mittels Zeigern verbunden sind. Ein Subtupel stellt die Basisspeichereinheit dar, vergleichbar mit einem Satz. Neben MD-Subtupeln gibt es noch Datensubtupel, welche die Daten eines komplexen Objekts speichern. Besitzt ein Objekt/Subobjekt atomare Attribute auf seiner obersten hierarchischen Stufe, so werden diese alle in einem Datensubtupel gespeichert. Im Beispiel gibt es also für „Department 314" ein Datensubtupel (314, 56194, 320000), welches die Abteilungsnummer 314, die Managernummer 56194 und das Budget 320000 enthält. Flache Objekte werden, wie es in relationalen DBMs üblich ist, komplett in einem Datensubtupel gespeichert und haben kein Mini Directory.

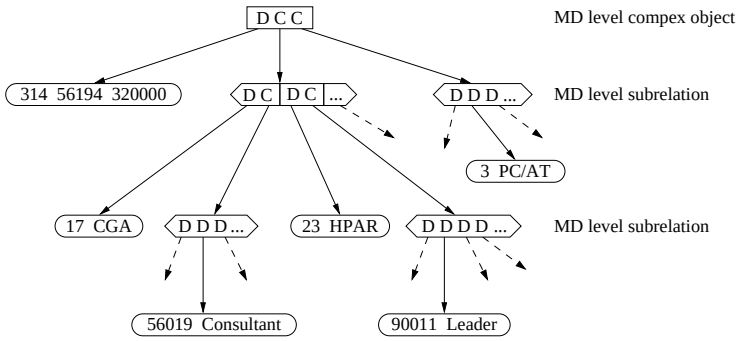
Zur Implementierung des Mini Directories werden drei Möglichkeiten vorgeschlagen, welche in Abbildung 3.17 als SS1–SS3 (Speicherstruktur 1–3) dargestellt sind. Alle drei besitzen ein ausgezeichnetes MD-Subtupel, welches die Wurzel des MD-Baumes ist. Neben



(a) SS1 generelle Lösung



(b) SS2 Unterobjekt orientiertes Mini Directory



(c) SS3 Untertabellen orientietes Mini Directory

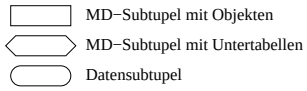


ABBILDUNG 3.17: Mögliche Speicherstrukturen eines komplexen NF^2-Objekts

den Zeigern auf untergeordnete Subtupel enthält dieses noch weitere Informationen über das komplexe Objekt. Folgende Eigenschaften unterscheiden die drei Möglichkeiten:

- ▷ SS1 nutzt MD-Subtupel für Untertabellen und für komplexe Unterobjekte.
- ▷ SS2 nutzt MD-Subtupel nur für komplexe Objekte
- ▷ SS3 nutzt MD-Subtupel nur für Untertabellen

Die grafische Darstellung in ABBILDUNG 3.17 verwendet eckige Knoten für MD-Subtupel

und abgerundete Knoten für Datensubtupel. Die verschiedenen Zeigertypen werden durch Markierungen unterschieden: Mit ‚D' werden Zeiger auf Datensubtupel und mit ‚C' werden Zeiger auf MD-Subtupel (Child Pointer) markiert.

Die Speicherstruktur SS1 wird in ABBILDUNG 3.17(a) dargestellt. Da strukturelle Änderungen der NF^2-Tabellen im Prototyp nicht betrachtet werden, hat hier das Wurzel-Subtupel eine feste Länge. Dies liegt daran, daß die variable Länge nur durch Untertabellen verursacht wird, und diese haben in SS1 eigene MD-Subtupel. Für jedes Element der Untertabellen ist in diesen ein Zeiger enthalten. Als Nachteil dieser Struktur wird die relativ hohe Zahl an kleinen MD-Subtupeln und damit eine hohe Knotenanzahl im Baum genannt. Dies liegt daran, daß für jedes komplexe Unterobjekt ein MD-Subtupel vorhanden ist, und in der Praxis hat ein komplexes Objekt viele davon. Relativ zu den MD-Subtupeln der Untertabellen sind die der Unterobjekte klein, da im allgemeinen davon ausgegangen werden muß, daß ein komplexes Objekt deutlich weniger Attribute als Elemente in mengenwertigen Attributen hat.

Aus diesem Grund werden in SS2 und SS3 die MD-Subtupel für Untertabellen und Subobjekte zusammengelegt. In SS2 (ABBILDUNG 3.17(b)) geschieht dies durch das Integrieren der MD-Subtupel der Untertabellen in die darüberliegenden MD-Subtupel der Unterobjekte. Daraus ergibt sich, daß das Wurzel-Subtupel seine feste Länge verliert und variabel wird. Für jedes komplexe Unterobjekt existiert hier ein MD-Subtupel. In SS3 (ABBILDUNG 3.17(c)) werden dagegen die MD-Subtupel der Unterobjekte nach oben in die MD-Subtupel der Untertabellen integriert. Für jede Untertabelle ist hier ein MD-Subtupel vorhanden.

Im Prototyp des AIM-Projekts wurde als Kompromiß zwischen verschiedenen Kriterien (Speicherplatz, Zugriffszeit und so weiter) SS3 gewählt. Um eNF^2-Tabellen zu unterstützen, kann die Reihenfolge der Einträge in den MD-Subtupeln der Untertabellen als Ordnung genutzt werden.

Wie weiter oben bereits erwähnt, enthält das Wurzel-Subtupel noch weitere Informationen. Dies ist eine Seitenliste aller Seiten, die von Subtupeln des komplexen Objekts belegt werden und repräsentiert den lokalen Adreßraum. Kommen nun neue Daten zum Objekt hinzu, so werden die bereits verwendeten Seiten nach freiem Platz durchsucht und wenn vorhanden genutzt. Ist kein Platz frei, so wird eine neue Seite in die Seitenliste eingetragen und die neuen Daten werden dorthin gespeichert. Damit wird auch das Prinzip der Clusterung unterstützt, da neue Daten möglichst in bereits vom Objekt belegte Seiten geschrieben werden. Ob jedoch eine Seite Daten von mehreren komplexen Objekttypen beinhalten kann, wird nicht erwähnt. Es ist jedoch anzunehmen, daß in eine Seite nur Daten von einem komplexen Objekttyp gespeichert werden können.

Zur Implementierung komplexer Objekte fehlt jetzt noch die Beschreibung der Zeiger (‚D' und ‚C'). Beide werden nach dem Mini-TID Konzept implementiert [HR01]. Sie bestehen also aus einem Paar (i, j), wobei i die Nummer der Seite angibt, welche das referenzierte Subtupel enthält und j die Position innerhalb dieser. Dabei ist i relativ zur Seitenliste im Wurzel-Subtupel. Um nun bei einem Zugriff auf die richtige Seite in der Datenbank zu gelangen, wird i durch die globale Seitennummer ersetzt, welche in der Seitenliste des komplexen Objekts an Stelle i steht. Ein Problem tritt auf, wenn Daten aus dem komplexen Objekt gelöscht werden und eine Seite frei wird. Würde diese nun aus der Seitenliste gelöscht werden, müssten alle Zeiger, welche auf nachfolgende Seiten zeigen, geändert werden. Daher werden ‚Lücken' in der Seitenliste zugelassen. Mit ‚Lücke' ist gemeint, daß in der Seitenliste an der zu löschenden Stelle ein Eintrag verbleibt. Wird eine neue Seite angefordert, so wird diese an die Lücke in der Liste gesetzt und so der Seitenlisteneintrag

wieder benutzt.

Für die in dieser Arbeit dargestellte Implementierungstechnik kann man folgende genutzte Konzepte zusammenfassen:

- ▷ Auslagerung von Kollektionselementen
- ▷ Trennung von Strukturinformationen und Daten
- ▷ Clusterung
- ▷ Kollektionen als Zeigerfeld (Pointer Array)
- ▷ Nutzung von Indexen

3.1.9 DASDBS

Die in [DPS86] beschriebenen komplexen Objekte sind Tupel des NF^2-Modells. Die dafür entwickelte Speicherstruktur wurde in einer Komponente eines Datenbanksystemkerns implementiert (DASDBS). Diese Komponente wird als Complex Record Manager (CRM) bezeichnet und sitzt direkt über der hier nicht weiter beschriebenen seitenorientierten Stable Memory Manager-Komponente.

Der Complex Record Manager (CRM) erhält seine NF^2-Objekte (NF^2-Tupel) in Form von abstrakten komplexen Sätzen, welche später genauer beschrieben werden (siehe ABBILDUNG 3.18). Dabei gibt es zwei Typen von Attributen: atomare und relationenwertige (Mengen/Listen). Über die Datentypen hat der CRM jedoch keine Kenntnis, sondern er kennt nur Bytefolgen. Atomare Attribute können fester Länge, variabler Länge oder extra lang sein. Letzteres ist für Attribute gedacht, die nicht in eine Seite passen. Ein relationenwertiges Attribut (Unterrelation) enthält eine Mengen von Subtupeln, die nach einem atomaren Attribut oder der Einfügereihenfolge sortiert sein können.

1. Anforderungen

Die folgenden Anforderungen werden an die Speicherarchitektur (Speicherkonzept, Speicherstruktur, Adressierungskonzept) gestellt.

- ▷ *Zugriff*
  - ◇ Da der CRM Anfragen durch hierarchische Zugriffe auf gewisse Komponenten der Objekte der Suchmenge bearbeitet, sollte ein schneller wahlfreier Zugriff (Random Access) auf das gesamte Objekt, wie auch auf Teile davon, möglich sein.
  - ◇ Es sollte sequentielles Arbeiten auf den Objekten möglich sein. Sind bei einer Anfrage mehrere Objekte betroffen, so sollten auf diese in einer vordefinierten Reihenfolge zugegriffen werden.
- ▷ *Änderung und Reorganisation*

  Durch Änderungen auf komplexen Objekten kann sich die Seitenbelegung ändern. Solche Änderungsoperationen können zum Beispiel das Einfügen neuer Elemente in Unterrelationen oder die Änderung von atomaren Attributen variabler Länge sein. Dadurch kann eine Verschiebung der Daten innerhalb einer Seite erforderlich werden. Externe Referenzen, die durch Byteadressen implementiert wären, müßten bei solchen Änderungen stets mitgeändert werden. Um dies zu vermeiden, muß das Adressierungskonzept dagegen stabil sein. Diese Stabilität sollte außerdem nicht die Gesamtleistung beeinträchtigen.

▷ *Speicherbedarf*
Der benötigte Speicher für Objekte (inkl. interner und externer Adressen) sollte klein sein.

2. Speicherkonzepte

Die Objekte des Datenmodells (NF^2-Tupel) werden auf Basissätze in Seiten abgebildet. Basissätze sind Bytefolgen, die in eine Seite passen und nur innerhalb des komplexen Satzes zugreifbar sind. Interne Adressen beziehen sich nur auf den Adreßraum des komplexen Satzes und sind außerhalb nicht gültig. Die vorgeschlagene Speicherungsform besitzt folgende Hauptmerkmale:

▷ *Clusterung*
Komplexe Sätze werden in so wenig Seiten wie möglich geclustert gespeichert.

▷ *Trennung von Struktur und Daten*
Interne Verknüpfungen eines komplexen Satzes und dessen Daten sind segmentiert, das heißt, sie werden voneinander getrennt und in Abschnitte eingeteilt.

- ◇ Basissätze mit Daten (atomare Abschnitte) werden geclustert ‚depth-first' gespeichert
- ◇ Basissätze mit Mengen von Verknüpfungen (Verknüpfungsabschnitte) werden geclustert ‚breadth-first' gespeichert

▷ *Keine Fragmentierung von Basisätzen mit Daten*
Komplexe Sätze, die kleiner als eine Seite sind, erfahren keine Fragmentierung, das heißt, sie werden nicht unnötig auf mehrere Seiten aufgeteilt. Sind sie größer als eine Seite, so enthalten die belegten Seiten keine anderen komplexen Sätze.

▷ *Directory Header*
Für jeden komplexen Satz gibt es einen sogenannten Directory Header. Dieser besteht aus allen internen Verknüpfungsabschnitten und einem sogenannten Seitenabschnitt (Page Segment), welches die Seitenliste enthält, die das komplexe Objekt belegt (zugeordnete DB-Seiten). Der Directory Header steht in der ersten Seite des Adreßraums des komplexen Satzes

Durch den Directory Header wird eine Zentralisierung notwendiger Informationen über interne Verknüpfungen (innerhalb des komplexen Objekts) und der verwendeten Seiten erreicht. Dadurch erreicht man ein einfaches Verschieben der Seitenmenge, die den komplexen Satz enthält. Werden Anfragen an den CRM gestellt, so werden erst die Directory Header durchsucht, dann wird auf die benötigten Seiten zugegriffen.

3. Speicherstruktur

Es ist Ziel, die Objekte des internen Datenmodells auf physische Seiten abzubilden, das heißt, NF^2-Tupel werden als abstrakte komplexe Sätze (Abstract Complex Record – ACR) betrachtet und in mehreren Schritten in konkrete komplexe Sätze umgewandelt, welche aus Basissätzen bestehen (siehe ABBILDUNG 3.18 und ABBILDUNG 3.20). Abstrakte komplexe Sätze (ACR) stellen die interne Darstellung der NF^2-Tupel dar. Als abstrakt werden sie

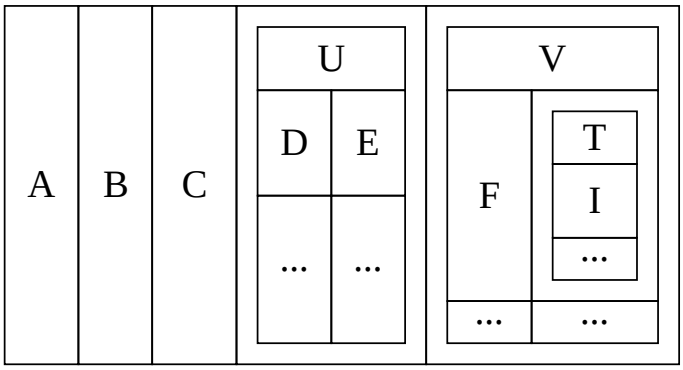
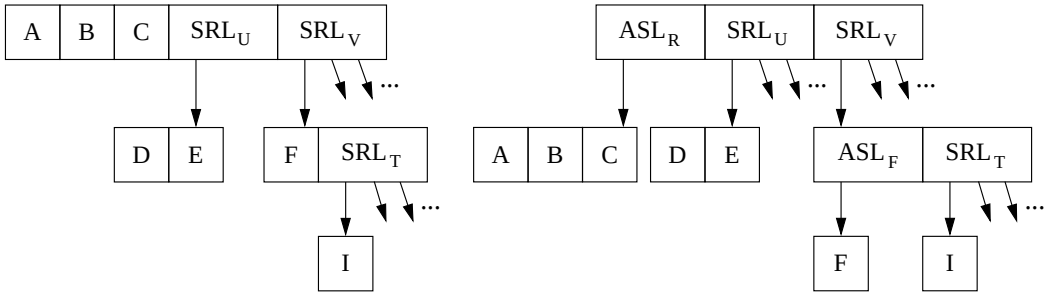


ABBILDUNG 3.18: Abstrakter komplexer Satz



(a) Segmentierung der Unterrelationen

(b) Segmentierung der atomaren Felder

ABBILDUNG 3.19: Segmentierung komplexer Sätze

bezeichnet, da die Daten nicht wirklich in diesem Format gespeichert werden, sondern eine Auftrennung der geschachtelten Daten erfolgt.

▷ *Segmentierung der Unterrelationen*

Unterrelationen werden von atomaren Attributen getrennt (siehe ABBILDUNG 3.19(a)). Dadurch wird der komplexe Satz in Untersätze geteilt, wobei der ursprüngliche Satz auch als Vatersatz bezeichnet wird. In diesem stehen nun anstelle der eingebetteten Unterrelationen sogenannte Unterrelationenverknüpfungen (Subrelation Links, SRL), welche den Vatersatz mit den abgeleiteten Teilen verbinden. Diese Unterrelationenverknüpfungen sind Mengen von einzelnen Verknüpfungen auf alle Elemente einer Unterrelation. Die so entstandenen Untersätze werden alle im gleichen Datenbanksegment gespeichert.

▷ *Segmentierung der atomaren Felder*

Untersätze, die atomare Felder und SRLs enthalten, werden in ein Atomsegment (Atomic Segment) und ein Verknüpfungssegment (Link Segment) geteilt (siehe ABBILDUNG 3.19(b)). Dabei finden die atomaren Werte in den Atomsegmenten Platz, und in den Verknüpfungssegmenten stehen nun die Verknüpfungen (Atomic Segment Links, ASL) zu diesen. Untersätze mit ausschließlich atomaren Feldern bleiben unverändert.

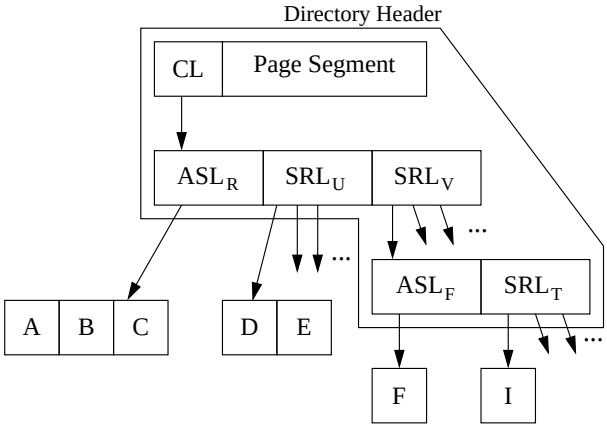


ABBILDUNG 3.20: Konkreter komplexer Satz

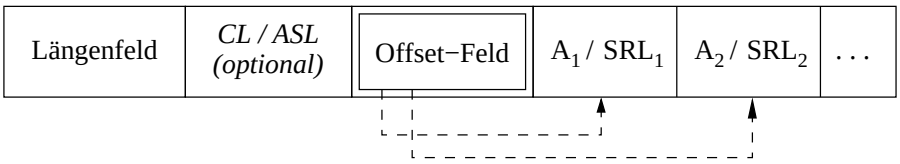


ABBILDUNG 3.21: Struktur eines Basissatzes

▷ *Kapselung*

Es wird ein Seitensegment (Page Segment) erzeugt, welches die Liste der verwendeten Seiten und deren Freispeicherinformationen enthält sowie einen sogenannten Connection Link (CL), welcher auf das oberste Verknüpfungssegment zeigt (siehe ABBILDUNG 3.20). Das Seitensegment versteckt die interne Struktur und bildet mit allen Verknüpfungssegmenten den Directory Header.

Wie schon erwähnt, ist eine Unterrelationenverknüpfung (SRL) eine Menge von Verknüpfungen, welche jeweils auf ein Element der Unterrelation verweist. Ein SRL wird als Feld mit Referenzen (Pointer Array) implementiert, an dessen Anfang ein Kardinalitätsfeld gesetzt wird, welches die Referenzen zählt und somit die Länge des Zeigerfeldes bestimmt. Zusätzliche Vaterzeiger gibt es nicht. Die nun entstandenen Basissätze sind Bytefolgen variabler Länge und bestehen aus einem Längenfeld, optional einer Verknüpfung (entweder CL oder ASL) und einem Offset-Feld (siehe ABBILDUNG 3.21. Ein Offset verweist auf den Beginn eines Feldes A_i (atomar) oder SRL_i (link) im Datenteil des Basissatzes. Basissätze können die Seitengröße nicht überschreiten, sondern finden komplett innerhalb einer Seite Platz.

4. Adressierung

Bei der Adressierung der Datensätze wird zwischen internen und externen Adressen unterschieden:

▷ *Interne Adressen*

Interne Adressen sind nur innerhalb des Seitensegments des komplexen Satzes gültig. Die Adressierung eines Basissatzes erfolgt nach dem Mini-TID-Konzept. Ein Mini-TID

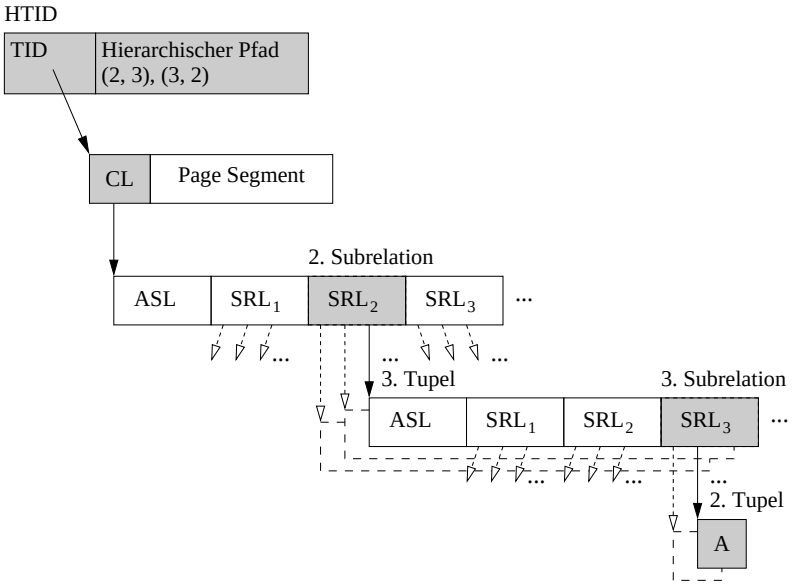


ABBILDUNG 3.22: Beispiel für eine hierarchische TID

besteht aus einer relativen Seitenkennung (Relative Page Identifier) und einer seitenlokalen Eintragsnummer (Page-Local Entry Number). Die Seitenkennung gibt hier die relative Nummer einer Seite des komplexen Satzes an. Belegt ein komplexer Satz zum Beispiel 4 Seiten, so sind die möglichen Werte 1 bis 4. Welche Seiten dies physisch sind, steht im Seitensegment (Page Segment) an der entsprechenden Stelle. Eine Seite beginnt mit einem Seitenkopf (Page Header), welcher zusätzliche Informationen über die Seite enthält. Der Rest der Seite, der Page Trailer, besteht aus Einträgen, die auf die in der Seite gespeicherten Basissätze zeigen. Ein solcher Verweis ist eine seitenlokale Byteadresse. Die seitenlokale Eintragsnummer der Mini-TID gibt nun, wie beim normalen TID-Konzept, die Nummer des Eintrags in der Seite an, an der sich der Basissatz befindet.

Wächst nun ein Basisatz an, so gilt es, zwei Fälle zu unterscheiden:

- ◇ *Der Seiteninhalt paßt trotz Wachstum noch in eine Seite.*

  Hierbei ist eine Verschiebung innerhalb der Seite notwendig, der aber lediglich eine Änderung des seitenlokalen Verweiseintrags und somit keine Mini-TID-Adressenänderung erfordert.

- ◇ *Der Seiteninhalt wächst über die Seite hinaus.*

  Es ist eine Verschiebung auf andere Seiten nötig. Eine Änderung der Mini-TID wird im übergeordneten Verknüpfungssegment (Link Segment) ausgeführt.

▷ *Externe Adressen*

Externe Adressen referenzieren einen komplexen Satz von außerhalb, etwa aus einem Index. Hierfür wurde das TID-Konzept zu einem hierarchischen TID-Konzept (HITID) erweitert (siehe ABBILDUNG 3.22). Mittels TID wird das Seitensegment (Page Segment) des komplexen Satzes referenziert. Das heißt, daß die Seite mit dem Directory Header des komplexen Satzes referenziert wird. Dies geschieht allerdings mit einer Seitennummer beziehungsweise -kennung (Page Identifier), die relativ zum jeweiligen

Datenbanksegment ist. Weiterhin umfaßt die TID eine Eintragsnummer, die wie bei der Mini-TID auf einen Eintrag im Seitenverzeichnis zeigt. Dieser enthält dann die Byteadresse des Basissatzes im Seitensegment. Soll nun auf innere Subtupel zugegriffen werden, so müssen die relativen Subrelationen-Nummern und Tupel-Sequenznummern aller Vorgänger in der Hierarchie vom Basissatz zum Subtupel angegeben werden. Die erste relative Relationennummer bestimmt die oberste Unterrelationenverknüpfung (SRL), die Tupelsequenznummer bestimmt in dieser SRL den Eintrag des Zeigerfeldes, welcher den gewünschten Zeiger enthält. Dies wird fortgesetzt, bis das Referenzziel erreicht ist. Die Erweiterung der TID zur HITID besteht also in einem hierarchischen Pfad, das heißt einer Folge von Nummern, die ein Navigieren durch die Verknüpfungshierarchie des komplexen Objekts erlauben.

Zusammenfassend kann man feststellen, daß folgende Konzepte genutzt werden:

- ▷ Auslagerung von Kollektionselementen bei Objekten, die größer als eine Seite sind
- ▷ Clusterung
- ▷ Kollektionen als Feld von Referenzen (Pointer Array)
  Die einzelnen Unterrelationenverknüpfungen (SRLs) sind Felder von Zeigern auf Mengenelemente.
- ▷ Sätze (Basissätze) sind durch die Seitengröße beschränkt (keine seitenübergreifenden Sätze)
- ▷ Trennung von Daten und Strukturinformationen
- ▷ Nutzung von Indexen

3.1.10 ADAPLEX

Das hier zugrunde gelegte semantische Datenmodell stellt einen Alternativvorschlag zum relationalen Modell zur Datenmodellierung dar. Zugehörige Implementierungsaspekte werden in [CDF+82] dargestellt. Darin wird ein Datenbank-Management-System beschrieben, das über eine in die Programmiersprache ADA integrierte Datenbanksprache namens DAPLEX angesprochen wird. Das Integrationsergebnis aus beiden Sprachen wird ADAPLEX genannt. Mit ADAPLEX soll es nun möglich sein, semantisch angereicherte Datenelemente für Datenbanken zu definieren.

Semantische Datenmodelle stellen Erweiterungen des ER-Modells dar. Sie wurden aus der Erkenntnis heraus entwickelt, daß manche Konzepte sich nur unvollständig im ER-Modell darstellen lassen. Daher wurden die allgemeinen Beziehungen des ER-Modells durch „semantisch angereicherte“ Beziehungstypen ersetzt [HS00]. Die Datenstrukturen werden ebenfalls mit Entitätstypen und Beziehungstypen zwischen diesen aufgebaut. Als Grundbausteine sind also Entitäten gegeben. Auf diesen Entitäten sind Funktionen definiert. Dies ist in etwa vergleichbar mit Objekten und ihren Eigenschaften (Attributen). Die Entität stellt das Objekt dar und die Funktionen auf diesem Objekt die Eigenschaften. Der Zusammenhang zwischen Funktionen und Eigenschaften ist der, daß Funktionen gewisse Eigenschaften der Entität zurückgeben (eine Funktion für eine Eigenschaft). Dabei kann eine Eigenschaft einen einzelnen Wert oder eine Menge von Werten darstellen. Entsprechend werden die Funktionen einwertig oder mengenwertig genannt. Die Werte können von nicht zusammengesetzten Typen sein, also durch ADA unterstützte Datentypen sowie Zeichenketten, als auch von zusammengesetzten Typen, das heißt, von in der Datenbank gespeicherten Entitäten. Im Fall, daß ein Wert eine Entität darstellt, verweist dieser nur auf diese Entität, stellt also eine Referenz dar.

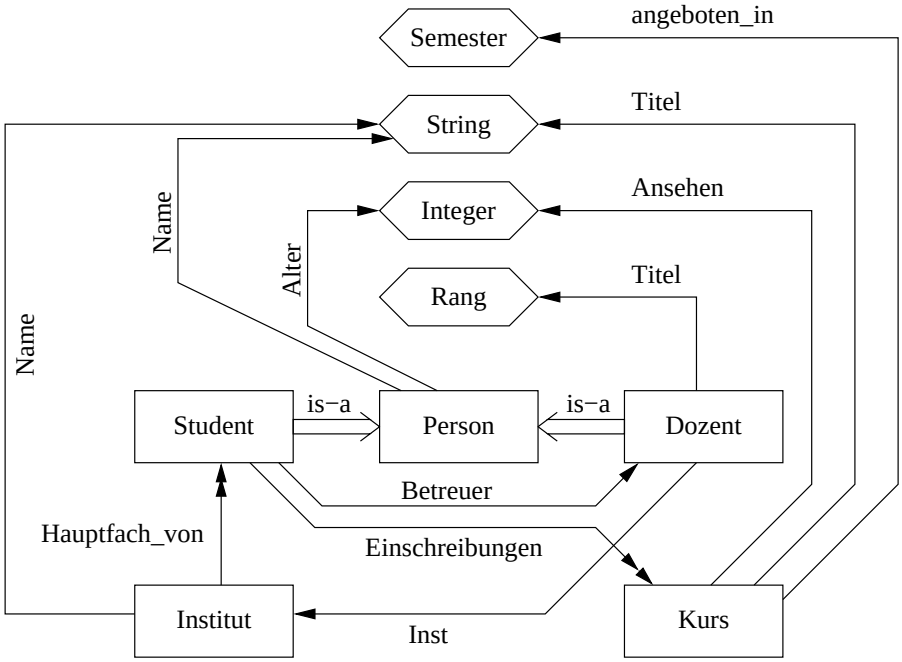


ABBILDUNG 3.23: Beispiel einer Datenbank für eine Universität

Haben Entitäten gleiche Eigenschaften, so werden sie Entitätstypen zugeordnet. Entitätstypen können in Spezialisierungs- beziehungsweise Generalisierungshierarchien eingebunden sein.

Zur Veranschaulichung der Generalisierung von Typen und der Zusammenhänge zwischen den Entitäten ist in ABBILDUNG 3.23 das in [CDF+82] dargestellte Datenbankschema abgebildet. Dabei werden zusammengesetzte Entitätstypen als Rechtecke dargestellt, einfache beziehungsweise mit ADA definierbare Typen werden in Diamantenform abgebildet. Pfeile mit doppelten Köpfen kennzeichnen mengenwertige Funktionen, und Doppelpfeile symbolisieren Is-A-Beziehungen in einer Generalisierungshierarchie.

Für die Speicherung komplexer Objekte in diesem System bietet sich eine 1:1-Beziehung zwischen Objekt und Entität an. Damit ist gemeint, daß jedes Objekt der Anwendung als eine Entität in der Datenbank gespeichert wird. Für jede Klasse von Objekten wird ein Entitätstyp definiert. Existiert also eine Klasse Student, die von einer Klasse Person abgeleitet wurde, so werden hierfür die Entitätstypen Student und Person erstellt, die wiederum in einer Is-A-Beziehung stehen (siehe Beispiel aus ABBILDUNG 3.23). Besitzt das Objekt Attribute einfacher (von ADA unterstützter) Datentypen, so wird für jedes Attribut eine Funktion auf dessen ADA-Datentyp definiert. Bei mengenwertigen Attributen von einfachen Datentypen sind die entsprechenden Funktionen mengenwertig. Besitzt das Objekt Attribute, die Teilobjekte oder Mengen von Teilobjekten sind, so wird für jedes Attribut eine Funktion (ein- oder mengenwertig) auf den entsprechenden Entitätstyp des Teilobjekts definiert.

Die physische Speicherung der so definierten komplexen Objekte (komplexe Entitäten) erfolgt auf blockorientierten Externspeichermedien. Dazu werden die Objekte entsprechend umgewandelt: Wird ein Objekt (eine Entität) angelegt, so wird ihm ein eindeutiger Identifikator zugewiesen. Hierdurch kann jedes Objekt unabhängig von seinen Daten eindeutig identifiziert werden. Das Prinzip der Objektidentität wird somit implementiert. Zusätzlich

müssen noch weitere Informationen für jede Entität gespeichert werden, da Entitäten zu mehreren Entitätstypen gehören können und somit auch Funktionen besitzen können, die von mehreren Typen stammen:

- ▷ *Werte der anwendbaren Funktionen*
  Dies entspricht den Werten der zugehörigen Attribute.

- ▷ *Typinformationen*
  Für gegebene Entitäten werden die zugehörigen Entitätstypen gespeichert.

- ▷ *Zusätzliche Informationen*
  Sollen Entitäten gelöscht werden, so müssen sie aus der Datenbank entfernt werden. Dies ist jedoch nur dann erlaubt, wenn keine Referenzen mehr auf die Entität bestehen. Zur effizienten Überprüfung wird ein Referenzzähler für jede Entität gespeichert.

Weiterhin werden die Entitäten beziehungsweise deren Funktionen in logische Sätze eingeteilt. Da Funktionen der Entitäten Werte von bestimmten Eigenschaften zurückgeben, kann man sich dies als Aufteilung der Attribute in Sätze vorstellen. Hierfür bestehen verschiedene Möglichkeiten, welche unterschiedliche Clusterungsstrategien repräsentieren:

- ▷ *Keine Gruppierung*
  Jede Funktion wird separat als binäre Relation gespeichert. Diese Relation hat je eine Spalte zur Repräsentaion des Definitionsbereiches der Funktion sowie zur Repräsentation des Wertebereiches der Funktion. Die Definitionsbereichsspalte nimmt Identifikatoren von Objekten des jeweiligen Entitätstyps als Funktionsargumente auf. Das sind im Endeffekt alle Identifikatoren der Enittätsobjekte, auf denen die Funktion beziehungsweise das (Funktions-)Attribut definiert ist. In der Wertebereichsspalte sind diesen Objekt-IDs die Funktionswerte zugeordnet, sie sind Werte (beziehungsweise Objekte) des in der Funktionsdefinition festgelegten Adatyps (beziehungsweise Entitätstyps). Ein Tupel dieser Relation hat somit die Form [E-ID, Wert].
  Jede so entstehende binäre Relation wird in ein separates Segment gespeichert.

- ▷ *Komplette Gruppierung*
  Die Werte aller Funktionen einer Entität (das heißt aller Funktionen der Entitätstypen denen die Entität angehört) werden zusammen in einem logischen Satz gespeichert.

- ▷ *Semantische Gruppierung*
  Die Werte aller Funktionen einer Entität werden so gruppiert, daß diejenigen im selben Satz gespeichert werden, die zum gleichen Entitätstyp gehören.

- ▷ *Beliebige Gruppierung*
  Die Werte aller Funktionen einer Entität können so gruppiert werden, wie es der Nutzer für seine Anwendungen als passend ansieht.

In der hier vorgeschlagenen Implementierung wird die semantische Gruppierung gewählt, und alle zum Entitätstyp gehörenden Funktionswerte werden im selben Satz gespeichert. Für den Fall, daß Felder mit beliebiger Länge auftreten (variabler Länge oder mengenwertig) und diese zu Speicherallokationsproblemen führen, wird jedoch die Option geboten,

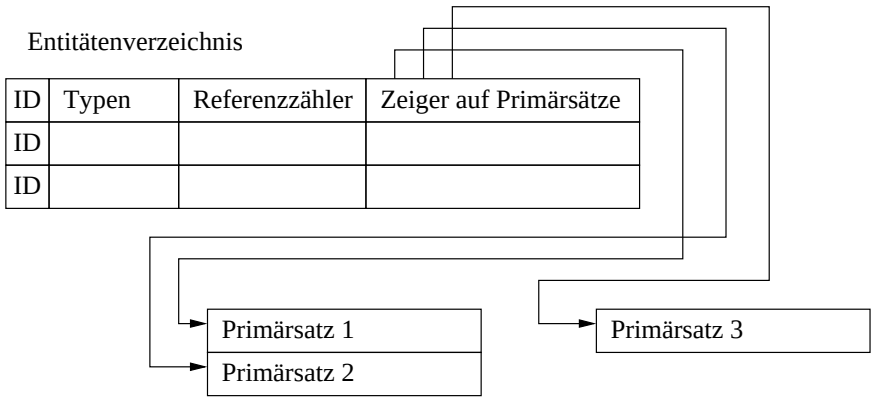


ABBILDUNG 3.24: Entitätenverzeichnis

diese Felder nach dem Gruppierungstyp 1 in jeweils einen Sekundärsatz auszulagern. Ein solches Speicherallokationsproblem könnte zum Beispiel sein, daß der Primärsatz nicht mehr in die für ihn vorgesehene Seite paßt.

Es existiert somit für jeden Entitätstyp ein primärer Satztyp. Zusätzlich wird in jedem Primärsatz der Identifikator der Entität gespeichert, für die der Satz Daten enthält. Der Rest des Satzes ist eine Anzahl von sich wiederholenden und nicht wiederholenden Feldern für jede mengenwertige und einwertige Funktion (die nicht in einen Sekundärsatz ausgelagert wurden).

Falls Sekundärsätze existieren, so werden diese in separate Segmente ausgelagert (nach Gruppierungstyp 1). Das Format dieser Sätze wird jedoch nicht weiter erklärt. Außerdem werden sekundäre Sätze nicht in die Clusterung einbezogen.

Für Objekte (Entitäten) gilt also, daß, wenn sie nur einem Entitätstyp zugeordnet sind, auch nur aus einem (logischem) Primärsatz bestehen.

Um die restlichen Informationen der Entitäten zu speichern (Referenzzähler und Typinformationen), wird ein Entitätenverzeichnis (Entity Directory) für jede Generalisierungshierarchie angelegt (siehe ABBILDUNG 3.24). Dieses dient der zentralisierten Speicherung der Informationen von Entitäten. Für jede Entität, die zu mindestens einem Entitätstyp der zugehörigen Generalisierungshierarchie gehört, wird ein Eintrag im Entitätenverzeichnis angelegt. Ein solcher Eintrag enthält folgende Informationen:

- ▷ Identifikator der Entität
- ▷ Typinformationen (zugehörige Entitätstypen)
- ▷ Referenzzähler
- ▷ physische Zeiger auf alle Primärsätze (einen für jeden Entitätstyp)

Die Einträge des Entitätenverzeichnisses werden ebenfalls in Sätzen gespeichert. Da Entitäten zu unterschiedlich vielen Entitätstypen gehören können, wurde hierfür eine Satzstruktur variabler Länge gewählt. Das gesamte Verzeichnis wird mittels linearem Hashing organisiert.

Sind bis jetzt die Entitäten in logische Sätze aufgeteilt worden, so ist es nun nötig, diese in physische Sätze zu wandeln (siehe ABBILDUNG 3.25). Hierfür besteht die Möglichkeit, die primären logischen Satztypen in mehrere getrennte Speichersatztypen aufzuteilen. Für einen logischen Satztyp seht immer ein Entitätstyp. Für diesen Entitätstyp können Entitäten existieren, die auch zu anderen Entitätstypen gehören. Somit können alle logischen Primärsätze danach eingeteilt werden, ob ihre zugehörigen Entitäten ebenfalls zu einem an-

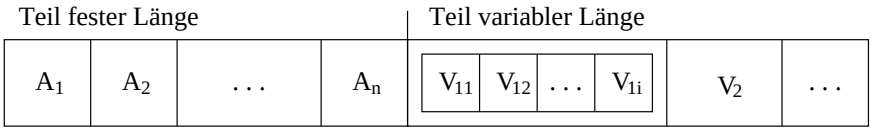


ABBILDUNG 3.25: Speichersatz

deren Entitätstyp gehören oder nicht. Im Beispiel kann der Satztyp für Person dahingehend eingeteilt werden, ob die Sätze auch zu Studenten gehören oder nicht.

Die entstandenen Speichersätze müssen nun in Seiten physisch abgelegt werden. Hierfür besteht wieder die Möglichkeit zu clustern. Dies soll mit Hilfe des Beispiels erklärt werden. Dort existiert der Entitätstyp Person und dessen logischer Satztyp. Diese Sätze wurden in diejenigen unterteilt, die zu einer Entität vom Typ Dozent gehören (a), und diejenigen, die nicht dazu gehören (b). Nun werden alle Speichersätze vom Typ Dozent mit denen vom Typ (a) geclustert. Dadurch wird eine objektbezogene Clusterung erreicht, das heißt, alle Sätze eines Objekts werden physisch dicht gespeichert. Um die Clusterung umzusetzen, werden sogenannte Hybridsätze genutzt, die Speichersätze unterschiedlichen Typs enthalten dürfen.

Vom System wird auch eine andere Clusterungsstrategie unterstützt. Dabei wird die Clusterung von den Funktionen der Entitäten bestimmt. Im Beispiel würde so der Datensatz eines Dozenten, der durch die Funktion Inst einem Institut zuordnet wird, möglichst in der Nähe des Insitutsdatensatzes gespeichert werden.

Zusammenfassend läßt sich hier die Nutzung folgender Speicherkonzepte für komplexe Objekte festellen:

- ▷ *Auslagerung von Kollektionselementen, falls Elemente Objekte sind und Speicherung von Kollektionen als Referenzenfeld (Pointer Array)*

  Die Elemente von Mengen (Kollektionen) werden als Referenzen im Objekt gespeichert
- ▷ *Keine Auslagerung von Kollektionselementen, falls Elemente primitiv sind*
- ▷ *Auslagerung sehr langer Attribute (Zeigerfelder oder lange Zeichenketten)*

  Sind Attribute zu lang, so werden sie ausgelagert. Auf ihnen ist dann jedoch keine Clusterung möglich.
- ▷ *Clusterung*
- ▷ *Keine Nutzung von seitenübergreifenden Sätzen*

  Sätze sind stets durch die Seitengröße beschränkt.
- ▷ *Nutzung von Indexen*

3.1.11 Objektrepräsentation nach Khoshafian und Valduriez

Die in [KV87] verwendete DBMS-Architektur soll kurz beschrieben werden. Grob betrachtet besteht diese aus einer von den Autoren als konzeptuell bezeichneten Schicht, welche auf die Manipulation von komplexen Objekten fokusiert, und aus der internen Schicht, welche sich den Fragen der physischen Speicherungsform widmet (siehe ABBILDUNG 3.26). Auf der konzeptuellen Schicht geht es um die Repräsentation und Manipulation der Objekte im Hauptspeicher. Die physische Repräsentation im Sekundärspeicher ist dagegen eine Frage, die auf der internen Schicht behandelt wird. Dabei kommuniziert das Anwendungsprogramm auf der Basis des konzeptuellen Schemas mit der konzeptuellen Schicht und diese auf der Grundlage des internen Schemas mit der internen Schicht.

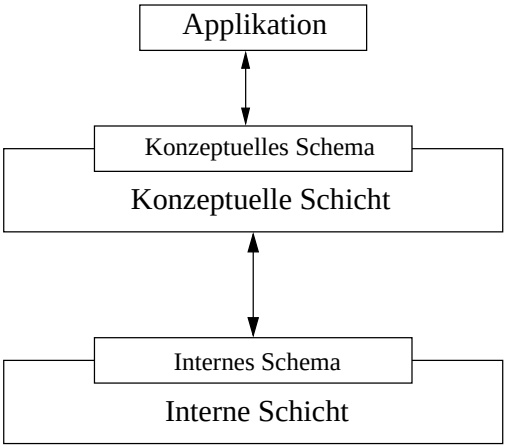


ABBILDUNG 3.26: Grundaufbau des DBMS

Objekte des konzeptuellen Modells haben die Form (Identifikator, Typ, Wert). Dabei ist der Identifikator ein Element einer gegebenen Identifikatormenge. Typ kann folgende drei Werte annehmen: Atom, Menge, Tupel. Der Wert ist abhängig vom Typ:

- ▷ Ist das Objekt atomar, so besteht er aus dem Wert des Atoms.
- ▷ Ist das Objekt eine Menge, so besteht er aus einer Menge von Identifikatoren für Elementobjekte
- ▷ Ist das Objekt skalar (vom Typ Tupel), so hat der Wert die Form (a_1:i_1,...,a_n:i_n), wobei a_j Attributnamen und i_j Identifikatoren für Elementobjekte sind.

Es wird zwischen zwei Arten von Objekten unterschieden: persistente und transiente. Um die durch das konzeptuelle Modell entstehenden Objekthierarchien zusammenzufassen, wird der Begriff des Datenbank-Wurzelobjekts (Database Root) eingeführt. Jedes von diesem Objekt erreichbare andere Objekt wird persistent, die nicht erreichbaren bleiben transient.

Die Objekte des konzeptuellen Modells werden auf das interne Modell abgebildet. Die Objekte des internen Modells sind ebenfalls aus Mengen-, Tupel- und Atomobjekten aufgebaut. Die im konzeptuellen Modell gegebene Objektidentität wird hier mittels Surrogaten umgesetzt. Diese sind systemgenerierte eindeutige Werte beziehungsweise Bezeichner oder Identifikatoren. Bei der Überführung zwischen den Modellen werden folgende Schritte durchgeführt:

- ▷ Für Mengen und Tupel wird mit Einführung der Surrogate die Identität implementiert.
- ▷ Referentiell gemeinsam genutzte Teilobjekte, das heißt Objekte mit mehreren Vaterobjekten, werden entweder getrennt von diesen gespeichert und alle Vaterobjekte erhalten Verknüpfungen mit ihren Teilobjekten oder die Teilobjekte werden bei einem beliebigen Vaterobjekt gespeichert und die restlichen erhalten Referenzen auf ihre Teilobjekte.
- ▷ Zur erweiterten assoziativen Suche unterstützt die interne Schicht Zugriffsstrukturen, wie zum Beispiel B*-Bäume oder Hashtabellen. Außerdem können mehrere Kopien der Objekte gespeichert werden, die jeweils nach einem anderen Attribut geclustert

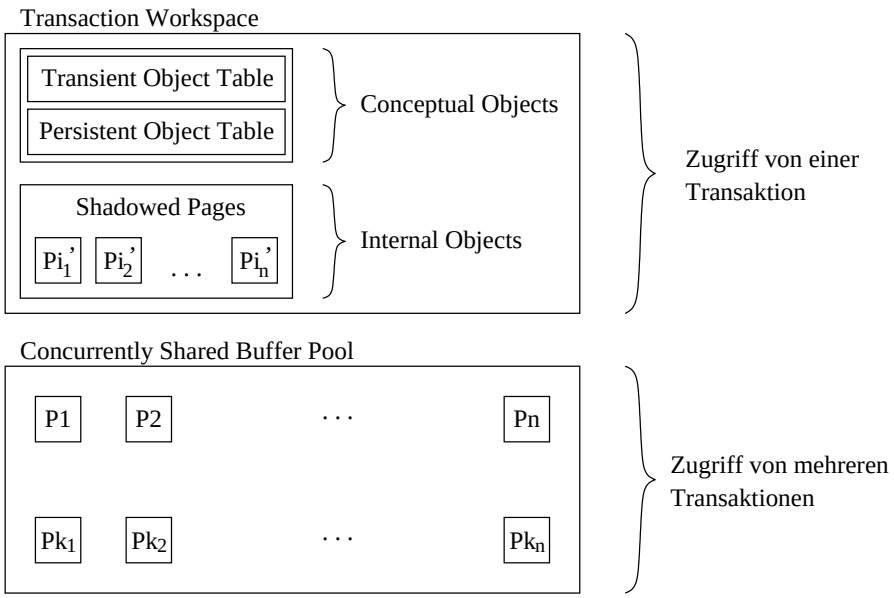


ABBILDUNG 3.27: Arbeitsbereich einer Transaktion

werden. Dies ist jedoch in erster Linie für Objekte, auf die nur lesend zugegriffen wird, gedacht, da bei diesen durch die redundante Speicherung keine unvertretbaren Aufwände bei Änderungsoperationen drohen.

Arbeitet ein Programm mit Objekten, so werden diese vom DBMS im ‚konzeptuellen' Format auf Speicherseiten gehalten (Speicherseitenbereich der Anwendung). Dies betrifft persistente sowie transiente Objekte. Objekte des internen Modells, die vom Programm geändert, jedoch noch nicht aus dem Puffer auf den Extenspeicher geschrieben wurden, werden in sogenannten Schattenseiten (Shadowed Pages) in einem weiteren Speicherbereich zwischengelagert. Außerdem werden die von mehreren Transaktionen gemeinsam genutzten Speicherseiten in einem wiederum anderen Pufferbereich aufbewahrt. Persistente und transiente Objekte (Objekte der konzeptuellen Schicht) werden in Persistent Object Tables und Transient Object Tables gespeichert (siehe ABBILDUNG 3.27). Persistente Objekte werden in die Persistent Object Table entweder aus den Schattenseiten oder aus den gemeinsam genutzten Datenseiten geladen. Jede Schattenseite entspricht genau einer Seite im Bufferpool oder auf dem Externspeicher. Wird die Transaktion beendet, so ersetzen die Schattenseiten die Seiten im Bufferpool oder die Externspeicherseiten, je nachdem, ob die Seiten gemeinsam genutzt wurden oder nicht.

Zur Veranschaulichung sind in ABBILDUNG 3.28 Objekttabellen abgebildet. Die Einträge in diesen Tabellen sind Tripel und haben die Form (Objekttyp, Referenzzähler, Objektwert). Objekttypen können hierbei wieder Menge, Tupel oder Atom sein. Der Referenzzähler zählt die Anzahl der Vaterobjekte. Dies dient der Freigabe des von dem Objekt belegten Speichers, falls kein Vaterobjekt mehr darauf verweist. Ist das Objekt atomar und von fester Länge, so wird dessen Wert direkt in der Objekttabelle gespeichert (‚Objektwert'). Sonst ist der ‚Objektwert' ein Zeiger. Es ist auch erlaubt, daß ein transientes Objekt mittels eines Zeigers ein persistentes Objekt referenziert, jedoch nicht umgekehrt; das stünde im Widerspruch zum Persistenzbegriff. Referenzen auf Objekte haben die Form (Table Selector, Index), wobei der Table Selector entweder die Transient oder die Persistent Object Table auswählt und die Index-Angabe einen Zugriffspfad der entsprechenden Tabelle be-

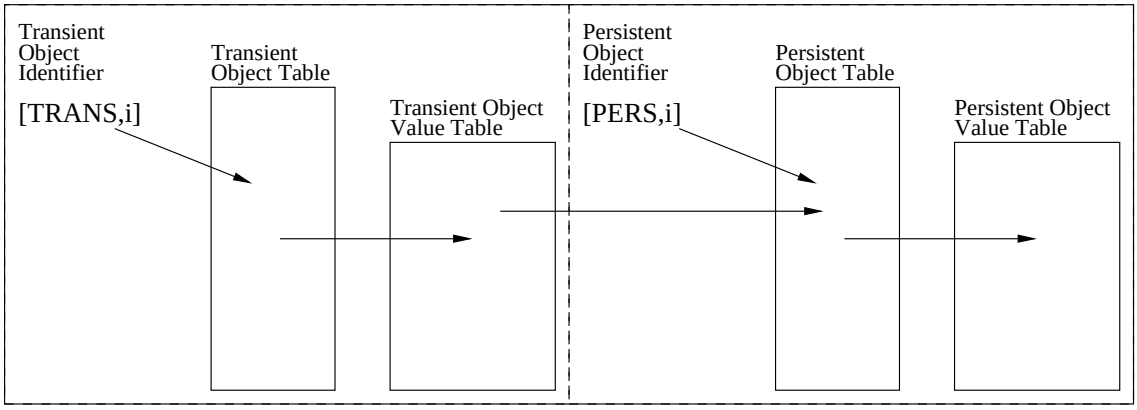


ABBILDUNG 3.28: Konzeptuelle Objekttabellen

zeichnet. Die Werte der Mengen und Tupelobjekte sowie der atomaren Objekte variabler Länge werden in separaten Tabellen gespeichert, den Objektwerttabellen. Zeichenketten haben dabei die Form (Attributname, Wert). Mengen und Tupel werden als Listen gespeichert, deren Elemente die Form (Attributname, Identifikator) haben. Bei Mengen bleibt der Attributname leer.

Die in den Objekttabellen verzeichneten Objekte werden mit allen zugehörigen Attributen und Unterobjekten möglichst in einer Seite gespeichert. Belegt also der Satz eines Objekts Platz in einer Seite, so werden möglichst alle weiteren Sätze mit Teilobjekten ebenfalls in dieser Seite gespeichert. Bei gemeinsam genutzten Teilobjekten ist dies jedoch nur bei einem Vaterobjekt der Fall. In den Seiten der anderen Objekte stehen statt dessen Sätze mit Referenzen. Dies geschieht jedoch nur so lange, wie die Objektgröße die Seitengröße nicht überschreitet. In diesem Fall muß das Objekt auf mehrere Seiten aufgeteilt werden. Innerhalb eines Satzes werden zur Trennung der einzelnen Daten eines Objekts nicht weiter beschriebene Separatoren genutzt. Auch die Umsetzung der mengenwertigen Attribute wird nicht genau beschrieben. Es wird jedoch die Nutzung von Feldern vorgeschlagen, die Referenzen auf alle Mengenelemente enthalten. Diese Speicherform soll in dieser Arbeit als Zeigerfeld (Pointer Array) bezeichnet werden.

Zusammengefaßt kann man die Nutzung folgender Speicherkonzepte bei [KV87] feststellen:

- ▷ Auslagerung von Kollektionselementen
- ▷ Kollektionen als Zeigerfelder (Pointer Arrays)
- ▷ Clusterung auf Objektebene
- ▷ Gemeinsam genutzte Teilobjekte möglich
- ▷ Seitenübergreifende Sätze (Spanned Records)
- ▷ Nutzung von Indexen

3.1.12 ORION

ORION ist ein Datenbankprojekt, welches Ende 1985 im Advanced Computer Technology Program (ACT) in der Microelectronics and Computer Technology Corporation (MCC) gestartet wurde. Im Laufe dieses Projektes entstanden mehrere Datenbanksysteme, die als OODBMS einzuordnen sind [KGBW90].

ORION besteht aus vier größeren Subsystemen:

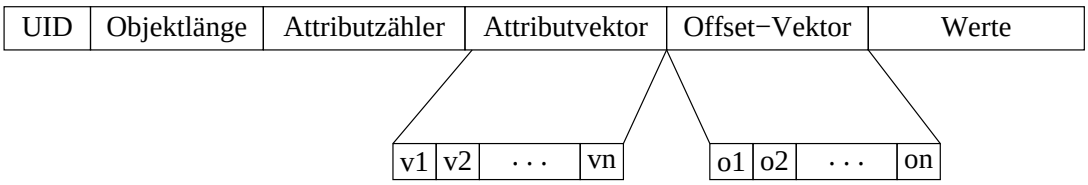


ABBILDUNG 3.29: Speicherformat von Externspeicherobjekten in ORION

▷ *Message Handler*

An den Message Handler gehen alle Nachrichten, die an das ORION-System geschickt werden.

▷ *Objektsystem*

Dies bietet Datenmanagement-Funktionen auf oberster Stufe an, enthält Anfrageoptimierung, Schemaverwaltung sowie Verwaltung großer Daten und unterstützt Objektversionen, zusammengesetzte Objekte und Multimediaobjekte.

▷ *Transaktionsmanagement-System*

Dies koordiniert den konkurierenden Zugriff auf Objekte und bietet Wiederherstellungsfunktionen.

▷ *Speichersubsystem*

Hier wird die persistente Speicherung von Objekten verwaltet und der Transfer der Objekte zwischen Hauptspeicher und sekundären Speichermedien.

Im folgenden soll ein Teil des Speichersubsystems von ORION dargestellt werden.

Werden Objekte in ORION gespeichert und wird darauf gearbeitet, so treten diese in zwei Formaten auf: ein Format für den Externspeicher und ein Format für den Arbeitsspeicher. Ist ein Objekt auf dem Externspeicher gespeichert, so hat es das Externspeicherformat (zur Speicher- und Anfrageeffizienz). Im Arbeitsspeicher hat es dagegen das Speicherformat, so daß Programme die Objekte manipulieren können, wie es die Programmiersprache vorgibt. Daraus ergibt sich, daß es in ORION zwei Pufferbereiche gibt. Begründet wird dies dadurch, daß kein einzelner Pufferbereich für beide Objektformate gleichzeitig geeignet ist. Objekte im Externspeicherformat werden im Seitenpufferbereich zwischengelagert, die anderen im Objektpufferbereich.

Wird nun auf ein Objekt zugegriffen, so wird die Seite, welche das Objekt enthält, in den Seitenpuffer geladen. Da Seiten mehrere Objekte enthalten können, wird anschließend innerhalb dieser Seite nach dem Objekt gesucht, es geladen und in einen Objektpuffer kopiert. Nun können Programme direkt auf diesen Objekten arbeiten. Werden Änderungen vorgenommen, so werden diese vom Transaktionsmanager auf Einhaltung der Integritätsbedingungen überprüft.

Für die persistente Speicherung der Objekte ist der Speichermanager zuständig. Dieser transferiert Seiten und Segmente in und aus dem Speicher und plaziert die Objekte auf dem Externspeicher. Dazu teilt der Speichermanager den Externspeicher in Partitionen ein, die ihrerseits in Segmente unterteilt werden. Segmente erfahren wiederum eine Unterteilung in Seiten, die die kleinste Einheit des Speichermanagers darstellen.

Sollen nun Objekte persistent gespeichert werden, so müssen sie in das Externspeicherformat überführt werden, welches in ABBILDUNG 3.29 gezeigt wird.

Um Objekte eindeutig identifizieren zu können, besitzen sie eine eindeutige logische Kennung. Dies ist hier die UID (Unique Logical Identifier), welche aus einer eindeutigen Kennung für die Klasse des Objekts und einer eindeutigen Kennung für das Objekt innerhalb der Klasse besteht. Das nächste Feld enthält die absolute Länge des Objekts. Im Feld darauf befindet sich die Anzahl der Attribute, welche im Externspeicherformat gespeichert werden. Der Attributvektor besteht aus Bezeichnern v_i aller Attribute, für die das Objekt direkt angegebene Werte hat. Der darauf folgende Offset-Vektor besteht aus den Offsets o_i der Attribute v_i, und zeigt in den Werteteil, welcher den letzten Teil darstellt. Darin werden nun die Werte der Attribute gespeichert. Ein Attribut kann dabei atomar oder mengenwertig sein und einen primitiven Typ oder einen Referenztyp besitzen. Für Referenzen auf andere Objekte werden deren UIDs verwendet.

Die somit beschriebene Speicherung löst hierarchisch strukturierte Objekte auf. Anstelle der Unterobjekte stehen deren Referenzen. Mengenwertige Attribute primitiver Typen werden im Objekt gespeichert, von Unterobjekten nur die Referenzen.

Eine automatische Clusterung ist somit nicht vorhanden. ORION clustert Instanzen der selben Klasse im selben physischen Segment. Ebenso die Instanzen einer nutzerdefinierten Kollektion von Klassen.

Zusammenfassend kann bei ORION die Nutzung folgender Speicherkonzepte für die Implementierung komplexer Objekte festgestellt werden:

- ▷ *Auslagerung von Kollektionselementen, falls diese Objekte sind, und Speicherung von Kollektionen als Zeigerfeld (Pointer Array)*

  Die Elemente von Mengen (Kollektionen) werden als Referenzen im Objekt gespeichert.
- ▷ *Inline-Speicherung von Kollektionselementen, falls Elemente primitiv sind*
- ▷ *Clusterung*

  Instanzen der gleichen Klasse werden im selben Segment gespeichert.
- ▷ *Nutzung von Indexen*

3.1.13 O_2

O_2 [Deu90] ist ein objektorientiertes Datenbanksystem, welches aus dem Altaïr-Projekt entstanden ist. Dieses begann im September 1986 und hatte das Ziel, ein Datenbanksystem der „nächsten Generation“ zu entwickeln.

Im folgenden soll das Objektmodell von O_2 beschrieben werden. Anschließend wird dessen Implementierung vorgestellt.

1. Objektmodell

In O_2 sind alle Informationen mittels Objekten organisiert. Diese haben eine Identität und kapseln ihre Daten sowie ihr Verhalten nach außen hin ab. Sollen die Daten der Objekte geändert werden, so ist dies nur über deren Methoden möglich.

In O_2 wird zwischen den Begriffen Klasse und Typ unterschieden. Instanzen einer Klasse sind Objekte und haben die oben genannten Eigenschaften (Kapselung von Daten und Verhalten). Dagegen sind die Instanzen der Typen Werte, die ihre Daten nicht kapseln. Dadurch ist ihre Struktur nach außen bekannt und sie können verändert werden. Jeder Klasse ist ein Typ zugeordnet, der die Struktur der Instanzen der Klasse bestimmt. Daher können Typen ebenfalls strukturiert sein. Können Klassen explizit durch Kommandos angelegt werden, treten Typen nur als Komponenten der Klassen auf. Gehören nun Objekte

der gleichen Klasse an, so ist ihre Struktur gleich. Außerdem können Klassen von anderen Klassen abgeleitet werden, wodurch sie deren Struktur erben und erweitern können. Als Wurzel des Vererbungshierarchiebaums ist in O_2 die Klasse Object vorhanden.

Ein Objekt ist in O_2 ein Paar eines Identifikators und eines Wertes ([Identifier, Value]). Dabei ist der Identifikator systemweit eindeutig und dient zur Bestimmung einzelner Objekte. Der Wert des Objekts bestimmt dessen Struktur und gehört einem bestimmten Typ an.

Typen sind in O_2 rekursiv aufgebaut und benutzen dafür die in O_2 eingebauten atomaren Typen (Integer, Float, Double, String, Char, Boolean und Bit) und bereits definierte Klassen. Diese Typen können als Elementtypen für die Typkonstruktoren Set, List, und Tuple dienen, so daß komplexe Objektstrukturen definierbar sind.

Der folgende Ausdruck stellt einen O_2-Typ dar:

```
tuple(Name: string,
      Map: bitmap,
      Hotels: set(Hotel))
```

Eine entsprechende Klassendefinition hat die folgende Form:

```
add class City
    type tuple(Name: string,
          Map: bitmap,
          Hotels: set(Hotel))
```

Und Objekte von dieser Klasse werden mit dem New-Kommando erstellt:

```
Berlin = new(City)
```

Detailliertere und formalere Angaben zum Objektmodel sind in [LRV88] zu finden.

2. Implementierung

Das Datenbank-Management-System O_2 ist, wie in ABBILDUNG 3.30 zu sehen ist, aus drei Teilen aufgebaut. Der Schema Manager stellt die oberste Schicht dar. Darunter liegt der Object Manager, unter dem der Disk Manager liegt.

2.1. Disk Manager

Als Disk Manager dient in O_2 das Wisconsin Storage System (WiSS). Dies bietet Persistenz, Externspeicherverwaltung und Zugriffssysnchronisation für physische Sätze. Sollen Daten persistent auf dem Externspeicher gespeichert werden, so bietet WiSS folgende Strukturen: satzstrukturierte Segmente, unstrukturierte Segmente und seitenübergreifende Sätze (Long Data Items). Diese Strukturen werden auf Seiten abgebildet, welche die kleinste Speichereinheit darstellen. Weiterhin unterstützt WiSS Indexe und bietet volle Kontrolle über die physische Position der Seiten. Außerdem umgeht WiSS das Dateisystem des Betriebssystems und bietet eigene Pufferung.

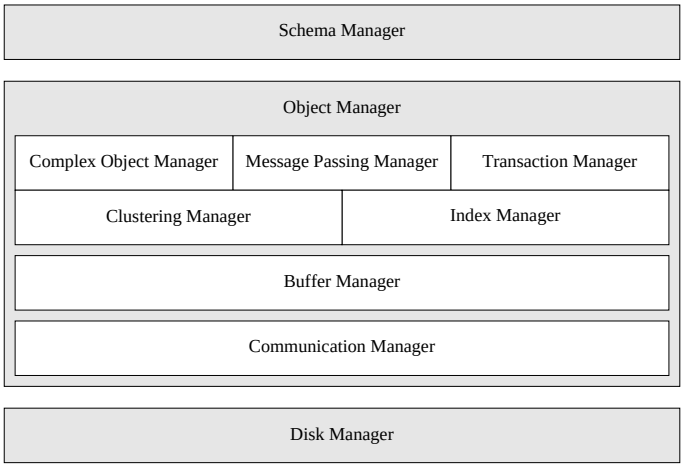


ABBILDUNG 3.30: Kernkomponenten von O_2

2.2. Schema Manager

Die oberste Schicht wird vom Schema Manager gebildet. Dieser ist für die Erstellung, Änderung und Löschung von Klassen, ihren Methoden und globalen Namen verantwortlich. Außerdem ist er für Anfragen auf Objekte zuständig. Werden Klassen von anderen Klassen abgeleitet, so sichert der Schema Manager die Semantik der Vererbung, das heißt, er stellt sicher, daß die abgeleiteten Klassen über die geerbten Daten und Methoden verfügen. Die Konsistenz des ganzen Schemas wird ebenfalls vom Schema Manager gesichert.

2.3. Object Manager

Der Object Manager stellt die im Zusammenhang mit dieser Arbeit interessanteste Komponente dar, da hier die Objekte zwischen Speicherformat, das heißt dem Format, welches O_2 den Programmen anbietet, und dem Externspeicherformat umgewandelt werden. Hierfür besteht der Object Manager aus einer äußeren, einer inneren Schicht und einer Kommunikationsschicht. Die äußere Schicht wird vom Complex Object Manager, dem Message Passing Manager, dem Transaction Manager, dem Cluster Manager und dem Index Manager gebildet. Die innere Schicht besteht dagegen aus dem Buffer Manager. Der Communication Manager bildet die Kommunikationsschicht.

Nachfolgend sollen der Complex Object Manager und der Cluster Manager näher beschrieben werden, da diese für die Speicherung komplexer Objekte von Interesse sind:

- ▷ *Complex Object Manager*

  Der Complex Object Manager kümmert sich unter anderem um die physische Darstellung von komplexen Objekten. Damit Objekte von anderen Objekten aus erreicht werden können, existiert das Prinzip der Objektidentität. Diese zu implementieren, ist Teil der Aufgaben des Complex Object Manager. In O_2 hat man sich für eine physische Kennung entschieden. Werden Objekte physisch gespeichert, so werden sie in vom WiSS gelieferte Sätze gespeichert. Jeder dieser Sätze (Records) besitzt einen Satzidentifikator (Record Identifier – RID), welcher die physische Position bestimmt. Diese RID wird als Objektidentifikator (OID) genutzt, das heißt zur eindeutigen Identifizierung des Objekts. Ein Problem tritt auf, wenn Objekte ihre physische Position wechseln. Dadurch würde sich neben der RID ebenfalls die OID ändern. Damit das

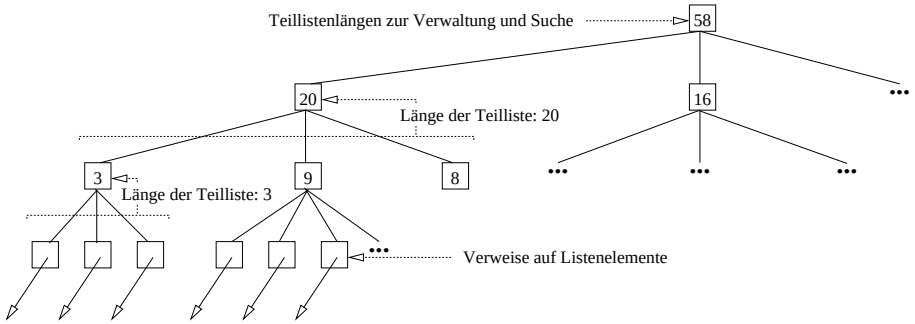


ABBILDUNG 3.31: Speicherung von Listen als geordnete Bäume

Objekt weiterhin referenziert werden kann, müßte man alle Referenzen auf das Objekt ändern. Da dies einen zu hohen Aufwand darstellt, wird die OID nicht geändert. Sie zeigt weiterhin auf den alten Platz des Objekts. An dieser Stelle steht nun jedoch eine Information über die neue RID, so daß das Objekt weiterhin erreichbar ist. Man spricht daher vom Stellvertreterprinzip.

Ein Objekt besteht jedoch nicht nur aus seinem Identifikator. Die Daten eines Objekts werden in einem oder mehreren Sätzen gespeichert. Die Struktur eines solchen Satzes wird von der Art der Daten bestimmt. Das O_2 Modell unterscheidet zwar zwischen Objekten und Werten, der Complex Objekt Manager vergibt dennoch strukturierten Werten einen Identifikator und speichert beziehungsweise lädt diese wie Objekte. Dies hat den Effekt, daß strukturierte Werte ausgelagert werden und nur ihre RIDs gespeichert werden.

Dies betrifft also auch die folgenden Typen. Hat ein Objekt ein Attribut vom Typ Tupel, Liste oder Menge, so wird im Objekt nur die RID für den Satz mit dem komplexen Wert gespeichert.

◇ *Tupel*

Ein Tupel wird physisch durch einen Satz repräsentiert. Wächst dieser über die Seitengröße hinaus, so wird eine andere Speicherungsform gewählt, das von WiSS unterstützte Long-Data-Item-Format. Ein Long Data Item (LDI) ist ein langer Satz, der aus mehreren normalen Sätzen zusammengesetzt ist, die auf verschiedenen Seiten plaziert werden. Einer dieser Sätze wird als Directory Record bezeichnet und enthält Zeiger (RIDs) auf alle zugehörigen Sätze. Die OID ist dabei die RID des Directory Records. Die LDIs entsprechen somit dem Konzept der seitenübergreifenden Sätzen.

◇ *Listen*

Listen werden als geordnete Bäume gespeichert. Ein solcher geordneter Baum ist wie ein B*-Baum aufgebaut, allerdings werden in den Knoten Längen von Teillisten gespeichert (ABBILDUNG 3.31), die zum auffinden von Teillisten und Listenelementen verwendet werden können. Diese Anzahlen müssen bei Änderungen der Liste ebenfalls geändert werden. Sind die Elemente der Liste Objekte, so werden nicht die Objekte selbst sondern nur ihre OIDs gespeichert. Falls es sich jedoch um einfache Daten handelt, werden diese direkt in den Sätzen der Liste gespeichert.

◇ *Mengen*

In einer Menge von Objekten werden nicht die Objekte selbst gespeichert, sondern nur deren OIDs. Außerdem stellt diese Menge selbst wieder ein Objekt dar, wodurch es eine OID besitzt. Über die physische Plazierung der Elementsätze wird mittels Clusterung entschieden. Dies ist jedoch Aufgabe des Cluster Managers. Da WiSS die Erstellung von Indexen unterstützt, ist es möglich, auf die Elemente einen Index zu setzen.

▷ *Cluster Manager*

In O_2 kann der Datenbankadministrator (DBA) die physische Speicherung beeinflussen. Dies geschieht, indem die von ihm angegebenen Kontrollinformationen an den Cluster Manager weitergegeben werden. Solche Kontrollinformationen sind in O_2 Plazierungsbäume (Placement Trees) und beschreiben, wie komplexe Objekte und ihre Komponenten gespeichert werden sollen. Ein Plazierungsbaum einer Klasse ist ein Teilbaum des Strukturbaums dieser Klasse. Dabei kann für jede Klasse die Clusterungsstrategie durch einen Plazierungsbaum angegeben werden, jedoch nicht mehr als einen für eine Klasse. Hier ein Beispiel eines Kommandos zur Erstellung eines solchen Baums und die dafür notwendigen Klassendefinitionen:

```
add class City
    type tuple(Name: string, Map: bitmap, Hotels: set(Hotel))

add class Monument
    type tuple(Name: string, MAddress: Address)

add class Address
    type tuple(Street: string, ACity: City)

add class Hotel
    type tuple(Name: string, HAddress: Address)

add cluster tree for class Monument
    desc tuple( Address: tuple(ACity: City [
                tuple(Hotels: set(Hotel))
            ]))
```

Wird nun ein Objekt der Klasse Monument gespeichert, so wird das zu ihm gehörende Adreßobjekt nahe dem Monumentobjekt gespeichert. Für das Adreßobjekt gilt, daß das zugehörige City-Objekt ebenfalls nahe dem Addreßobjekt gespeichert wird. Und schließlich gilt für das City-Objekt, daß alle zugehörigen Hotelobjekte physisch nahe dem City-Objekt gespeichert werden. Der Clusterungsbaum einer Klasse gibt also an, welche Teilobjekte physisch dicht an einem Objekt der Klasse gespeichert werden sollen. Für genauere Angaben über die Clusterung in O_2 sei auf [BDH92] verwiesen.

Die in O_2 genutzten Konzepte zur Speicherung komplexer Objekte sind in folgender Liste zusammengefaßt:

- ▷ Auslagerung von Kollektionselementen, falls diese strukturiert sind
- ▷ Keine Auslagerung, falls die Elemente atomar sind
- ▷ Zeigerfelder (Pointer Arrays) bei Mengen
- ▷ Indexstruktur bei Listen
- ▷ Teilobjekte in Sekundärsätze

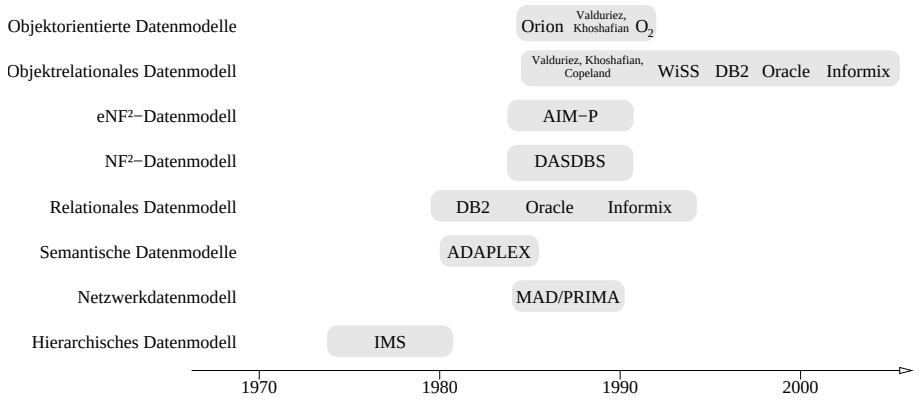


ABBILDUNG 3.32: Auf Speicherkonzepte hin untersuchte Auswahl von Ansätzen, Prototypten und Systemen

- ▷ Clusterung
- ▷ Seitenübergreifende Sätze
- ▷ Nutzung von Indexen

3.1.14 Vergleich und Zusammenfassung der Speichertechniken

Die in den vorigen Abschnitten vorgestellten Implementierungstechniken komplexer Objekte in Datenbanksystemen griffen auf unterschiedliche aber auch auf gemeinsame Methoden zurück. In diesem Abschnitt sollen die gemachten Vorschläge anhand dieser Methoden verglichen werden. Außerdem wird ein Überblick gegeben, aus welchen Datenbanktypen und -modellen diese Implementierungstechniken stammen. Daraus kann man ersehen, daß die durch die objektorientierte Programmierung gemachten Forderungen durchaus schon bei älteren Systemen berücksichtigt wurden. ABBILDUNG 3.32 führt alle hier auf Speichertechniken für komplexe Objekte hin untersuchten Ansätze, Prototypen und Systeme auf. Dabei wurde versucht, die Techniken von möglichst vielen unterschiedlichen Modellen darzustellen, um deren Techniken für Vorschläge zur Objektrepräsentation in einem objektrelationalen DBMS zu nutzen. In den TABELLEN 3.1 und 3.2 werden die untersuchten Ansätze und die darin gefundenen Techniken zusammengefaßt und eingeordnet. Aus dieser Zusammenstellung kristllisiert sich folgender Kanon von grundlegenden Konzepten und Techniken zur Speicherung komplexer Objekte heraus:

- ▷ *Clusterung*

  Offensichtlich ist die Clusterung ein grundlegendes Prinzip, da es in fast allen Vorschlägen eingesetzt wird. Dabei wird es jedoch auf unterschiedliche Weise umgesetzt:

  - ◇ *Objektbezogene Clusterung*

    Bei dieser Strategie wird jedes Objekt für sich als Ganzes geclustert gespeichert, das heißt, alle Komponenten des komplexen Objekts werden physisch dicht gespeichert. Dies ist besonders günstig, falls häufig auf komplexe Objekte als Ganzes zugegriffen wird.

    Eine solche Clusterungsstrategie wird in dem in Abschnitt 3.1.9 beschriebenen Vorschlag verfolgt. Dort werden Objekte, die kleiner als eine Seite sind, komplett in eine Seite gespeichert. Clusterung ist damit selbstverständlich. Objekte, die

TABELLE 3.1: Vergleich der Speichertechniken der betrachteten Vorschläge

Betrachtete Vorschläge		**Techniken**							
		Kollektionen							
		Auslagerung	**Inline Array**	**Pointer Array**	**Verkettete Liste**		**sequentielle Reihung**	**logische Rückwärts-referenzen**	**index-organisiert**
					einfach	**doppelt**			
WiSS+ (3.1.4)		$\oplus$	$\ominus$	$\ominus$	$\oplus$	$\ominus$	$\ominus$	$\ominus$	$\ominus$
[KV87] (3.1.11)		$\oplus$	$\ominus$	$\oplus$	$\ominus$	$\ominus$	$\ominus$	$\ominus$	$\ominus$
[VKC86] (3.1.3)	DS[1]	$\ominus$	$\oplus$[2]	$\ominus$	$\ominus$	$\ominus$	$\ominus$	$\ominus$	$\ominus$
	NS[1]	$\oplus$	$\ominus$	$\ominus$	$\ominus$	$\ominus$	$\ominus$	$\oplus$	$\ominus$
DASDBS (3.1.9)		$\oplus$	$\ominus$	$\oplus$	$\ominus$	$\ominus$	$\ominus$	$\ominus$	$\ominus$
AIM-P (3.1.8)		$\oplus$	$\ominus$	$\oplus$	$\ominus$	$\ominus$	$\ominus$	$\ominus$	$\ominus$
ORION (3.1.12)		$\oplus$[3]	$\oplus$	$\oplus$[4]	$\ominus$	$\ominus$	$\ominus$	$\ominus$	$\ominus$
PRIMA (3.1.2)		$\oplus$[3]	$\oplus$	$\oplus$[4]	$\ominus$	$\ominus$	$\ominus$	$\ominus$	$\ominus$
ADAPLEX (3.1.10)		$\oplus$	$\oplus$	$\oplus$	$\ominus$	$\ominus$	$\ominus$	$\ominus$	$\ominus$
Oracle (3.1.7)		$\oplus$	$\ominus$	$\ominus$	$\ominus$	$\ominus$	$\ominus$	$\oplus$	$\ominus$
DB2 (3.1.5)		$\ominus$[5]	$\ominus$	$\ominus$	$\ominus$	$\ominus$	$\ominus$	$\ominus$	$\ominus$
IMS (3.1.1)		$\oplus$[6]	$\oplus$[6]	$\ominus$	$\oplus$	$\oplus$	$\oplus$	$\ominus$	$\ominus$
Informix (3.1.6)		$\ominus$	$\oplus$	$\ominus$	$\ominus$	$\ominus$	$\ominus$	$\ominus$	$\ominus$
O_2 (3.1.13)		$\oplus$[3]	$\ominus$	$\oplus$	$\ominus$	$\ominus$	$\ominus$	$\ominus$	$\oplus$

[1] DS = Direct Storage, NS = Normalized Storage
[2] Auch andere Reihenfolgen möglich
[3] Keine Auslagerung, falls Elemente atomar sind
[4] Falls Kollektionen ausgelagert werden
[5] Es werden keine Kollektionen unterstützt
[6] Abhängig vom gewählten DB-Typ

TABELLE 3.2: Vergleich der Speichertechniken der betrachteten Vorschläge (Fortsetzung)

Betrachtete Vorschläge		**Techniken**				
		Clusterung		**seitenüber-greifende Sätze**	**Indexe**	**Trennung von Daten & Strukturinformationen**
		objektbezogen	**objektübergreifend**			
WiSS+ (3.1.4)		⊖	⊕	⊖	⊕	⊖
[KV87] (3.1.11)		⊕	⊖	⊕	⊕	⊖
[VKC86] (3.1.3)	DS[1]	⊕	⊖	⊖	⊕	⊖
	NS[1]	⊖	⊕	⊖	⊕	⊖
DASDBS (3.1.9)		⊕	⊖	⊕	⊕	⊕
AIM-P (3.1.8)		⊕	⊖	k.A.	⊕	⊕
ORION (3.1.12)		⊕	⊖	⊖	⊕	⊖
PRIMA (3.1.2)		⊕	⊕	⊕	k.A.	⊖
ADAPLEX (3.1.10)		⊕	⊖	⊖	⊕	⊕
Oracle (3.1.7)		⊕	⊕	⊕	⊕	⊖
DB2 (3.1.5)		⊕	⊖	⊖	⊕	⊖
IMS (3.1.1)		⊕	⊕[2]	⊖	⊕[2]	⊖
Informix (3.1.6)		⊕	⊖	⊕	⊕	k.A.
O_2 (3.1.13)		⊕	⊕	⊕	⊕	⊖

[1] DS = Direct Storage / NS = Normalized Storage

[2] Abhängig vom gewählten DB-Typ

größer als eine Seite sind, werden in einer möglichst geringen Anzahl zusammenhängender Seiten geclustert abgespeichert.

Bei AIM-P (Abschnitt 3.1.8) wird ähnlich geclustert. Neue Daten werden in bereits vom Objekt belegten Seiten abgelegt und möglichst dicht beieinander gespeichert.

ORION (Abschnitt 3.1.12) ist dagegen nicht so streng. Dort werden Instanzen der gleichen Klasse im selben Segment gespeichert.

Werden komplexe Objekte mit Kollektionsattributen in Oracle gespeichert, so kann dort ein gemeinsames Clusterkriterium für mehrere Tabellen definiert werden. Damit kann zum Beispiel ein Objekt einer Objekttabelle mit seinen zugehörigen Kollektionselementen, die in einer Nested Table gespeichert sind, geclustert werden. Es ist aber auch möglich, die Daten der Nested Table nach einem eigenen Attribut zu clustern. Wird dazu die logische Referenz auf das übergeordnete Objekt genutzt, so entsteht ebenfalls eine objektbezogene Clusterung.

DB2 clustert stets auf Objektebene, da alle Objekte kompakt gespeichert werden.

In IMS ist die Clusterung vom Datenbanktyp abhängig. Werden in HSAM-Datenbanken Objekte in hintereinanderliegenden Seiten gespeichert, so sind sie in anderen Typen nicht mehr hintereinanderliegend. Es ist also nur ein Clusterkriterium möglich.

Werden komplexe Objekte in Informix in einer Tabelle gespeichert, so kann mittels eines Clusterindexes die Clusterung erreicht werden. Dadurch werden die Objekte mit gemeinsamen Clusterungswerten hintereinander gespeichert.

Objektbezogen clustert auch ADAPLEX. Dort ist es möglich, alle Primärsätze eines Objekts physisch nahe beieinander zu speichern.

Der Vorschlag aus Abschnitt 3.1.11 clustert ebenfalls auf Objektbasis, da dort möglichst alle Sätze der Teilobjekte in der Seite des Hauptobjekts gespeichert werden.

Da Kollektionen nicht ausgelagert werden und sonst auch keine Aufteilung in Teilobjekte vorgenommen wird, clustert das Direct Storage Model nach Valduriez, Khoshafian und Copeland aus Abschnitt 3.1.3 ebenfalls objektbezogen.

◇ *Objektübergreifende Clusterung*

In Abschnitt 3.1.3 wird neben dem Direct Storage Model auch das Normalized Storage Model vorgestellt. Dieses clustert die Daten nur objektübergreifend, da dort Komponenten in unterschiedlichen Segmenten gelagert werden.

Eine ähnliche Clusterungsstrategie wird auch bei WiSS+ in Abschnitt 3.1.4 verfolgt. Dort werden die Objekte nicht mit ihren Kollektionen geclustert, sondern getrennt von ihnen gespeichert. Dafür werden die Kollektionselemente zusammen geclustert. Außerdem werden alle Wurzelobjekte (Wurzel des Hierarchiebaums) im gleichen Segment gespeichert. Für die Komponenten gilt das gleiche, das heißt, alle Komponentensätze werden gemeinsam in ein separates Segment gespeichert. Innerhalb der Segmente ist dann eine Clusterung möglich. So können beispielsweise alle Wurzelsätze mit dem gleichen Attributwert physisch eng gespeichert werden. Auch für Komponentensätze ist das möglich.

◇ *Kombinierte Clusterung*

Bei PRIMA (Abschnitt 3.1.2) ist die Clusterung noch flexibler. Dort können gan-

ze Objekte zusammen geclustert werden, ebenso aber auch Komponenten von unterschiedlichen Objekten (einzelne Atome).

Ebenfalls sehr flexibel in der Clusterungsstrategie ist O_2. Dort können Objekte als Ganzes gespeichert werden, indem für die entsprechende Klasse ein Plazierungsbaum angegeben wird, der alle Komponentenobjekte umfaßt. Werden dagegen nur Plazierungsbäume der Komponentenobjekte angegeben, so werden diese geclustert gespeichert.

▷ *Freiheitsgrade bei der Speicherung von Kollektionen*

Für Kollektionen stehen ebenfalls mehrere Optionen zur Verfügung. Die erste Unterscheidung ist die nach der Möglichkeit der Teilsatzauslagerung. In TABELLE 3.1 ist zu sehen, daß nicht bei jedem Vorschlag aus der Literatur die Elemente einer Kollektion ausgelagert werden können.

In ORION und in PRIMA betrifft dies lediglich die Kollektionen, deren Elemente von atomarem Typ sind. Andernfalls werden diese ausgelagert.

Eine völlig integrierte Speicherung von mengenwertigen Attributen wird dagegen im direkten Speichermodell aus Abschnitt 3.1.3 verfolgt. Eine solche integrierte Implementierung einer Kollektion wird auch als Inline Array bezeichnet und kann zu Platzproblemen führen, falls keine seitenübergreifenden Sätze unterstützt werden.

Erfolgt jedoch eine Auslagerung der Kollektionselemente, so werden hierfür wiederum mehrere Alternativen angeboten. Kollektionen können als Feld von Zeigern auf Kollektionselemente implementiert werden (Pointer Array), oder aber auch als verkettete Liste der Elemente (Linked List). Diese Techniken werden auch in der Literatur genutzt. Aber auch andere Varianten, wie zum Beispiel sequentielle Hintereinanderreihung der Kollektionselemente oder logische Rückwärtsreferenzen, sind anzutreffen.

Die Varianten können in folgenden Gruppen zusammengefaßt werden:

◇ *Zeigerfeld*

Bei dieser Implementierungsart der mengenwertigen Attribute werden im Satz des Objekts Zeiger auf jeweils ein Element der Kollektion gespeichert. Dadurch verringert sich die Größe des (Primär-)Satzes des Objekts unter Umständen enorm, kann bei extrem großen Mengen jedoch immer noch zu Problemen führen, falls die Seitengröße erreicht wird. Diese Art der Auslagerung wird von AIM-P (Abschnitt 3.1.8) genutzt sowie von dem in Abschnitt 3.1.11 beschriebenen Ansatz und dem auf DASDBS aufbauenden System aus Abschnitt 3.1.9. Aber auch ORION, PRIMA, O_2 und ADAPLEX verwenden Zeigerfelder für ihre Kollektionen. O_2 nutzt diese jedoch nur für Attribute vom Typ Set. Der ebenfalls unterstützte Typ List wird anders implementiert.

◇ *Verkettete Liste*

Werden Kollektionen nach diesem Verfahren umgesetzt, so wird im Primärsatz des Objekts für ein mengenwertiges Attribut lediglich ein Zeiger auf einen Satz mit einem Element der Menge gespeichert. Die restlichen Elemente werden nach dem Listenprinzip verkettet, das heißt, ein Satz eines Elements zeigt auf den Satz des nächsten Elements. Dadurch ergibt sich die Möglichkeit, mengenwertige Attribute beliebiger Größe anzulegen. Falls lediglich auf bestimmte Elemente der Menge zugegriffen werden soll, so müssen jedoch alle Vorgängerelemente ‚abgelaufen' werden. Im Fall einer geclusterten Speicherung wäre dies nicht so schwerwiegend,

da in diesem Fall die Elemente physisch dicht liegen (möglichst in der selben Seite) und diese Seite(n) als Ganzes in den Speicher geladen werden kann.

Die in Abschnitt 3.1.4 dargestellte Implementierung komplexer Objekte verwendet verkettete Listen für mengenwertige Attribute. Diese kommen dort jedoch nur in einfach verketteter Form vor. Doppelt verkettete Listen verwendet IMS für die Implementierung. Es sind aber auch einfach verkettete Listen möglich. Außerdem obliegt es dem Nutzer, ob ein zusätzlicher Zeiger auf das letzte Element der Liste gespeichert wird.

◇ *Sequentielle Reihung*

Werden die Kollektionselemente in Sekundärsätze ausgelagert und diese physisch hintereinander plaziert, so wird von einer sequentiellen Reihung gesprochen. Dabei werden die Elemente nicht mittels Zeigern verbunden. Allein ihre physische Reihenfolge bewirkt ihren Zusammenhalt und ihre Wiederauffindbarkeit. Diese Variante ist nur für Datenbanken gedacht, deren Elemente (Objekte) nicht oder sehr selten geändert werden. Beim HSAM-Datenbanktyp von IMS ist diese Methode zu finden.

◇ *Elementauslagerung mit logischer Rückwärtsreferenzierung*

In Oracle werden Kollektionen vom Typ Nested Table (entsprechen den Multimengen) in separate Tabellen ausgelagert. Dabei wird die Verbindung zwischen dem Objekt und seinen Mengenelementen mittels logischer Werte erreicht, die mit Fremdschlüsseln vergleichbar sind. ABBILDUNG 3.15 aus Abschnitt 3.1.7 verdeutlicht dies. Auch das normalisierte Speichermodell aus Abschnitt 3.1.3 arbeitet nach diesem Prinzip.

◇ *Indexorganisierte Speicherung*

Hierbei werden die Kollektionselemente nicht als Liste organisiert, sondern mittels eines Baumes, dessen Schlüsselwerte die Kollektionselemente selbst oder spezielle Attribute dieser sind. O_2 verwendet eine solche Umsetzung für Attribute vom Typ List. Dabei fungieren eindeutige Kennungen der Elemente als Schlüssel. Zur Indexierung wird eine B^*-Baumvariante verwendet.

▷ *Seitenübergreifende Sätze (Spanned Records)*

Werden Sätze in Seiten gespeichert, so ist gewöhnlich deren Größe durch die Größe der Seite beschränkt. Dadurch ergibt sich der Nachteil, daß Objekte, wenn sie nur aus einem Satz bestehen, nicht größer als eine Seite werden können. Dies kann besonders bei nicht ausgelagerten Kollektionen einschränkend wirken. Aber auch Zeigerfelder (Pointer Arrays) sind davon betroffen. Um diese Einschränkung zu umgehen, verfügt die Hälfte der vorgestellten Systeme über die Fähigkeit, Sätze auf mehr als eine Seite zu verteilen (Spanned Records).

▷ *Trennung von Daten und Strukturinformationen*

Nach diesem Prinzip liegen die eigentlichen Daten und die Informationen über die (hierarchische) Struktur (Zeiger, ...) nicht in den selben physischen Sätzen. Hiervon kann man bei Anfragen auf Teile von Objekten profitieren, falls diese Strukturinformationen ihrerseits geclustert gespeichert sind. Diese Eigenschaft besitzen jedoch nur die

wenigsten der vorgestellten Systeme. Gründe hierfür zu finden, wäre an dieser Stelle jedoch Spekulation.

- ▷ *Nutzung von Indexen*
  Soll auf die gespeicherten Daten zugegriffen werden, so gelingt dies immer mittels eines einfachen Table Scans. Da dies jedoch bei großen Datenmengen sehr kostspielig sein kann, werden Indexe zur Beschleunigung eingeführt. Hierfür werden ein oder mehr Attribute der Objekte als Indexschlüssel angegeben, über den dann eine beschleunigte Suche erfolgt. Solche Indexe sind meist als B*-Bäume oder Hashstrukturen implementiert und werden in Kapitel 4 näher behandelt.

Zusammenfassend muß festgestellt werden, daß sich in den betrachteten vielfältigen Vorschlägen und Systemen zur Speicherung komplexer Objekte eine Reihe wiederkehrender und sich ergänzender Grundkonzepte finden.

So ist die Möglichkeit, die Daten geclustert zu speichern, in allen Ansätzen vorhanden. Ebenso werden Indexe für eine beschleunigte Anfrageabarbeitung unterstützt. Die Einschränkung der Satzgröße auf die Seitengröße wird dagegen nicht von jedem System aufgehoben. Die Hälfte der dargestellten Systeme unterstützte dieses Merkmal. Eine größere Übereinkunft ist bei der Implementierung von Kollektionen zu beobachten. Zwar werden diese auf verschiedene Weise umgesetzt, eine Auslagerung ist jedoch im Großteil aller Vorschläge möglich. Hat man sich für eine Auslagerung entschieden, so stehen in der Regel weitere Wahlmöglichkeiten zur Verfügung: am häufigsten wird die Variante des Referenzenfeldes (Pointer Arrays) gewählt.

Aus den untersuchten Ansätzen heraus wurde die in diesem Abschnitt präsentierte Systematisierung von Konzepten zur Speicherung komplexer Objekte zusammengefaßt. Im nächsten Abschnitt wird auf dieser Grundlage eine vollständige Konzeption zu Techniken der Speicherung komplexer Obkete in ORDBMS entworfen.

3.2 Speicherstrukturen zur Nutzung in ORDBMS

Dieser Abschnitt stellt, basierend auf der im letzten Abschnitt durchgeführten Untersuchung der Literatur, ein in sich schlüssiges Konzept der Speicherung komplexer Objekte, wie sie in ORDBMS genutzt werden sollten, vor und diskutiert die darin enthaltenen Freiheitsgrade und Speicherungsalternativen. Dabei wird auf Vor- und Nachteile der verschiedenen Optionen eingegangen. Darauf aufbauend wird in Kapitel 5 der auf die physische Speicherung von komplexen Objekten bezogene Teil der Spezifikationssprache PRDL präsentiert, der die Freiheitsgrade umsetzt.

Zu Beginn der eigentlichen Ausführungen ist noch eine grundlegende Begriffsdefinition notwendig.

3.2.1 Aufteilung von Objekten auf mehrere physische Sätze

Ebenso wie in relationalen Systemen erfolgt die Ablage der Daten stets innerhalb interner Sätze. Wie bereits erwähnt, sind dabei einem logischen Tupel (fast) beliebig viele interne beziehungsweise physische Sätze zugeordnet. Da pro logischem Objekt mindestens ein physischer Satz existiert, bietet sich die Unterteilung der internen Sätze in zwei Klassen an:

▷ *Primärsätze*
Bei einem Primärsatz handelt es sich um genau den einen Satz, der beim Anlegen eines Tabellentupels auf jeden Fall, also unabhängig von irgendwelchen Speicheroptionen, existiert. Es gibt pro Relation also genauso viele Primärsätze wie logische Tupel.

▷ *Sekundärsätze*
Alle anderen internen Sätze, die in irgendeiner Weise Daten der gespeicherten Objekte beinhalten, werden Sekundärsätze genannt.

Da jede Variante der Speicherung von Datenbankobjekten prinzipiell auch ohne zusätzliche Zugriffspfade (Indexe) auskommen muß, ist es erforderlich, die Sekundärsätze mit ihrem zugehörigen Primärsatz zu verknüpfen. Wie sich später zeigen wird, führt dies letzten Endes zu einer Baumstruktur, in der der Primärsatz als Wurzel und die Sekundärsätze als Knoten und Blätter fungieren.

3.2.2 Speicherort interner Sätze und Clusterstrategien

Die erste wichtige Speicheroption betrifft die Frage, wo und auf welche Weise interne Sätze abgespeichert werden sollen. Bei der Aufteilung eines logischen Objekts auf mehrere Sätze entsteht pro Relation eine bestimmte Anzahl physischer Satztypen. Für jeden Typ kann unabhängig voneinander ein gewünschter Speicherort angegeben werden. Zu beachten ist, daß ein Satztyp immer objektübergreifend zu verstehen ist. Es ist also nicht möglich, pro Objekt unterschiedliche Speicherangaben zu treffen.

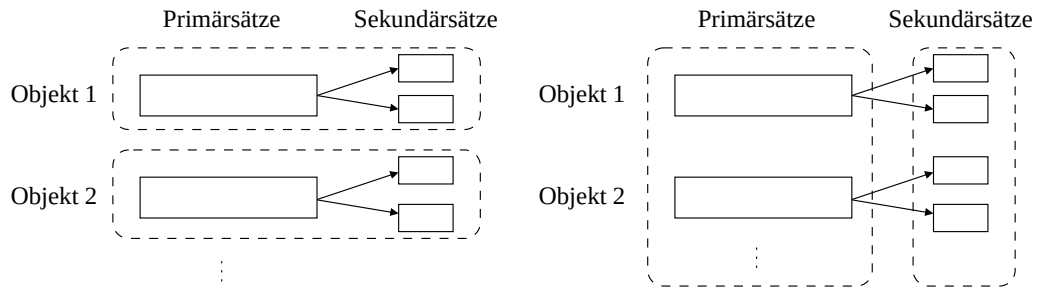
Bezogen auf die Frage, welche Sätze einer Relation dicht gespeichert werden sollen, kann man (nach [Keß95]) zwei Aspekte der Clusterung unterscheiden:

▷ *Objektbezogene Clusterung*
Hierbei steht die Forderung im Mittelpunkt, bestimmte interne Sätze, die zu einem logischen Objekt gehören, möglichst dicht zu speichern. Zusätzlich können in die Clusterung auch Sätze von Objekten miteinbezogen werden, die von dem betreffenden Objekt referenziert werden (klassenübergreifende objektbezogene Clusterung).

▷ *Objektübergreifende Clusterung*
Dieser Aspekt der Clusterung zielt darauf ab, interne Sätze mehrerer Objekte einer Klasse dicht zu speichern. Es existieren zwei Abstufungen bezüglich der konkreten Clusterstrategie:
  ◇ Das einzige Clusterkriterium ist die gemeinsame Klassenzugehörigkeit.
  ◇ Es besteht eine zusätzliche Verbindung zwischen den Sätzen innerhalb einer Seite bzw. eines Clusters. Dies kann z.B. eine wertebasierte Ähnlichkeit bzw. Äquivalenz sein.

Die Anwendung dieser zwei Aspekte ist keinesfalls ausschließlich; vielmehr erfolgt sie in der Regel kombiniert. Abbildung 3.33(a) und Abbildung 3.33(b) skizzieren dieses Prinzip.

Bei der Zuweisung eines Speicherorts für einen internen Satz innerhalb eines vorgegebenen Segments (beziehungsweise Tablespaces) können fünf verschiedene Möglichkeiten unterschieden werden, wobei die ersten drei bereits in RDBMS verbreitet sind:



(a) Objektbezogene Cluster (b) Objektübergreifende Cluster

ABBILDUNG 3.33: Clusterungsalternativen

▷ *Ungeordnet*

In der einfachsten Variante wird für einen Typ von internen Sätzen als Speicherort ein vorab definierter Tablespace angegeben. Die Tupel werden ungeordet auf Seiten des Tablesspace abgelegt (Prinzip der Halde), wobei eine Seite immer genau einem Satztyp zugeordnet wird. Diese Art der objektübergreifenden Speicherung ist einfach zu verwalten und eignet sich insbesondere gut für Scans über die Gesamtmenge der Sätze eines Typs.

▷ *Geordnet*

Sätze werden als Blätter einer Baumstruktur in einem logischen Segment eines bestimmten Tablespace gespeichert. Diese Speichermethode eignet sich besonders für Satztypen, auf die entweder oft in einer bestimmten logischen Reihenfolge zugegriffen wird oder auf die der Zugriff vorwiegend wertebasiert und über die ordnungsbestimmenden Attribute erfolgt. Auch bei dieser Speicherart handelt es sich um objektübergreifende Clusterung.

▷ *Wertebasierte Cluster*

Hier wird ein Satz einem vorab definierten Clustertyp zugewiesen. Kriterium für eine gemeinsame Speicherung sind die gleichen Werte bestimmter Attribute.

▷ *Referenzbasierte Cluster*

Diese Form der Speicherung ähnelt der der wertebasierten Clusterung, da auch hier der Wert eines bestimmten Attributs, in diesem Fall der eines Referenzattributs, über den Speicherort eines Satzes entscheidet. Ein Satz wird dicht bei dem Primärsatz gespeichert, dessen zugehöriges logisches Objekt aus ihm referenziert wird.

▷ *Kollektionsbezogene Cluster*

In den bisher genannten Möglichkeiten waren die Angaben zum Speicherort entweder rein objektbezogen (erster Fall) oder standen in direkter Beziehung zu Attributwerten der jeweiligen Sätze. Für Sekundärsätze ergibt sich eine weitere Variante, wobei sich die Speicherposition aus der Zugehörigkeit zu ein und demselben kollektionswertigen Attribut ableitet. Im Rahmen der Erläuterungen zu PRDL wird in Abschnitt 5.2 auf diese Form der Speicherung noch näher einzugehen sein.

3.2.3 Seitenübergreifende Sätze

Da interne Sätze innerhalb von Speicherseiten abgelegt werden, deren Größe sich an den Blöcken gängiger Externspeichermedien orientiert und demnach zwischen 4 und 32 KBytes liegt, ist die Länge eines internen Satzes bei einfachen Speichersystemen auf die Größe einer Seite des entsprechenden Segments beschränkt. Um auch umfangreichere Objekte speichern zu können, erweist es sich als notwendig, daß das Speichersystem dem Zugriffssystem Datencontainer anbietet, deren Größe nicht diesen Restriktionen ausgesetzt ist.

Möglich wird dies, indem aus einzelnen internen Sätzen Satzketten gebildet werden (Spanned Records). Das Speichersystem zerlegt die ihm vom Zugriffssystem übergebenen langen Datensätze in hinreichend kleine Portionen, die für sich genommen jeweils auf eine Speicherseite passen. Aus Sicht des Speichersystems werden die Teilsätze wie gewohnt behandelt, was bedeutet, daß die Ketten mittels Zeigern aus physischen Adressen realisiert werden. Dieser Vorgang verläuft aus Sicht des Zugriffssystems vollständig transparent; die Adresse eines langen Satzes entspricht der seines ersten Teilsatzes.

Neben der sequentiellen Verkettung der Datensatzteile sind natürlich auch andere Strukturen denkbar. So könnten die Datenportionen etwa auch über ein Verzeichnis am Satzanfang, das einen schnellen Direktzugriff auf einzelne Satzteile erlaubt, oder eine Baumstruktur, die Daten- und Längenänderungen auch in der Mitte sehr großer Datenmengen effizient gestaltet, referenziert werden. Allerdings ist dabei die Einschränkung zu beachten, daß das Speichersystem keine Kenntnisse über Inhalt und Struktur der in den seitenübergreifenden Sätzen abgelegten Informationen voraussetzen kann und sollte. Ausdrücklich soll betont werden, daß seitenübergreifende Sätze nicht für Datenzugriffe über Inhalt und Struktur der enthaltenen Daten geeignet sind. Soll der Datenzugriff durch solche weitergehenden Kenntnisse unterstützt werden, so sollten Aufteilung und Vernetzuung der Daten auf der Ebene des Zugriffsystems erfolgen, da dort die entsprechenden Informationen vorliegen, und nicht auf der Ebene des Speichersystems.

Voraussetzung für eine sinnvolle Aufteilung eines Satzes auf mehrere Teilsätze ist, daß seine Größe die einer Seite des jeweiligen Tablespaces übersteigt. Die Alternative dazu wäre, auch dann schon die Aufteilung auf Teilsätze zuzulassen, wenn ein sich vergrößernder Satz nicht mehr auf die aktuelle Seite paßt, obwohl er noch in eine andere, gegebenenfalls neue Seite passen würde. Diese Taktik verringert unter Umständen den Aufwand von Änderungen, arbeitet mit der Zeit aber gegen die eigentlich gewollte dichte Speicherung von logisch zusammengehörenden Daten. Insofern wird in dieser Arbeit für seitenübergreifende Sätze gefordert, daß diese stets größer als eine Speicherseite sein müssen.

Da das Speichersystem keine Kenntnisse über Inhalt und Struktur der Daten in den Sätzen voraussetzen darf, kann es neben dem „Zugriff als Ganzes“ höchstens noch den längen- und positionsbasierten Zugriff anbieten. Ohne spezielle Optimierungen im Zugriffsystem würde dieses außerdem seitenübergreifende Sätze genauso wie ‚normale‘ Sätze behandeln. Es ist also als Regelfall anzunehmen, daß ein Satzzugriff aus dem Zugriffssystem immer einen Zugriff auf den Datensatz als Ganzes impliziert und alle Teilsätze eines seitenübergreifenden Satzes zusammen geladen bzw. gespeichert werden.

Um die Anzahl notwendiger Ein-/Ausgabe Operationen bei einem solchen Zugriff möglichst gering zu halten, sollte das Speichersystem dafür sorgen, daß die Speicherung der Teilsätze auf hintereinanderliegenden Seiten erfolgt. Es kann - im Idealfall - demnach auch von einer zusammenhängenden Seitenkette oder Seitenfolge gesprochen werden, in der ein seitenübergreifender Satz gespeichert ist.

Ein Nachteil von seitenübergreifenden Sätzen liegt darin, daß aus der Sicht des Optimie-

rers u.U. nur sehr ungenaue Prognosen über reale Zugriffskosten getroffen werden können. Aus diesem Grund sollte es für jeden Satztyp möglich sein, verkettete Sätze explizit zuzulassen oder zu verbieten.

Die bisherigen Überlegungen gehen davon aus, daß der Mechanismus zur Speicherung großer seitenübergreifender Sätze innerhalb des Speichersystems realisiert wird. Denkbar wäre natürlich auch eine Variante, in der das Zugriffssystem diese Aufgabe selbst übernimmt. Als ein Hauptvorteil erwiese sich dabei, daß das Speichersystem nicht geändert werden müßte. Nachteile wären eine höhere Kommunikation auf der internen Satzschnittstelle sowie die eingeschränkte Kontrolle über dichte Speicherung der Teilsätze, da sich letzteres nur über Cluster steuern läßt.

Verkettete Sätze dürfen nicht mit LOBs verwechselt werden. Während LOBs sehr große Datenmengen aufnehmen, die dem Vielfachen einer Seitengröße entsprechen, sollten verkettete Sätze nur die eine Möglichkeit bieten, mit Satzgrößen nicht strikt an die Größe einer Seite gebunden zu sein. Verkettete Sätze sollten jedoch nur über wenige Seiten gehen.

3.2.4 Objektreferenzen

Hier geht es um die Frage, wie Objektreferenzen in ORDBMS implementiert werden können. Letztendlich handelt es sich bei solchen Referenzen um nichts anderes als um Referenzen auf die Primärsätze der logischen Objekte.

Daher bietet es sich zunächst einmal an, auf die selben Techniken zurückzugreifen, wie sie auch bei der Referenzierungen aus Indexen in RDBMS genutzt werden. Die Vor- und Nachteile der bereits in Abschnitt 2.5.2.2 diskutierten unterschiedlichen Ansätze der logischen und physischen Referenzierung lassen sich auch auf Objektreferenzen übertragen.

Allerdings genügt keine der beiden Techniken den in Abschnitt 2.1 vorgestellten Anforderungen für Objektreferenzen. Diese lassen sich ausschließlich mit einer erweiterten Form der logischen Referenzierung hinreichend erfüllen, in der die Objektidentifikatoren entsprechend den gewünschten Forderungen (räumliche und zeitliche Eindeutigkeit) angepaßt werden. Solche Erweiterungen wirken sich jedoch negativ auf die durchschnittlichen Kosten für die Transformation von der logischen in die physische Adresse aus, da es nicht mehr möglich ist, den Transformationsindex mittels schnellen Feldindexes zu implementieren. Gerade deswegen ist ein zusätzlicher Einsatz von PPP-Einträgen bei der erweiterten Form der logischen Referenzen unumgänglich.

Eine Besonderheit liegt in dem Fall vor, in dem die Zieltabelle einer Referenz indexorganisiert gespeichert ist. Indexe benutzen, wie bereist diskutiert, hier den Schlüssel des Indexeintrags als Referenz auf das gewünschte Objekt. Die Referenz basiert also auf veränderlichen Werten. Auf Referenzattribute kann dies in dieser Form nicht angewendet werden. Im Fall einer Änderung des Schlüssels müßten alle Objekte gefunden und verändert werden, die das entsprechende Objekt referenzieren, was (effizient) quasi unmöglich ist.

In den meisten Fällen wird die Wahl für Objektreferenzen daher auf erweiterte logische Referenzen fallen. Innerhalb dieser Ausführungen wird darauf verzichtet, näher auf die zur Handhabung dieser Form von Referenzen notwendigen Konzepte einzugehen. Die Ausführungen beschränken sich darauf, daß es bei Objektreferenzen prinzipell die Wahlmöglichkeit zwischen physischen Referenzen, logischen Referenzen und logischen Referenzen verbunden mit PPP-Einträgen gibt.

Ein Beispiel, wie mit PRDL die Art der Referenz festgelegt werden kann, findet sich im nächsten Abschnitt.

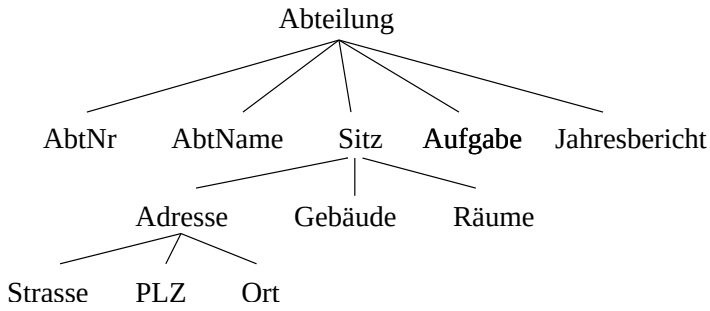


ABBILDUNG 3.34: Objekt mit verschachtelten, strukturierten Attributen

3.2.5 Objektfragmentierung und ausgelagerte Speicherung von Attributen

Die letzten zwei darzulegenden Optionen haben bei ihrer Anwendung direkte Auswirkungen auf die Anzahl der internen Sätze pro logischem Objekt. Abweichend von der Speicherung aller Attribute in einem gemeinsamen Satz, dem Primärsatz, können die zu speichernden Objekte durch die Nutzung sogenannter Sekundärsätze fragmentiert werden, in die gezielt einzelne Attribute oder Attributunterstrukturen ausgelagert werden können. Für Sekundärsätze können, wie weiter oben erläutert, separate Speicherorte angegeben werden, so daß verschiedene Bereiche eines logischen Objekts jeweils objektübergreifend dicht gespeichert werden.

Beispielweise ist es möglich, häufig gebrauchte Attribute zum schnellen Zugriff zusammen in einem Primärsatz zu speichern, während selten benötigte Attribute in Sekundärsätze ausgelagert werden.

Ein Objekt besteht aus logischer Sicht aus einer verschachtelten Hierachie strukturierter Attribute, die sich graphisch als Baum darstellen läßt. Das in ABBILDUNG 3.34 dargestellte Beispiel wird auch im folgenden zur Demonstration verschiedener Speichervarianten eingesetzt. Zum jetzigen Zeitpunkt seien die einzelnen Attribute des Objekt einfachen Typs und vorerst nicht kollektionswertig. Je nach Anforderung wird die Struktur später noch um weitere Attribute erweitert bzw. wird der Typ vorhandener Attribute verändert.

Neben der Auslagerung von vollständigen Attributen der obersten Ebene – im Beispiel also den Attributen AbtNr, AbtName, Sitz und Aufgabe – muß es auch möglich sein, Attribute bzw. Teilattribute tieferer Verschachtelungsebenen in Sekundärsätzen abzuspeichern. In diesem Zusammenhang bieten sich zwei unterschiedliche Herangehensweisen an, wobei erstere, die wahlfreie Attributauslagerung (auch attributorientierte Attributauslagerung), völlige Freiheit in der Verteilung von Teilattributen auf Sekundärsätzen ermöglicht, während die zweite, die strukturorientierte Attributauslagerung, die logische Position der Teilattribute innerhalb der Strukturierungshierarchie mitberücksichtigt.

3.2.5.1 Wahlfreie Attributauslagerung

Bei der wahlfreien Attributauslagerung können bei der Definition eines Sekundärsatzes und der in diesem Zusammenhang notwendigen Angabe der darin abzuspeichernden Daten beliebige Attribute und Teilattribute angegeben werden. Zu diesem Zweck ist die Nutzung von Pfadausdrücken sinnvoll, mit deren Hilfe die gewünschten Teildaten eines logischen Objekts adressiert werden können. In dieser Arbeit wird die allgemein gebräuchliche Punktnotation

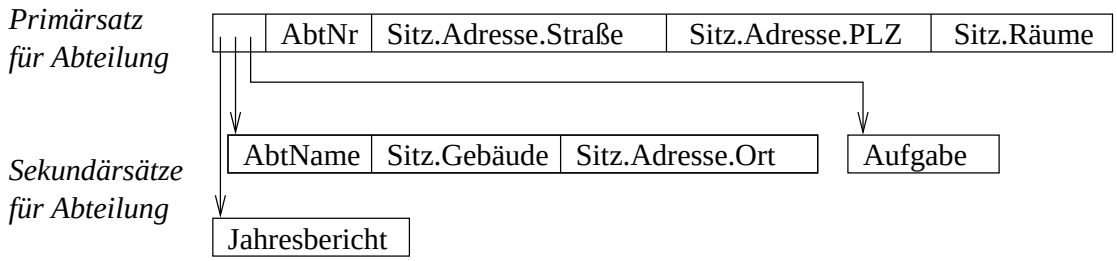


ABBILDUNG 3.35: Sekundärsatzdefinition durch wahlfreie Attributauslagerung

zur Beschreibung von Pfadausdrücken verwendet. Das Attribut Straße würde demnach mit Sitz.Adresse.Straße adressiert werden.

In dieser freien Variante der Attributauslagerung wird es möglich, auch Attribute zusammenzulegen, deren logische Zusammengehörigkeit nur in der gemeinsamen Objektzugehörigkeit besteht. Ein Beispiel wäre ein Sekundärsatz bestehend aus AbtName, Sitz.Gebäude und Sitz.Adresse.Ort, wie ihn ABBILDUNG 3.35 zeigt.

Um eine physische Verbindung zwischen Primär- und Sekundärsatz zu erhalten, existiert im Primärsatz ein Zeiger auf den Speicherort des Sekundärsatzes (physische Referenz). Man erhält physisch gesehen eine 2-stufige Baumstruktur mit dem Primärsatz als Wurzel und je nach Definition beliebig viele Sekundärsätze als oberste Knotenschicht, die auch gleichzeitig die Blätter des Baumes sind.

Neben der Referenz vom Primärsatz auf einen Sekundärsatz ist es auch denkbar, Zeiger von den Sekundärsätzen auf den Primärsatz einzurichten. Daraus ergibt sich der Vorteil, daß bei Selektionsabfragen auch Scans über Sekundärsätzen möglich sind. Im Beispiel könnte das eine Abfrage sein, die alle Abteilungen mit einem bestimmten Ortsnamen in Sitz.Adresse.Ort zurückgibt. Nachdem über einen Index oder Table Scan alle Sekundärsätze mit dem gesuchten Ort ermittelt worden sind, kann über die Rückwärtsreferenzen ein Zugriff auf den Primärsatz und, falls nötig, weitere Sekundärsätze erfolgen.

Ohne zusätzliche Angaben bei der Sekundärsatzdefinition werden alle Sekundärsätze direkt mit dem Primärsatz verknüpft, so daß es zu der eben bereits erwähnten einstufigen flachen Baumstruktur kommt. Bei sehr vielen Sekundärsätzen kann es allerdings sein, daß die entsprechend hohe Anzahl solcher Verknüpfungszeiger im Primärsatz eine nicht zu vernachlässigende Größe von Speicherplatz in Beschlag nimmt. Diese Tatsache spricht für die Einführung erweiterter Möglichkeiten bei der Verknüpfung der Sekundärsätze mit dem Primärsatz.

Durch eine explizite Benennung der Sekundärsätze wird es möglich gemacht, weitere Sekundärsätze als Kind nicht des Primärsatzes, sondern eines bereits vorhanden Sekundärsatzes anzulegen. Auf diese Weise können beliebig tiefe Baumstrukturen definiert werden (ABBILDUNG 3.36).

Optional können für tiefergelegene Sekundärsätze sowohl Rückwärtsreferenzen auf den Vater-Sekundärsatz als auch auf den Primärsatz angelegt werden. Der Hauptvorteil für diese Datenverteilung auf eine Baumstruktur kann bei bestimmten Selektionsanfragen über mehreren Attributen zum Tragen kommen. In der Weiterführung des Beispiels könnte das eine Anfrage nach Abteilungen in bestimmten Orten und mit bestimmten Aufgaben sein, bei deren Abarbeitung durch die hierarchische Struktur ein Zugriff auf die Gesamtmenge der Primärsätze umgangen werden kann. Nachdem über einen Table Scan alle Sekundärsätze herausgefiltert wurden, die den gesuchten Abteilungsnamen verbunden mit dem

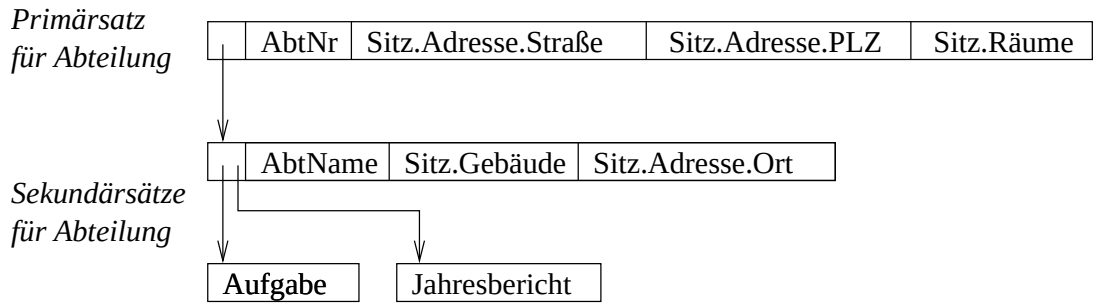


ABBILDUNG 3.36: Baumstruktur bei wahlfreier Attributauslagerung

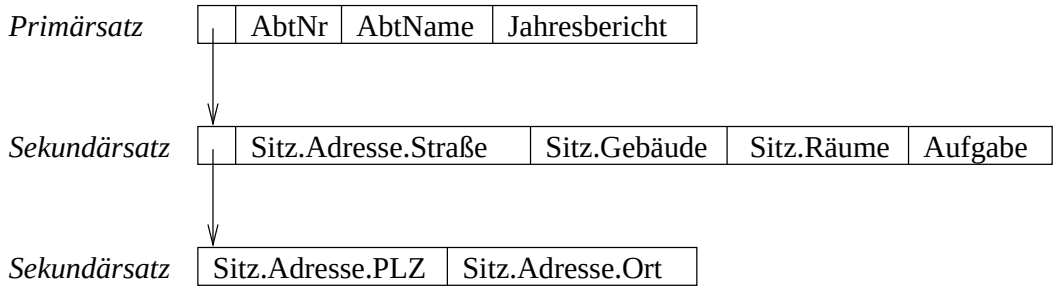


ABBILDUNG 3.37: Baumstruktur bei strukturorientierter Sekundärsatzdefinition

passenden Ort enthalten, kann über den direkten Zeiger zum Sekundärsatz mit dem Attribut Aufgabe ohne Zugriff auf den Primärsatz das dritte Selektionskriterium überprüft werden.

Bei der wahlfreien Attributauslagerung in Sekundärsätze lassen sich also Datenstrukturen definieren, deren Aufbau vollständig unabhängig von der logischen Struktur des Objekts ist.

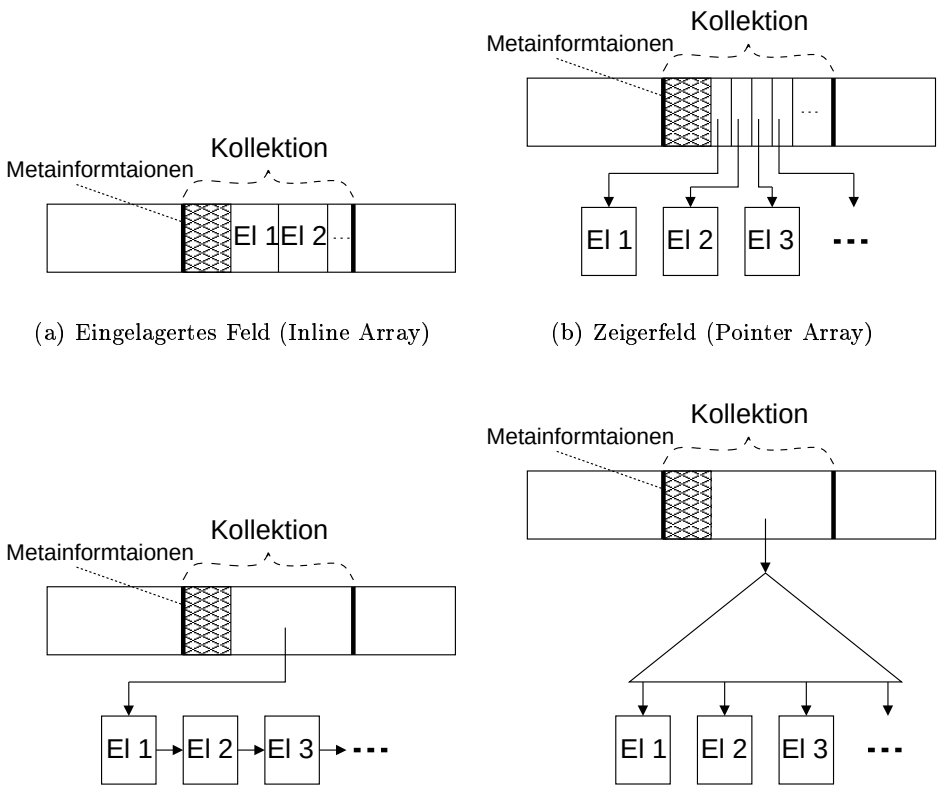
3.2.5.2 Strukturorientierte Attributauslagerung

Im Gegensatz zur wahlfreien Attributauslagerung steht der strukturorientierte Ansatz, der zwar ebenfalls die Definition von Baumstrukturen ermöglicht, diese aber zu einem gewissen Teil von dem logischen Aufbau des Objekt abhängig macht.

Im Rahmen des strukturorientierten Ansatzes ist das Erstellen von Sekundärsätzen nur unter Angabe eines strukturierten Attributs möglich. Letzteres kann auch das Objekt selbst sein, da die Attribute der obersten Ebene des logischen Objektaufbaus stets eine Struktur bilden.

Bei der Angabe der Attribute, die ausgelagert werden sollen, können ausschließlich Attribute oder Teilattribute der angegebenen Struktur genannt werden. Im Beispiel könnte man mit dieser Technik zunächst die Attribute Sitz und Aufgabe des Objekts und anschließend die Attribute PLZ und Ort der Struktur Adresse auslagern. Als Ergebnis würde man die hierarchische Baumstruktur aus ABBILDUNG 3.37 erhalten.

Zwar ist dieser Ansatz der Sekundärsatzdefinition nicht ganz so flexibel wie der wahlfreie, er dürfte jedoch für die meisten Fälle ausreichen und bringt dabei den Vorteil einer einfacheren Zuordnung von logischen Attributen zu deren Speichersatz mit sich. PRDL wird beide Ansätze unterstützen.



(a) Eingelagertes Feld (Inline Array)

(b) Zeigerfeld (Pointer Array)

(c) Verkettete Liste

(d) Indexiert/Indexorganisiert

ABBILDUNG 3.38: Physische Varianten von Kollektionsstrukturen

3.2.6 Speicherung kollektionswertiger Attribute

Eine der wichtigsten und prägnantesten Neuerungen beim Übergang von relationalen zu objektrelationalen Datenbankmanagementsystemen ist die Unterstützung kollektionswertiger Attribute [Luf05]. Entsprechend neuartig müssen auch die Überlegungen zur Speicherung von Kollektionen sein.

3.2.6.1 Grundlegende physische Speicherstrukturen

In der Literatur werden verschiedenste Speicherstrukturen für kollektionswertige Attribute vorgeschlagen (siehe Abschnitt 3.1). Diese sollen dementsprechend auch in die Vorschläge dieser Arbeit zur Speicherung komplexer Objekte in ORDBMS aufgenommen und eingearbeitet werden. Die folgenden wichtigsten Spielarten dieser Speicherstrukturen für kollektionswertige Attribute sollen in den nachfolgenden Abschnitten kurz vorgestellt werden. ABBILDUNG 3.38 skizziert die Anwendung der Varianten.

- ▷ *Inline Array*
  Unter der Bezeichnung Inline Array sollen alle Speichervarianten zusammengefaßt werden, bei denen die Elemente einer Kollektion zusammen innerhalb eines internen Satzes gespeichert werden.

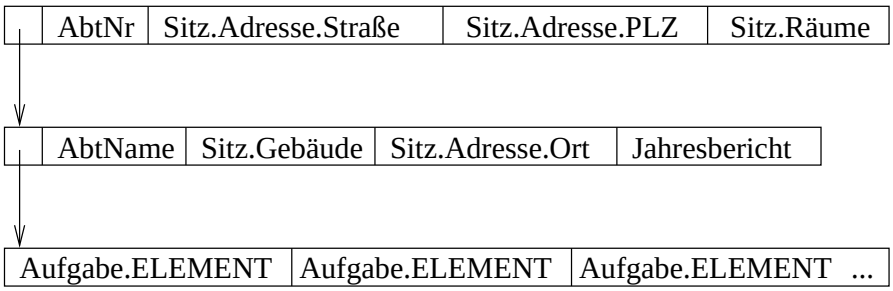


ABBILDUNG 3.39: Speicherstruktur Inline Array für Kollektionen

▷ *Pointer Array*
Werden die Elemente einer Kollektion auf die eine oder andere Art dynamisch auf mehrere interne Sätze verteilt, die aus einem gemeinsamen übergeordneten Satz heraus nur noch referenziert werden, so soll von der Speichervariante Zeigerfeld gesprochen werden.

▷ *Linked List*
Werden die Elemente einer Kollektion dagegen dynamisch auf mehrere interne Sätze verteilt, die nach dem Prinzip der verketteten Liste miteinander verknüpft sind, so stellt das eine Variante der verketteten Satzliste (Linked List) dar.

▷ *Indexiert/indexorganisiert*
Wenn die Elemente einer Kollektion dynamisch auf mehrere interne Sätze verteilt und aus einer Zugriffsstruktur heraus referenziert werden, so soll von einer indexierten oder indexorganisierten Speicherungsform die Rede sein. Pro Kollektion soll dabei in der Regel eine separate Indexstruktur vorhanden sein. Aber auch gemeinsame Indexstrukturen für eine Menge von eingeschachtelten Kollektionen sollen unter diese Kategorie fallen. Zur Indexierung kommen unterschiedlichste Zugriffsstrukturen in Betracht: B*-Bäume, Hashstrukturen, Suffix- sowie RD-Bäume und so weiter.

▷ *Alternative Kollektionsrepräsentationen*
Auch eine Reihe von Signatur- und Bitmap-Verfahren zur Speicherung von Kollektionen durch alternative Repräsentation ihrer Elemente werden mit berücksichtigt und unter diesem Begriff zusammengefaßt.

Man beachte, daß die durch Kollektionsspeicherstrukturen zusätzlich erzeugten internen Sätze auch hier als Sekundärsätze bezeichnet werden. Außerdem können die einzelnen Strukturen parametrisiert, abgewandelt und kombiniert eingesetzt und an die jeweiligen Bedürfnisse angepaßt werden, denn jede der Alternativen zur Kollektionsspeicherung bringt Vor- und Nachteile für den Einsatz bei bestimmten logischen Kollektionsarten mit sich.

Im folgenden werden diese wichtigsten möglichen Strukturen näher erläutert. Dabei wird auf geeignete Zuordnungen zu den in Abschnitt 2.2.4.2 vorgestellten logischen Kollektionsstrukturen eingegangen. Vorher ist es jedoch Zeit, dem geschätzten Leser den Begriff *kronen* zur späteren Auflösung zu nennen.

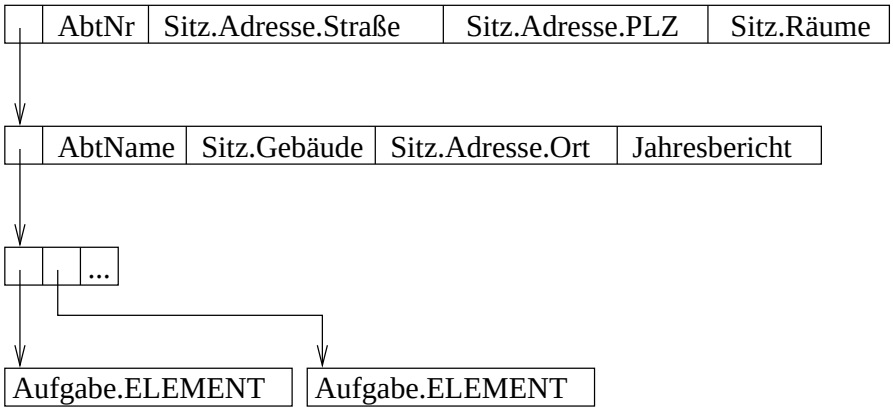


ABBILDUNG 3.40: Speicherstruktur Pointer Array für Kollektionen

1. *Eingelagerte Speicherung*

Beim der eingelagerten Speicherung (Inline Array) werden die Elemente einer Kollektion innerhalb des physischen Satzes des übergeordneten Attributs oder des Tabellentupels gespeichert, das heißt innerhalb eines einzigen internen Satzes. Ein Beispiel ist in ABBILDUNG 3.39 gegeben. Im Vergleich zu den anderen Alternativen ist diese Struktur bei einer Erweiterung von RDBMS zu ORDBMS noch vergleichsweise einfach zu implementieren. Kennzeichnend ist insbesondere, daß die Anzahl der Sekundärsätze konstant bleibt.

Einen großen Einfluß auf die Aufnahmekapazität der Datenstruktur hat hier – bei der Anwendung auf den Satz, der die Kollektion speichert – die Verfügbarkeit von seitenübergreifenden Sätzen (Abschnitt 3.2.3), die seitenübergreifende verkettete Sätze variabler Länge darstellen. Sind verkettete Sätze nicht erlaubt, beschränkt sich die maximale Größe der Kollektion damit drastisch.

Eine zusätzliche Erweiterung besteht in der optionalen Definition eines Überlaufsatzes, der ab einer bestimmten Anzahl von Elementen beziehungsweise ab einer überschrittenen Gesamtgröße der Kollektion zusätzlichen Speicherplatz zur Verfügung stellt. Auf diese Weise kann einerseits die zulässige Kollektionsgröße erhöht werden und andererseits die Maximalgröße des internen Satzes kontrolliert werden, der die Kollektion enthält.

Eingelagerte Speicherungen eignen sich besonders für kleinere Kollektionen und Kollektionen mit kleinen Elementen, auf die in der Regel gemeinsam und nicht einzeln zugegriffen wird. Davon ausgehend läßt sich jede Spielart weiterer Kriterien wie sortierte Speicherung oder schneller Elementzugriff hinreichend gut befriedigen.

2. *Zeigerfeld*

Die Speicherstruktur Zeigerfeld (Pointer Array) verteilt die Elemente einer Kollektion auf die eine oder andere Art dynamisch auf mehrere interne Sätze, die aus dem Satz des übergeordneten Attributs heraus referenziert werden (ABBILDUNG 3.40). Je nach Größe der Kollektion werden dazu unterschiedlich viele Sekundärsätze angelegt. Der dabei entstehenden Gruppe von internen Sätzen wird einheitlich ein Speicherplatz zugewiesen. Der Benutzer ist in der Lage, die Anzahl von Kollektionselementen pro internem Satz zu steuern. Dies kann entweder mittels Angabe einer maximalen Elementanzahl oder einer maximalen Größe der Sätze geschehen.

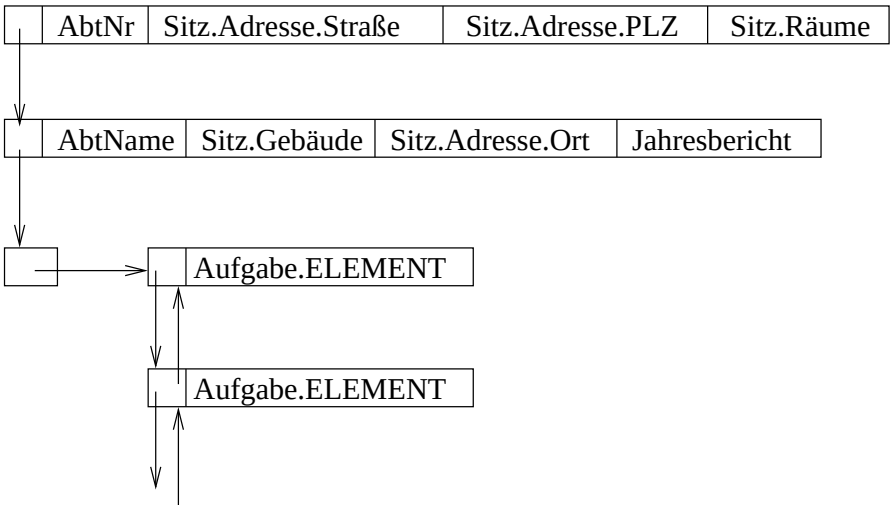


ABBILDUNG 3.41: Speicherstruktur Linked List für Kollektionen

Die Vorteile dieser Struktur liegen darin, daß einerseits sehr umfangreiche Elemente abgespeichert werden können, andererseits aber auch – unter der Vorraussetzung, daß pro Elementsatz eine konstante Anzahl von Elementen erlaubt ist – ein schneller Positionszugriff auf einzelne Elemente möglich wird. Legt man zusätzlich Indexstrukturen über den Elementsätzen an, erhöht dies außerdem die Geschwindigkeit des wertebasierten Zugriffs. Hervorzuheben ist ebenfalls das schnelle Einfügen und Löschen von Kollektionselementen.

Die Struktur Zeigerfeld eignet sich insbesondere für die logische Kollektionsart Feld sowie auch für Felder mit variabel langen Elementen. Der Grund liegt nicht zuletzt darin, daß der Zugriffspfad zu den Elementen und die eigentlichen Daten strikt voneinander getrennt sind. Da die Zeiger als Felder organisiert sind und eine fixe Größe besitzen, kann ein Positionszugriff in jedem Fall sehr schnell erfolgen.

Neben den Zeigern auf die Sätze mit den Kollektionselementen werden in dem übergeordneten Sekundärsatz zusätzliche Meta-Daten, wie die aktuelle Gesamtzahl vorhandener Elemente, gespeichert.

Die Speicherung sehr großer Kollektionen wird in gewisser Weise dadurch begrenzt, daß pro angelegtem Elementsatz ein Verweis innerhalb des übergeordneten Satzes angelegt werden muß. Sind für letzteren verkettete Sätze verboten, so ist die mögliche Anzahl von Zeigern durch die maximale Satzlänge, das heißt die Seitengröße, beschränkt.

Wie auch bisher bei Sekundärsätzen, können zusätzliche Rückwärtsreferenzen zum übergeordneten Sekundärsatz beziehungsweise zum Primärsatz erstellt werden, was besonders für den Aspekt der Indexierung von Kollektionselementen interessant ist.

3. Verkettete Liste

Eine sehr flexible Speicherstruktur für beliebig große Kollektionen findet sich in der Struktur der verketteten Liste, bei der die Kollektionselemente dynamisch auf mehrere interne Sätze verteilt und verkettet werden (ABBILDUNG 3.41). Da das Hinzufügen von Elementen keine Vergrößerung des einzelnen Satzes zur Folge hat, sondern die Struktur stets um einen vollständigen Satz erweitert wird, ist die Maximalgröße dieser Kollektionsstruktur theoretisch nur durch die Größe des zugewiesenen Tablesspace begrenzt.

Wie auch bei Zeigerfeldern (Pointer Arrays), kann der Benutzer Einfluß auf die Anzahl

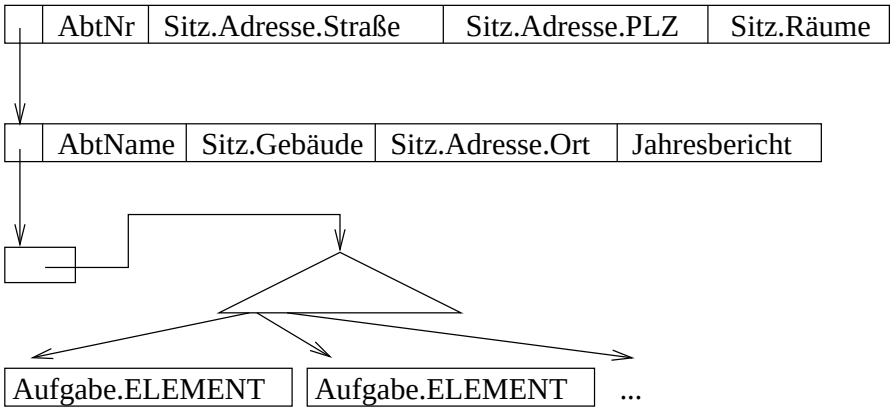


ABBILDUNG 3.42: Indexierte beziehungsweise indexorganisierte Speicherung von Kollektionen

der enthaltenen Elemente und die Größe der dynamisch erstellten Sekundärsätze nehmen. Auch Rückwärtsreferenzen zum übergeordneten Satz sind möglich. Zusätzlich kann bei dieser Speicherstruktur zwischen einfach und doppelt verketteter Liste gewählt werden.

Verkettete Listen eignen sich – wie man leicht nachvollziehen kann – sehr gut für die logische Kollektionsart List, deren besondere Eigenschaften sich ja gerade aus der physischen Implementierung her ableiten.

4. Indexierte beziehungsweise indexorganisierte Speicherung

Als umfangreichste Speicherstruktur stellt die indexierte beziehungsweise indexorganisierte Variante die höchsten Anforderungen an die Erweiterung von RDBMS zu ORDBMS. Sie unterscheidet sich von den bisher genannten Alternativen insofern, als daß die Verbindung zu den eigentliche Daten über einen eigene Zugriffsstruktur realisiert wird (ABBILDUNG 3.42).

Am Speicherort des kollektionswertigen Attributs wird (neben den üblichen möglichen Meta-Daten) ausschließlich ein Zeiger auf die Wurzel einer Zugriffsstruktur angelegt. Die Blätter, Knoten oder Einträge der Zugriffsstruktur wiederum zeigen auf Sekundärsätze mit je einem Kollektionselement.

Im Rahmen der Kollektionsstruktur werden demnach zwei Typen von Sekundärsätzen angelegt: Sätze, die zu der Zugriffstruktur gehören, und Sätze, die die Elementdaten enthalten.

Falls ein Ordnungskriterium zur Verwaltung der Zugriffsstruktur benötigt wird (wie beispielsweise bei B*-Bäumen), kann dieses in der Regel aus dem Typ der Elemente abgeleitet werden, da selbst tief verschachtelte Typen letztendlich auf eine bestimmte Anzahl von Elementartypen reduzierbar sind. Somit existiert meist ein Ordnungskriterium auch für den Fall, in dem der Benutzer nicht explizit einen Primärschlüssel angegeben hat. Problematisch ist nur der Fall von verschachtelten Mengen, denn er erfordert die Definition einer nicht natürlich vorgegebenen linearen Ordnung über Mengen (beispielsweise mittels einer Hashfunktion oder über die Anzahl der Elemente).

Eine weitere Konsequenz aus dem Prinzip des integrierten Zugriffspfads besteht darin, daß im Gegensatz zu den letzten beiden Speicherstrukturen pro Elementsekundärsatz nur ein Element der Kollektion sinnvoll ist. Es besteht wieder die Möglichkeit der Definition von

Rückwärtsreferenzen. Als weitere Option ist eine zusätzliche Verknüpfung der Elementsätze nach dem Prinzip der verketteten Liste denkbar.

Als Zugriffsstruktur können verschiedene Baumindexe, Hash- und Signaturbaumverfahren eingesetzt werden. Letztendlich kommen alle auch zur reinen Indexierung verwendeten Verfahren in Frage. Sie werden im Kapitel 4 behandelt.

Je nach eingesetzter Zugriffsstruktur eignet sich die indexierte beziehungsweise indexorganisierte Primärspeicherstruktur für Elementtests, schnellen Elementzugriff und andere Operationen, wie sie bei den logischen Kollektionsarten Set und Multiset üblich sind. Im Gegensatz zur verketteten Liste können selbst bei sehr großen Mengen noch passable Antwortzeiten garantiert werden.

5. Alternative Kollektionsrepräsentationen

Zur Speicherung von Kollektionen, deren Elemente „einfach numerierbar" sind, können Bitmaps als alternative Repräsentation genutzt werden. Unter „einfach numerierbar" sollen hier Elementtypen verstanden werden, die in natürlicher Art und Weise auf ein endliches Intervall ganzer Zahlen abgebildet werden können. Anstatt die Elemente selbst zu speichern, kann in einer Liste beziehungsweise in einem Feld, das für jedes mögliche Element ein Bit enthält, das Vorhandensein der Elemente durch entsprechend gesetzte Bits gespeichert werden. Zusätzlich können verschiedene verlustfreie Kompressionsverfahren für diese Bitmap genutzt werden, so daß es möglich ist, Kollektionen sehr kompakt zum Beispiel im selben physischen Satz wie die übergeordnete Struktur (inline) zu speichern. Ein weiterer Vorteil solcher Bitmap-Verfahren ist die Unterstützung von Abfragen und Änderungen mit Mengenoperationen, die durch einfache boolesche Bitoperationen realisiert werden können.

Neben diesen verlustfreien Bitmap-Verfahren zur alternativen Repräsentation von Kollektionen existieren auch verschiedene Signaturverfahren. Diese sind jedoch in der Regel verlustbehaftet, das heißt, sie stellen nur eine approximative Repräsentation dar. Sie erlauben zwar eine sehr schnelle Ausführung von Abfragen und Änderungen durch Mengenoperationen und eine sehr kompakte Repräsentation, sind aber nicht zur primären Speicherung geeignet. Als sehr effiziente Möglichkeit zur Indexierung von Kollektionen werden sie jedoch in Abschnitt 4.1.3.1 behandelt.

3.2.6.2 Kollektionskonstruktoren zur flexiblen Definition von Speicherstrukturen

Zur Definition von Kollektionsstrukturen wird in [Kis02] ein Sprachansatz vorgestellt der sich an den ersten vier der im lezteten Abschnitt vorgestellten Grundspeicherformen orientiert. Er beschreibt Konstrukte zur Definition der vier festen Speichervarianten eingelagerte Speicherung (Inline Array), Zeigerfeld (Pointer Array), verkettete Liste (Linked List) und B-/B*-Baum, die jeweils mehr oder weniger durch eine Reihe von Parametern veränderbar waren (ABBILDUNGEN 3.38(b), 3.38(d), 3.38(a) und 3.38(c)). So ist es beispielsweise möglich, bei der Struktur Linked List anzugeben, ob innerhalb eines Elements der verketteten Liste auch mehrere Elemente als Feld gespeichert werden können.

Allerdings ist dieser Ansatz – vier feste Fälle und jeweilige Parametrisierung – zu unflexibel, denn er läßt viele sinnvolle Speichervarianten nicht zu, selbst Varianten, die von heutigen kommerziellen DBMS angeboten werden, wie zum Beispiel die ausgelagerte Speicherung von Kollektionselementen zusammen mit dem Einsatz von Rückwärtsreferenzen.

Aus diesen Gründen wird in dieser Arbeit, aufbauend auf [Kis04], das Konzept mit einer bestimmten Anzahl von festen Varianten zugunsten eines Ansatzes mit einer festen Anzahl von Konstruktoren verworfen.

Jeder dieser Konstruktoren steht für eine Art Strukturbaustein, mit dessen Hilfe durch Kombination und gegenseitige Einsetzung eine Vielzahl von unterschiedlichsten physischen Speicherstrukturen für Kollektionen entstehen können. Die Konstruktoren und ihre Sprachbausteine werden in Abschnitt 5.2.6 im Detail beschrieben.

Hier sei abschließend nur eine Auswahl an Konstruktoren aufgeführt, die das Aufgreifen der grundlegenden Kollektionsstrukturen des letzten Abschnitts (3.2.6.1) verdeutlicht. Unter anderem werden folgende Kostruktoren vorgeschlagen:

- ▷ `ARRAY`
- ▷ `LINKED LIST`
- ▷ `B-TREE`

Im Zusammenspiel mit weiteren Konstruktoren wie zum Beispiel `REFERENCE` und `RECORD` lassen sich die unterschiedlichsten Speicherstrukturen definieren. So ergibt beispielsweise `ARRAY OF ELMENT` eine eingelagerte Speicherung, `ARRAY OF REFERENCE` dagegen ein Zeigerfeld.

3.2.7 Speicherung von Objekten aus Typhierarchien

Das wesentliche Merkmal an dem Objektorientierung oft festgemacht wird, ist die Vererbung beziehungsweise die Spezialisierung und Generalisierung von Klassen. Dementsprechend muß ein objektrelationales DBMS auch angemessene Möglichkeiten zur Speicherung von Objekten aus Vererbungshierarchien bieten.

Aus logischer Sicht führt das bei Spaltentypen dazu, daß in einer Tabellenspalte polymorphe Objekte gespeichert werden. Ihr Subtyp (unterhalb des Spaltentyps) kann sich von Tupel zu Tupel ändern.

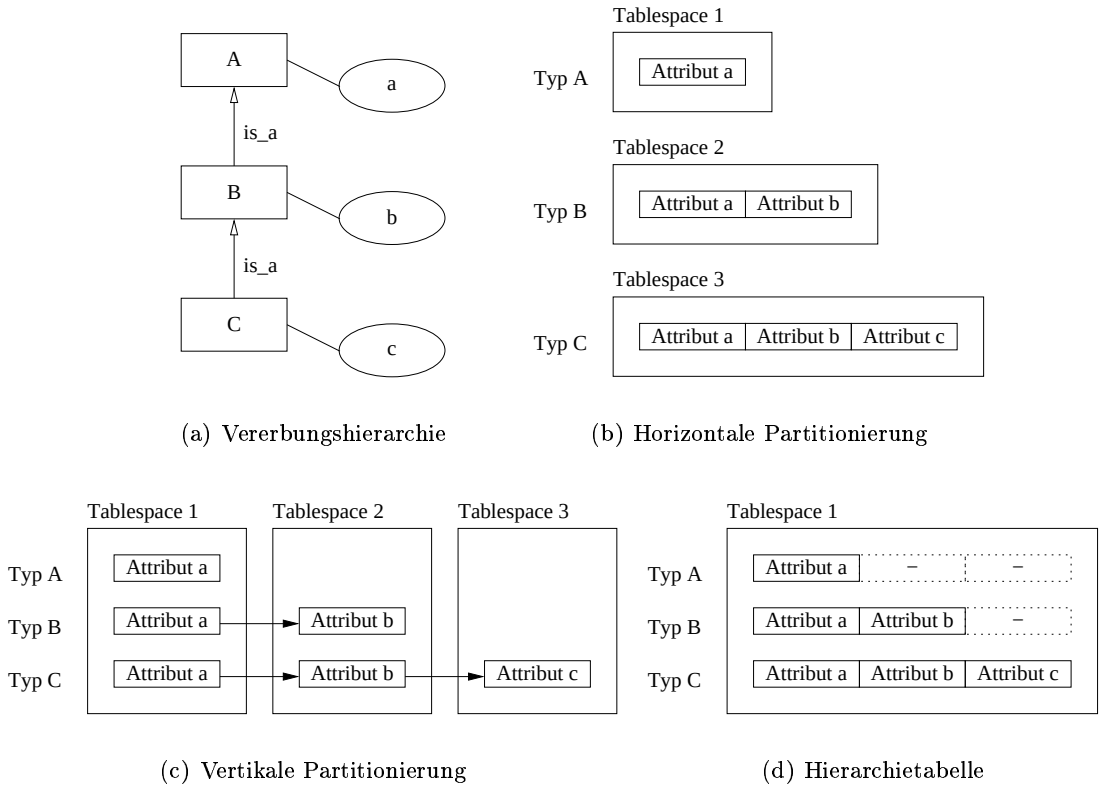
Für Tabellentpyen können auf der logischen Repräsentationsebene für die Subtypen entsprechende Subtabellen explizit (durch eine Subtabellendefinition) oder implizit (für alle betroffenen Tabellen durch die Subtypdefinition) eingeführt werden. Diese haben dann einen entsprechend dem Subtyp erweiterten Satz an Attributen. Bei Anfragen an die Supertabelle müssen dann die Supertabelle und alle Subtabellen vereinigt werden, so daß aus logischer Sicht eine Teilmengenbeziehung zwischen Sub- und Supertabellen besteht.

Aus physischer Sicht sind für die Speicherung der Subtypobjekte beziehungsweise ihrer zusätzlichen Attribute in der Literatur im wesentlichen 3 Grundverfahren bekannt [Luf00]:

- ▷ *Horizontale Partitionierung*

  Dabei werden die Objekte von Super- und Subtypen entsprechend der Typhierarchie auf Tablespaces (beziehungsweise bei relationaler Ablage auf der logischen Ebene auf Tabellen) aufgeteilt: für jeden Subtyp eine eigene Tabelle mit allen ererbten und direkt definierten Attributen (siehe ABBILDUNG 3.43(b)).

  Beim Einfügen müssen die Objekte entsprechend ihrem Typ in die richtige Subtabelle und damit in den richtigen Tablespace eingeordnet werden. Und bei der Abfrage einer Supertabelle muß diese mit allen ihren Subtabellen, jeweils auf die Supertypattribute projeziert, vereinigt werden.



(a) Vererbungshierarchie (b) Horizontale Partitionierung

(c) Vertikale Partitionierung (d) Hierarchietabelle

ABBILDUNG 3.43: Verfahren zur Speicherung von Objekten aus Vererbungshierarchien

▷ *Vertikale Partitionierung*

Bei diesem Verfahren werden alle Objekte aller Typen einer Vererbungshierarchie in einer gemeinsamen Tabelle, das heißt, physisch in einem gemeinsamen Tablespace, gehalten (siehe ABBILDUNG 3.43(c)). Jedoch besitzt diese Tabelle nur die Attribute des Supertyps. Die zusätzlichen Attribute jedes Subtyps werden dann in neue Tabellen und damit in andere Tablespaces ausgelagert.

Beim Einfügen von Objekten werden die Attribute entsprechend der konkreten Subklassenzugehörigkeit dann auf die Tablespaces aufgeteilt und über Referenzen miteinander verbunden. Abfragen erfordern dann umgekehrt wieder das Zusammenfügen der Objektteile über Verbundoperationen.

▷ *Hierarchietabelle*

Hier werden die Objekte aller Typen in einem gemeinsamen Tablespace gehalten (siehe ABBILDUNG 3.43(d)). Die Sätze dieses Tablespaces haben jeweils unterschiedliche Attributmengen, entsprechnd dem konkreten Typ des darin abgelegten Objekts. Es wird dabei von unterschiedlichen Satztypen (ein Satztyp pro Objekttyp) gesprochen. Die Ablage in einer Hierarchietabelle erfordert also vom Speichersystem die Fähigkeit zur Speicherung heterogener Satztypen. Alternativ könnten die physischen Sätze auch die Obermenge aller Attribute einer Vererbungshierarchie enthalten und die jeweils nicht benötigten Attribute mit `NULL` belegen. Aber dann muß der jeweilige Objekttyp festgehalten werden, um zwischen im Typ nicht vorhandenen und vorhandenen, aber mit `NULL` belegten Attributen unterscheiden zu können.

Alle diese Grundverfahren müssen von einem objektrelationalen Speichersystem unterstützt werden.

Dazu sind jedoch die bereits vorgestellten Konzepte ausreichend und keine neuen Sprachkonstrukte notwendig. Für die Speicherung und Definition gelten dazu folgende Regeln:

1. *Eine Subtabelle beziehungsweise ein Subtyp ohne eigene Speicherspezifikation übernimmt (‚erbt') die Spezifikation seiner Supertabelle beziehungsweise seines Supertyps.*

   Dies bewirkt, daß die ererbten Attribute in gleicher Weise wie bei der Superttabelle gespeichert werden. Neue Attribute werden ohne Spezifikation zum Primärsatz hinzugefügt.

   Diese Regelung gilt sowohl für typisierte Tabellen als auch für Spaltentypen und führt zur Speicherung nach dem *Hierarchietabellenansatz.*

2. *Für eine Subtabelle beziehungsweise einen Subtyp können Speicherspezifikationen für neue Attribute vorgenommen werden.*

   Dabei kann für diese neuen Attribute eine Auslagerung in neue Sekundärsätze in einem anderen Tablespace definiert werden. Gleichzeitig kann angegeben werden, wie die Referenzierung zwischen den Primär- und Sekundärsätzen erfolgen soll.

   Auf diese Weise ergibt sich eine *vertikale Partitionierung* von Objekten in einer Vererbungshierachie.

3. *Schließlich kann bei der Subtabellen- beziehungsweise Subtypdefinition auch eine Speicherungsspezifikation für alle Attribute angegeben werden. Die Speicherung neuer Attribute wird darin definiert und die Speicherung ererbter Attribute redefiniert.*

   Die Speicherredefinition gilt nur für Objekte des jeweiligen Subtyps und aller seiner Subtypen, aber nicht für Objekte von Supertypen. Wird für Subtypobjekte Speicherung in einem neuen Tablespace definiert, so führt das zur *horizontalen Pratitionierung.*

Gleichzeitig können über diese Art der Speicherspezifikation beliebige Kombinationen der vorgestellten Verfahren ermöglicht werden.

Insofern eröffnen die vorgestellten Konzepte ein großes Spektrum zur flexiblen Anpassung der Speicherstrukturen von komplexen Objekten in Vererbungshierarchien an unterschiedliche Situationen und Arbeitslasten.

3.3 Zusammenfassung des Kapitels

Als Grundlage für die im Kapitel 5 vorgestellte Speicherbeschreibungssprache PRDL zur Trennung des physischen Datenbankentwurfs von der Definition der logischen Datenstrukturen mit SQL wurden im ersten Abschnitt dieses Kapitels Speicherstrukturen für komplexe Objekte aus der Literatur, aus DBMS-Prototypen und -Produkten untersucht. Es wurden die wesentlichen bekannten Konzepte präsentiert und auf ihre Eignung für objektrelationale DBMS hin diskutiert.

Dabei stellte sich heraus,

- daß sich in allen Datenbankgenerationen, von den vorrelationalen über die relationalen bis hin zu den objektorientierten Ansätzen, relavante Speicherkonzepte finden lassen und

- ▷ daß sich Grundkonzepte, wie die Ein- oder Auslagerung von Teilobjekten, die Clusterung und andere mehr, zwar in verschiedenen Verbrämungen, aber doch über fast alle Generationen hinweg ähneln.

Auf diese Weise konnte im zweiten Abschnitt des Kapitels eine möglichst vollständige, abgeschlossene und konsistente Konzeption zur Speicherung komplexer Objekte in ORDBMS aufgestellt werden. Diese umfaßt einen Satz von Speicherkonzepten und -alternativen, die es ermöglichen, die verschiedenen Verarbeitungsanforderungen bei verschachtelten Objektstrukturen durch die Anpassung der physischen Repräsentation zu unterstützen.

Insofern wurde mit diesem Kapitel die Speicherstrukturbasis für eine verbesserte Komplexobjektunterstützung in ORDBMS sowie für eine Trennung von logischen und physischen Modellierungsaspekten geschaffen. Das nächste Kapitel wird die Überlegungen zur Speicherung um Indexierungskonzepte ergänzen, so daß dann die Grundlagen für die Präsentation der Speicherbeschreibungssprache PRDL in Kapitel 5 gelegt sein werden.

Kapitel 4

Zugriffspfade für komplexe Objekte

4.1 Zugriffspfade in Literatur und Produkten

Im folgenden soll ein Überblick über verschiedene in der Fachliteratur vorgestellte Indexstrukturen gegeben werden. Dabei werden die Entwicklungen seit Einführung des B- und B*-Baums zur Sprache kommen, wobei die Auswahl auf die für objektrelationale Datenbanksysteme relevanten Indexe eingeschränkt wird.

4.1.1 Überblick

Es kann grob zwischen zwei Richtungen bei den Entwicklungen von Zugriffsstrukturen unterschieden werden. Zum einen gibt es Indexe, die auf eine spezielle Anwendung beziehungsweise einen speziellen Datentyp hin entworfen wurden, und zum anderen universelle Zugriffsstrukturen für vorhandene oder neue Generationen von Datenbanken, wie NF^2- oder objektorientierte Datenbanken. In die erste Gruppe fallen insbesondere alle Arten von geometrischen (spatial) Zugriffsstrukturen oder solche für Multimedia-Datenbanken. Da objektrelationale ebenso wie schon rein relationale Systeme bestrebt sind, möglichst anwendungsunabhängig und universell zu sein, werden diese Arten von Indexen – soweit nicht nützliche Techniken und Ansätze übernommen werden können – nachfolgend nicht weiter betrachtet.

Interessanter und ertragreicher ist hingegen ein Blick auf Entwicklungen in den Bereichen OODBMS, NF^2 und deren Varianten. Auch hier kann zwischen Lösungen unterschieden werden, die eng an spezielle Voraussetzungen innerhalb der grundlegenden Konzepte dieser Datenbanken gebunden sind (zum Beispiel die Existenz von Klassenhierarchien in OODBMS) und sich daher nicht auf ORDBMS übertragen lassen, sowie allgemein gehaltenen Lösungen, welche auch in die objektrelationale Welt übernommen werden können. Ein umfassender Vergleich von Zugriffsstrukturen speziell für NF^2 findet sich in [Keß95].

In dieser Arbeit liegt der Fokus auf Indexstrukturen, die eine Unterstützung für objektrelationale Datenstrukturkonzepte bieten, wie zum Beispiel für frei ineinander verschachtelbare und miteinander kombinierbare Strukturen und Kollektionsarten. Abbildung 4.1 zeigt eine Übersicht von Indexstrukturen im Bereich objektorientierter Indexe sowie Pfad- und Verbundindexe. Die Pfeile symbolisieren, auf welchen bereits vorhandenen Konzepten eine neue Entwicklung aufgebaut wurde beziehungsweise woher Techniken übernommen wurden. Anhand der Pfeile ist also auch eine zeitliche Entwicklung ablesbar.

Einen weiteren großen, für objektrelationale Systeme relevanten Bereich bilden Indexe zur Unterstützung von Operationen auf Kollektionen (siehe Abbildung 4.4). Hervorzu-

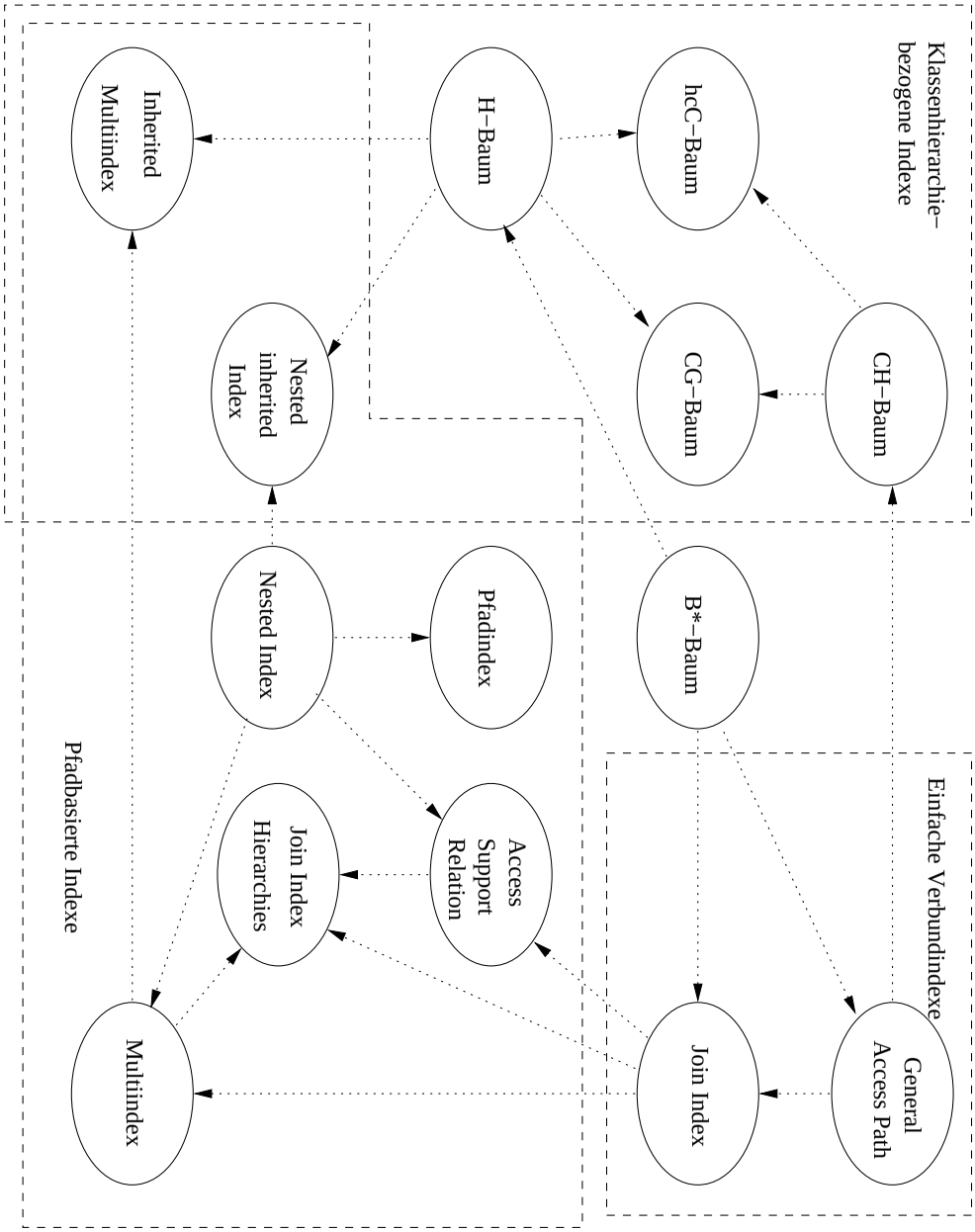


ABBILDUNG 4.1: Übersicht über objektorientierte, pfadbezogene und Verbundindexe

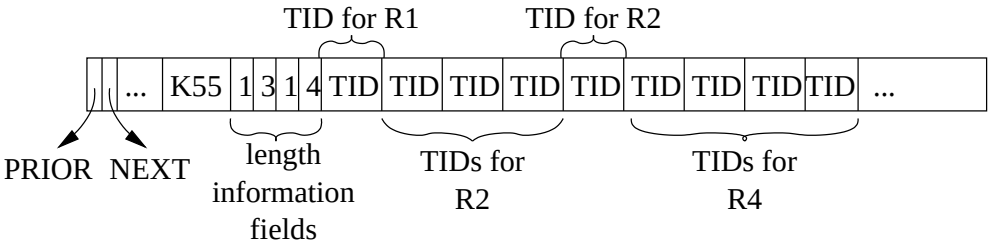


ABBILDUNG 4.2: Indexeintrag bei der Generalized Access Path Structure (auf Blattebene des B*-Baumes)

heben ist, daß es – wie auch bei den übrigen Zugriffsstrukturen – um universelle, auf die generischen Mengenkonstruktoren abgestimmte Lösungen geht. Ein großes Feld nehmen hierbei Strukturen ein, die sich signaturbasierter Verfahren bedienen.

4.1.2 Pfad- und klassenbezogene Indexe

4.1.2.1 Generalized Access Path Structure

Die bereits im Jahre 1978 von Härder vorgestellte Zugriffsstruktur Generalized Access Path [Här78] zielt darauf ab, die Vorteile von direkt auf Tupelebene verketteten Daten mit den Vorteilen klassischer B*-Baum-Strukturen zu vereinen. Erstere Technik ähnelt stark den aus Netzwerkdatenbanken bekannten physischen Strukturen und dient der Implementierung einer 1:n-Beziehung zwischen den Tupeln zweier Relationen, welche durch eine Fremdschlüssel-Beziehung miteinander verbunden sind. Beispielsweise kann so ein Satz einer Relation Abteilung – in dieser Beziehung Owner genannt – mit den zugehörigen Sätzen der Relation Angestellter (Member) verknüpft werden. Dabei wird im Satz der Abteilung ein Verweis auf den Satz des ersten Angestellten dieser Abteilung gespeichert, während sich in Sätzen der Angestellten ein Verweis zu dem vorherigen und nachfolgenden Angestellten sowie zu dem Satz der dazugehörigen Abteilung befindet. Der Hauptvorteil dieser Technik gegenüber einem einfachen Index über dem Fremdschlüssel in der Angestelltenrelation liegt darin, daß zum Zugriff auf alle Angestellten einer Abteilung eben kein Zugriff auf eine externe Indexstruktur und die damit verbundenen Externspeicherzugriffe notwendig sind. Das bessere Verhältnis der Seitenzugriffe kann des weiteren durch eine geclusterte Speicherung eines Abteilungssatzes mit den dazugehörigen Angestelltensätzen verstärkt werden. Im optimalen Fall ist kein einziger zusätzlicher Seitenzugriff notwendig, das heißt mit einem einzigen Seitenzugriff werden sowohl die Abteilungs- als auch die zugehörigen Angestelltendaten bereitgestellt.

Da die beiden Relationen über eine Fremdschlüsselbeziehung miteinander verbunden sind, also die Zusammengehörigkeit auf logischer Ebene durch Attribute gleichen Typs gegeben ist, können die bekannten Vorteile einer B*-Baum-Struktur mit denen von Satzverknüpfung kombiniert werden. Härder führt die Generalized Access Path Structure ein, bei der durch einen Baumindex mehrere Relationen indexiert werden können. Dabei besitzen die Relationen jeweils ein Attribut gleichen Typs, welche gemeinsam in einer Zugriffsstruktur indexiert werden. Eine Abfrage in diesem Index liefert demnach für jede der beteiligten Relationen eine Menge von Verweisen auf die entsprechenden Sätze. ABBILDUNG 4.2 soll das Prinzip am Beispiel von vier Relationen R1 bis R4, in denen jeweils das Attribut Abteilungsnummer enthalten sei, verdeutlichen. Im Beispiel sei ‚K55' eine solche Abteilungsnummer.

Der große Vorteil dieser Struktur liegt in ihrer Vielseitigkeit bei der Unterstützung von Abfragen: Einerseits kann eine Generalized Access Path Structure für die einzelne Tabelle wie jeweils ein eigener B*-Baum-Index betrachtet werden, andererseits kann die Struktur zur Unterstützung von Verbundabfragen über das Schlüsselattribut (Abteilungsnummer im Beispiel) herangezogen werden.

Ein Nachteil dieser Struktur liegt darin, daß bei der Beteiligung einer Relation mit verhältnismäßig vielen Tupeln die Baumstruktur sehr groß wird, so daß bei einem Zugriff, der nur einen Teil der im Index vorhandenen Relationen betrifft, zu viele nicht benötigte Index-Daten geladen werden müssen. Ein weiteres in [Val87] aufgeführtes Argument betrifft den Einsatz solcher Strukturen in Hochleistungs-DBMS mit vielen parallelen Transaktionen. Ein einzelner universeller Index für viele, auch unterschiedliche Zugriffe führt unweigerlich zu einem Hot Spot und entsprechenden Leistungseinbußen, vor allem bei vielen Indexänderungen.

Zusammenfassend läßt sich aber sagen, daß mit dieser Indexstruktur für viele Anwendungsfälle eine einfache und flexible Art der Unterstützung von Verbundanfragen zur Verfügung steht, indem der Verbund im Index vorberechnet wird.

4.1.2.2 Verbundindexe

Ebenfalls zur schnelleren Auswertung von Verbundanfragen (Joins) wurde 1987 von Valduriez das Konzept der Verbundindexe (Join Indexes) vorgestellt [Val87]. Für zwei Relationen, die häufig über bestimmte Attribute miteinander verbunden werden sollen, werden zwei B*-Bäume angelegt, die jeweils die Tupel-IDs (TIDs) der Sätze der einen zu verbindenden Relation als Schlüssel haben. Einträge der Baumindexe sind die dem Verbund entsprechenden TIDs der anderen Relation.

Durch das Vorabberechnen des Verbundes ist ein Einsatz dieser Hilfsstrukturen vor allem bei besonders komplexen, über nutzerdefinierte Funktionen, andere Berechnungen oder mehrere Attribute gebildeten Verbundprädikaten sinnvoll. Gleichzeitig entstehen allerdings gegebenenfalls sehr hohe Kosten bei häufigen Änderungen der für den Verbund relevanten Attribute. Der Einsatz von Verbundindexen ist gegenüber Lösungen mit klassischen Indexen über den zu verbindenden Attributen zudem immer dann zuüberlegen, wenn der Verbund weder eine sehr hohe noch eine sehr geringe Selektivität hat [Val87].

Optimierungsmöglichkeiten ergeben sich im Einsatz spezieller, angepaßter Verbundalgorithmen (hybrider Hash-Join-Algorithmus) sowie durch das Anwenden von Bitlisten als Indexstruktur. Letztere erweisen sich insbesondere für Warehouse-Applikationen als sehr effizient.

4.1.2.3 Pfadbezogene Indexe

Ein wichtiger Schritt in der Entwicklung objektorientierter Datenbanksysteme war die Vorstellung der drei Indexarten Pfad-, Nested und Multiindex durch Bertino und Kim 1989 [BK89]. Ausgangspunkt der Überlegungen war es, für die – in vielen Fällen hierarchischen – Objektnetzwerke in OO-Datenbanken Mechanismen bereitzustellen, die eine schnelle Selektion von Objekten einer Klasse anhand von Attributen ermöglicht, welche außerhalb der gesuchten Objekte liegen. Vom Prinzip her handelt es sich dabei um einen Verbund über mehrere Stufen hinweg. Dieses Konzept war bereits 1986 erstmals von Maier und Stein [MS86] im Kontext des GemStone-Datenbank-Projektes diskutiert worden.

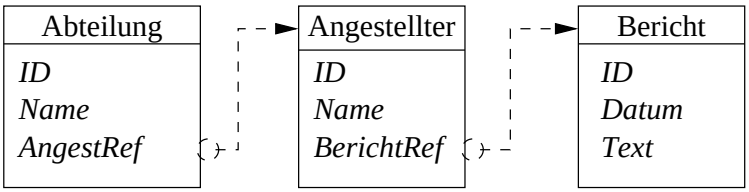


ABBILDUNG 4.3: Beispiel einer Klassenhierarchie

Als Beispiel seien drei Klassen Abteilung, Angestellter und Bericht gegeben (ABBILDUNG 4.3). Die Attribute AngestRef und BerichtRef sind jeweils Mengen von Referenzen. Klassen-Attribut-Pfade in diesem Modell wären zum Beispiel:

```
Abteilung.AngestRef.BerichtRef.Datum

Angestellter.BerichtRef.Text
```

Allgemein gesagt besteht ein solcher Pfad aus einer Kette von miteinander verknüpften Klassen. Nur am Ende des Pfades befindet sich ein Attribut, welches zu der letzten Klasse gehört.

Ziel dieser Gruppe von Indexen ist eine effiziente Auswertung von Fragen wie: „In welcher Abteilung wurde in der letzten Woche von einem Angestellten ein Bericht erstellt?“. Zur Beantwortung solcher Anfragen würde ein Index angelegt werden, dessen Schlüssel die Datumswerte der Objekte aus der Klasse Bericht sind und dessen Einträge eine Menge von Verweisen auf Objekte der Klasse Abteilung enthalten.

Die drei Arten von Indexen unterscheiden sich folgendermaßen: Ein Nested Index enthält als Indexeintrag neben dem Schlüssel nur Verweise auf die Zielobjekte (im Beispiel also auf die Abteilungen). Bei einem Pfadindex werden hingegen Identifikatoren zu allen Objekten eines konkreten Pfads gespeichert. Ein Multiindex setzt sich aus mehreren einstufigen Indexen – einer je Verschachtelungsebene – zusammen. Im vorliegenden Beispiel wären das drei: ein Index, der anhand eines Datums alle dazugehörigen Berichte liefert, ein Index, der zu der ID eines Berichtes den entsprechenden Angestellten liefert, und ein Index, der einen Angestellten seiner Abteilung zuordnet. Bis auf den ersten Index handelt es sich also bei den Teilindexen eines Multiindex immer um sogenannte Identitätsindexe (siehe 3. Multiindex).

Auf die Vor- und Nachteile der drei Indexarten gehen die folgenden Abschnitte ein.

1. Nested Index

Ein großer Vorteil des Nested Index [BK89] besteht darin, daß er mit Hilfe eines einfachen B*-Baums implementiert werden kann. Er ermöglicht eine ausgesprochen schnelle Selektion von Objekten anhand von Eigenschaften, die sonst nur durch umfangreiche Verbundoperationen abzufragen wären. Ein breiter Einsatz dieser Technik wird nur durch das sehr ineffiziente Verhalten bei Änderungsoperationen verhindert. So müssen beispielsweise bei Änderungen einer Referenz einer inneren Klasse deren Auswirkung auf den Index überprüft werden. Da in einem Indexeintrag selbst keine Informationen darüber enthalten sind, auf welchem Pfad von Instanzen eine konkrete Verknüpfung zustande gekommen ist, müssen – ausgehend vom veränderten Objekt – alle möglichen Instanzpfade abgeschritten werden, an denen dieses Objekt beteiligt ist. Erst so ist es möglich, die relevanten Indexeinträge –

die durch Indexschlüssel und Zielobjekt-ID eindeutig bestimmt werden können – zu identifizieren und anzupassen. Entsprechend viele Dereferenzierungen sind notwendig, wobei eine Unterstützung durch Hilfsindexe möglich ist. Allerdings müssen letztere natürlich dann bei Änderungsoperationen ebenfalls aktualisiert werden.

Eine etwas genauere Analyse der Einsetzbarkeit von Nested Indexes mit Blick auf objektrelationale DBMS wird im Abschnitt 4.2 gegeben. Schon hier läßt sich aber sagen, daß ein praktischer Einsatz für längere Pfade beziehungsweise durchschnittlich dynamische Umgebungen quasi nicht möglich ist. Sinnvoll hingegen kann die Anwendung des Index bei sehr kurzen Pfaden (zum Beispiel zwei Stufen) und gleichzeitig relativ statischem Datenprofil sein.

Ein weiterer Nachteil des Nested Index ist, daß nur Prädikate ausgewertet werden können, die den Gesamtpfad betreffen. Für die Auswertung von Ausdrücken, bei denen Objekte selektiert werden sollen, die sich innerhalb des Pfades befinden, sind weitere unabhängige Nested Indexes notwendig. Eventuell ist es dabei jedoch möglich, einzelne Hilfsindexe mehrfach zu nutzen.

2. Pfadindex

Bei Pfadindexen [BK89] werden in einem Eintrag zu einem konkreten Instanzpfad neben der Zielklasse auch die Identifikatoren aller inneren Objekte dieses Pfades gespeichert. Im direkten Vergleich mit einem analogen Nested Index sind daher die Indexeinträge hier wesentlich länger, was einen größeren Index und somit langsamere Zugriffszeiten mit sich bringt. Insbesondere steigt durch die längeren Verweislisten an einem Schlüsselwert der Verwaltungsaufwand.

Hauptvorteil des Pfadindexes gegenüber dem Nested Index ist sein weitaus flexiblerer Einsatz. So lassen sich über die Werte des betreffenden Attributs aus jeder Klasse entlang des indexierten Pfades die entsprechenden Objekte selektieren. Änderungen von Objekten, die innerhalb des indexierten Pfades liegen, haben ähnlich umfangreiche Auswirkungen wie beim Nested Index. Auch hier müssen zunächst alle Pfade ausgehend vom geänderten Objekt in Richtung indexiertem Attribut evaluiert werden. Durch den höheren Informationsgehalt in den Indexen kann allerdings vermieden werden, daß die Pfade auch in die andere Richtung verfolgt werden müssen.

Trotz dieser Verbesserung scheitert ein breiterer Einsatz an den immer noch sehr hohen Kosten bei Änderungsoptionen. Für Fälle, in denen die Zahl der Anfragen deutlich über denen der Änderungen liegt, kann jedoch der Vorteil der schnelleren Selektion stärkeres Gewicht bekommen als die Nachteile durch die schlechten Änderungszeiten.

3. Multiindex

Als Ausweg aus dem Problem der hohen Kosten bei Änderungsoperationen wurde ein dritter Indextyp, der Multiindex propagiert [BK89]. Bei diesem wird pro Stufe innerhalb des Pfades ein einzelner Index angelegt. Bis auf den Index, der das eigentliche Attribut indexiert, handelt es sich um sogenannte Identitätsindexe, also Indexe, deren Schlüssel Objekt-IDs sind.

Wie auch bei Pfadindexen können Multiindexe sehr flexibel eingesetzt werden. Durch das Indexieren weiterer Attribute beliebiger Klassen des Pfades ist es möglich, einzelne Indexe mehrfach zu nutzen; sowohl beim Nested Index als auch beim Multiindexen wären dafür separate Indexe notwendig. Der nächste große Vorteil von Multiindexen besteht in

den günstigen Kosten bei Änderungsoperationen. Wird ein Objekt innerhalb des Pfades verändert, reicht eine lokale Änderung des entsprechenden Index aus.

Im Gegensatz zu Nested Index und Pfadindex sind bei dem Multiindex pro Abfrage jedoch mehrere Indexzugriffe notwendig, was die Gesamtleistung stark reduziert. Es gibt jedoch eine Anzahl von Einsatzfällen, in denen die Vorteile von schnellen Änderungen die Nachteile der langsameren Abfrage überwiegen [MS86, KKD89, Deu90].

Aufbauend auf Multi- und Verbundindexe wurde 1994 von Xie et al. das Konzept der Verbundindex-Hierarchien (Join Index Hierarchy) vorgestellt [XH94]. Entlang eines Referenzpfades werden für ausgewählte Teilpfade Verbundindexe angelegt. Diese bilden, bezogen auf die Start- und Endposition des indexierten Teilpfades, eine Hierarchie von Indexen. Verbundabfragen entlang des Pfades können je nach Verfügbarkeit durch Kombination von Indexen beantwortet werden.

4.1.2.4 Zugriffsrelationen

Eine Erweiterung der pfadbezogenen Indexe erfolgte in zwei Richtungen. In der ersten fand durch Kemper und Moerkotte [KM90] eine Klassifikation der Pfadindexe statt. Dabei wurde ein Pfadindex formal zunächst als Relation betrachtet, wobei pro Stufe des Pfades eine Spalte in der Relation vorgesehen ist. Ein Tupel der Relation repräsentiert dann einen konkreten Pfad aus Objektinstanzen.

Da auch Null-Werte in der Relation zugelassen wurden, können Indexe hinsichtlich ihrer Vollständigkeit in vier Klassen – sogenannte Extensionen – eingeteilt werden:

- ▷ *Kanonische Extension*
  Der Index enthält nur Information über vollständige Pfade und entspricht damit dem Resultat des inneren Verbundes (Inner Join).

- ▷ *Links-vollständige Extension*
  Der Index enthält zusätzlich Information über Pfade, die in der Startklasse des Pfades beginnen, ohne daß ein vollständiger Pfad bis zu dem Attribut am Ende des Pfades existiert und entspricht somit dem Ergebnis eines linken äußeren Verbundes (Left Outer Join).

- ▷ *Rechts-vollständige Extension*
  Neben Informationen über vollständige Pfade werden auch Teilpfade gespeichert, die zwar das Ende des Pfades beinhalten, jedoch von keinem Pfad ausgehend von der ersten Klasse des Pfades erreicht werden können. Die Indexeinträge entsprechen in diesem Fall dem Ergebnis eines rechten äußeren Verbundes (Right Outer Join).

- ▷ *Vollständige Extension*
  Dieser Index enthält jede Art von Teilpfaden, also auch solche, die weder den Anfang noch das Ende des Pfades einschließen. Damit entspricht er dem Berechnungsresultat eines vollständigen äußeren Verbundes (Full Outer Join).

Wie auch bei einfachen Nested- und Pfadindexen sind bei dieser Art von Strukturen Änderungsoperationen nur unter Zuhilfenahme geeigneter Hilfsindexe in akzeptabler Zeit durchzuführen. Im Grunde muß jede Spalte – mindestens aber die erste sowie die letzte – der Relation mit einem eigenen Index versehen werden. Durch die zentrale Haltung des Index in einer einzigen Tabelle verbunden mit der Möglichkeit, viel mehr als nur vollständige

Pfade in ihm zu speichern, besteht im Gegensatz zu dem einfachen Pfadindex oder dem Multiindex die Gefahr eines Hot Spots.

Die zweite Richtung der Erweiterung der pfadbezogenen Indextypen bezieht das Konzept der in der OO-Welt gebräuchlichen Klassenhierarchien, einschließlich Vererbung in den Indexentwurf mit ein [CBBC94]. Daraus resultieren eine Reihe neuer Indextypen wie der Nested Inherited Index oder der Multi Inherited Index. Letztere beide sind Varianten des Multi beziehungsweise Nested Index, bei denen im Indexeintrag neben der Referenz auf ein Objekt zusätzlich Informationen zu der Klassenzugehörigkeit gespeichert sind.

Auf weitere Indexe speziell für Klassenhierarchien wird im folgenden Abschnitt eingegangen.

4.1.2.5 Indexe für Klassenhierarchien

Speziell für die attributbasierte Suche in OO-Datenbanken wurden 1989 und 1992 zwei spezielle Indexstrukturen vorgestellt. Beiden Ansätzen liegt das Problem zugrunde, aus einer Menge von Objekten einer Klassenhierarchie durch die Suche über ein gemeinsames, vererbtes Attribut zu selektieren. Die erste Lösung, der CH-Baum [KKD87], baut auf einem herkömmlichen B*-Baum auf, speichert – analog zur Generalized Access Path Structure – in seinen Blättern jedoch Verweise auf Objekte verschiedener Klassen. Es handelt sich hierbei also um einen Ansatz, der Objekte aus mehreren Mengen innerhalb der Zugriffsstruktur nach einem gemeinsamen Attributwert clustert. Ein anderer Ansatz, nämlich die Clusterung nach Klassenzugehörigkeit, wurde 1992 von Low et al. in Form des H-Baums präsentiert [LOL92]. Hier werden für jede Klasse separate Baumstrukturen erstellt, welche durchgängig miteinander verknüpft sind. Diese Verknüpfungen sind direkt an die zugrunde liegende Klassenhierarchie gebunden, so daß der Index einer übergeordneten Klasse ausschließlich Referenzen auf Indexe untergeordneter Klassen beinhaltet. Durch zusätzliche Regeln bei der Erstellung und Pflege dieser Querverweise wird gewährleistet, daß alle Anfragen, ausgehend von dem am höchsten gelegenen Index, korrekt und vollständig beantwortet werden können.

Der Vorteil gegenüber dem CH-Baum liegt in der schnellen Beantwortung von Anfragen, die nur eine Klasse innerhalb der Objekthierarchie betreffen. Auf der anderen Seite sind in der Regel mehr Seitenzugriffe bei einer Abfrage auf der gesamten Hierarchie notwendig.

Um die Vorteile dieser beiden Strukturen zu vereinen, wurde 1994 von Sreenath der hcC-Baum eingeführt [SS94]. Diesem liegt ein Generalized Access Path Index zugrunde, dessen Blätter pro Klasse einen Verweis auf einen externen Verzeichnissatz beinhalten. In diesem genau einer Klasse zugeordneten Verzeichnissatz sind die Referenzen auf die eigentlichen Objekte gespeichert. Alle Verzeichnissätze, die zu einer Klasse gehören, sind des weiteren verknüpft. Darüber hinaus gibt es von jedem Schlüsseleintrag einen Verweis auf eine zusätzliche Kette von Verzeichnissätzen, die klassenübergreifend Referenzen auf alle relevanten Objekte dieses Eintrags beinhalten.

Mit dieser Struktur sind sowohl effiziente klassenbezogene als auch klassenübergreifende Abfragen möglich.

Die gleiche Struktur verwenden Klinger und Moerkotte in ihrem CG-Baum [KM94]. Einziger Unterschied zum hcC-Baum ist das Fehlen der klassenübergreifenden Verzeichnissatzkette, was den Nachteil hat, daß bei einer Abfrage über den Objekten aller Klassen mehrere Verzeichnissätze geladen werden müssen.

4.1.2.6 Übersicht

TABELLE 4.1 gibt einen Überblick über die vorgestellten Indexstrukturen und stellt ihre wesentlichen Eigenschaften sowie Vor- und Nachteile vor.

TABELLE 4.1: Erweiterte Indexstrukturen

Index	Beschreibung	Vor-/Nachteile
General Access Path	▷ B*-Baum-Index über Attribute gleicher Domain aus verschiedenen Relationen ▷ Blatt enthält für einen Schlüsselwert pro beteiligter Relation eine Liste von TIDs	⊕ Einfach zu implementieren: basiert auf bekanntem B*-Baum ⊕ Einzelne Änderung billig ⊖ Hot Spot bei vielen beteiligten Relationen (Update)
Verbundindex	▷ Unterstützt Verbunde über zwei Relationen ▷ Schlüssel der beiden Relationen jeweils in B*-Baum-Index	⊕ Läßt sich einfach implementieren ⊕ Unterstützt effizient Verbund mit komplexen Verbundausdrücken ⊖ Änderungen der Verbundattribute in den ursprünglichen Relationen bedingen Änderungen im Verbundindex ⊖ Bei komplexen Verbundausdrücken Änderung sehr aufwendig
Nested Index	▷ Für über Referenzen verkettete Objekte (Referenzpfad) ▷ Die Objekte am Beginn der Kette werden über Attribute von Objekten am Ende der Kette indexiert ▷ In den Indexeinträgen werden nur OIDs zu den Objekten am Ende des Pfades gespeichert	⊕ Sehr schnelle Selektion von Objekten ohne (umfangreiche) Verbunde ⊕ Kompakter Index basierend auf bekanntem B*-Baum ⊖ Aktualisierung nach Änderungen unter Umständen sehr aufwendig ⊖ Für effiziente Änderungen Rückwärtsreferenzen notwendig

Index	**Beschreibung**	**Vor-/Nachteile**
Pfadindex	▷ Für über Referenzen verkettete Objekte (Referenzpfad) ▷ Die Objekte am Beginn der Kette werden über Attribute von Objekten am Ende der Kette indexiert ▷ In den Indexeinträgen werden OIDs zu allen Objekten entlang des Pfades gespeichert	⊕ Sehr schnelle Selektion ⊕ Ein Index ersetzt mehrere Nested Indexes ⊖ Aktualisierung nach Änderungen unter Umständen sehr aufwendig ⊖ Für effiziente Änderungen Rückwärtsreferenzen notwendig ⊖ Viel Information pro Indexeintrag ⊖ Gefahr von Hot-Spots
Multiindex	▷ Für über Referenzen verkettete Objekte (Referenzpfad) ▷ Die Objekte am Beginn der Kette werden über Attribute von Objekten am Ende der Kette indexiert ▷ Pro Stufe des Referenzpfades wird ein Index angelegt	⊕ Effiziente Änderungen ⊕ Vielseitig einsetzbar ⊕ Basiert auf einfachem B*-Baum-Index ⊖ Suche langsamer als bei Nested beziehungsweise Pfad Index, da viele Indexzugriffe notwendig
Access Support Relation	▷ Ähnlich wie Pfadindex ▷ Indexinformationen werden in einer Relation gespeichert ▷ Unterscheidung in vier Extensionen	⊕ Sehr flexibel ⊖ Effizient erst durch viele Indexe auf der Relation ⇒ im Extremfall Multiindex ⊖ Gefahr von Hot-Spots
Verbundindex-Hierarchie	▷ Ausgewählte Verbundindexe entlang eines Pfades	⊕ Relativ flexibel ⊕ Auswirkungen von Änderungen gut kontrollierbar ⊖ Nicht so effizient wie Nested Index
CH-Baum	▷ Index über ein gemeinsames Attribut mehrerer Klassen einer Klassenhierarchie ▷ Mehrere Indexeinträge wie bei General Access Path ▷ Ermöglicht Zugriff auf alle Objekte innerhalb einer Hierarchie anhand eines Attributs	⊕ Basiert auf B*-Baum-Prinzip: einfach zu implementieren ⊕ Schnelle klassenübergreifende Abfrage ⊕ Effiziente Änderungen ⊖ Abfrage von einzelnen Klassen langsamer, da Baum nicht nach Klassenzugehörigkeit trennt ⊖ Gefahr von Hot-Spots

Index	Beschreibung	Vor-/Nachteile
H-Baum	▷ Index über ein Attribut mehrerer Klassen einer Klassenhierarchie ▷ Pro Klasse ein einzelner Index ▷ Indexe sind analog der zugrunde liegenden Klassenhierarchie miteinander verknüpft	⊕ Schnelle klassenspezifische Abfrage ⊕ Klassenübergreifende Abfrage schneller als nicht verknüpfte einzelne Indexe ⊖ Klassenübergreifende Abfrage langsamer als CH-Index ⊖ Relativ schwierige und kostenintensive Änderungen
hcC-Baum	▷ Erweiterter H-Baum ▷ Faßt Objektreferenzen auf Blattebene sowohl klassenbezogen als auch klassenübergreifend über spezielle Verzeichnissätze zusammen	⊕ Sowohl in klassenbezogenen als auch klassenübergreifenden Abfragen effizient ⊖ Viel Redundanz, Änderungen teuer
CG-Baum	▷ Ähnlich hcC-Baum	⊕ Sowohl in klassenbezogenen als auch klassenübergreifenden Abfragen effizient ⊕ Weniger Redundanz als hcC-Baum ⊖ Komplizierte Wartung
Nested Inherited Index	▷ Nested Index unter Berücksichtigung von Klassenhierarchien	⊕ Sehr schnelle Selektion von Objekten ohne (umfangreiche) Verbunde ⊖ Aktualisierung nach Änderungen in der Regel sehr aufwendig
Inherited Multiindex	▷ Multiindex unter Berücksichtigung von Klassenhierarchien	⊕ Leichter wartbar als Nested Index

4.1.3 Operationsunterstützung auf Kollektionen

Im folgenden werden einige Strukturen vorgestellt, die speziell für die Unterstützung der in TABELLE 2.1 aufgeführten Operationen auf Kollektionen konzipiert sind. Dazu gehört zum einen die große Gruppe der Strukturen, die auf signaturbasierte Verfahren zurückgreifen. Zum anderen wurden in unterschiedlichen Kontexten verschiedene Spezialindexe entwickelt, die sich mit der Speicherung beziehungsweise dem Zugriff auf eine Kollektion von Kollektionen beschäftigen. ABBILDUNG 4.4 zeigt eine Übersicht dieser Strukturen. Zusätzlich mit aufgeführt sind verschiedene grundlegende Indexstrukturen, von denen Techniken für die erweiterten Indexformen übernommen wurden. ABBILDUNG 4.5 skizziert diese Zusammenhänge speziell für Indexe zur Operationsunterstützung auf Listen.

Zunächst werden allgemeine Techniken und Ansätze zur Berechnung von Signaturen vorgestellt.

ABBILDUNG 4.4: Indexstrukturen zur Unterstützung von Operationen auf Mengen

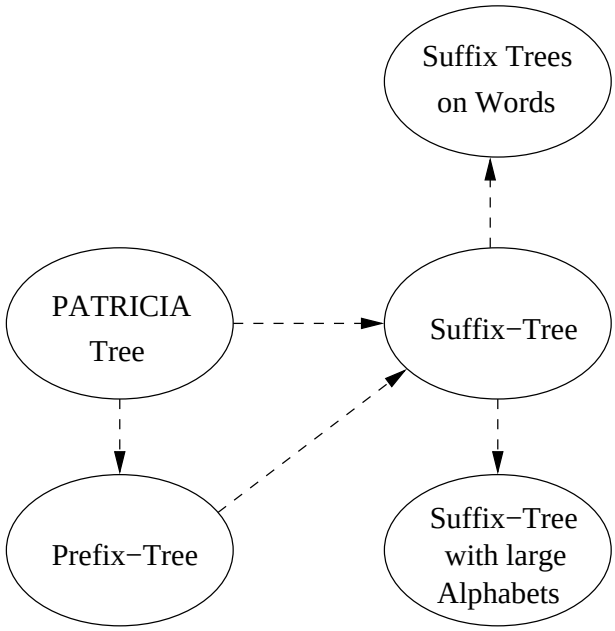


ABBILDUNG 4.5: Indexstrukturen zur Unterstützung von Operationen auf Listen

4.1.3.1 Verfahren zur Bildung von Signaturen

Es können zwei Arten von Signaturen unterschieden werden:

- ▷ *Exakte oder verlustfreie Signaturen*
  als eine vollständige, verlustfreie Darstellung von Daten

- ▷ *Approximative oder verlustbehaftete Signaturen*
  als ein mit Informationsverlust behafteter Hash-Schlüssel von Daten

Unter den in ABBILDUNG 4.4 aufgeführten Verfahren sind auch vier verlustfreie Signaturbildungen zu finden: Perfect Encoding [DLM96], Exact Reversible Signature Files [DLM96], S-Index [DLM97] und der Hierarchical Bitmap Index [MMNM03]. Beim Perfect Encoding wird auf der Basis aller potenziell möglichen Elemente eine sortierte Liste mit allen Mengen einer festgelegten Größe durchnummeriert. Über diese Nummerierung wird die Zielmenge kodiert und indexiert. Das Verfahren hat den Nachteil, daß die Berechnung der Signatur sehr kostenaufwendig ist.

Exact Reversible Signature Files bilden Mengen auf Bitmaps ab und sind damit zwar schneller, verbrauchen jedoch viel Speicherpatz pro Signatur. Ein Kompromiß zwischen diesen beiden Ansätzen wurde unter der Bezeichnung S-Index vorgestellt; diese Zugriffsstruktur konnte sich jedoch für einen praktischen Einsatz nicht durchsetzen.

Eine weitere Technik der verlustfreien Codierung mit Bitfeldern ist der hierarchische Bitmap-Index.

Sehr wichtig ist ebenfalls die Gruppe der Signaturverfahren, in der Signaturen im Sinne einer verlustbehafteten Hash-Abbildung aufgebaut werden. Signaturen werden in diesem Zusammenhang beispielsweise für beschleunigte Vergleichsoperationen eingesetzt. Kennzeichnend für diese Konzepte ist, daß bei einem positiven Ergebnis des Vergleichs zweier

Signaturen (‚Drop'[1]) immer noch eine Verifizierung anhand der Originaldaten erfolgen muß. Je nach dem, wie dieser endgültige Vergleich dann ausfällt, spricht man von einem ‚False Drop' beziehungsweise einem ‚Right Drop'.

1. Bildung von Signaturen mittels Superimposed Coding

Die zentrale Technik zur Erstellung von (in der Regel approximativen) Signaturen nennt sich Superimposed Coding. Sie basiert auf dem Prinzip, daß eine Signatur für ein Objekt, ausgehend von einem mit Nullen gefüllten Bitvektor bestimmter Länge, berechnet wird. Für jedes Attribut des Objekts wird eine schwach mit Einsen besetzte Signatur gleicher Länge angelegt. Durch bitweise OR-Verknüpfung dieser Signaturen wird die Signatur des gesamten Objekts erstellt [Knu97]. Wichtig ist, daß jede Signatur der Attribute die gleiche Anzahl von Einsen im Bitvektor hat.

Diese Technik läßt sich sowohl für Attribute einer Struktur als auch für die Elemente einer Kollektion anwenden. In letzterem Fall lassen sich die Signaturen zweier Mengen für Gleichheits-, Superset-, Subset- und Überschneidungstests einsetzen.

Veränderbare Parameter des Algorithmus sind

- das Verfahren zur Erstellung der Signaturen für die Attribute,
- die Gesamtlänge des Bitvektors der Signatur sowie
- die Anzahl der Stellen, die pro Attribut in der Signatur mit Einsen besetzt werden.

Das wichtigste Optimierungskriterium beim approximativen Superimposed Coding ist die Senkung der False-Drop-Rate[2]. Entsprechnde Techniken wurden in der Literatur bereits ausführlich diskutiert [FC84, IKO93, KFIO93].

2. Hierarchical Bitmap Index

Dieses Verfahren ist darauf ausgelegt, sehr große, schwach besetzte Bitmaps, die zur Repräsentation von Mengen eingesetzt werden, verlustfrei komprimiert abzuspeichern.

Eine Bitmap besteht dabei aus so vielen Stellen, wie es unterschiedliche Elemente geben kann. In der Darstellung einer konkreten Menge ist dann an den entsprechenden Stellen der Bitmap das Bit auf 1 gesetzt. Beispielsweise wird für Mengen von 2-Byte-Integer-Zahlen ein Bitfeld aus 65536 Bits benötigt.

In der Literatur sind eine Reihe von Verfahren zur Repräsentation von Bitlisten und zur Unterstützung der Anfragebearbeitung durch diese zu finden. Als frühe Arbeiten seien hier [Här75b, Här75a] genannt.

Hier soll jedoch auf einen neueren Vertreter dieser Ansätze eingegangen werden: auf den hierarchischen Bitmap-Index (Hierarchical Bitmap Index), der in [MMNM03] vorgestellt wurde. Sein Aufbau ist in ABBILDUNG 4.6 dargestellt. Er gründet auf der Voraussetzung, daß sich in konkreten Mengen relativ zu der Anzahl potenziellen Elemente sehr wenige wirklich vorhandene Elemente befinden. Die entsprechenden Bitmaps bestehen also zum größten Teil aus Nullen.

In einem ersten Schritt wird die n Bit lange Bitmap in kurze, gleich große Abschnitte der Länge l aufgeteilt. Jedem Abschnitt mit mindestens einer gesetzten 1 wird eine 1 zugeordnet, jedem Abschnitt, der nur aus Nullen besteht, eine 0. Die aus diesem Prozeß

[1] Bei positivem Vergleichsergebnis wird in der Literatur eigenartigerweise im Zusammenhang mit Signaturen von ‚Drops' gesprochen, im Zusammenhang mit Zeichenkettenvergleichen dagegen in der Regel von ‚Matches'.

[2] Anteil der Fälle in denen zwar der approximative Signaturvergleich positiv aber der exakte Datenvergleich negativ verläuft.

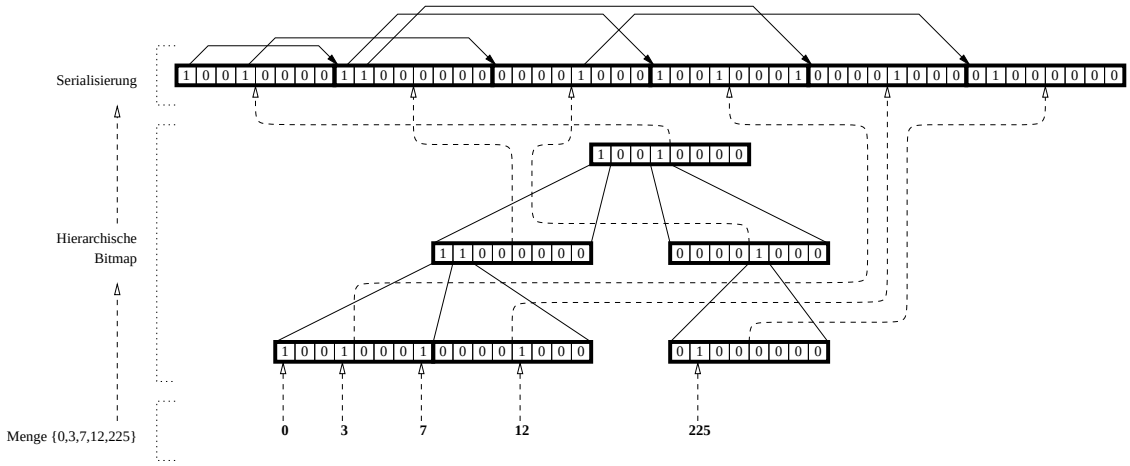


ABBILDUNG 4.6: Hierarchical Bitmap Index

resultierende Bitmap der Länge $\frac{n}{l}$ wird wiederum in Abschnitte der Länge l geteilt und rekursiv auf die eben beschriebene Methode behandelt. Das Verfahren stoppt, wenn nur noch ein Abschnitt übrigbleibt.

Der eben beschriebene Algorithmus konstruiert eine Baumstruktur. Die Blätter sind die Abschnitte der Anfangs-Bitmap. In den inneren Knoten werden jeweils l Abschnitte zu einem zusammengefaßt. Um ausgehend von der Wurzel des Baums alle auf 1 gesetzten Bits der Gesamtbitmap zu finden, müssen von einem inneren Knoten aus nur die Unterbäume betrachtet werden, denen eine 1 zugeordnet ist.

Diese Tatsache ermöglicht eine sehr kompakte Speicherung der Baumstruktur. Beginnend mit der Wurzel müssen nur die Abschnitte (bzw. Blätter) des Baums abgespeichert werden, die mindestens eine 1 beinhalten. Die Abarbeitung der Speicherung erfolgt in Level-Order und erfordert keine Speicherung von Zusatzinformationen zur Verknüpfung der Baumknoten, da alle benötigten Informationen aus den Bitlisten der Knoten einfach berechnet werden können. Mit Hinblick auf diese Speicherform können hierarchische Bitmap-Indexe nicht nur als Index, sondern auch als kompakte Repräsentation von dünnbesetzten Mengen betrachtet werden. Insofern könnte man auch von Hierarchical Bitmap oder Hierarchical Bitmap Representation sprechen, was sich jedoch (bisher) nicht eingebürgert hat.

Die Wurzeln von hierarchischen Bitmap-Indexen können als Signaturen betrachtet werden. Mit dem im nächsten Kapitel vorgestellten signaturbasierten Verfahren ist eine effiziente Verwaltung einer Menge von hierarchischen Bitmap-Indexen möglich.

4.1.3.2 Signaturbasierte Indexe

Es haben sich im wesentlichen drei unterschiedliche Ansätze zur Speicherung und Verwaltung von einer Menge von Signaturen durchgesetzt. Die Signaturen werden entweder flach hintereinander gespeichert, in hierarchischen Baumstrukturen verwaltet oder über Hashtabellen indexiert. Ausführliche Vergleiche dieser drei Ansätze finden sich zum Beispiel in [Hel97] oder [HM99].

1. Signaturfile

Diese Struktur ist die einfachste Form, mehrere Signaturen zu verwalten. Die Signaturen werden zusammen mit den Referenzen auf die entsprechenden Objekte einfach hinterein-

ander, gegebenenfalls geordnet abgespeichert [LL92].

Für einen schnelleren Zugriff wurden verschiedene mehrstufige Verfahren entwickelt, bei denen über Bitoperationen mehrere Signaturen zu einer zusammengefaßt werden (Multi Level Signature File) [PBC80, CS89]. Dieser Ansatz eignete sich allerdings nur für relativ statische Daten.

2. S-Baum

1986 wurde von Deppisch der S-Baum (S-Tree bzw. Signature Tree) vorgestellt [Dep86]. Ähnlich wie bei den mehrstufigen Verfahren werden Signaturen mit Bitoperationen zusammengefaßt und in einer Hierarchie organisiert. Da allerdings eine dynamische Balancierung dieser Struktur nach dem Prinzip von B*-Bäumen erfolgt, eignen sich S-Bäume auch für den nicht-statischen Kontext und umfangreiche Datenbestände.

Neben Abfragen auf Gleichheit sind mit S-Bäumen auch Subset-Abfragen möglich.[3]

3. Hashverfahren

Eine weitere Gruppe signaturbasierter Strukturen baut auf unterschiedlichen Hashverfahren auf. Ein erster Algorithmus, der aufgrund teilweiser Übereinstimmung aus einer Hash-Tabelle selektiert, wurde durch Otoo in [Oto84] präsentiert. Es folgten mehrere Varianten, die Superimposed Coding mit Hashindexen kombinierten [LL89], unter anderem auch mit linearem Hashing [ZRT91].

Da die bisher genannten Verfahren weder Super- noch Subset-Operationen unterstützen, wurden 1997 von Helmer das Extendible Signature Hashing und das Recursive Linear Signature Hashing vorgestellt. Beide Varianten unterstützen sowohl Super- als auch Subset-Operationen für Signaturen, allerdings keine Tests auf Überschneidung.

Extendible Signature Hashing basiert auf erweiterbarem Hashing, das 1979 als Variante des dynamischen Hashings vorgestellt wurde [FNPS79]. Bei diesem Verfahren wird – vereinfacht gesagt – die Größe des Hash-Verzeichnisses dynamisch an die Anzahl der gespeicherten Elemente angepaßt. Gleichzeitig werden gegebenenfalls mehrere Verzeichniseinträge einem einzelnen Bucket zugeordnet.

Auch Recursive Linear Signature Hashing ist eine an Signaturen angepaßte Variante des dynamischen Hashings (aufbauend auf [RSD84]). Die Überlaufbehandlung basiert auf der Annahme, daß sich neue Einträge relativ gleichmäßig über die bestehende Hashtabelle verteilen. Die Hashstruktur besteht aus einer Reihe von Hashtabellen, wobei eine Tabelle eines niedrigeren Levels immer etwa halb so groß ist wie eine der nächst höheren. Paßt ein Element nicht mehr in die größte Haupttabelle, werden der Reihe nach alle Untertabellen abgearbeitet, um einen freien Platz zu finden. Gleichzeitig wird pro Tabelle ein Zähler verwaltet, der nach einer bestimmten Anzahl eingefügter Elemente Bucket-Splits durchführt. Sind alle zu einem bestimmten Schlüssel gehörenden Buckets aller Tabellen besetzt, wird ein Tabellenlevel hinzugefügt.

Bei den an Signaturen angepaßten Varianten dieser beiden Verfahren werden Signaturen als Hashschlüssel verwendet. Mit Hilfe spezieller Algorithmen sind neben Gleichheitsabfragen auch Super- und Subset-Abfragen möglich. Die Vorteile dieser beiden Strukturen sind damit ihre Universalität bezüglich möglicher Abfragetypen und sehr schnelle Gleichheitsabfragen. Nachteilig ist, daß Sub- und Superset-Abfrage im Vergleich dazu relativ langsam

[3] Als Subset-Abfragen werden bei gegebener Suchmenge Q und Elementmenge O Abfragen der Form $\{o_i \in O | Q \subseteq o_i\}$ bezeichnet. Bei $\{o_i \in O | Q \supseteq o_i\}$ handelt es sich um Superset-Abfragen.

sind. Des weiteren entarten beide Varianten bei ungünstiger Datenlage, Recursive Linear Signature Hashing etwas stärker als das erweiterbare Hashing [Hel97].

4.1.3.3 Spezielle Indexe

Neben signaturbasierten Strukturen gibt es auch eine Reihe weiterer Entwicklungen zur schnellen Abfrage einer Menge von Kollektionen. Neben zwei Baumstrukturen wird abschließend ein Verfahren zur Unterstützung von Abfragen auf Listen vorgestellt.

1. Invertierter Index

Als eine sehr einfache Form zur Suche in einer Menge von Mengen wurde von Inglis 1974 der Inverted Index eingeführt [Ing74]. Mit einem B*-Baum oder einem vergleichbaren Index wird mengenübergreifend ein Attribut der Elemente indexiert. Die Indexeinträge enthalten Referenzen auf die jeweiligen Mengen, die ein entsprechendes Element besitzen.

Die Hauptvorteile des invertierten Indexes sind seine einfachen und flexiblen Umsetzungsmöglichkeiten. Auch zeichnet er sich durch ein schnelles Abfrageverhalten aus. Dagegen eignet sich diese Zugriffsstruktur schlecht für die Indexierung sehr vieler Mengen mit wenigen unterschiedlichen Elementen.

2. RD-Baum

RD-Bäume (‚Russian Doll Trees') [HP94] bauen auf R-Bäumen (R-Trees bzw. Range Trees) [Gut84] auf und sind analog zu diesen aufgebaut. Blätter eines R-Baumes enthalten je ein Objekt und die Angabe des minimalen n-dimensionalen Rechtecks, das dieses Objekt umfaßt. Innere Knoten enthalten eine Reihe von Zeigern auf Kind-Knoten sowie ein minimales Rechteck, das alle Rechtecke der Kinder einschließt.

Es gibt verschiedene Verfahren, diese Technik auf Mengen von Mengen zu übertragen. In dem einfachen Fall, daß eine Menge von Integer-Mengen verwaltet werden soll, könnte anstelle des minimal umfassenden Rechtecks die Menge selbst genutzt werden. In diesem Fall würde in inneren Knoten die Vereinigungsmenge der Mengen aller Kind-Knoten gespeichert werden.

Die so entstandene Struktur eignet sich gut für Superset-Abfragen, aber nicht zu Ermittlung von Subsets. Durch das Invertieren der Funktion zur Berechnung des umfassenden Rechtecks (Schnitt- anstelle von Vereinigungsmenge) entsteht jedoch ein Baum, mit dem dann Subset-Abfragen durchgeführt werden können.

Ein Nachteil dieser Struktur ist, daß die Verwaltungs- und Abfragekosten bei der Indexierung einer großen Menge von Mengen hoch sind.

3. Suffix-Baum

Der Suffix-Baum (Suffix Tree) dient der Suche nach Teillisten innerhalb einer Liste von Elementen [Wei73]. Für den Aufbau der Struktur werden alle möglichen Suffixe der Liste gebildet (von jedem Element aus eine bis zum Ende der Liste gehende Teilliste). Anschließend werden diese Suffixe in einen Präfix-Baum [YM01] oder PATRICIA-Baum [Mor68] eingefügt.

PATRICIA-Bäume dienen der komprimierten Speicherung einer Menge von Präfixen. Ausgehend von einer gemeinsamen Wurzel werden die Präfixe abgelaufen. Unterscheidet sich ein Element an einer bestimmten Stelle von den restlichen Präfixen, zweigt ein neuer

Unterbaum ab. Kennzeichnend für PATRICIA-Bäume ist, daß nur Knoten mit Abzweigungen gespeichert werden und in den inneren Knoten außer der Anzahl keine Informationen zu den Elementen zwischen den inneren Knoten existieren. Präfix-Bäume unterscheiden sich von PATRICIA-Bäumen dadurch, daß bei ersteren in inneren Knoten zusätzlich alle Elemente seit dem letzten Knoten gespeichert werden.

Suffix-Bäume können effizient auf Teillisten abgefragt werden. Zusätzlich zu den Schlüsselwerten der Elemente befinden sich in den Knoten Referenzen auf die entsprechenden Elemente der Liste. Ein Nachteil dieser Struktur ist ihre umfangreiche Größe bei Listen mit vielen unterschiedlichen Elementen.

Ursprünglich wurden Suffix-Bäume für eine einzelne Liste von Elementen konzipiert. Durch das Abspeichern weiterer Informationen bei den Referenzen ist es denkbar, einen einzelnen Suffix-Baum für eine Menge von Listen zu erstellen.

4.1.3.4 Übersicht der vorgestellten Techniken

Tabelle 4.3 zeigt eine Übersicht der in diesem Abschnitt angesprochenen Indexe. Neben einer kurzen Beschreibung ihrer wesentlichen Eigenschaften und den von ihnen unterstützten Operationen sowie der Gegenüberstellung ihrer Vor- und Nachteile werden die einzelnen Strukturen auf einen möglichen Einsatz in ORDBMS hin beurteilt.

Tabelle 4.3: Techniken zur Unterstützung von Operationen auf Kollektionen

$\subseteq$	Subset		
$\supseteq$	Superset	$\oplus\oplus$	Volle Unterstützung
$=$	Gleichheit	$\oplus$	Eingeschränkte Unterstützung
$\bigcirc\!\!\!\!\bigcirc$	Test auf Überschneidung	—	Für keine Unterstützung

Index / Hilfs-struktur	**Beschreibung, Vor-/Nachteile, Einsatz in ORDBMS**	$\subseteq$	$\supseteq$	$=$	$\bigcirc\!\!\!\!\bigcirc$
Signatur	**Beschreibung** ▷ Approximative Kurzrepräsentation für Mengen oder komplexe Objekte ▷ Meist Bitlisten **Vor-/Nachteile** $\oplus$ Schnelle Berechnung $\oplus$ Eignung für Mengenprädikate $\ominus$ Prädikatauswertung approximativ $\ominus$ Verlustbehaftet **Einsatz in ORDBMS** ▷ einzelne Signatur als alternative Repräsentation lokal pro Menge	$\oplus\oplus$	$\oplus\oplus$	$\oplus\oplus$	$\oplus\oplus$

Index / Hilfs-struktur	**Beschreibung, Vor-/Nachteile, Einsatz in ORDBMS**	$\subseteq$	$\supseteq$	$=$	$\cap\!\!\cap$
Signatur-File	**Beschreibung** ▷ Flache Auflistung der Signaturen aller Mengen ▷ Unscharfer Index (Vorfilterung) **Vor-/Nachteile** ⊕ Sehr einfach zu implementieren ⊕ Einfügen in O(1), wenn nicht sortiert ⊕ Wenig Speicherplatz ⊖ O(n) für Suche und Löschen **Einsatz in ORDBMS** ▷ Index über Signaturen, gut für Äquivalenzabfragen	⊕⊕	⊕⊕	⊕⊕	⊕⊕
Multilevel Signature File	**Beschreibung** ▷ Mehrstufige Organisation der Signaturen (OR-Verknüpfung) **Vor-/Nachteile** ⊕ Schneller als sequenzielle Signatur-Files ⊕ Gut für relativ statische Daten ⊖ Schlecht bei änderungsintensiven Daten **Einsatz in ORDBMS** ▷ Kein Einsatz, da schlecht in dynamischen Umgebungen	⊕⊕	—	⊕⊕	—
S-Baum	**Beschreibung** ▷ Ähnlich wie Multilevel Signatur-Files ▷ Beruht auf balancierter Baumstruktur **Vor-/Nachteile** ⊕ Schnelles Suchen ⊕ Gut bei dynamischen Daten ⊕ Basiert auf bekanntem B*-Baum **Einsatz in ORDBMS** ▷ Index über Menge von Mengen	⊕⊕	—	⊕⊕	—

Index / Hilfs-struktur	**Beschreibung, Vor-/Nachteile, Einsatz in ORDBMS**	$\subseteq$	$\supseteq$	$=$	$\bigcirc\!\!\bigcirc$
Hierarchical Bitmap Index	**Beschreibung** ▷ Kompakte, verlustfreie Repräsentation von einzelnen, dünnbesetzten Mengen ▷ Mengen-Elemente werden eindeutig auf ganze Zahlen abgebildet ▷ Pro Menge wird eine große Bitmap gebildet, das in hierarchischer Form abgespeichert und darurch komprimiert wird **Vor-/Nachteile** ⊕ Keine Nachbearbeitung bei Abfragen notwendig ⊕ Hoher Komprimierungsgrad ⊕ Navigation durch einfache (Bit-)Operationen ⊕ Gut für viele kleine Mengen bei vielen unterschiedlichen Elementen ⊖ Hoher Speicherplatzbedarf bei größeren Mengen und weit gestreuter Verteilung der Elemente **Einsatz in ORDBMS** ▷ Einzelne hierarchische Signaturen als alternative Repräsentation lokal pro Menge ▷ S-Baum oder ähnliches indexiert Wurzeln mehrerer Strukturen	⊕⊕	⊕	⊕⊕	⊕

Index / Hilfs-struktur	**Beschreibung, Vor-/Nachteile, Einsatz in ORDBMS**	$\subseteq$	$\supseteq$	$=$	$\cap\!\!\cap$
RD-Baum	**Beschreibung** ▷ Abwandlung des R-Baumes ▷ Baumstruktur über Menge von Mengen ▷ Pro Blatt eine Menge ▷ Innere Knoten enthalten Informationen über alle vorkommenden Elemente im untergeordneten Teilbaum (Subset-Abfrage möglich) ▷ Invertierte Variante: innere Knoten enthalten Informationen über Elemente, die in allen Mengen des untergeordneten Teilbaums vorkommen (Superset-Abfragen möglich) **Vor-/Nachteile** ⊕ Basiert auf bekanntem R-Baum ⊕ Gut für kleine Anzahl von Mengen ⊕ Schnelles und einfaches Suchen bei wenig Mengen ⊖ Skaliert schlecht: Leistungseinbußen bei vielen Mengen, Degenerierung ⊖ Für Sub- und Superset-Anfragen zwei beziehungsweise kombinierte Baumstrukturen notwendig **Einsatz in ORDBMS** ▷ Lokaler Index auf kleiner Menge von Mengen	⊕⊕	⊕	—	⊕⊕
PE: Perfect Encoding	**Beschreibung** ▷ Struktur zur Textsuche auf Wortbasis; bei sortierten Mengen: Wörter $\hat{=}$ Elemente ▷ Suchstruktur für einzelne große Mengen **Vor-/Nachteile** ⊕ Keine Nachbearbeitung des Abfrageergebnisses notwendig ⊕ Hohe Kompression ⊖ Hoher CPU-Overhead ⊖ Nur für einzelne große Mengen **Einsatz in ORDBMS** ▷ Kein Einsatz, da zu langsam	⊕	—	⊕	⊕

Index / Hilfs-struktur	**Beschreibung, Vor-/Nachteile, Einsatz in ORDBMS**	$\subseteq$	$\supseteq$	$=$	$\cap\!\!\cap$
ERSF: Exact Reversible Signature Files	**Beschreibung** ▷ Große Bitmap mit einem Bit pro möglichem Element **Vor-/Nachteile** ⊕ Vollständige Repräsentation der Menge ⊕ Schnelle Auswertung ⊖ Nur für einzelne Mengen mit wenigen unterschiedlichen Elementen sinnvoll ⊖ Lange Signatur **Einsatz in ORDBMS** ▷ Kein Einsatz, da zu großer Speicher-Overhead	⊕⊕	⊕⊕	⊕⊕	⊕⊕
S-Index	**Beschreibung** ▷ Kombination aus PE und ERSF mittels Teile- und Herrsche Ansatz ▷ Baumstruktur mit verketteten Listen an jedem Knoten **Vor-/Nachteile** ⊕ Optimum zw. PE und ERSF ⊕ Keine Nachbearbeitung einer Abfrage notwendig ⊖ Kompliziert; basiert nicht auf Standardbäumen **Einsatz in ORDBMS** ▷ Kein Einsatz, da Implementierung zu komplex ist und nicht auf Standard-Strukturen basiert	⊕	—	⊕	⊕
Inverted Index	**Beschreibung** ▷ Mengenübergreifender B*-Baum: Indexschlüssel sind die Schlüssel der Elemente, Rückgabewerte einer Abfrage sind Verweise auf die Sätze der Kollektionen, die die entsprechenden Elemente beinhalten **Vor-/Nachteile** ⊕ Gut bei vielen unterschiedlichen Elementen ⊕ Keine Nachbearbeitung eines Abfrageergebnisses notwendig ⊖ Schlecht bei vielen Mengen und wenig unterschiedlichen Elementen **Einsatz in ORDBMS** ▷ Index über Menge von Mengen	⊕⊕	—	⊕⊕	—

Index / Hilfs-struktur	**Beschreibung, Vor-/Nachteile, Einsatz in ORDBMS**	$\subseteq$	$\supseteq$	$=$	$\cap\!\!\cap$
Extendible Signature Hashing	**Beschreibung** ▷ Index für eine Menge von Mengen ▷ Prinzip: ◇ Pro Menge Signatur ◇ Erweiterbares Hashing auf den Signaturen **Vor-/Nachteile** ⊕ Sehr gut bei $=$ ⊖ Vergleichsweise schlecht bei $\subset$ und $\supset$ ⊖ Entartet bei ungünstiger Datenlage **Einsatz in ORDBMS** ▷ Index für Menge von Mengen	⊕⊕	⊕⊕	⊕⊕	—
Recursive Linear Signature Hashing	**Beschreibung** ▷ Index über Menge von Mengen ▷ Prinzip: ◇ Pro Menge eine Signatur ◇ Hierarchie aus mehreren Hashtabellen, die nächste ist doppelt so groß wie die vorhergehende ◇ Elemente für volle Hash-Buckets werden in nächste Ebene ausgelagert **Vor-/Nachteile** ⊕ Sehr gut bei $=$ ⊖ Vergleichsweise schlecht bei $\subset$ und $\supset$ ⊖ Entartet bei ungünstiger Datenlage stärker als Extendible Signatur Hashing **Einsatz in ORDBMS** ▷ Index für eine Menge von Mengen	⊕⊕	⊕⊕	⊕⊕	—
Suffix-Baum	**Beschreibung** ▷ Aus allen Suffixen einer Liste wird ein Präfix oder PATRICIA-Baum erstellt **Vor-/Nachteile** ⊕ Schnelle Suche von Teillisten ($O(\|q\|)$)und Elementen ⊕ Aufbau in $O(n)$ (plus Sortieren) mit $O(n)$ Speicher **Einsatz in ORDBMS** ▷ Index für sortierte Kollektionen	⊕⊕	—	⊕⊕	—

4.2 Zugriffspfade zur Nutzung in ORDBMS

In diesem Abschnitt wird auf der Basis der im letzten Abschnitt vorgestellten Indexkonzepte aus Forschung und Produktentwicklung diskutiert, an welchen Stellen innerhalb von objektrelationalen DBMS ein Einsatz solcher oder davon abgeleiteter Zugriffsstrukturen sinnvoll und möglich ist.

Zuerst werden grundlegende Begriffe und Ziele definiert und inhaltlich abgegrenzt. Es schließt sich eine Diskussion der objektrelationalen Anforderungen an Zugriffsstrukturen an. Die Grundlage dazu bildet eine kurze Betrachtung der in relationalen Systemen vorhandenen Zugriffspfadkonzepte. Das Ziel dieser Diskussion liegt darin, eine einheitliche Sichtweise auf Zugriffsstrukturen für Kollektionen in objektrelationalen Systemen zu erreichen, was letztendlich eine geeignete Voraussetzung dafür ist, die gestiegene Komplexität handhaben zu können.

Darauf aufbauend werden die verschiedenen Freiheitsgrade dargestellt, die bei der Definition von Zugriffsstrukturen beachtet und angegeben werden müssen. Besondere Beachtung wird der Auswahl des Indexschlüssels und des Indextyps geschenkt.

4.2.1 Begriffsbestimmung und Abgrenzung

4.2.1.1 Zugriffsstruktur versus Index

Der Begriff ‚Zugriffsstruktur' wird von seiner Bedeutung her im folgenden relativ weit gefaßt, das heißt, daß beispielsweise auch eine Signatur als Zugriffsstruktur verstanden wird, obwohl mit einer Signatur kein gezielter Zugriff auf einzelne Elemente der Kollektion möglich ist. Vielmehr soll es sich bei einer Zugriffsstruktur um eine Konstruktion handeln, mit der ganz allgemein Informationen einer Kollektion schnell abgefragt werden können. Die Begriffe ‚Index' und ‚Zugriffsstruktur' werden im weiteren synonym benutzt.

4.2.1.2 Globale versus lokale Zugriffsstruktur

Bezieht sich eine Zugriffsstruktur auf alle (Teil-)Objekte einer Relation, spricht man von globalen Indexen bzw. Zugriffsstrukturen. In diesem Fall befindet sich der Einstiegspunkt zu dem Index direkt im Datenbankkatalog. Bei Zugriffsstrukturen für einzelne kollektionswertige Attribute wird hingegen beispielsweise für jedes Tabellentupel ein eigener ‚kleiner' Index über alle Elemente dieses Kollektionsattributs angelegt. Der Einstiegspunkt für diese lokalen Zugriffsstrukturen befindet sich an der Speicherposition des entsprechenden kollektionswertigen Attributs.

4.2.1.3 Benutzerdefinierte Indexe im objektrelationalen Umfeld

Unter dem Oberbegriff „objektrelationale Erweiterungen" werden in kommerziellen DBMS auch Mechanismen zur Definition sog. benutzerdefinierter Indexe bereitgestellt. Mit diesem Ansatz soll es möglich sein, für bestimmte Anwendungen und bestimmte Datenstrukturen (zum Beispiel zur Verarbeitung von räumlichen Daten; Stichwort: Spatial Data) spezielle Zugriffsstrukturen zu definieren, die eine schnelle Verarbeitung diverser Anfragen und Prädikate zum Ziel haben. Zu diesem Zweck stellen Datenbanksysteme wie Oracle, DB2 oder Informix Schnittstellen zur Verfügung, über die ein Anwender dem System neue Indexstrukturen bekannt machen bzw. definieren kann [DCC+01]. Diese nutzerdefinierten Indexstrukturen werden dann vom DBMS ähnlich wie die fest vorgegebenen Indexstrukturen (in der Regel B*-Baum) genutzt, indem über die vordefinierten Schnittstellen Änderungs-

und Anfrageoperationen aufgerufen werden. Abgesehen von dem oftmals sehr großen Aufwand einer guten Implementierung der notwendigen Funktionalität und der notwendigen Integration auf den Ebenen von Anfragesprache (nutzerdefinierte Prädikate), Anfrageoptimierung (Auswertungskosten und Selektivitäten nutzerdefinierter Prädikate) und Anfrageausführung (Auswertung nutzerdefinierter Prädikate durch nutzerdefinierte Indexe), zeigt es sich jedoch häufig als problematisch, daß das DBMS gegebenenfalls keine vollständige Kontrolle über die Daten- und Zugriffsstruktur besitzt und dementsprechend nur unflexibel damit umgehen kann. Dem gegenüber steht nichts desto trotz die ausgesprochen gute Anpassungsfähigkeit solcher Lösungen an die Anforderungen einer Anwendung.

In Abgrenzung zu anwendungsabhängigen Indexstrukturen stellen die folgenden Ausführungen Techniken und Konzepte vor, die – unabhängig von konkreten Anwendungen – Möglichkeiten bieten, vorhandene Daten angemessen zu strukturieren und zu indexieren. Die Konzepte sollen direkt im DBMS implementiert werden und orientieren sich an vorhandenen logischen Konstruktor-Vorlagen wie `LIST` und `SET`. Es werden also verschiedene Konstruktoren bereitgestellt, die konkrete Datenstrukturen durch geeignete Zugriffsstrukturen unterstützen. Letztere sind dann dem DBMS in ihrem Aufbau, insbesondere für den Prozeß der Anfrageoptimierung, vollständig bekannt.

Abgesehen von den Unterschieden dieser zwei Ansätze muß deutlich klargestellt werden, daß es sich bei benutzerdefinierten Indexen und den in dieser Arbeit propagierten Zugriffsstrukturen keineswegs um sich ausschließende, sondern vielmehr um sich ergänzende Konzepte handelt.

So gibt es auf der einen Seite viele Fälle, in denen Anforderungen spezieller Anwendungen den Einsatz benutzerdefinierter Indexe unabdingbar machen, weil selbst die vorhandenen erweiterten Konzepte für Zugriffsstrukturen diesen nicht gerecht werden können. Ein Beispiel dafür sind geographische oder geometrische Anwendungen, wie die Suche nach Überlappungen in Polygonmengen: Obwohl Polygone als (geordnete) Menge von Eckpunkten aufgefaßt werden können, hilft eine generische Kollektionsindexstruktur zur schnellen Bestimmung von Teil- und Obermengen nicht viel weiter. Es hilft hier nur eine am speziellen Anwendungsfall ausgerichtete nutzerdefinierte Indexstruktur weiter, über die beispielsweise ein R-Baum oder eine Grid-File-Struktur bereitgestellt werden kann.

Auf der anderen Seite erfordert gerade das Vorhandensein kollektionswertiger Attribute eine angemessene Erweiterung der in die DBMS integrierten Zugriffsmechanismen. Letztere nur mit benutzerdefinierten Indexen abdecken zu wollen, stellt keine ernstzunehmende Alternative da.

4.2.2 Anforderungen und Eigenschaften von Zugriffsstrukturen

4.2.2.1 Relationaler Fall

Werden im relationalen Fall die Indexe nur für Relationen gebraucht, so ist es dagegen im objektrelationalen Fall notwendig, Zugriffsstrukturen für eine neue Klasse von Mengen zu durchdenken und bereitzustellen. Vor diesem Schritt soll erst einmal kurz der Begriff Zugriffsstruktur im Zusammenhang mit relationalen Systemen betrachtet werden.

Zugriffsstrukturen haben primär die Aufgabe, bestimmte Objekte anhand ihrer Eigenschaften aus einer Menge gleichwertiger Objekte zu selektieren. Mengen gibt es im relationalen Fall auf zwei Ebenen:

a. *Datenbankschema: Menge von Relationen*

Trivialerweise können die einzelnen Relationen eines Datenbankschemas als eine Menge (von Relationen) gesehen werden. Ein Zugriff auf diese Relationen erfolgt über den Datenbankkatalog. Diese Betrachtungsweise ist zugegebenerweise etwas ‚philosophisch', trägt aber zur hier entstehenden einheitlichen Betrachtungsweise auf Indexstrukturen bei.

b. *Relation: Menge von Tupeln*

Die zweite ‚handfestere' Ebene von Mengen in relationalen Systemen sind die Relationen selbst, die sich aus einer Menge von Tupeln zusammensetzen. Selektionskriterien können immer direkt von dem einzelnen Tupel abgeleitet werden, so daß Indexe eigentlich eine schnellere Alternative zu einem Scan über die gesamte Relation darstellen.

Wie sich an diesen zwei Fällen zeigt, kann eine Klassifizierung einer Menge von Objekten entweder über eine externe Zugriffsstruktur erfolgen, die anhand einer Eingabe Verweise auf das entsprechende Objekt liefert (Fall a), oder das Klassifizierungskriterium ist Teil der Objekte selbst (Fall b). Ist im ersten Fall die Zugriffsstruktur obligatorisch und beinhaltet Informationen, so kann im zweiten Fall eine Selektion auch per Scan erfolgen, und Zugriffsstrukturen sind nur ein fakultatives Hilfsmittel mit redundanter Information. Im rein relationalen Fall liegt also eine zweistufige Verschachtelung von Mengen vor.

4.2.2.2 Objektrelationaler Fall

1. Kernfunktionen von Zugriffsstrukturen

Im wesentlichen werden vier Kernfunktionen einer Zugriffsstruktur im objektrelationalen Kontext unterschieden:

1. Gezielter Zugriff auf einzelne Elemente der Kollektion unter Umgehung des Ladens der gesamten Kollektion.

2. Zugriff auf die Elemente der Kollektion in einer bestimmten Ordnung.

3. Indexe als alternative Repräsentation der gesamten Kollektion (zum Beispiel Signatur).

4. Selektion von Elementen anhand bestimmter Eigenschaften, die zum Beispiel das Resultat aufwendiger Berechnungen oder Verbunde sind. Die Zugriffsstruktur berechnet diese Eigenschaften im Voraus und ermöglicht eine beschleunigte Auswertung von Anfragen.

Die letzten beiden Forderungen sind neu gegenüber relationalen Systemen. Forderung 3 ist dabei eine notwendige Folge des Vorhandenseins kollektionswertiger Attribute, also vieler Mengen gleichen Typs, die dann jeweils in ihrer Gesamtheit zum Beispiel als Operanden in Prädikaten auftreten können. Der vierte Punkt erweist sich, besonders mit Blick auf Referenzattribute, als sinnvoll und notwendig. So können mit einer Zugriffsstruktur wie dem Nested Index bespielsweise sehr schnell Attribute von anderen Objekten ausgewertet werden, die ohne den Index erst durch umfangreiche Verbund-Operationen erfaßt werden können.

2. Einsatzgebiete von Zugriffsstrukturen

Ein weiterer wichtiger Aspekt im objektrelationalen Fall ist, daß sich die im Relationalen gegebene zweistufige Verschachtelung von Kollektionen durch das Vorhandensein kollektionswertiger Attribute potenziell zu einer Baumstruktur beliebiger Tiefe erweitern kann. Kennzeichnend ist, daß der Einstiegspunkt zum Zugriff auf ein Element in einer Ebene immer in der höher gelegenen Ebene liegt.

Zusammen mit den bereits in Abschnitt 4.1 angedeuteten erweiterten Möglichkeiten beim Einsatz von Zugriffsstrukturen sowie den vier Kernfunktionen von Zugriffsstrukturen ergeben sich insgesamt gesehen sechs zentrale Einsatzgebiete:

1. *Primäre Zugriffsstruktur für Elemente einer Kollektion*
   Zugriffsstrukturen zum Zugriff auf die Elemente von kollektionswertigen Attributen oder Relationen über Schlüsselattribute; teilweise mit zusätzlichem Informationsgehalt, beispielsweise bei verketteten Listen; Elemente und Indexeinträge sind einander eineindeutig zugeordnet.

2. *Sekundäre Zugriffsstrukturen auf Elemente einer Kollektion*
   Zugriffsstrukturen zum Zugriff auf Elemente von kollektionswertigen Attributen oder Relationen über Nichtschlüsselattribute.

3. *Verschachtelte Sekundärzugriffsstrukturen innerhalb eines logischen Objekts*
   Zugriffsstrukturen, die über mehrere Verschachtelungsebenen hinweg Zugriff auf Subelemente ermöglichen.

4. *Objektübergreifende Sekundärzugriffsstrukturen*
   Im Zusammenhang mit Referenzattributen können sekundäre Zugriffsstrukturen auch auf Attributen basieren, die in anderen Objekten liegen (zum Beispiel beim Nested Index).

5. *Kompakte alternative Repräsentation einer Kollektion*
   Ein Index als kompakte redundante Speicherungsform einer Kollektion, die schnelle Aussagen über die Elemente ermöglicht, ohne auf die Elemente selbst zugreifen zu müssen.

6. *Repräsentation einer Kollektion*
   Indexorganisierte Speicherung der Elemente von kollektionswertigen Attributen oder Relationen; Elemente und Indexeinträge sind wie bei primären Zugriffsstrukturen einander eineindeutig zugeordnet, allerdings werden die Elemente direkt in den Indexeinträgen gespeichert.

3. Verknüpfung von separat gespeicherten Kollektionselementen mit dem übergeordneten Objekt

Sind die einzelnen Elemente eines kollektionswertigen Attributs in separaten Sätzen (Sekundärsätzen) gespeichert, stellt sich die Frage, auf welche Art und Weise eine Zuordnung zwischen einer einzelnen Kollektion und ihren Elementen erfolgen kann. Gegeben sei folgendes einfaches Beispiel:

```
Angestellter(Name, Vorname, SET(Berichte))
```

Wird als Speicherort der Berichte ein Segment angegeben, so befinden sich alle Berichte ungeordnet in diesem Segment. Allein das Wissen um den Speicherort – das Segment – reicht also nicht aus, um alle zusammengehörenden Elementsätze einer einzelnen Kollektion zu lokalisieren.

Die Assoziation zwischen Kollektion und Elementen kann entweder durch Hin- oder durch Rückwärtsreferenzen repräsentiert werden. In letzterem Fall werden in jedem Element Verweise auf die entsprechende Kollektion gespeichert. Auf diverse Probleme bei mehrstufig verschachtelten Kollektionen wird an späterer Stelle im Zusammenhang mit eingebettet gespeicherten Kollektionen eingegangen.

Hinreferenzen können auf zwei Arten realisiert werden:

a. Die Kollektion beinhaltet eine Zugriffsstruktur, in der Referenzen auf alle zugehörigen Elemente gespeichert sind.

b. Innerhalb des Segments wird pro Kollektion ein Cluster angelegt. Diese grenzen sich in irgendeiner Art und Weise voneinander ab, so daß eine einzige Referenz aus dem übergeordneten Objekt auf das zugehörige Cluster notwendig ist. Mit zusätzlichen Rückwärtsreferenzen in den Clustern sind auch Scans über das Segment denkbar.

Es ergeben sich also drei Möglichkeiten, wie Elemente innerhalb eines Segments ihrer Kollektion zugeordnet werden können:

- ▷ Direkte Zugriffsstruktur von dem übergeordneten Objekt zu den einzelnen Elementen.
- ▷ In den einzelnen Elementen sind Informationen gespeichert, die eine Zuordnung zu der Kollektion ermöglichen.
- ▷ Von dem übergeordneten Objekt zeigt ein Verweis auf ein Cluster innerhalb des Segments, welcher die Elemente der Kollektion enthält.

4. Spracheinbindung zur Definition von Zugriffsstrukturen

Für Zugriffsstrukturen auf Tupeln von Relationen steht der `CREATE-INDEX`-Befehl zur Verfügung. Im objektrelationalen Fall soll also zum einen der bekannte `CREATE-INDEX`-Befehl erweitert werden. Da die Deklaration der Speicherstruktur einer Kollektion im wesentlichen der Definition einer Zugriffsstruktur auf den Kollektionselementen entspricht, wird zum anderen auch innerhalb der PRDL-Konstrukte für kollektionswertige Attribute in den `CREATE-TYPE`- und `CREATE-TABLE`-Befehlen eine sehr ähnliche Syntax verwendet. Eine ausführliche Diskussion darüber wird im nächsten Kapitel stattfinden.

4.2.3 Freiheitsgrade beim Erstellen von Indexen

In Erweiterung der bisherigen Möglichkeiten wird vorgeschlagen, die Definition von Indexen in ORDBMS auf folgende vier Parameter zu stützen:

- ▷ Die Schlüssel (Eingabewerte) des Index
- ▷ Die Art und die damit verbundenen Eigenschften des Index (B*-Baum, Hashstruktur, ... und unique/non-unique, exakt/approximativ, ...)
- ▷ Die Objektmenge, auf die sich der Index bezieht

▷ Die genauen Referenzziele, also die Ausgabe einer Indexsuche.

Zusätzlich ist noch die Angabe eines Namens für den Index vorgesehen.

Im Relationalen ist die Angabe des letzten Parameters nicht notwendig, da indexierbare Objekte immer in nur genau einem physischen Satz gespeichert sind.

Im folgenden wird auf diese vier Freiheitsgrade im Zusammenhang mit objektrelationalen Datenbanken eingegangen, als erstes auf die Auswirkungen der objektrelationalen Erweiterungen auf die Art der möglichen Schlüssel bei Indexen.

4.2.3.1 Indexschlüssel im OR-Kontext

Im rein relationalen Kontext werden als Indexschlüssel einfache Attribute oder Attributkombinationen der indexierten Objekte eingesetzt. In mehrfacher Hinsicht erweitern sich bei objektrelationalen Systemen die Möglichkeiten für Indexschlüssel:

▷ Indexschlüssel aus kollektionswertigen Subattributen

▷ Indexschlüssel aus anderen Objekten

▷ Vollständige Kollektionen als Indexschlüssel

Diese Varianten werden nachfolgend diskutiert.

1. Indexschlüssel aus kollektionswertigen Subattributen

Die Benutzung von Pfadausdrücken zur Navigation innerhalb der Subattribute der indexierten Elemente erlaubt auch das Auswählen von kollektionswertigen Attributen bzw. Teilattributen. Als direkte Folge hat ein indexiertes Objekt also potenziell mehrere Ausprägungen einer Eigenschaft eines bestimmten Typs. Entsprechend kann ein konkretes Objekt auch mehrere Einträge in ein und demselben Index haben. Als Beispiel sei hier wieder eine Relation Angestellter gegeben, die u.a. ein kollektionswertiges Attribut Berichte hat, das u.a. einen Datumswert umfaßt. Ein Index auf den Angestellten mit dem Schlüsselattribut Bericht.Datum würde also für ein bestimmtes Datum auf alle Angestellten verweisen, die einen Bericht mit diesem Datum haben. Ein Angestellter mit mehreren unterschiedlich datierten Berichten wäre dann gegebenenfalls mehrmals im Index verzeichnet. Bei der Definition des Index sollte es daher möglich sein festzulegen, ob das Resultat einer Suche im Index eine Menge oder Multimenge sein soll. Entscheidet man sich für die Multimenge, würde beim Vorhandensein von zwei Berichten mit dem gleichen Datum die Referenz auf den entsprechenden Angestellten auch zweimal zurückgeliefert werden, sonst jedoch nicht.

Als Syntaxelement bietet es sich an, das aus der `SELECT`-Klausel bekannte Schlüsselwort `DISTINCT` auch innerhalb des `CREATE-INDEX`-Befehls zuzulassen.

2. Indexschlüssel aus anderen Objekten

Die Verfügbarkeit von Referenzattributen bei der Objektdefinition wirft die Frage auf, ob es möglich sein sollte, die Schlüsselattribute aus einem anderen als dem jeweils indexierten Objekt zu beziehen. Es zeigt sich schnell, daß eine Bejahung dieser Frage aufgrund der hier propagierten Unabhängigkeit von logischem OR-Modell und physischem Entwurf unausweichlich ist. Es ist also notwendig, die Möglichkeiten bei Deklaration eines Indexschlüssels stark zu erweitern und zu verallgemeinern.

Mit Referenzattributen und kollektionswertigen Attributen ist die Implementierung beliebiger Beziehungen möglich. Speziell 1:n-Beziehungen können dabei auf zwei Arten umgesetzt werden: Die Beziehung zwischen einer Relation Angestellter und einer Relation Abteilung kann beispielsweise entweder dadurch wiedergegeben werden, daß in einem Angestelltentupel eine Referenz auf seine entsprechende Abteilung gespeichert wird, oder indem in jeder Abteilung eine Kollektion von Referenzen auf die zugehörigen Angestellten verweist. Soll ein Index erstellt werden, der aus der Menge aller Angestellten aufgrund des Abteilungsnamens selektiert, muß es bei der Definition irgendwie möglich sein, den Abteilungsnamen als Schlüssel anzugeben.

Wurde die Beziehung zwischen Angestelltem und Abteilung per Referenzattribut aus dem Objekt Angestellter implementiert, ist ein Bezug einfach durch einen Dereferenzierungsoperator möglich. Denkbar wäre es, die Semantik des Punktoperators, der bereits einen navigierenden Zugriff auf untergeordnete, strukturierte Attribute innerhalb eines logischen Objekts zuläßt, entsprechend zu erweitern.

Erfolgt die Implementierung der 1:n-Beziehung dagegen mit einem kollektionswertigen Attribut, so ist eine Angabe des Schlüssels nicht ohne weiteres möglich. Konsequenterweise müssen als Schlüssel auch komplexere Verbund-Operationen erlaubt sein, die über die in der Datenbank vorhandenen Beziehungsinformationen Eigenschaften von Objekten als Indexschlüssel objektübergreifend zugänglich machen. Letztendlich sollten ganz allgemein Ausdrücke als Indexschlüssel zugelassen werden, beispielsweise die Summe zweier numerischer Felder. Entsprechend kann dann bei einer Abfrage, deren Auswahlkriterium von der Summe dieser zwei Felder abhängt, von diesem Index Gebrauch gemacht werden. Vergleichbares ist mit Einschränkungen bereits in kommerziellen Datenbank-Management-Systemen realisiert, bei DB2 zum Beispiel, indem Indexe auf virtuellen Attributen zugelassen werden [IBM00a].

3. Vollständige Kollektionen als Indexschlüssel

Durch das Vorhandensein kollektionswertiger Attribute ist es syntaktisch prinzipiell möglich, daß bei der Deklaration der Schlüsselattribute eines Index eine Kollektion als Ganzes angegeben wird. Bei einigen Zugriffsstrukturen wie zum Beispiel dem RD-Baum ist die Auswahl einer Kollektion als Schlüssel zwingend. Bei anderen ergeben sich je nach gewähltem Indextyp folgende Probleme:

- ▷ Wie ist die Gleichheit zweier Ausprägungen dieser Kollektion definiert?
- ▷ Existiert ein Ordnungskriterium auf den Ausprägungen dieser Kollektion?

Während die erstgenannte Frage bei jedem Indextyp zur Identifizierung von Indexeinträgen relevant ist, stellt sich die zweite nur bei solchen Indexen, deren Konzeption und Implementierung das Vorhandensein einer Ordnung voraussetzt. Wichtiger Vertreter der ersten Klasse ist der Hashindex, während der B*-Baum zur zweiten gehört.

Um zwei Kollektionen zu vergleichen, müssen deren Elemente verglichen werden. Des weiteren entscheidet der genaue Typ der Kollektion (zum Beispiel `LIST` oder `SET`) darüber, in welcher Reihenfolge welche Elemente betrachtet werden.

3.1. Elementvergleich als Basis für den Vergleich von Kollektionen

Um zwei Elemente einer Kollektion zu vergleichen, kann jedes Element als Ganzes oder aber nur der Primärschlüssel der Elemente berücksichtigt werden. Daß nicht das ganze Element

sondern nur der Primärschlüssel zum Vergleich herangezogen wird, ist Voraussetzung dafür, daß je nach logischer Kollektionsart gegebenenfalls schnelle Algorithmen für den Vergleich genutzt werden können, die nicht den internen Aufbau der Elemente betrachten müssen.

Des weiteren muß bei der Behandlung von Referenzattributen unterschieden werden, ob nur die Referenzen selbst oder auch die referenzierten Objekte verglichen werden. Aufbauend auf [Luf02a] wird für diese Arbeit die letzte Frage zugunsten der flachen Variante entschieden, denn bei einem Vergleich erfolgt kein Zugriff auf das referenzierte Objekt. Zu beachten ist, daß bei ungünstig gewählter physischer Implementierung der logischen Referenzen dennoch eine teilweise Dereferenzierung notwendig sein könnte. Werden physische Referenzen basierend auf dem TID-Konzept gewählt, könnten etwa zwei unterschiedliche Referenzen auf dasselbe physische Objekt zeigen, wenn ein referenziertes Objekt verschoben wird und nach der Verschiebung neue phyische Referenzen nicht auf die alte Satzadresse sondern direkt auf die neue Position zeigen würden. So etwas muß entweder vollständig verhindert oder beim Vergleich durch Dereferenzierung entsprechend behandelt werden.

3.2. Signaturen zum Vergleich von Kollektionen

Um bei dem Vergleich zweier Kollektionen einen Zugriff auf die Elemente zu vermeiden, bietet sich der Einsatz von alternativen Repräsentationen der Kollektion, beispielsweise in Form von Signaturen, an. Werden zwei Kollektionen auf Gleichheit getestet, ist bei zwei gleichen Signaturen ein Zugriff auf alle Elemente der Kollektion für eine letzte Überprüfung unvermeidlich. Eine weitere Möglichkeit zur Leistungsverbesserung könnte die Verwaltung zusätzlicher Strukturen sein, in der die Primärschlüssel der Kollektionselemente redundant gespeichert werden.

4.2.3.2 Verschiedene Typen von Zugriffsstrukturen

Dieser Abschnitt diskutiert zunächst, welche Indexstrukturen in ORDBMS angeboten werden sollten und erörtert dann ihre Eignung für Kollektionen und Relationen.

1. Berücksichtigte Zugriffsstrukturen

Basierend auf den in [Kis02] und Abschnitt 4.1 vorgestellten sowie den in bestehenden Systemen vorhandenen Konzepten werden in der hier präsentierten Speicherbeschreibungssprache PRDL folgende Typen von Zugriffsstrukturen berücksichtigt:

- *B*-Baum*
  Klassisch: Referenzen auf Objekte werden in einem Suchbaum anhand von Attributwerten aus eben diesen Objekten verwaltet.

- *Nested Index*
  Dieser Index unterscheidet sich von der Implementierung her nicht von einem B*-Baum. Der Unterschied besteht allein in den Indexeinträgen im Baum. Bei einem Nested Index stammen sie von Objekten, die über Referenzattribute bzw. sonstige Verbundoperationen mit den Objekten der indexierten Menge assoziiert sind. Nur wenn das indexierte Attribut direkt zu dem selben logischen Objekt gehört, wird in dieser Arbeit zur sprachlichen Unterscheidung im folgenden noch von B*-Baum gesprochen.

▷ *Pfadindex*

Der Pfadindex ist wie der Nested Index konstruiert, allerdings enthält er zusätzlich Referenzen auf alle Objekte, die sich entlang des Pfades zwischen Schlüsselattribut und indexiertem Objekt befinden.

▷ *Multiindex*

Die Definition eines Multiindex entspricht der Definition mehrerer einfacher B*-Baumindexe entlang des Pfades zwischen Schlüsselattribut und indexiertem Objekt. Da die jeweiligen Objektklassen entlang des Pfades vollständig indexiert werden, geht der Informationsgehalt der einzelnen Indexe über den eines einzelnen Pfadindex hinaus. Insbesondere bei langen Pfaden bedeutet die Definition einzelner Indexe einen großen Mehraufwand. Im Rahmen der Definition von Nested Indexen wird indes eine Möglichkeit zur kompakteren Deklaration der gewünschten Strukturen zur Verfügung gestellt.

▷ *Hashing*

Mengen werden hier mittels Hashtabellen indexiert. Welche konkrete Variante des Hashing zum Einsatz kommt, wird in dieser Arbeit nicht weiter thematisiert. Da Signaturen als Hashschlüssel eingesetzt werden können, werden auch die im Zusammenhang mit Kollektionszugriffsstrukturen erwähnten Hashverfahren berücksichtigt.

▷ *Verkettete Liste, Zeigerfeld, Eingelagertes Feld*

Diese drei bereits in [Kis02] vorgestellten Speicherstrukturen sollen weiterhin als elementare Varianten zur Speicherung kollektionswertiger Attribute unterstützt werden.

▷ *Signatur*

Für vorhandene oder ad hoc definierte kollektionswertige Attribute sollen Signaturen zur schnellen Suche und zum schnellen Mengenvergleich definierbar sein. Darüber hinaus wird es möglich sein, die relevanten Attribute für die Erstellung der Signatur zu deklarieren. Des weiteren können generell Signaturen für komplexe (Teil-)Objekte erstellt werden.

▷ *Signatur-File*

Im Prinzip handelt es sich hierbei um eine Aneinanderreihung von Indexeinträgen, wobei der Indexschlüssel eine Signatur ist und das Ziel der Referenzen die Elemente der indexierten Kollektion.

▷ *Signaturbaum*

Basierend auf Signaturen als Schlüssel wird eine Baumstruktur aufgebaut. Signaturen in inneren Knoten werden durch logische Verknüpfung der untergeordneten Signaturen gebildet.

▷ *Hierarchischer Bitmap-Index*

Anders als bei dem Signaturbaum ist diese Struktur nur für Mengen konzipiert. Wie im letzten Abschnitt beschrieben, besteht sie erstens aus der charakteristischen, vollständigen Repräsentation einzelner Mengen und zweitens aus einem Signaturbaum, der den schnellen Zugriff auf die Wurzeln dieser Mengen gewährleistet. Wenn im folgenden die Eigenschaften und Einsatzmöglichkeiten von hierarchischen Bitmap-Indexen

analysiert werden, bezieht sich das stets nur auf die hierarchische Darstellung der einzelnen Mengen.

▷ *RD-Baum*

Diese Zugriffsstruktur indexiert eine Menge von Mengen. Da RD-Bäume bei vielen unterschiedlichen Elementen sehr groß werden können, wird es notwendig sein, eine Einbindung dieser Struktur sowohl für bestehende als auch für dynamisch erzeugte Kollektionen zuzulassen. Letztere entstehen aus Teilen der Kollektionselemente.

▷ *Suffix-Baum*

Suffix-Bäume werden dann zum Einsatz kommen, wenn eine Reihenfolge bei der Elementspeicherung vorhanden ist. Grundlage kann sowohl eine einzelne Kollektion als auch eine Menge von Kollektionen sein.

▷ *Generalisierter Zugriffs Pfad*

Der Generalized Access Path wird in zweierlei Weise in etwas vereinfachter Form angewandt: Zum einen kann ein Index, der beispielsweise tupelübergreifend alle Elemente eines kollektionswertigen Attributs berücksichtigt, als eine vereinfachte Form dieser Zugriffsstruktur angesehen werden, da Elemente aus mehreren Mengen gleichzeitig indexiert werden. Zum anderen wird es möglich sein, mehrere Satztypen eines indexierten logischen Objekts als Ausgabe einer Indexsuche zu deklarieren. Auch hier berücksichtigt eine Zugriffsstruktur basierend auf einem Attribut mehrere Mengen von Satztypen.

▷ *Verbundindex*

Verbundindexe setzen sich im Prinzip aus zwei einfachen, einstufigen Nested Indexen zusammen. Daher ist es sinnvoll, Verbundindexe bei der Definition auf Nested Indexe zurückzuführen und auf eine explizite Definitionsmöglichkeit in der zu entwerfenden Sprache zu verzichten.

Im folgenden wird beschrieben, in welchem Kontext die Strukturen eingesetzt werden können.

2. Zugriffsstrukturen für kollektionswertige Attribute

2.1. Klassifikation von Zugriffsstrukturen

Zugriffsstrukturen für kollektionswertige Attribute können nach vier Merkmalen klassifiziert werden:

▷ *Gezielter Elementtest*

Hierbei ist es möglich, über die Zugriffsstruktur das Vorhandensein eines Elements ohne Zugriff auf die gesamte oder wesentliche Teile der Menge zu klären. Zugriffsstrukturen können diesem Merkmal in dreierlei Weise genügen:

◇ *Uneingeschränkt möglich*

Ein Elementtest ist uneingeschränkt mit dem alleinigen Zugriff auf den Index möglich.

◇ *Eingeschränkt möglich*

Die Zugriffsstruktur liefert auf eine Anfrage entweder die Antwort „nicht enthalten“ oder „möglicherweise enthalten“. Im negativen Fall liegt das Ergebnis demnach sofort vor, während im positiven Fall noch durch das (sequentielle) Durchsuchen der gesamten Menge die Richtigkeit des Enthaltenseins sichergestellt werden muß.

◇ *Nicht möglich*

Ein Elementtest ist nicht schneller möglich, als es auch eine sequentielle Suche in der Menge wäre.

▷ *Gezielter Elementzugriff*

Es geht bei diesem Punkt um die Frage, ob die Zugriffsstruktur eine Möglichkeit bietet, gezielt auf einzelne Elemente der Menge zuzugreifen, ohne daß in der gesamten restliche Menge gesucht werden muß. Ein klassischer Vertreter mit dieser Fähigkeit ist der B*-Baum-Index.

▷ *Geordneter Elementzugriff*

Eine Zugriffsstruktur kann einen geordneten Zugriff auf die Elemente zulassen. Dabei muß es sich nicht unbedingt um eine semantisch sinnvolle Reihenfolge, wie etwa nach aufsteigendem (Schlüssel-)Wert, handeln, da zum Beispiel für Verbundoperationen eine andere, technisch bedingte Reihenfolge hilfreich sein kann.

In Abgrenzung zum folgenden Punkt ist die Feststellung wichtig, daß die Ordnung durch eine innere Eigenschaft der Elemente in natürlicher Weise bedingt ist.

▷ *Elementzugriff über eine künstliche, nur in der Zugriffsstruktur gespeicherte Reihenfolge*

Hierbei erfolgt die Anordnung der Elemente und die Zugriffsreihenfolge nicht nach einer natürlichen Ordnung, sondern es kann zum Beispiel durch ein navigierendes Einfügen und Löschen eine künstliche Ordnung erreicht werden. Letztendlich bedeutet dies, daß in der Zugriffsstruktur zusätzlich Reihenfolgeinformationen enthalten sind. Ein Beispiel für eine solche Struktur ist die verkettete Liste.

TABELLE 4.5 zeigt, welche der physischen Zugriffsstrukturen die eben aufgeführten Eigenschaften erfüllen.

Neben dem B*-Baum (vorausgesetzt, die Elemente sind nicht auf Blattebene eingebettet gespeichert) eignen sich auch Hashing und RD-Baum für einen Elementtest ohne Zugriff auf das Element selbst. Bei Referenzfeldern (Pointer Arrays) erfolgt ein Zugriff immer über die Position: ist pro physischem Satz nur genau ein Element gespeichert, so ist immerhin ein kostengünstiger Test darüber möglich, ob ein Speicherplatz belegt ist oder nicht. Strukturen, die auf Signaturen basieren, können nur eingeschränkt zum Elementtest benutzt werden. Das Fehlen eines Elements kann sehr schnell überprüft werden, für einen sicheren positiven Test muß jedoch auf die Elemente zugegriffen werden. Eine Ausnahme bildet der hierarchische Bitmap-Index. Nicht geeignet für effiziente Elementtests sind die Strukturen verkettete Liste und eingelagerte Speicherung.

Außer den eben genannten Strukturen verkettete Liste und eingelagerte Speicherung finden sich in einfachen Signaturen keine Referenzen auf die Elemente der Strukturen. Sonst

TABELLE 4.5: Eigenschaften von Kollektionszugriffsstrukturen

⊕⊕ Uneingeschränkt möglich, ⊕ Eingeschränkt möglich, — Nicht möglich

Zugriffs-struktur	**El.-Test gezielt**	**El.-Zugriff gezielt**	**El.-Zugriff natürliche Reihenfolge**	**El.-Zugriff künstliche Reihenfolge**
B*-Baum, ...*	⊕⊕	⊕⊕	⊕⊕	—
Hashing	⊕⊕	⊕⊕	—	—
Verkettete Liste	—	—	⊕⊕	⊕⊕
Zeigerfeld	⊕	⊕⊕	⊕⊕	⊕⊕
Eingebettetes Feld	—	—	⊕⊕	⊕⊕
Signatur	⊕	—	—	—
Signatur-File	⊕	⊕⊕	⊕⊕	⊕⊕
Signaturbaum	⊕	⊕⊕	⊕⊕	—
Hierarchischer Bitmap-Index	⊕⊕	⊕⊕	⊕⊕	—
RD-Baum	⊕⊕	⊕⊕	⊕⊕	—
Suffix-Baum	⊕⊕	⊕⊕	⊕⊕	—

* Nested Index, Pfadindex und Multiindex basieren alle auf dem B*-Baum-Index, daher sind sie in der Tabelle nicht separat erwähnt.

bieten alle anderen Indexe die Möglichkeit, auf Elemente zuzugreifen, ohne die gesamte Kollektion geladen zu haben.

Bis auf Signaturen und Hashstrukturen können alle Zugriffsstrukturen die Elemente der Kollektion in einer bestimmten Ordnung ausgeben.

Die auf Baumstrukturen basierenden Indexarten eignen sich nur eingeschränkt für sehr schnelle navigierende Elementzugriffe und -änderungen, wie sie zum Beispiel für die Implementierung der logischen Kollektionsart Liste benötigt werden. Flache Strukturen hingegen ermöglichen sehr effiziente ($O(1)$) Einfüge- und Löschoperationen an beliebigen Listenpositionen.

2.2. Einsatz als Speicherstruktur

Anhand dieser Eigenschaften erschließt sich, welche physische Zugriffsstruktur sich für welche logische Kollektionsart eignet (TABELLE 4.6). Berücksichtigt wird hier nur die Frage nach der Speicherstruktur und damit der primären Zugriffsstruktur für die Elemente einer Kollektion, also nach der Struktur, die die Assoziation zwischen einer Kollektion und ihren Elementen sicherstellt. Nicht aufgeführt ist der Suffix-Baum, der nur als Sekundärzugriffsstruktur für Listen eingesetzt werden kann, sowie die Signatur, die ebenfalls nur als Sekundärzugriffsstruktur einen Sinn hat.

Prinzipiell kann jede der hier betrachteten physischen Strukturen als Primärspeicherstruktur einer Menge angewandt werden. Je nach Charakteristik der Menge (groß versus klein, Lese- versus Schreiboperationen und so weiter) eignen sich bestimmte Strukturen besonders gut beziehungsweise eher schlecht. Erwähnt werden sollen hier nur B*-Baum und Hashing, die bei sehr großen Kollektionen ihre Stärke bei der Mengenverwaltung ausspielen können.

Bei Multimengen verhält es sich wie bei Mengen. Allein die Struktur des hierarchischen Bitmap-Indexes ist von ihrer Konzeption her nicht fähig, das mehrfache Vorkommen eines

TABELLE 4.6: Mögliche Speicherstrukturen für logische Kollektionsarten

⊕⊕ Gut geeignet, ⊕ Geeignet, ⊖ Schlecht geeignet, — Nicht möglich

Physische Speicherstruktur	**Eignung für logische Struktur**			
	Set	**Multiset**	**List**	**Array**
B*-Baum	⊕⊕	⊕⊕	—	⊕
Hashing	⊕⊕	⊕⊕	—	⊕
Verkettete Liste	⊕	⊕	⊕⊕	⊖
Zeigerfeld	⊕	⊕	⊖	⊕⊕
Eingebettetes Feld	⊕	⊕	⊖	⊕⊕
Signatur-File	⊕	⊕	⊖	⊕
Signaturbaum	⊕	⊕	—	—
Hierarchischer Bitmap-Index	⊕	—	—	—
RD-Baum	⊕	⊕	—	—

Elements zu beschreiben.

Für Listen eignen sich nur Strukturen, die den navigierenden Zugriff unterstützen. Hervorzuheben ist natürlich verkettete Liste: die konzeptionellen Möglichkeiten der logischen Kollektionsart Liste leiten sich ja von dem Aufbau der physischen Datenstruktur ab.

Bei Feldern kann neben den sehr gut geeigneten Strukturen eingelagerte Speicherung und Zeigerfeld im Falle von großen Feldern auch der Einsatz eines B*-Baums beziehungsweise von Hashing sinnvoll sein, da hier auf der Basis von Feldindexwerten zugegriffen wird und keine Nachbarschaftsbeziehungen zwischen den Feldelementen benötigt werden. Ein Signatur-File als primäre Zugriffsstruktur entspricht vom Prinzip her einem Zeigerfeld, mit dem Unterschied, daß durch die Signatur noch eine Information über das Element an dem entsprechenden Platz zur Verfügung steht.

Einschränkend muß gesagt werden, daß ein hierarchischer Bitmap-Index nur dann als Primärspeicherstruktur eingesetzt werden kann, wenn es sich bei der Kollektion um eine Menge von Zahlenwerten oder um etwas vergleichsweise ‚Einfaches' (beispielsweise einen Referenztyp) handelt. Für komplexe logische Strukturen, wie zum Beispiel verschachtelte Mengen, ist diese physische Struktur eher ungeeignet, weil in diesen Fällen die benötigte Abbildung der Kollektionselemente auf Positionen im Bitvektor nur mit Aufwand herzustellen und zu berechnen ist.

2.3. Einsatz als Zugriffsstruktur

Eine weitere Frage betrifft die Zugriffsstrukturen für die Elemente einer Kollektion. Abgesehen vom hierarchischen Bitmap-Index müssen für die Wahl einer Zugriffsstruktur keine weiteren Einschränkungen berücksichtigt werden. Daß einige Varianten sowohl aus Leistungsgesichtspunkten als auch der Funktionalität nicht unbedingt sehr sinnvoll sind (zum Beispiel auf einer Kollektion, deren Elemente als eingebettetes Feld zusammen in einem einzigen Satz gespeichert werden, einen B*-Baum als Zugriffsstruktur zu definieren), rechtfertigt das Verbot solcher Kombinationen jedoch nicht.

Bis auf Inline Array können also alle in TABELLE 4.6 aufgeführten Strukturen als Zugriffsstrukturen definiert werden. Einschränkungen ergeben sich gegebenenfalls nur im Zusammenhang mit der Wahl der Indexschlüssel.

3. Zugriffsstrukturen für Relationen

Bei der Vielzahl der evaluierten Zugriffsstrukturen für kollektionswertige Attribute wäre es nur konsequent, diese auch für den Zugriff auf Relationen und sogar als Speicherform für Relationen – die natürlich auch nichts anderes als eine bestimmte Art von Kollektionen sind – zuzulassen. Was die Speicherungsform von Tupeln betrifft, finden sich mittlerweile auch in kommerziellen Systemen eine Reihe von Alternativen zur einfachen, ungeordneten Speicherung in Segmenten:

- ▷ *Speicherung in Clustern*

  Wie in Abschnitt 2.5.2.1 beschrieben, werden hier Sätze in abgegrenzten Clustern, also physisch dicht beieinanderliegend, abgelegt. Die Clusterung kann eindimensional nach einem Attribut oder mehrdimensional nach Attributkombinationen erfolgen. Weiterhin kann zwischen tabellen- beziehungsweise objektbezogener und übergreifender Clusterung unterschieden werden [Kis02]. Ein Umsetzungsbeispiel ist Oracle mit seinen Clustered Tables.

- ▷ *Indexorganisierte Speicherung*

  Bei dieser Speichervariante werden Tupel direkt in den Blättern eines B*-Baums gespeichert. Ein Umsetzungsbeispiel sind die Index Organized Tables (IOT) in Oracle.

- ▷ *Speicherung als Feld*

  In dieser zum Beispiel in DB2 als Range Clustered Table (RCT) bezeichneten Speicherungsform werden Daten anhand festgelegter Attributwertintervalle in den Speicherplätzen eines Feldes abgelegt. Da für jedes gespeicherte Objekt so ein fester Speicherbereich vorgesehen ist, kann über das zugrunde liegende Attribut sehr schnell auf einzelne Objekte zugegriffen werden.

Da den Relationen als oberste Ebene der Hierarchie von Kollektionen immer eine Sonderrolle zukommt, wird sich diese Arbeit speziell mit Zugriffsstrukturen für eingebettete Kollektionen, also mit kollektionswertigen Attributen, beschäftigen. Für globale Indexe werden keine umfangreichen Neuerungen im Vergleich zu [Kis02] eingeführt.

Als mögliche Speicherungsformen für die Primärsätze der Tupel einer Relation stehen also nach wie vor zur Verfügung:

- ▷ Ungeordnet in einem Segment
- ▷ Indexorganisiert
- ▷ Innerhalb eines Clusters

Als Zugriffsstrukturen für globale Indexe werden berücksichtigt:

- ▷ B*-Baum,
- ▷ Nested Index,
- ▷ Pfadindex,
- ▷ Multiindex und
- ▷ Hashing.

4.2.3.3 Auswahl der indexierten Menge

Die möglichen Mengen, auf denen eine Zugriffsstruktur definiert werden kann, leiten sich direkt aus der Hierarchie von Kollektionen ab. Die Menge kann dann durch zwei Angaben deklariert werden:

- ▷ Die Ebene beziehungsweise Position, in der sich der Einstiegspunkt zu der Zugriffsstruktur befindet
- ▷ Ein Objekttyp, der sich bezüglich des Einstiegspunkts tiefer in der Kollektionshierarchie befindet

Wird im relationalen Kontext mit `CREATE INDEX` ein Index angelegt, so ist der Einstiegspunkt immer in der obersten Ebene, also dem Datenbankschema. Entsprechend muß man für die zweite Angabe, den untergeordneten Objekttyp, immer nur den Namen einer Relation angeben. In der `CREATE-INDEX`-Syntax stand dieser Name zwischen `ON` und der Klammer, die die Auflistung der Schlüsselattribute einleitete. Wie im nächsten Kapitel im Detail gezeigt wird, ist eine Erweiterung der Syntax nötig, um eine Angabe des Einstiegspunktes zu ermöglichen.

Hingewiesen werden soll noch auf eine weitergehende Möglichkeit für die Deklaration der zu indexierenden Objektmenge. Allerdings wird dem aufmerksamen Publikum an dieser Stelle das nächste Schlüsselwort *pfad* übergeben, um sogleich fortzufahren. Angenommen, in einem Objekt Abteilung befindet sich ein kollektionswertiges Attribut mit Referenzen auf alle Angestellten, die zu dieser Abteilung gehören, dann wäre es denkbar, als Einstiegspunkt zu einer Zugriffsstruktur das Abteilungstupel zu wählen. Des weiteren könnte man es ermöglichen, die Menge aller zu dieser Abteilung gehörenden Angestellten als zugrunde liegende Menge zu wählen. Dadurch wäre man in der Lage, einen Index zu definieren, der innerhalb einer Abteilung alle Angestellten mit bestimmten Eigenschaften auswählt und somit die Grenzen einer einzelnen Kollektionshierarchie überschreitet.

Diese erweiterte Form der Definitionsmöglichkeit für Zugriffsstrukturen erlaubt es also, die Beziehung zwischen Einstiegspunkt eines Index und den indexierten Objekten mittels Referenzattributen über die Grenzen einer einzelnen Kollektionshierarchie hinaus festzulegen, sozusagen „baumübergreifend im Wald der Kollektionen und Relationen".

Da das eben erwähnte Beispiel auch in ähnlicher Form durch eine einzelne große Zugriffsstruktur mit einem zusammengesetzten Indexschlüssel implementiert werden kann (erste Schlüsselkomponente wäre die Abteilung eines Angestellten und zweite Schlüsselkomponente das eigentliche Merkmal, nach dem selektiert werden soll), wird in dieser Arbeit auf diese Möglichkeit bei der Definition von Indexen verzichtet: Ein Index wird genau einer Relation zugeordnet, was letztendlich eine einfachere Handhabung (sowohl von Seiten des Anwenders als auch von Seiten diverser DBMS-Komponenten wie dem Optimierer) von Zugriffsstrukturen in objektrelationalen Systemen ermöglicht.

4.2.3.4 Referenzen auf die indexierten Objekte

Schon allein das Vorhandensein von ausgelagerten Feldern in strukturierten Attributen macht es nötig, die genauen Referenzziele eines Indexes, der für einen gezielten Elementzugriff eingesetzt wird, festzulegen. Ergebnis einer Abfrage eines solchen Index ist immer eine Liste von Referenzen auf die Elemente der Kollektion, die der Suchabfrage genügen. Werden die Elemente als Ganzes referenziert, beziehen sich die Referenzen auf die Primärsätze der Elemente.

Da Attribute ausgelagert in Sekundärsätzen gespeichert werden können, muß es möglich sein, daß sich das Ergebnis einer Indexabfrage auf Sekundärsätze bezieht. Bei der Definition von Indexen wird man daher eine Reihe von Attributen angeben können, deren Inhalt bei einer Indexabfrage relevant ist, und so festlegen können, welche Sätze (ob Primär- oder Sekundärsatz) aus dem Index heraus referenziert werden sollen.

4.3 Zusammenfassung des Kapitels

Im ersten Teil dieses Kapitels wurden Indexstrukturen aus der Literartur und aus Produkten zusammengetragen und vorgestellt, die sich zur Indexierung komplexer Objekte eignen. Dabei wurde besonders auf zwei Klassen eingegangen:

- ▷ Pfad- und klassenbezogene Indexe die den Zugriff auf vernetzte, potentiell tief verschachtelte und in Vererbungshierarchien organisierte Objektstrukturen unterstützen sowie
- ▷ Indexe die eine Operationsunterstützung für Kollektionen von Objekten und Attributwerten bieten.

Im zweiten Teil wurde auf dieser Gundlage ein Konzept zur Indexierung in objektrelationalen DBMS erarbeitet. Dieses faßt Indexstrukturen zusammen, verallgemeinert sie und schafft ein System, in dem Indexstrukturen miteinander kombiniert, an konkrete Objektstrukturen angepaßt und sowohl global als auch lokal eingesetzt werden können. Dies wird durch die Untersuchung und Gegenüberstellung der Indexstrukturen und ihrer Eignung für bestimmte Anfragesituationen vorbereitet und durch die Aufstellung von sogenannten Freiheitsgraden der Indexierung erreicht.

Auf diese legt das Kapitel einen weiteren Grundstein für die Spezifikation der Speicherbeschreibungssprache PRDL. Da also in Kapitel 3 Speicherstrukturen und in Kapitel 4 Indexstrukturen für komplexe Objekte unterschucht wurden, kann auf dieser Basis in Kapitel 5 PRDL vorgestellt werden.

Kapitel 5

Spezifikation physischer Speicherstrukturen mit PRDL

Um dem DBMS die gewünschten Speichervarianten mitteilen zu können, bietet sich die Einführung einer Speicherbeschreibungssprache an. Diese Erweiterung adressiert die physische Repräsentation von Objekten, so daß die Sprache Physical Representation Definition Language, kurz PRDL, genannt wird. PRDL und ihre Syntax und Semantik sollen in diesem Kapitel vorgestellt werden.

5.1 Entwurf einer Speicherbeschreibungssprache

Der Einsatz von PRDL soll im Kontext von SQL-Anweisungen, genauer im Kontext der Data Definition Language (DDL), erfolgen. Daher sollte die neue Sprache sich an den vorhandenen Definitionen und Eigenheiten der SQL-Norm orientieren. In gewisser Weise könnte man bei PRDL also von einer Art Erweiterung der DDL „nach unten“ sprechen. Da sich (Norm-)SQL allerdings per Definition nur auf konzeptuelle Aspekte bezieht, wurde eine deutliche Trennung zwischen PRDL und der SQL-DDL eingehalten. Bei PRDL handelt es sich also um eine eigene Speicherspezifikationssprache, die allerdings auf einige SQL-Klauseln zurückgreift und physische Definitionen zu den mit der SQL-DDL erzeugten logischen Datenstrukturen erlaubt.

5.1.1 Entwurfskriterien für PRDL

Die zentralen, angestrebten Eigenschaften der Sprache werden nachfolgend aufgelistet und beschrieben[1]:

- *Einfachheit und Orthogonalität*
  PRDL sollte nach Möglichkeit auf einer kleinen Anzahl wiederkehrender Syntaxelemente aufgebaut sein. Teilkonstrukte einer ähnlichen oder gleichen Semantik sollten an einer Stelle definiert und dann an mehreren Stellen eingesetzt werden können. In diesem Zusammenhang steht auch die Forderung nach Orthogonalität.

[1] Dabei handelt es sich in gewissem Sinne um angestrebte Eingenschaften für *alle* formalen Sprachen.

▷ *Lesbarkeit und intuitive Verständlichkeit für den Menschen*

Bereits beim Entwurf von SQL galt als ein Kriterium, daß Anweisungen möglichst intuitiv verständlich sein sollen[2]. Auch PRDL sollte diesem Anspruch genügen. Wenn möglich, sollten in dieser Sprache formulierte Ausdrücke bei einer allgemeinen Kenntnis der Thematik selbsterklärend sein.

▷ *Maschinelle Lesbarkeit*

Neben einer guten Lesbarkeit für Anwender sollte der Sprachentwurf ebenso berücksichtigen, daß sich die maschinelle Analyse (Parsen) verhältnismäßig einfach implementieren und durchführen läßt. Zu diesem Zweck wird die Syntax von PRDL mittels einer regulären Grammatik (Typ 3) beschrieben. Leider steht dieser Punkt jedoch in gewissem Widerspruch zur vorhergehenden Anforderung, so daß versucht wurde, einen sinnvollen Ausgleich zwischen beiden zu finden.

▷ *Definition von Vorgabewerten (Defaults) und Optionalität der Spezifikationen*

Mit der Sprache sollen für jede mögliche konzeptuelle Variante Vorgabewerte für die physische Repräsentation festgelegt werden. Auf diese Weise ist gewährleistet, daß selbst dann, wenn keine oder nur eingeschränkte Angaben zur Speicherung gemacht werden, eine eindeutige Art der Speicherung definiert ist. Die Voreinstellungen sollten so angelegt sein, daß sie nach Möglichkeit für viele Fälle sinnvoll sind.

▷ *Weitgehende semantische Zulässigkeit syntaktisch möglicher Ausdrücke*

Ausdrücke, die mit der Grammatik erzeugt werden können, also syntaktisch korrekt sind, sollten weitestgehend semantisch zulässig sein. Bei dem Entwurf der Sprache PRDL zeigte sich jedoch, daß eine optimale Erfüllung dieses Kriteriums angesichts der Vielzahl zu berücksichtigender Fälle recht schwierig erreichbar ist und auf Kosten der Einfachheit und Orthogonalität gehen würde. Es wurde daher ein Entwurf gewählt, bei dem die einfache Handhabung der Sprache im Vordergrund stand. Als direkte Folge davon müssen bei der Benutzung diverser Konstrukte je nach Kontext eine Reihe von semantischen Beschränkungen beachtet werden.

5.1.2 Speicherbeschreibungssprachen

PRDL ordnet sich in eine Reihe anderer Speicherbeschreibungssprachen (SSLs – Storage Specification Languages oder SSDLs – Storage Structure Definition Languages) für Datenbanksysteme ein. Dieser Abschnitt will auf einige andere Ansätze hinweisen sowie Parallelen und Unterschiede zu PRDL aufzeigen.

Parallelen sind dabei hauptsächlich in den Konzepten zur physischen Repräsentation von einfachen und verschachtelten Datenstrukturen zu finden. Es war ein durchaus angestrebter Effekt beim Entwurf von PRDL, Impulse und Ideen von anderen Speicherbeschreibungssprachen auch aus nicht-relationalen Datenmodellen ‚aufzusammeln' und für die objektrelationale Welt nutzbar zu machen. Da es in allen SSLs um die physische Strukturierung mehr oder weniger komplexer Datenstrukturen geht, ähneln sich die dabei auftretenden Konzepte und Varianten natürlich. Insgesamt kann von einem weitgehend modell- und sprachunabhängigen Repertoire von Speicherkonzepten gesprochen werden.

Unterschiede zwischen den SSLs ergeben sich gerade durch ihren jeweiligen Bezug auf ein Datenmodell und eine Datendefinitionssprache (DDL – Data Definition Language). SSLs

[2] Ob das heutige SQL diesem Kriterium ohne Einschränkungen genügt, soll dahingestellt bleiben.

müssen sich natürlich auf die datenmodellabhängigen Definitionen logischer Datenstrukturen beziehen, denn diese definieren gerade die Objekte, deren Strukturierung mit der SSL vorgenommen werden soll. Insofern ist es auch einsichtig, daß sich SSLs am jeweiligen Sprachstil der DDL ausrichten. (Mit Sprachstil ist dabei die geläufige empirische Einordnung von Syntax und Semantik von Datenbank- und Programmiersprachen in Klassen wie C-artig, COBOL-artig et cetera gemeint.) Sehr häufig ist bei „praxisnahen Ansätzen“ und Produkten auch eine Vermischung von DDL und SSL oder sogar ihre Verschmelzung zu einer Sprache festzustellen.

In den folgenden kurzen Unterabschnitten soll jeweils eine SSL aus verschiedenen Datenmodellwelten vorgestellt werden.

5.1.2.1 UDS-SSL als Beispiel aus der Netzwerk-Modellwelt

Die Speicherbeschreibungssprache des seit vielen Jahren existierenden (Netzwerk-)Datenbanksystems UDS/SQL von Fujitsu Siemens [FSC04] orientiert sich sehr eng an der CODASYL-SSDL, die in [COD70] vorgeschlagen und festgeschrieben wird. Da sie sich auf die UDS-DDL [COD78] bezieht und eng mit dieser integriert ist, erstreckt sich diese Darstellung auf beide Sprachen.

1. UDS-DDL

Die UDS-DDL erlaubt es, in COBOL-ähnlicher Syntax logische Datenstrukturen zu definieren. Datengruppen entsprechen dabei den Strukturtypen in RDBMS. Sie enthalten Felder mit einfachen Datentypen (Attribute), die noch zu Vektoren (Arrays) zusammengefaßt werden können. Auch Datengruppen können als Array auftreten und werden dann als Wiederholungsgruppe bezeichnet. Aus all diesen Attributstrukturen können sogenannte Satzarten aufgebaut werden. Die entsprechen den Tabellen in RDBMS. Sets sind benannte (hierarchische) Beziehungstypen zwischen den Satzarten. Gespeichert werden die Ausprägungen der Satzarten und Sets in sogenannten Realms, benannten Speicherbereichen, die im wesentlichen den Tablespaces oder Segmenten in RDBMS entsprechen.

Neben den logischen Datendefinitionen, auf die hier nicht weiter eingegangen werden soll, umfaßt die UDS-DDL aufgrund nicht sehr strikter Ebenentrennung auch folgende Aspekte des physischen Datenbankentwurfs:

Mit der Festlegung sogenannter Data Base Key-Felder und der anwendungsgesteuerten Vergabe von Data Base Key-Werten kann die physische Plazierung und Reihenfolge der Datensätze gesteuert werden. Damit wird vor allem die Möglichkeit zum direkten oder sequentiellen Zugriff per Data Base Key Translation Table (DBTT) gesteuert. Die entsprechenden UDS-DDL-Schlüsselworte lauten `TYPE IS DATABASE-KEY` und `LOCATION MODE IS DIRECT`.

Alternativ kann auch eine per Hashfunktion gestreute Speicherung der Datensätze mit `LOCATION MODE IS CALC` erfolgen. Als Eingabeparameter für die Hashfunktion ist dabei ein Feld der jeweiligen Satzart festzulegen. Da über `DUPLICATES ARE [NOT] ALLOWED` gesteuert werden kann, ob Duplikate bei den Feldwerten zugelassen werden sollen, kann auf diesem Wege auch eine gewisse Clusterung der Datensätze nach diesen Feldwerten realisiert werden, denn so werden Sätze mit gleichem Feldwert auch physisch nahe beieinander abgelegt.

Weiterhin lassen sich per `SEARCH KEY IS` Zugriffspfade für den wertebasierten Zugriff auf Datensätze definieren. Die Zuordnung zwischen Suchschlüssel (Search Key) und Da-

tenbankschlüssel (Data Base Key) kann dabei entweder ebenfalls über eine Hashfunktion (`USING CALC`) oder über eine Tabelle mit Wertepaaren (`USING INDEX`[3]) erfolgen.

Auch lassen sich physische Verkettungsreihenfolgen innerhalb von Beziehungsausprägungen, den Sets, per `ORDER IS` festlegen. Dabei lassen sich sowohl die Einfügereihenfolge als auch ein Feldwert als auch die explizite Navigation an die Einfügeposition zur Einreihung nutzen.

Analog zu den Zugriffspfaden auf Datensätzen lassen sich Zugriffspfade für Sets per `ORDER is SORTED INDEXED` anlegen. Diese realisieren dann für die Datensätze in einer Set-Auspägung einen wertebasierten Zugriff.

Ebenfalls noch im Rahmen der DDL können Datensätze per `RECORD ... WITHIN ... AREA-ID IS ...` in Realms plaziert und damit einem Speicherbereich zugewiesen werden.

Insgesamt ist festzustellen, daß die UDS-DDL neben den logischen Datendefinitionen, auf die hier nicht eingegangen wurde, einen erheblichen Anteil an Spezifikationen für physische Speicherstrukturen umschließt. Dies ist sicher auch durch ihre Entstehung „in frühen Jahren“ zu erklären.

2. UDS-SSL

Mit der Speicherbeschreibungssprache UDS-SSL lassen sich folgende Aspekte der physischen Speicherung spezifizieren [RK84]:

- ▷ Im Mengengerüst werden Größen und Mengen von physischen Speicherbereichen festgelegt.
- ▷ Die Speicherungsform für die logischen Satzverknüpfungen läßt sich definieren.
- ▷ Die Lage von Datensätzen, Hashbereichen und Zuordnungtabellen kann angegeben werden.
- ▷ Es können Schwellwerte für physische Reorganisationen angegeben werden.

Bei der Spezifikation des Mengengerüsts können neben der Größe der DBTT auch die Größen von Primärschlüssel- und Zugriffspfad-Hashbereichen, Zugriffstabellen und Verbindungsausprägungen (Set-Occurences) mit `POPULATION IS` festgelegt werden. Ihr Größenwachstum wird mit `INCREASE IS` angegeben.

Für Verbindungsausprägungen kann per UDS-SSL eine der folgenden drei Speicherungsformen ausgewählt werden: Mit `MODE IS CHAIN` werden alle Mitglieder (Member) einer Set-Ausprägung (Set-Occurrence) in Form einer verketteten Liste miteinander verbunden. Bei `MODE IS POINTER ARRAY` wird ein Feld mit Referenzen auf alle Set-Mitglieder angelegt und vom Besitzerdatensatz (Owner) aus referenziert. Und `MODE IS LIST` verlagert die Member-Datensätze direkt in das Feld beziehungsweise speichert diese als physisch dichte Satzfolge. Dabei kann für alle drei Set-Formen mit `ORDER IS` eine Speicherreihenfolge festgelegt werden. Mit `LINKED TO OWNER` kann weiterhin eine Referenz von jedem Member-Satz auf den Owner-Satz zusätzlich angelegt werden. Außerdem kann jeweils mit der Option `WITH PHYSICAL LINK` von der standardmäßigen Referenzierung per Data Base Key über die DBTT auf die Verwendung physischer Referenzen, das heißt, die direkte Verwendung von Speicheradressen, umgeschaltet werden. Letztlich sei erwähnt, daß bei `MODE IS CHAIN` die Angabe von `LINKED TO PRIOR` eine doppelte Verkettung bewirkt.

Die Lage von Member-Datensätzen, Hashbereichen und Zuordnungstabellen kann innerhalb der ihnen per UDS-SSL zugewiesenen Realms über `ATTACHED TO OWNER` oder `DETACHED WITHIN` gesteuert werden. Zusätzlich ist auch eine optimierte Anordnung der

[3]Bei der Angabe von `USING INDEX` wird von UDS laut Dokumentation [FSC04] kein (baumförmiger) Index angelegt, sondern im Widerspruch zum üblichen Sprachgebrauch eine Zugriffstabelle.

Datensätze durch Speichervorreservierung über die Angabe von `PLACEMENT OPTIMIZATION` möglich.

Schließlich lassen sich Schwellwerte für die automatische physische Reorganisation von Zuordnungstabellen mit `DYNAMIC REORGANIZATION` angeben.

3. Fazit

Obwohl bei CODASYL (hier konkret UDS) die physischen Speicherkonzepte nicht sauber von den logischen Datendefinitionen getrennt sind und sich sowohl auf die SSL und DDL verteilen, lassen sich schon wichtige Konzepte für die hier vorzustellende, objektrelationale Speicherbeschreibungssprache PRDL finden:

- Erstens wird zumindest der Versuch einer Trennung von logischen und physischen Aspekten in eine DDL und eine SSL unternommen, zumal die Vermischung von DDL und SSL in erster Linie der geringeren Datenunabhängigkeit des Netzwerkmodells im Vergleich zum relationalen Datenmodell geschuldet ist.

  Die Absonderung von logischen und physischen Definitionen findet sich bei unserem Ansatz in der Trennung von PRDL von SQL wieder.

- Zweitens beruht die notwendige Kopplung von SSL an die DDL nicht auf der Integration der Sprachen. Vielmehr baut die CODASYL-SSDL/UDS-SSL insofern auf der CODASYL-/UDS-DDL auf, als daß die in der DDL definierten Objektbezeichner in der SSDL/SSL genutzt werden.

  Auch dieses Konzept ist in PRDL zu finden.

- Und drittens lassen sich die Speicherkonzepte von CODASYL-SSDL/UDS-SSL direkt in die objektrelationale Welt übertragen und finden sich dementsprechend in PRDL wieder. Beispielhaft seien hier genannt:
  - Nutzung verschiedener Speicherstrukturen für Kollektionen.
  - Steuerung der Plazierung von Datenobjekten.
    - Ein- und Auslagerung von untergeordneten Datenstrukturen.
    - Physische Plazierung nach einer Sortierordnung.
    - Clusterung von Datensätzen.
  - Nutzung unterschiedlicher Zugriffspfade, hier Hashtabellen und Zugriffstabellen.
  - Nutzung von logischen und physischen Referenzen.

Obwohl CODASYL-SSDL/UDS-SSL also eine wichtige Grundlage für PRDL bildet, stellt PRDL neben der Adaption an objektrelationale Konzepte doch eine wesentliche Weiterentwicklung dar. Dazu gehören insbesondere die Trennung logischer und physischer Aspekte, die Orientierung am SQL-Sprachkonzept (im Gegensatz zur COBOL-Ähnlichkeit von CODASYL-SSDL/UDS-SSL) und die Vielfalt der zur Verfügung stehenden Speicher- und Indexierungskonzepte.

5.1.2.2 SSL-Erweiterungen von Oracle-SQL als Beispiel aus der relationalen Modellwelt

Als Vertreter der relationalen und objektrelationalen DBMS soll Oracle im Hinblick auf SSL-Konzepte untersucht werden.

Zunächst ist festzustellen, daß es bei Oracle keine deutliche Trennung zwischen der logischen Datenmodellierung und der Definition physischer Speicherungsaspekte gibt. Vielmehr sind beide Aspekte in die jeweiligen Oracle-SQL-Befehle integriert.

1. Speicherspezifikationen in Oracle-SQL

Der wohl wichtigste Befehl aus physischer Sicht ist dabei `CREATE TABLE`. Neben den logischen Datendefinitionen für die Attribute der jeweiligen Tabelle besitzt `CREATE TABLE` einen großen ‚Rucksack' an Optionen zur Festlegung der physischen Speicherung ihrer Datensätze:

- ▷ Über das Schlüsselwort `TABLESPACE` läßt sich der Speicherort der Tabelle festlegen. Dort können mit dem Befehl `CREATE TABLESPACE` eingerichtete und über diverse Optionen physisch konfigurierte Tablespaces verwendet werden.

- ▷ Die physische Tabellenorganisation kann über das Schlüsselwort `ORGANIZATION` gewählt werden. Zur Verfügung stehen die ‚klassische' Organisation als `HEAP` mit der Datensatzplazierung nach der Einfügereihenfolge. Mit `INDEX` kann die Speicherung als indexorganisierte Tabelle erfolgen. Und schließlich kann mit `EXTERNAL` auf Daten außerhalb des DBS zugegriffen werden (sogenannte External Tables).

- ▷ Mit `CLUSTER` und der Angabe der Clusterschlüsselattribute kann eine Tabelle auch einem vorab mit `CREATE CLUSTER` definierten Cluster zugeordnet werden.

- ▷ Gleichfalls lassen sich mit `PARTITION BY` für Tabellen Partitionierungen nach den verschiedenen Strategien vornehmen. Zur Verfügung stehen die Bereichspartitionierung und die Hashpartitionierung.

- ▷ In allen Varianten lassen sich im `CREATE-TABLE`-Befehl auch noch weitere Parameter, wie zum Beispiel zur physischen Seitenbelegung, angeben.

Leider lassen sich Spezifikationen zur physischen Speicherung nur am `CREATE-TABLE`-Befehl vornehmen. Eine Angabe im Rahmen des `CREATE-TYPE`-Befehls, die als Vorgabe für spätere Tabellendefinitionen und so als einheitliche physische Spezifikation an zentraler Stelle dienen kann, ist im Gegensatz zu PRDL nicht vorgesehen.

2. Fazit

Weitergehend soll Oracle-SQL hier nicht hinsichtlich seiner Speicherspezifikationen untersucht werden. Jedoch bleiben folgende Punkte festzuhalten:

Bei Oracle läßt sich, streng genommen, nicht von der Existenz einer SSL sprechen. Vielmehr sind im produktspezifischen SQL-Dialekt logische und physische Aspekte eng miteinander verwoben. Obwohl diese Situation typisch für RDBMS und ORDBMS ist, konterkariert sie die an sich weit fortgeschrittene Datenunabhängigkeit in RDBMS: Einige Änderungen der Speicherstruktur wirken sich auf das logische Datenmodell aus und erfordern dort Strukturänderungen. Deshalb treten leider auch physisch motivierte Änderungen bis hinauf auf die Anwendungsebene auf.

Dieser Aspekt soll durch die in dieser Arbeit vorgeschlagene Einführung der Speicherbeschreibungssprache PRDL für objektrelationale DBMS wesentlich verbessert werden. Die

konsequente Trennung der Definition logischer Objektstrukturen mit SQL von der Definition physischer Speicherstrukturen mit PRDL verspricht eine wesentliche Verbesserung der Datenunabhängigkeit in ORDBMS.

Allerdings stellt Oracle, wie auch andere relationale und objektrelationale DBMS, eine Reihe interessanter Konzepte zur Speicherung komplexer Objekte zur Verfügung. Diese sollen, wie schon in Abschnitt 3.1.7 deutlich gemacht wurde, in PRDL einfließen.

5.1.2.3 Zusammenfassung und Einordnung von PRDL

Zusammenfassend läßt sich zur Einordnung von PRDL festhalten, daß bisher für DBMS keine Speicherbeschreibungssprachen bekannt sind, die die in Kapitel 3 und Kapitel 4 vorgestellten spezifischen objektrelationalen Anforderungen zur effizienten und optimierten Speicherung komplexer Objekte berücksichtigen. Es gibt in der Produktwelt nicht einmal separate Speicherbeschreibungssprachen für relationale DBMS. Alles, was in diesem Bereich verfügbar ist, ist der ‚Rucksack' mit vielfältigen physischen Spezifikationen an SQL-DDL-Befehlen in aktuellen DBMS-Produkten. Bei diesen schlagen jedoch Änderungen der physischen Speicherung mitunter bis auf die Anwendungsebene durch. Insofern herrscht ein Mangel an Datenunabhängigkeit bei der Datenmodellierung, -speicherung und -verarbeitung.

Diese Situation soll durch die hier vorgeschlagene Speicherbeschreibungssprache PRDL, insbesondere im Hinblick auf die komplexen physischen Strukturen verschachtelter Objekte mit kollektionswertigen Attributen, verbessert werden. PRDL bietet für objektrelationale DBMS ein Konzept zur Erhöhung der Datenunabhängigkeit durch die Trennung von logischer und physischer Datenmodellierung, wie sie interessanterweise bereits für CODASYL-Datenbanken durch die Aufspaltung in CODASYL-SSDL und -DDL in Ansätzen existiert(e).

Die Speicher- und Indexierungskonzepte, die in PRDL integriert sind, stellen eine Zusammenfassung von Vorschlägen und Techniken aus DBMS-Produkten und aus der Literatur dar, die nach ihrer Eignung für komplexe Objekte in objektrelationalen DBMS ausgesucht worden sind. Die berücksichtigten Konzepte reichen von vorrelationalen DBMS über relationale Datenbankprodukte bis hin zu NF^2-Prototypen und objektorientierten Techniken. Zur Integration wurden sie miteinander abgeglichen, teilweise auf Basiskonzepte zurückgeführt und zur Anwendung an objektrelationale Konzepte angepaßt. Die vielen Parallelen zu bekannten Speicherverfahren sind also beabsichtigt und Kern von PRDL. Gleichzeitig ist der Entwurf von PRDL soweit offen und flexibel gehalten, daß auch neue Speicher- und Indexierungtechniken integrierbar sind. Darauf wird unter anderem im folgenden näher eingegangen.

5.1.3 Einbettung der PRDL-Konstrukte

Ziel von PRDL ist es, für zwei verschiedene Aufgaben geeignete Sprachkonstrukte zur Verfügung zu stellen:

1. *Spezifikation der physischen Speicherungsform für komplexe Objekte*
   - ◇ Aufteilung der Attribute der möglicherweise verschachtelten Objektstruktur auf einen Primärsatz und potentiell mehrere Sekundärsätze.
   - ◇ Spezifikation der Verkettung der physischen Sätze eines komplexen Objekts.
   - ◇ Festlegung des Speicherorts der physischen Sätze.
   - ◇ Festlegen der primären Speicherstruktur von eingeschachtelten Kollektionen.

2. *Deklaration sekundärer Zugriffsstrukturen*
   - ◇ Für eingeschachtelte Kollektionen
   - ◇ Für ganze Relationen

Für den ersten Fall werden in SYNTAXREGEL 1 die `CREATE-TYPE` und in SYNTAXREGEL 2 die `CREATE-TABLE`-Anweisung von SQL erweitert. Das Erstellen eines Datenbankschemas, in dem komplexe Attribute benutzt werden, erfolgt in verschiedenen Schritten und unter der Berücksichtigung bestimmter Reihenfolgen. So müssen zum Beispiel zuerst alle für Attribute benötigte Typen definiert sein, bevor eine entsprechende Relation erzeugt werden kann. Auch bei der Erstellung komplexer Typen selbst müssen alle eingebundenen Untertypen bereits definiert sein.

Für den Zeitpunkt der Einbringung von PRDL-Fragmenten zur Steuerung der physischen Speicherung unterstützt PRDL prinzipiell zwei Möglichkeiten. Zum einen kann bereits bei der Typdefinition festgelegt werden, wie die Objekte dieses Typs zu speichern sind. Zum anderen erlaubt es PRDL ebenso, diese Angaben erst zum Zeitpunkt der Nutzung des Typs im Datenmodell – beispielsweise bei der Tabellendefinition – zu machen. Die erste Möglichkeit schließt die zweite nicht aus. PRDL behandelt Angaben zum Definitionszeitpunkt als Festlegung von Vorgabewerten (Defaults) speziell für diesen Typ. Sie können dann zu einem späteren Zeitpunkt bei der Nutzung des Typs in dieser Weise übernommen oder gezielt an die jeweiligen Anforderungen angepaßt werden.

Für den zweiten Fall wird in SYNTAXREGEL 4 der `CREATE-INDEX`-Befehl entsprechend erweitert (siehe Abschnitt 5.2.1.2). Dieser Befehl ist bereits aus Datenbanksystemen in verschiedenen Varianten bekannt. Er gehört zwar nicht (mehr) zur SQL-Norm, da es bei dem Anlegen von Indexen um rein physische und nicht um konzeptuelle Aspekte geht, ist aber natürlich die korrekte Stelle zur Erweiterung um zusätzliche Zugriffsstrukturen.

Die für PRDL relevanten DDL-Konstrukte sind also `CREATE TYPE`, `CREATE TABLE` und `CREATE INDEX`. PRDL wäre auch auf die entsprechenden `ALTER`-Anweisungen erweiterbar. Diese Erweiterung wirft die Frage nach der Reorganisation von physischen Speicher- und Indexierungsstrukturen auf, die in dieser Arbeit jedoch nicht behandelt werden soll. Eine eingehende Behandlung von Reorganisationsfragen sind im Umfeld von [SD03, Dor03] zu finden.

5.2 Syntax und Semantik von PRDL

Die Spezifikation der Sprache PRDL soll es dem Anwender (Datenbankadministrator) ermöglichen, die zur jeweiligen Anwendung passende Speicherstruktur für den zugehörigen logischen Entwurf zu definieren. Zur ‚Formulierung' des logischen Entwurfs dient bisher und auch weiterhin SQL, das sich, zumindest aus Sicht der Norm, prinzipiell auf rein konzeptuelle Aspekte beschränkt und beschränken soll. Zur Festlegung der physischen Aspekte soll die neue Sprache PRDL dienen.

Die Präsentation der Sprache erfolgt über die Erläuterungen der zugehörigen Grammatik. Sie setzt sich aus einer Menge reservierter Wörter und Sprachbildungsregeln (auch Produktionen genannt) zusammen, die in diesem Kapitel in Form von Syntaxdiagrammen mit zugehörigen Erläuterungen und Beispielen dargestellt werden. Im Anhang B ist die PRDL-Syntax noch einmal vollständig in der auch in der SQL-Norm gebräuchlichen Variante der Backus-Naur-Form (BNF) zusammengefaßt.

5.2.1 Anbindung von PRDL an SQL

Die Anbindung von PRDL erfolgt durch das rein textuelle Anhängen von Speicherspezifikationen an die SQL-Anweisungen `CREATE TYPE` und `CREATE TABLE` sowie an die `CREATE-INDEX`-Anweisung. Zwar werden diese Anweisungen dazu um zusätzliche Klauseln erweitert, doch bleiben duch die Kapelung der Speicherspezifikationen SQL und PRDL dabei sauber voneinander getrennt. Das rein textuelle Anhängen erfolgt lediglich der Einfachheit halber. Ohne Probleme könnten die PRDL-Spezifikationen in separate Anweisungen ausgelagert werden. In diesen müsste jeweils lediglich der entsprechende Typ, die entsprechende Tabelle oder der entsprechende Index als Kontext angegeben werden.

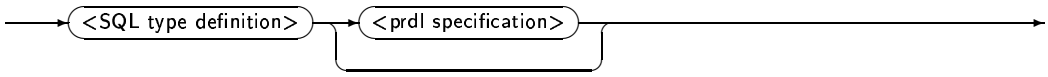
PRDL erlaubt die Spezifikation der physischen Speicherform von komplexen Objekten und die Definition von Zugriffsstrukturen für diese. Angaben zur Speicherform werden an `CREATE TYPE` und `CREATE TABLE` angehängt, für Angaben zu Zugriffsstrukturen wird `CREATE INDEX` erweitert.

5.2.1.1 Spezifikation der Speicherform mit PRDL

Zur Spezifikation der physischen Speicherform komplexer Objekte werden Erweiterungen an den SQL-Anweisungen `CREATE TYPE` und `CREATE TABLE` vorgenommen.

Syntaxregel 1

<type definition> ::=



Erläuterung

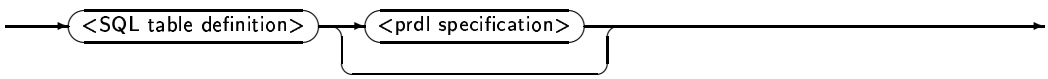
Nach einer Typdefinition kann optional eine PRDL-Spezifikation zur Festlegung der physischen Speicherungsform festgelegt werden. Dabei dienen die Angaben zur Typdefinition als Vorgaben (Defaults) für Tabellendefinitionen, in denen entsprechende Typen verwendet werden.

Teilklauseln

- *<SQL type definition>*
  Typdefinition nach der SQL-Norm.
- Syntaxregel 3 *<prdl specification>*
  PRDL-Spezifikation für die physische Speicherstruktur.

Syntaxregel 2

<table definition> ::=



Erläuterung

Nach einer Tabellendefinition kann optional eine PRDL-Spezifikation zur Festlegung der physischen Speicherungsform festgelegt werden. Angaben zur Typdefinition dienen als Vorgaben, falls keine Festlegungen getroffen werden. Allerdings ist es möglich, bei der Tabellendefinition diese Vorgaben durch neue PRDL-Spezifikationen zu ändern oder zu ersetzen.

TEILKLAUSELN

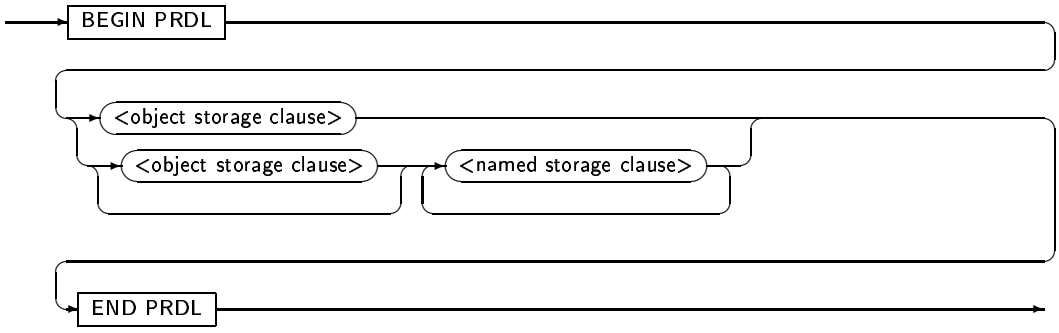
- *<SQL table definition>*
  Tabellendefinition nach der SQL-Norm.
- SYNTAXREGEL 3 *<prdl specification>*
  PRDL-Spezifikation für die physische Speicherstruktur.

Wie bereits in Abschnitt 5.1.1 erläutert, sind alle PRDL-Angaben prinzipiell optional. Als Vorgabewerte für die physische Speicherung wird deshalb jeweils eine in kommerziellen Datenbanksystemen gängige Variante festgelegt, um die Kompatibilität exisitierender Systeme mit PRDL zu sichern. Auf diese Weise ist selbst eine leere PRDL-Spezifikation gültig. Folgende Vorgaben existieren:

- Die Primärsätze werden im Standard-Tablespace des DBMS abgelegt.
- Alle Character-Large-Object- und Binary-Large-Object-Attribute der obersten Strukturierungsebene der Objekte werden in je einem LOB-Sekundärsatz ausgelagert. Innerhalb von Kollektionen wird für LOB-Attribute pro Element ein weiterer Sekundärsatz angelegt.
- Es werden sonst keine Attribute ausgelagert.
- Die Sekundärsätze für Large Objekts werden in einem eigens vom DBMS dafür vorgesehenen Standard-Tablespace für Large Objects gespeichert.
- Es werden keine Rückwärtsreferenzen von den Sekundärsätzen auf den Primärsatz angelegt.
- Kollektionsattribute werden im Normalfall als eingelagertes Feld ohne zusätzlichen Überlaufsatz gespeichert. Handelt es sich bei dem Elementtyp um ein LOB, wird die Struktur Verweisfeld (ein Element pro Satz) benutzt.
- Sowohl bei den Primärsätzen als auch bei allen Sekundärsätzen sind verkettete Sätze erlaubt.

SYNTAXREGEL 3

<prdl specification> ::=



ERLÄUTERUNG

PRDL-Anweisungen zur Spezifikation der physischen Strukturen werden durch die Ausdrücke `BEGIN PRDL` und `END PRDL` eingeklammert. Die möglichen PRDL-Anweisungen sind in zwei Teile aufgeteilt: In der <object storage clause> werden

Angaben zur Speicherung des Primärsatzes gemacht, in der <named storage clause> können mehrere Abschnitte mit Angaben zur Speicherung von einzelnen Attributen beziehungsweise Sekundärsätzen auftreten.

TEILKLAUSELN

- ▷ SYNTAXREGEL 5 *<object storage clause>*
  Angaben zur Speicherung des Primärsatzes
- ▷ SYNTAXREGEL 19 *<named storage clause>*
  Angaben zur Speicherung von einzelnen Attributen

5.2.1.2 Spezifikation von Zugriffsstrukturen mit PRDL

Im Gegensatz zur Spezifikation der Speicherstrukturen komplexer Objekte, bei der PRDL-Befehle an SQL-DDL-Konstrukte angehängt wurden, wird für die Definition sekundärer Zugriffsstrukturen der `CREATE-INDEX`-Befehl modifiziert. In relationalen DBMS hat dieser Befehl üblicherweise die Struktur:

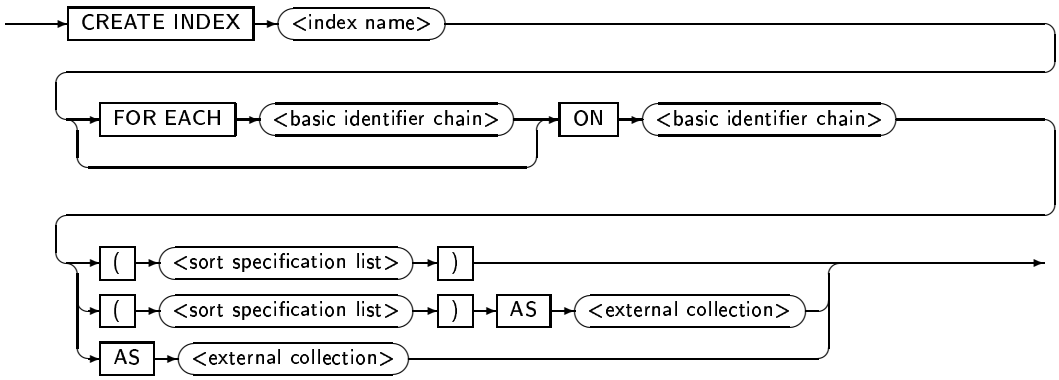
```
CREATE INDEX <index name>
    ON <basic identifier chain> (<sort specification list>)
```

Meist folgen diesem Teil noch diverse Optionen, wie Speicherort oder Füllgrad der Indexseiten. In fast allen Fällen ist der Indextyp implizit auf den B*-Baum beschränkt.

Um mit `CREATE INDEX` erstens auch Indexe für kollektionswertige Attribute und zweitens die vielen verschiedenen Indextypen berücksichtigen zu können, ist eine Erweiterung dieses Befehls unumgänglich.

SYNTAXREGEL 4

<create index> ::=



ERLÄUTERUNG

Diese Syntaxregel definiert den Befehl zur Erzeugung von Indexen. Es gibt darin zwei Neuerungen im Vergleich zur üblichen `CREATE-INDEX`-Syntax:

- ▷ Mit `FOR EACH` kann per <basic identifier chain> der Einstiegspunkt oder können die Einstiegspunkte für den Zugriffspfad ausgewählt werden. Für das Element oder bei Kollektionen für die Elemente dieser Einstiegsebene wird jeweils ein lokaler Index angelegt.

  Entfällt die Angabe, so ist die Einstiegsebene standardmäßig der Datenbankkatalog, d.h. es handelt sich um einen globalen Index. Die aus relationalen DBMS

bekannte Definition globaler Indexe auf Relationsattributen ist also als Spezialfall mit abgedeckt.

Ausgehend von der mit `FOR EACH` ausgewählten Einstiegsstruktur kann nach `ON` wieder über eine <basic identifier chain> eine untergeordnete Kollektion ausgewählt werden. Über den Elementen dieser ausgewählten Kollektion wird dann der Index aufgebaut. Die Indexschlüssel werden danach in Klammern relativ zur ausgewählten Kollektion angegeben.

Wie auch bei `FOR EACH` und `ON` kann auch bei der Angabe der Indexschlüssel in eine Relation ‚hineinnavigiert' werden, so daß tief in komplexe objektrelationale Strukturen verschachtelte Indexschlüsselattribute definiert werden können.

- ▷ Der Typ der Zugriffsstruktur wird nach `AS` innerhalb der <external collection>-Klausel spezifiziert. Entfällt die Angabe, so wird standardmäßig ein B*-Baum genutzt.

  Die durch SYNTAXREGEL 45 <external collection> spezifizierte Zugriffsstruktur muß sich dabei natürlich stets auf Kollektionselemente oder deren Teile beziehen und ‚endet' so stets mit SYNTAXREGEL 46 <collection element>.

Zu bemerken ist, daß aus Kompatibilitätsgründen die Angabe des Indexschlüssels an zwei Stellen möglich ist: Entweder klassisch innerhalb der Klammern des obigen Ausdrucks, oder, falls ein Indextyp spezifiziert wird, innerhalb des `AS`-Ausdrucks.

TEILKLAUSELN

- ▷ *<index name>*
  Name des Indexes, Bezeichner nach der SQL-Norm.
- ▷ *<basic identifier chain>*
  Bezeichner nach der SQL-Norm.
- ▷ *<sort specification list>*
  Liste von Bezeichnern nach der SQL-Norm, zur Bestimmung des Indexschlüssels.
- ▷ SYNTAXREGEL 45 *<external collection>*
  Angabe der Indexstruktur. Diese beginnt mit einer Auslagerung, das heißt einer Referenz vom Datenbankkatalog auf die Wurzel der Indexstruktur.

BEISPIEL

Mit folgender Anweisung wird für jedes Element des Listenattributs `Herstellungsablauf`, das heißt jeden Fertigungsschritt, ein lokaler Index aufgebaut.

```
CREATE INDEX Werkstückart_Bearbeitungskonfiguration
    FOR EACH Herstellungsablauf
    ON Bearbeitungskonfiguration
    (Parametername)
```

Er indexiert die Elemente des eingeschachtelten Mengenattributs `Bearbeitungskonfiguration`, die Einrichtungsparameter, anhand der Parameternamen.

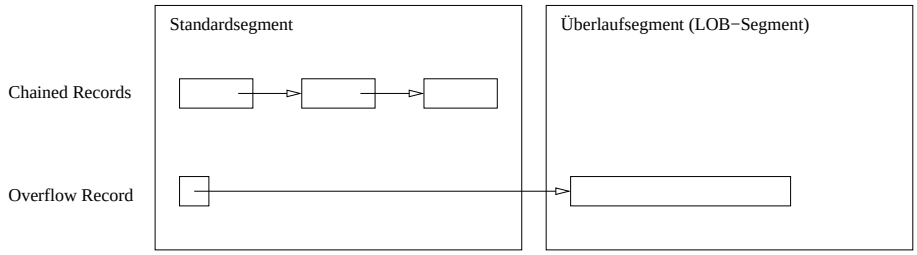
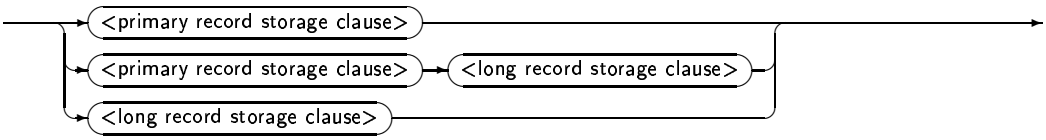


ABBILDUNG 5.1: Speicherung ‚übergroßer' Sätze: Satzverkettung versus Überlaufsatz

5.2.2 Speicherangaben zum Primärsatz

SYNTAXREGEL 5

<object storage clause> ::=



ERLÄUTERUNG

Zur Speicherung von Primärsätzen können zwei Optionen festgelegt werden:

- ▷ *Wo sollen die Primärsätze gespeichert werden?*
- ▷ *Wie sollen Sätze gespeichert werden, die zu groß für eine physische Seite sind?*

  Sollen verkettete Sätze (Record Chaining – mehrere verkettete physische Sätze im gleichen Segment zur Aufnahme der Teile eines großen logischen Satzes) erlaubt sein? Oder sollen sogenannte Überlaufsätze (Overflow Records – ein großer Satz, möglichst als LOB in einem LOB-Segment, der aus dem ursprünglichen (‚kleinen') Satz heraus referenziert wird und die ‚übergroßen' Daten aufnimmt) angelegt werden? Und wo erfolgt ihre Speicherung? ABBILDUNG 5.1 verdeutlicht die grundsätzlichen Alternativen.

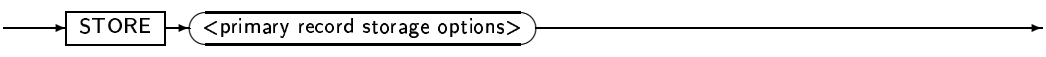
Wird eine dieser Festlegungen nicht angegeben, so greifen die Standardstrategien des jeweiligen DBMS (siehe Abschnitt 5.2.1.1).

TEILKLAUSELN

- ▷ SYNTAXREGEL 6 *<primary record storage clause>*
  Festlegung des Speicherplatzes für Primärsätze.
- ▷ SYNTAXREGEL 14 *<long record storage clause>*
  Festlegung des Speicherplatzes für Sätze, die zu groß für eine physische Seite sind.

SYNTAXREGEL 6

<primary record storage clause> ::=



ERLÄUTERUNG

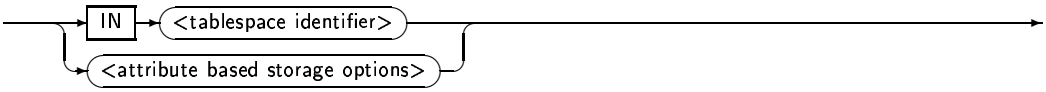
In <primary record storage clause> wird die Festlegung des Speicherorts der Primärsätze mit dem Schlüsselwort `STORE` eingeleitet. Die eigentliche Spezifikation folgt danach in <primary record storage options>.

TEILKLAUSELN

▷ SYNTAXREGEL 7 *<primary record storage options>*
Festlegung des Speicherorts von Primärsätzen.

SYNTAXREGEL 7

<primary record storage options> ::=



ERLÄUTERUNG

Mit dieser Regel wird der Speicherort von Primärsätzen festgelegt. Die Variante `IN` <tablespace identifier> erlaubt die Angabe eines Tablespaces zur Speicherung. Weitere Möglichkeiten können über <attribute based storage options> angegeben werden.

Wie bereits in Abschnitt 3.2.2 vorgestellt, sind prinzipell fünf verschiedene Varianten bei der Speicherung von physischen Sätzen möglich. Für Primärsätze sind jedoch nur die ersten vier relevant. Sie ergeben sich aus der Kombination der Syntaxregeln 7 und 8, die zum Zwecke der Wiederverwendung von der <attribute based storage options>-Klausel getrennt wurden.

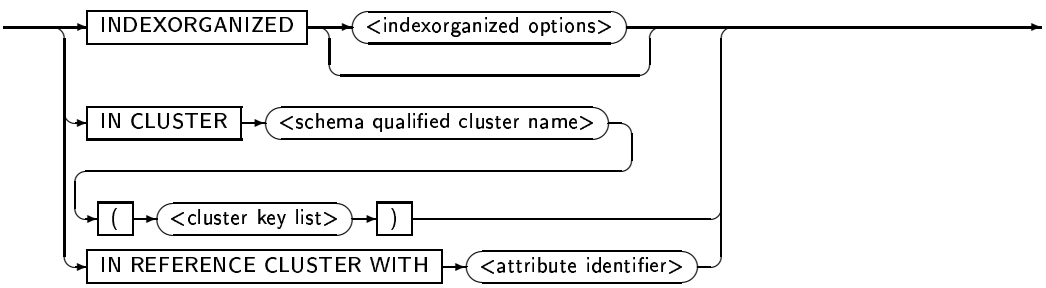
TEILKLAUSELN

▷ SYNTAXREGEL 9 *<tablespace identifier>*
Angabe eines vorab definierten Tablespaces beziehungsweise der Verweis auf den systemabhängigen Standard-Tablespace.

▷ SYNTAXREGEL 8 *<attribute based storage options>*
Attributbasierte Spezifikationen des Speicherorts von Primärsätzen.

SYNTAXREGEL 8

<attribute based storage options> ::=



ERLÄUTERUNG

Attributwertbasierte Spezifikationen des Speicherorts von Primärsätzen können mit dieser Regel angegeben werden.

Die erste Variante für die Speicherung von Primärsätzen besteht in der sortierten Speicherung, also nach dem Prinzip der indexorganisierten Tabelle. Das Schlüsselwort `INDEXORGANIZED` ist gefolgt von weiteren (optionalen) Angaben.

Eine weitere Möglichkeit für die Speicherung des Primärsatzes ist seine Zuordnung zu einem vorab definierten benannten Clustertyp über `IN CLUSTER`. Zwingend ist die Festlegung der Attribute, die dem Clusterschlüssel zugeordnet werden.

Anzumerken ist, daß in einem Cluster der Clusterschlüssel nur einmal gespeichert wird. Zusätzliche Speicherangaben für die dem Clusterschlüssel zugeordneten Attribute sind daher nicht zulässig.

Die letzte Variante zur Primärsatzspeicherung ist die referenzbezogene Speicherung mit `IN REFERENCE CLUSTER WITH`. Hier ist die Angabe eines Referenzattributs notwendig. Der Primärsatz wird im gleichen Cluster wie der Primärsatz gespeichert, auf den die Referenz verweist. Die Referenz ist sozusagen der Clusterschlüssel.

TEILKLAUSELN

- ▷ SYNTAXREGEL 10 *<indexorganized options>*
  Optionen zur indexorganisierten Speicherung.
- ▷ *<schema qualified cluster name>*
  Name des Clusters, Bezeichner nach Norm-SQL.
- ▷ SYNTAXREGEL 42 *<cluster key list>*
  Attribute, die dem Clusterschlüssel zugeordnet werden.
- ▷ SYNTAXREGEL 13 *<attribute identifier>*
  Angabe eines Referenzattributs zur referenzbezogenen Clusterung.

BEISPIEL

Ein Bespiel für die indexorganisierte Speicherung ist bei der Vorstellung von SYNTAXREGEL 10 angegeben.

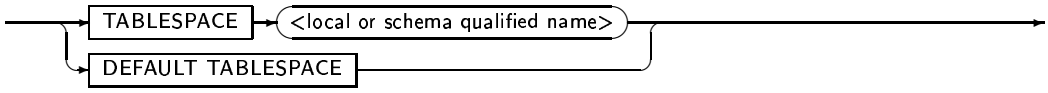
Die wertebasiert geclusterte Speicherung zeigt folgendes Beispiel. Es werden die Primärsätze der Relation einem Clustertyp zugewiesen, der die Objekte nach zwei Attributwerten gliedert:

```
CREATE TABLE Fertigungsmaschine AS Fertigungsmaschine_t
BEGIN PRDL
    STORE IN CLUSTER Standortcluster
        (Standort.Werksgebäude, Standort.Etage)
END PRDL
```

Zu beachten ist, daß die Attribute des Clusterschlüssels nicht unbedingt atomar sein müssen, sondern auch strukturiert oder kollektionswertig sein können. Dies geht allerdings nur, wenn auf ihnen eine Sortierordnung oder zumindest die Äquivalenz definiert ist. Dieses Themengebiet würde jedoch den Rahmen dieser Arbeit sprengen und soll deshalb hier nicht weiter behandelt werden.

SYNTAXREGEL 9

<tablespace identifier> ::=



ERLÄUTERUNG

Diese Regel dient der Angabe eines Tablespaces. Dieser kann entweder über den Namen identifiziert werden, oder es kann der Standard-Tablespace gewählt werden.

Die Definition von Tablespaces könnte im übrigen prinzipiell ebenfalls Bestandteil der PRDL-Spezifikation sein. Im Rahmen dieser Arbeit wird darauf jedoch nicht weiter eingegangen. Es wird angenommen, daß ein Tablespace-Name datenbankweit eindeutig

ist und, wie im unten folgenden Beispiel, über einen einfachen Bezeichner identifiziert werden kann.

TEILKLAUSELN

▷ *<local or schema qualified name>*
Name des Tablespaces. Bezeichner nach Norm-SQL.

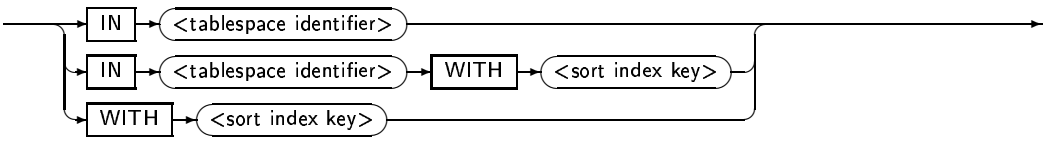
BEISPIEL

Das Beispiel zeigt, wie eine Tabelle vom Typ `Fertigungsmaschine_t` erzeugt und im Tablespace `TS_1` abgelegt wird:

```
CREATE TABLE Fertigungsmaschine AS Fertigungsmaschine_t
BEGIN PRDL
    STORE IN TABLESPACE TS_1
END PRDL
```

SYNTAXREGEL 10

<indexorganized options> ::=



ERLÄUTERUNG

Für die indexorganisierte Speicherung können folgende Angaben gemacht werden:

- ▷ Die erste Alternative ist die explizite Zuweisung eines Tablespaces. Fehlt diese, wird der Standard-Tablespace des Systems bzw. des Benutzers gewählt.
- ▷ Des weiteren kann der Schlüssel festgelegt werden, nach dem die sortierte Speicherung erfolgen soll. Ohne Festlegung wird standardmäßig der Primärschlüssel genutzt. Wenn dieser nicht definiert ist, ist die Angabe zwingend.

TEILKLAUSELN

▷ SYNTAXREGEL 9 *<tablespace identifier>*
Angabe eines Tablespaces.

▷ SYNTAXREGEL 11 *<sort index key>*
Angabe des Indexschlüssels.

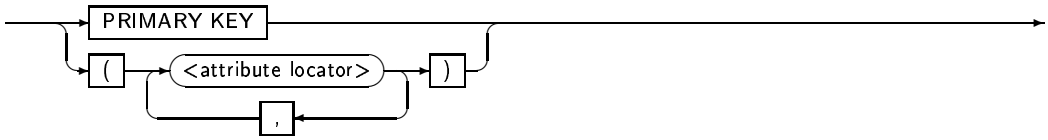
BEISPIEL

Folgendes Beispiel zeigt, wie die Fertigungsmaschinen nach Werksgebäude und Etage indexorganisert abgespeichert werden können. Als Speicherort ist der Standard-Tablespace angegeben. Diese Angabe könnte auch weggelassen werden, da sie die Voreinstellung ist.

```
CREATE TABLE Fertigungsmaschine AS Fertigungsmaschine_t
BEGIN PRDL
    STORE INDEXORGANIZED
        IN DEFAULT TABLESPACE
        WITH (Standort.Werksgebäude, Standort.Etage)
END PRDL
```

SYNTAXREGEL 11

<sort index key> ::=



ERLÄUTERUNG

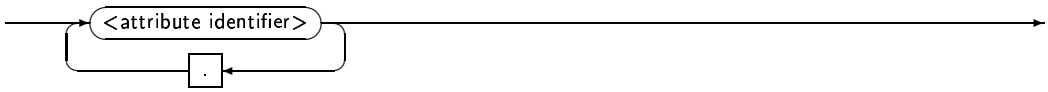
Die Festlegung des Schlüssels für die indexorganisierte Speicherung kann, sofern einer definiert ist, über den Primärschlüssel oder die explizite Angabe der Schlüsselattribute erfolgen. PRDL unterstützt in der jetzigen Form nicht die Angabe kollektionswertiger Attribute als Schlüssel.

TEILKLAUSELN

▷ SYNTAXREGEL 12 *<attribute locator>*
Pfadausdruck zur Angabe eines Attributs.

SYNTAXREGEL 12

<attribute locator> ::=



ERLÄUTERUNG

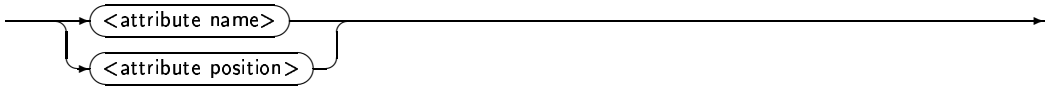
Mit dieser Regel kann ein Pfadausdruck zur Bezeichnung eines Attributs angegeben werden, das auch aus tieferen Verschachtelungsebenen stammen kann und daher über die Punkt-Notation adressiert wird.

TEILKLAUSELN

▷ SYNTAXREGEL 13 *<attribute identifier>*
Bezeichnung eines Attributs.

SYNTAXREGEL 13

<attribute identifier> ::=



ERLÄUTERUNG

Die Identifikation eines Attributs kann über seine Namen oder seine Position innerhalb einer Struktur erfolgen.

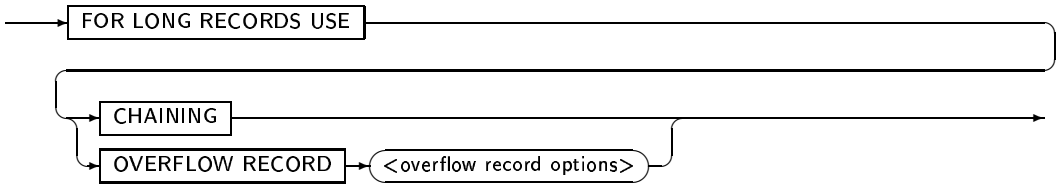
TEILKLAUSELN

▷ *<attribute name>*
Attributname nach Norm-SQL.

▷ *<attribute position>*
Attributnummer nach Norm-SQL.

Syntaxregel 14

<long record storage clause> ::=



Erläuterung

Mit dieser Klausel kann die Nutzung von seitenübergreifenden, verketteten Sätzen oder von Überlaufsätzen für den Fall spezifiziert werden, daß die Größe eines Datensatzes die Größe einer einzelnen Seite überschreitet (vergleiche Abbildung 5.1).

Teilklauseln

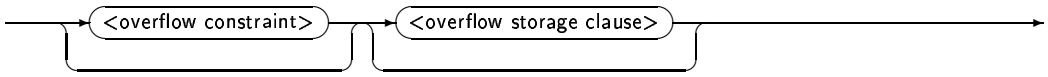
- ▷ Syntaxregel 15 *<overflow record options>*
  Angaben zur Nutzung von Überlaufsätzen.

Beispiel

```
CREATE TABLE Fertigungsmaschine AS Fertigungsmaschine_t
BEGIN PRDL
    FOR LONG RECORDS USE CHAINING
END PRDL
```

Syntaxregel 15

<overflow record options> ::=



Erläuterung

Bei der Spezifikation von Überlaufsätzen werden mit dieser Regel weitere Angaben gemacht. Die Überlaufbedingung gibt dabei an, ab welcher Satzgröße Daten ausgelagert werden sollen. Die Speicherung des Überlaufsatzes kann in der zweiten, optionalen Teilklausel festgelegt werden.

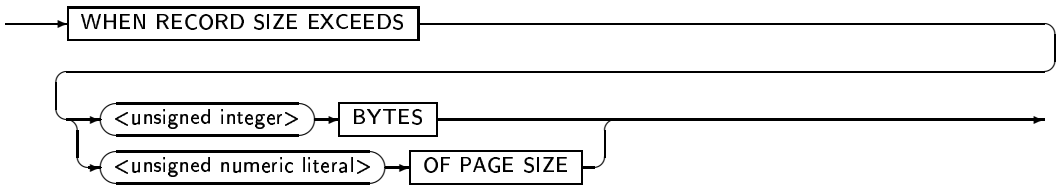
Fehlt die Angabe zur Speicherung, so wird standardmäßig `STORE IN SUPERIOR RECORD TABLESPACE` angenommen (siehe Syntaxregel 18), was bedeutet, daß der Überlaufsatz im gleichen Tablespace wie der ursprüngliche Satz (der nur noch einen Verweis auf den Überlaufsatz enthält) gespeichert wird.

Teilklauseln

- ▷ Syntaxregel 16 *<overflow constraint>*
  Bedingung für die Auslagerung.
- ▷ Syntaxregel 17 *<overflow storage clause>*
  Angaben zur Speicherung des Überlaufsatzes.

Syntaxregel 16

<overflow constraint> ::=



Erläuterung

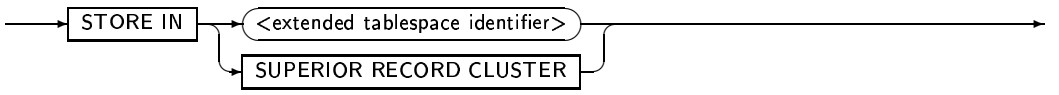
Die Überlaufbedingung gibt dabei an, ab welcher Satzgröße Daten ausgelagert werden sollen. Die Angabe der Grenze kann entweder absolut (in Bytes) oder relativ zur Seitengröße (Dezimalzahl zwischen 0 und 1) erfolgen.

Teilklauseln

- ▷ *<unsigned integer>*
  Größenangabe als nicht negative ganze Zahl nach Norm-SQL.
- ▷ *<unsigned numeric literal>*
  Größenangabe zwischen 0 und 1 als nicht negative Dezimalzahl nach Norm-SQL.

Syntaxregel 17

<overflow storage clause> ::=



Erläuterung

Mit dieser Regel kann der gewünschte Speicherplatz für den Überlaufsatz festgelegt werden. Es kann entweder ein <extended tablespace identifier> angegeben werden oder `STORE IN SUPERIOR RECORD CLUSTER`. Damit kann der Überlaufsatz zusammen mit dem übergeordneten Satz – in diesem Fall also dem Primärsatz – als Cluster gespeichert werden.

Teilklauseln

- ▷ Syntaxregel 18 *<extended tablespace identifier>*
  Identifikation des Tablespaces zur Speicherung des Überlaufsatzes.

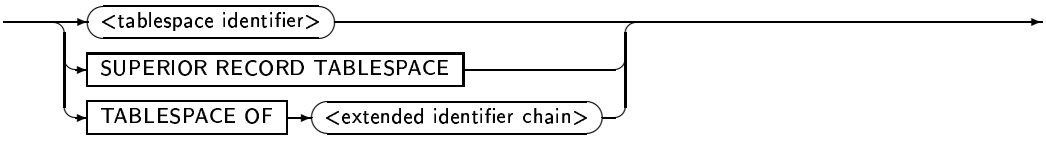
Beispiel

Sätze, die größer als eine halbe Speicherseite sind, sollen in den Überlauf-Tablespace `TS_2` ausgelagert werden:

```
CREATE TABLE Fertigungsmaschine AS Fertigungsmaschine_t
BEGIN PRDL
    FOR LONG RECORDS USE OVERFLOW RECORD
        WHEN RECORD SIZE EXCEEDS 0.5 OF PAGE SIZE
        STORE IN TABLESPACE TS_2
END PRDL
```

SYNTAXREGEL 18

<extended tablespace identifier> ::=



ERLÄUTERUNG

Diese Regel legt die Identifikationsmöglichkeiten für einen Tablespace fest. Neben der expliziten Angabe des Namens ist es mit `SUPERIOR RECORD TABLESPACE` auch möglich, auf den Tablespace zu verweisen, der von dem übergeordneten Satz benutzt wird. Weiterhin kann mit `TABLESPACE OF` auch auf den Tablespace verwiesen werden, in dem ein anderes Objekt liegt.

TEILKLAUSELN

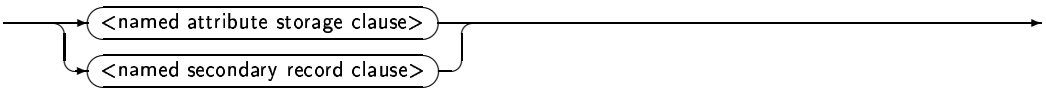
- ▷ SYNTAXREGEL 9 *<tablespace identifier>*
  Identifikation eines Tablespaces.
- ▷ SYNTAXREGEL 41 *<extended identifier chain>*
  Identifikation eines Datenobjektes mit erweiterten Möglichkeiten.

5.2.3 Speicherangaben zu einzelnen Attributen

Der zweite große Abschnitt des PRDL-Blocks aus SYNTAXREGEL 3 <prdl specification> besteht aus mehreren Teilabschnitten der Produktion <named storage clause>.

SYNTAXREGEL 19

<named storage clause> ::=



ERLÄUTERUNG

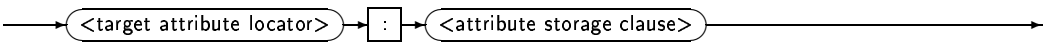
Beide Alternativen dieser Regel ermöglichen es, die Speicherung von Objektteilen, das heißt von ganzen Substrukturen und von einzelnen Attributen, zu steuern und damit die zu speichernden Objekte je nach Anforderung auf Speicherebene (vertikal) zu partitionieren. Die erste Alternative realisiert die Möglichkeit, Einfluß auf die Speicherung einzelner Attribute zu nehmen. Die zweite Alternative realisiert den wahlfreien (attributorientierten) Ansatz zur Zuordnung der Objektattribute zu Sekundärsatzen.

TEILKLAUSELN

- ▷ SYNTAXREGEL 20 *<named attribute storage clause>*
  Speicherangaben zu einzelnen Attributen.
- ▷ SYNTAXREGEL 68 *<named secondary record clause>*
  Speicherangaben zu benannten Sekundärsätzen.

SYNTAXREGEL 20

<named attribute storage clause> ::=



ERLÄUTERUNG

Die Speicherangaben zu einzelnen Attributen werden mit dem Bezeichner des angesprochenen Attributs eingeleitet. Nach einem Doppelpunkt folgt die Spezifikation der Speicherung.

TEILKLAUSELN

- ▷ SYNTAXREGEL 21 *<target attribute locator>*
  Identifikation des Attributs.
- ▷ SYNTAXREGEL 23 *<attribute storage clause>*
  Angaben zur Speicherung des Attributs.

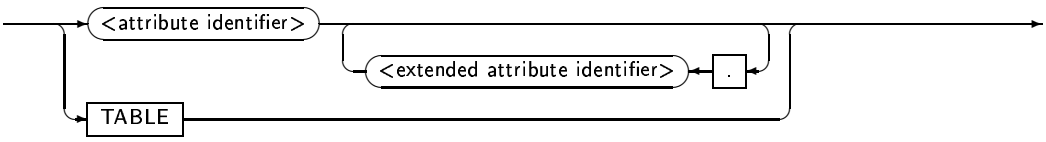
BEISPIEL

Das Grundgerüst für Speicherungsangaben zu einzelnen Attributen, wie hier zum Beispiel für das Attribut `Maschinenlogbuch`, sieht so aus:

```
CREATE TABLE Fertigungsmaschine AS Fertigungsmaschine_t
BEGIN PRDL
    Maschinenlogbuch : <attribute storage clause>
END PRDL
```

SYNTAXREGEL 21

<target attribute locator> ::=



ERLÄUTERUNG

Bei der Identifikation des Attributs kommt der Pfadausdruck mit Punktnotation zum Einsatz. Durch die sukzessive Angabe von Attributbenennungen kann so in Strukturen hineinpositioniert werden. Die erweiterte Identifikationsform kommt dabei für Kollektionen mit unbenannten Elementen zum Einsatz.

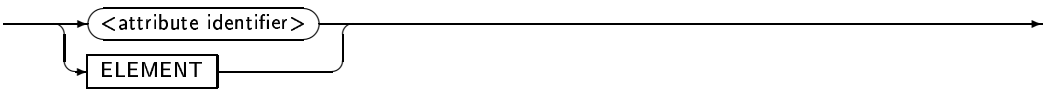
Einen Sonderfall bildet die oberste Ebene des Objekts. Sie kann mit Hilfe des Schlüsselworts `TABLE` adressiert werden.

TEILKLAUSELN

- ▷ SYNTAXREGEL 13 *<attribute identifier>*
  Bezeichnung eines Attributs im Pfadausdruck.
- ▷ SYNTAXREGEL 22 *<extended attribute identifier>*
  Erweiterte Bezeichnung eines Attributs bei Kollektionen.

SYNTAXREGEL 22

<extended attribute identifier> ::=



ERLÄUTERUNG

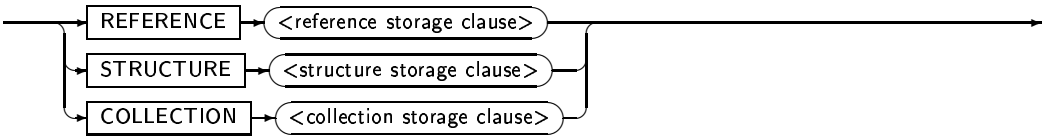
Eine Besonderheit ergibt sich bei der Identifikation von kollektionswertigen Attributen. Um eine Kollektion als Ganzes zu adressieren, genügt der Pfadausdruck mit dem Namen des Attributs. Soll der Elementtyp einer Kollektion adressiert werden, wird an die Adresse das Suffix `.ELEMENT` (mit Trennzeichen Punkt) angehängt. Auf diese Weise ist es auch möglich, Teilattribute innerhalb von Kollektionen zu erreichen. Bei Strukturen wird dagegen ein <attribute identifier> genutzt.

TEILKLAUSELN

▷ SYNTAXREGEL 13 *<attribute identifier>*
Bezeichnung eines Attributs im Pfadausdruck.

SYNTAXREGEL 23

<attribute storage clause> ::=



ERLÄUTERUNG

Diese Regel ermöglicht die Festlegung der Speicherung für dreierlei Arten von Attributen: für strukturierte Attribute, für kollektionswertige Attribute sowie für Referenzattribute. Für sonstige einfache Attribute, wie `INTEGER` oder `CHARACTER`, sind Speicheroptionen nicht sinnvoll und sind dementsprechend auch nicht vorgesehen.

Je nach Art des Attributs kommt eines der Terminale `REFERENCE`, `STRUCTURE` oder `COLLECTION` zum Einsatz, gefolgt von jeweils spezifischen Optionen.

TEILKLAUSELN

▷ SYNTAXREGEL 24 *<reference storage clause>*
Speicherangaben zu Referenzattributen. Ihre Beschreibung folgt im nächsten Abschnitt.

▷ SYNTAXREGEL 25 *<structure storage clause>*
Speicherangaben zu Strukturattributen. Diese werden in Abschnitt 5.2.5 vorgestellt.

▷ SYNTAXREGEL 33 *<collection storage clause>*
Speicherangaben zu Kollektionsattributen, deren Präsentation in Abschnitt 5.2.6 folgt.

BEISPIEL

Da das Attribut `Bearbeitungsfähigkeiten.ELEMENT.Id_Bearbeitungsvorgang` der Beispieltabelle eine Referenz ist und das Attribut `Maschinenlogbuch` von `Fertigungsmaschine` eine Liste ist, werden Speicherungsangaben zu diesen wie folgt eingeleitet:

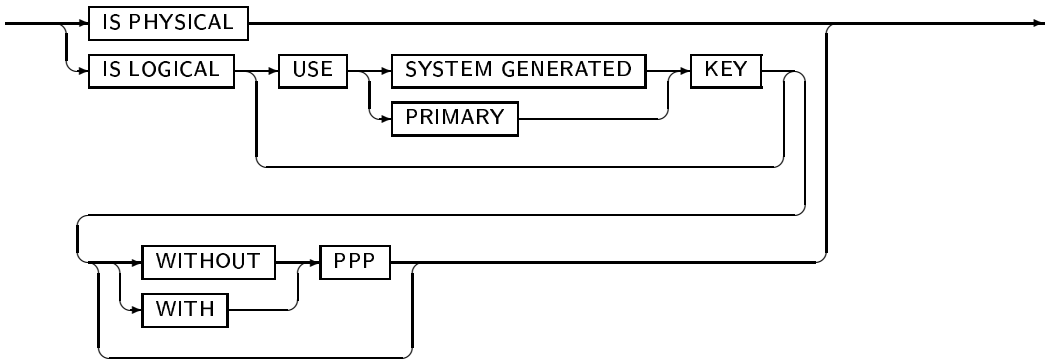
```
CREATE TABLE Fertigungsmaschine AS Fertigungsmaschine_t
BEGIN PRDL
    Bearbeitungsfähigkeiten.ELEMENT.Id_Bearbeitungsvorgang :
        REFERENCE <reference storage clause>
    Maschinenlogbuch :
        COLLECTION <collection storage clause>
END PRDL
```

5.2.4 Speicherangaben zu Referenzattributen

In diesem Abschnitt werden die Möglichkeiten zur Speicherung von Referenzattributen behandelt.

SYNTAXREGEL 24

<reference storage clause> ::=



ERLÄUTERUNG

Diese Regel dient der Spezifikation der Speicherung von Referenzattributen. Es kann bestimmt werden, ob physische Referenzen, logische Referenzen oder logische Referenzen mit PPP benutzt werden sollen (vergleiche Abschnitt 2.5.2.2 und Abschnitt 3.2.4). Werden keine Angaben für ein Referenzattribut gemacht, so wird die physische Referenzierung benutzt.

Bei physischen Referenzen wird die Tupel-ID (TID) des entsprechenden Satzes als Referenz genommen. Über das TID-Konzept wird gewährleistet, daß eine physische Referenz auch nach dem Verschieben des Satzes noch gültig ist. Der Vorteil dieser Variante liegt in der hohen Geschwindigkeit des Dereferenzierens, vorausgesetzt, es fand keine Verschiebung des gewünschten Satzes statt. Ein Nachteil ist daher der steigende Aufwand, wenn Sätze häufig verschoben werden und ein Zugriff sehr oft mit zwei Seitenzugriffen verbunden ist. Ist die Zieltabelle allerdings indexorganisiert gespeichert, so ist die Angabe `IS PHYSICAL` nicht erlaubt, und es muß die bei indexorganisierter Speicherung übliche logische Referenzierung über den Indexschlüssel angewandt (siehe Abschnitt 3.2.4) werden.

Logische Referenzen basieren auf einer expliziten Abbildung eines logischen Schlüssels auf die aktuelle physische Adresse des Satzes. Infolgedessen ist für jede Dereferenzierung eine Abfrage des Abbildungsindexes (oder der Abbildungstabelle) notwendig. Letzterer könnte im Falle von `SYSTEM GENERATED` beispielsweise mit Hilfe eines Feldes implementiert werden, so daß Abfragen in $O(1)$ möglich wären. Wird `PRIMARY KEY` gewählt, so ist die Abbildungsfunktion der Index auf dem Primärschlüssel.

Der Vorteil logischer Referenzen ist, daß die physische Position nur noch an genau einer Stelle – in der Abbildungsfunktion – verwaltet werden muß. Der Hauptnachteil liegt in der geringeren Geschwindigkeit dieses Konzepts gegenüber einer direkten physischen Adresse als Referenz, da bei jeder Dereferenzierung definitiv eine Abfrage an den Abbildungsindex notwendig ist.

Um die Vorteile beider Varianten miteinander zu verknüpfen, ist eine Mischform (hybride Variante) vorgesehen, in der zwar logische Referenzen benutzt werden, bei jeder Referenz aber zusätzlich die Adresse der wahrscheinlichen physischen Position

(PPP – probable position pointer) mit abgespeichert wird. Diese Unterstützung wird für logische Referenzen mit `WITH PPP` aktiviert oder mit `WITHOUT PPP` explizit deaktiviert. Standard ist, daß diese auch als „physical guess“ bezeichnete Zusatzinformation bei logischen Referenzen mitgeführt wird. Deshalb kann `WITH PPP` auch weggelassen werden.

BEISPIEL

Das folgende Beispiel demonstriert, wie für eine Referenz die Benutzung logischer Verweise auf der Basis des Primärschlüssels unter Einbeziehung physischer Positionshinweise festgelegt wird.

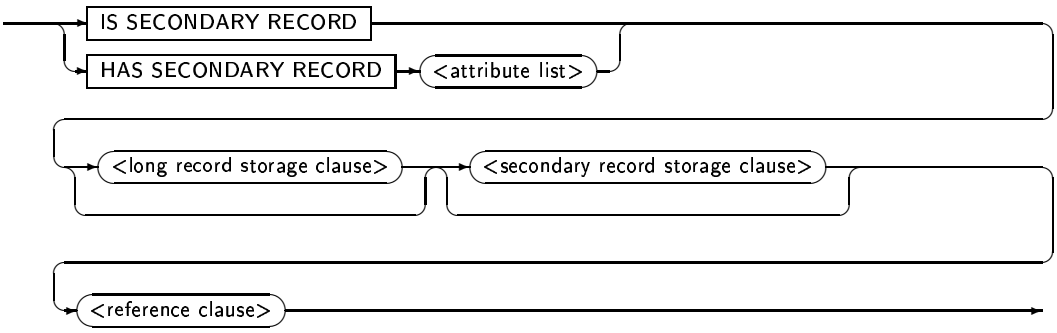
```
CREATE TABLE Fertigungsmaschine AS Fertigungsmaschine_t
BEGIN PRDL
    Bearbeitungsfähigkeiten.ELEMENT.Id_Bearbeitungsvorgang :
        REFERENCE IS LOGICAL USE PRIMARY KEY WITH PPP
END PRDL
```

5.2.5 Speicherangaben zu strukturierten Attributen

Die Spezifikationsmöglichkeiten von PRDL für die Speicherung strukturierter Attribute beschreibt dieser Abschnitt.

SYNTAXREGEL 25

<structure storage clause> ::=



ERLÄUTERUNG

Diese Regel gestattet es, bei der physischen Speicherung von Strukturen einerseits für einzelne Attribute eigene Speicherspezifikationen vorzunehmen und andererseits gezielt Attribute oder Teilattribute der Struktur in eigens dafür definierte Sekundärsätze auszulagern.

Bei `IS SECONDARY RECORD` wird das komplette strukturierte Attribut mit allen seinen Subattributen in einen physischen Sekundärsatz ausgelagert.

Bei `HAS SECONDARY RECORD` werden jedoch nur die in der nachfolgenden Attributliste aufgeführten Subattribute in einen Sekundärsatz ausgelagert.

Optional können per <long record storage clause> weitere Angaben zur Behandlung langer Sätze, die zu groß für eine Speicherseite sind, und per <secondary record storage clause> zur Speicherung des jeweiligen Sekundärsatzes gemacht werden.

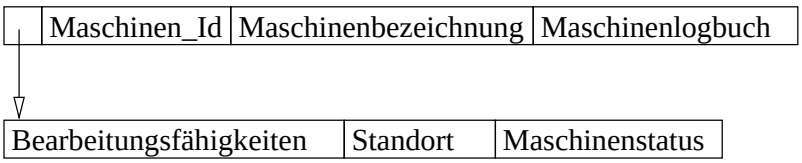


ABBILDUNG 5.2: PRDL-Beispiel für die Auslagerung von Attributen

Als letztes wird mit <reference clause> die Anbindung des definierten Sekundärsatzes an den Primär- oder den übergeordneten Sekundärsatz durch Referenzen spezifiziert.

Die Pfadausdrücke zur Identifizierung von Attributen werden immer relativ zu der Position der betreffenden Struktur angegeben. Das gilt für die auszulagernden Attribute der <attribute list> genauso, wie zum Beipiel für die Schlüsselangaben bei der indexorganisierten beziehungsweise werteclusterbasierten Speicherung im Rahmen der <secondary record storage clause>-Klausel. Wie allgemein üblich, heißt das, daß zum Beispiel der relative Pfadausdruck `Adresse.PLZ` im Kontext von `Abteilung.Sitz` zum absoluten Pfad `Abteilung.Sitz.Adresse.PLZ` erweitert wird.

TEILKLAUSELN

- ▷ SYNTAXREGEL 26 *<attribute list>*
  Auflistung der Attribute, die zum neu definierten Sekundärsatz gehören.
- ▷ SYNTAXREGEL 14 *<long record storage clause>*
  Angaben zur Behandlung langer Sätze.
- ▷ SYNTAXREGEL 30 *<secondary record storage clause>*
  Angaben zur Speicherung des neu definierten Sekundärsatzes.
- ▷ SYNTAXREGEL 27 *<reference clause>*
  Anbindung des neu definierten Sekundärsatzes per Referenzierung.

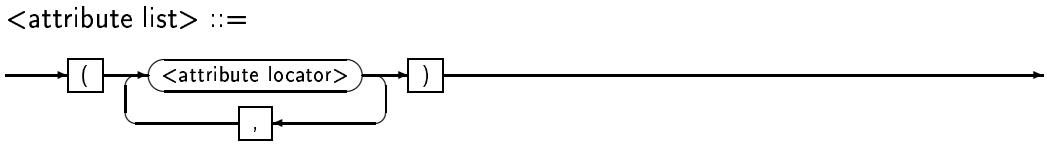
BEISPIEL

Folgendes Beispiel zeigt, wie von der Auslagerung von Teilattributen Gebrauch gemacht wird:

```
CREATE TABLE Fertigungsmaschine AS Fertigungsmaschine_t
BEGIN PRDL
    TABLE : STRUCTURE
        HAS SECONDARY RECORD (Bearbeitungsfähigkeiten,
            Maschinenstandort, Maschinenstatus)
END PRDL
```

Die resultierende Speicherstruktur ist in ABBILDUNG 5.2 dargestellt und zeigt die Auslagerung der Attribute `Bearbeitungsfähigkeiten`, `Maschinenstandort` und `Maschinenstatus` in einen Sekundärsatz. Bei der Auflistung der Attribute entspricht die Angabe eines strukturierten Attributs, als abkürzende Schreibweise, der Aufzählung sämtlicher Unterattribute. Nur wenn ein Unterattribut an anderer Stelle explizit genannt wird, hat diese einzelne Nennung Vorrang vor der impliziten in der abgekürzten Form (vergleiche Attribut `Maschinenstandort`). Attribute, die in keiner Klausel aufgelistet sind, werden standardmäßig dem Primärsatz zugeordnet.

SYNTAXREGEL 26



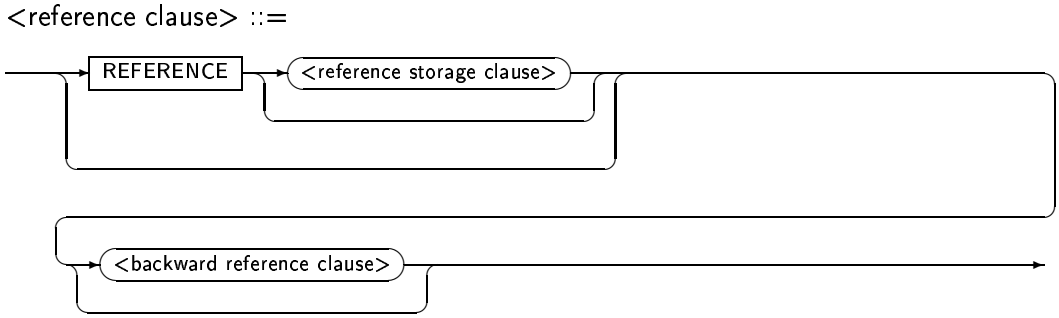
ERLÄUTERUNG

Mit Hilfe dieser Regel kann eine kommagetrennte, nicht leere Attributliste festgelegt werden.

TEILKLAUSELN

▷ SYNTAXREGEL 12 *<attribute locator>*
Identifikation eines Attributs.

SYNTAXREGEL 27



ERLÄUTERUNG

Die Art der Verknüpfung eines neu entstehenden Sekundärsatzes mit den bereits vorhanden Sätzen kann über diese Syntaxregel definiert werden.

Sekundärsätze können entweder mit einer Vorwärtsreferenz, mit einer Rückwärtsreferenz oder mit beiden oder aber überhaupt nicht mit dem übergeordneten Satz verbunden werden.

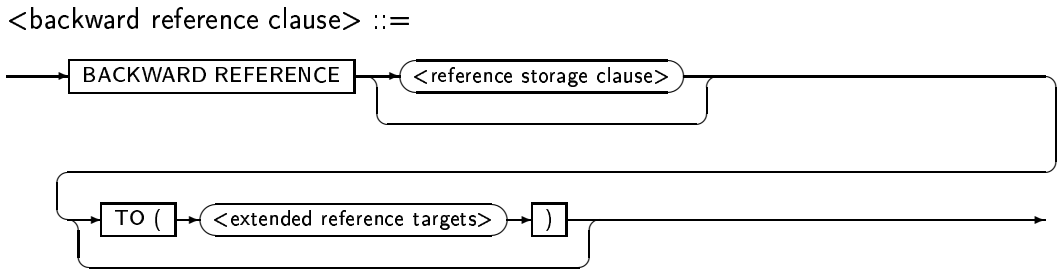
Die Verwendung von Vorwärtsreferenzen wird mit `REFERENCE`, gefolgt von einer optionalen Speicherspezifikation, und Rückwärtsreferenzen über die <backward reference clause> definiert.

TEILKLAUSELN

▷ SYNTAXREGEL 24 *<reference storage clause>*
Angabe zur Speicherung der Vorwärtsreferenz.

▷ SYNTAXREGEL 28 *<backward reference clause>*
Angabe zur Definition einer Rückwärtsreferenz.

Syntaxregel 28



Erläuterung

Diese Syntaxregel erlaubt die Verwendung von Rückwärtsreferenzen von Sekundärsätzen auf den jeweiligen Primär- oder übergeordneten Sekundärsatz. Sie bieten Unterstützung für Anfragen, die beispielsweise anhand von Subobjekteigenschaften auf übergeordnete Objekte zugreifen.

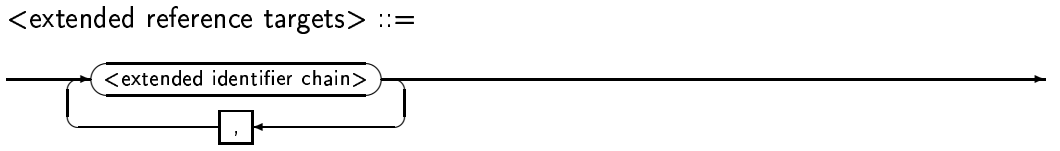
Die Regel ermöglicht die optionale Definition der Speicherung der Rückwärtsreferenz und die optionale Angabe von Referenzzielen, das heißt physischen Sätzen, auf die zurückverwiesen werden soll.

Das Fehlen eines expliziten Referenzziels erzeugt standardmäßig eine Rückwärtsreferenz auf den direkt übergeordneten physischen Satz.

Teilklauseln

- ▷ Syntaxregel 24 *<reference storage clause>*
  Festlegungen zur Speicherung der Rückwärtsreferenzen.
- ▷ Syntaxregel 29 *<extended reference targets>*
  Definition der Ziele der Rückwärtsreferenzen.

Syntaxregel 29



Erläuterung

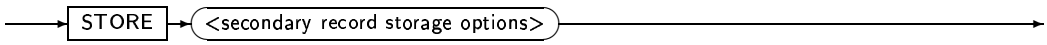
Die Syntax zeigt, daß auch immer mehrere Rückwärtsreferenzen angelegt werden können. Die Definition einer Rückwärtsreferenz besteht jeweils aus der Angabe eines erweiterten Pfadausdrucks.

Teilklauseln

- ▷ Syntaxregel 41 *<extended identifier chain>*
  Erweiterter Pfadausdruck zur Angabe des Rückreferenzziels.

Syntaxregel 30

<secondary record storage clause> ::=



Erläuterung

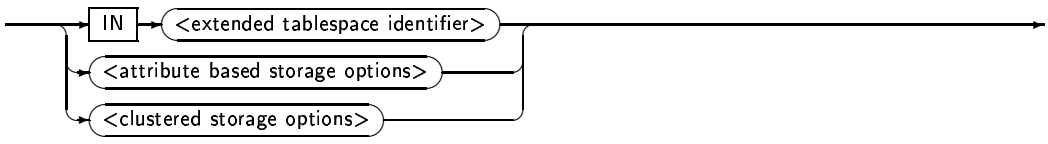
Speicherungsangaben für Sekundärsätze werden in dieser Regel mit `STORE` eingeleitet. Die folgende Angabe des Speicherorts selbst ist zur Wiederverwendung in Syntaxregel 31 <secondary record storage options> ausgelagert.

Teilklauseln

- ▷ Syntaxregel 31 *<secondary record storage options>*
  Angabe des Speicherorts für Sekundärsätze.

Syntaxregel 31

<secondary record storage options> ::=



Erläuterung

Diese Regel dient der Angabe des Speicherorts für Sekundärsätze. Die Produktion <extended tablespace identifier> soll der Vereinfachung dienen und ermöglicht den Verweis auf den Tablespace des um eins höher gelegenen Satzes. Diese Einstellung entspricht auch dem Standardspeicherort von Sekundärsätzen. Mit der Klausel <attribute based storage options> kann eine über definierbare Attribute gesteuerte indexorganisierte oder geclusterte Speicherung spezifiziert werden. Und die letzte Variante eröffnet die Möglichkeit, den Sekundärsatz als Cluster zusammen mit einem bereits existierenden internen Satz zu speichern.

Teilklauseln

- ▷ Syntaxregel 18 *<extended tablespace identifier>*
  Festlegung eines Tablespaces als Speicherort.
- ▷ Syntaxregel 8 *<attribute based storage options>*
  Angaben zur attributbasierten Speicherung.
- ▷ Syntaxregel 32 *<clustered storage options>*
  Angaben zur geclusterten Speicherung.

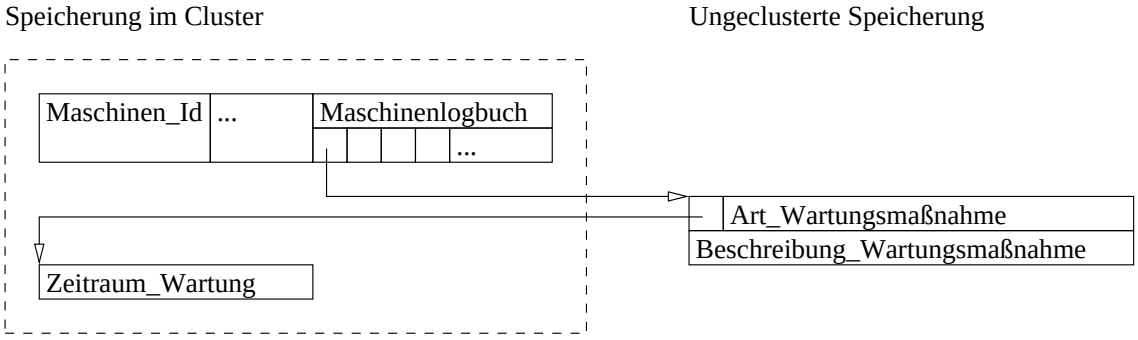
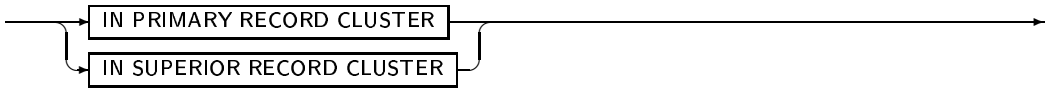


ABBILDUNG 5.3: PRDL-Beispiel für die geclusterte Speicherung von Sekundärsätzen

SYNTAXREGEL 32

<clustered storage options> ::=



ERLÄUTERUNG

Mit dieser Regel können Sekundärsätze zusammen mit ihrem Primärsatz (`IN PRIMARY RECORD CLUSTER`) oder mit ihrem übergeordneten Satz (`IN SUPERIOR RECORD CLUSTER`) in einem Cluster gespeichert werden.

BEISPIEL

Folgendes PRDL-Beispiel und die in ABBILDUNG 5.3 dargestellte resultierende Speicherstruktur demonstrieren die Speicherung im Cluster des Primärsatzes:

```
CREATE TABLE Fertigungsmaschine AS Fertigungsmaschine_t
BEGIN PRDL
    Maschinenlogbuch.ELEMENT : STRUCTURE
        IS SECONDARY RECORD
            REFERENCE IS PHYSICAL
    Maschinenlogbuch.ELEMENT.Zeitraum_Wartung : STRUCTURE
        IS SECONDARY RECORD
            STORE IN PRIMARY RECORD CLUSTER
            REFERENCE IS PHYSICAL

END PRDL
```

5.2.6 Speicherangaben zu Kollektionsattributen

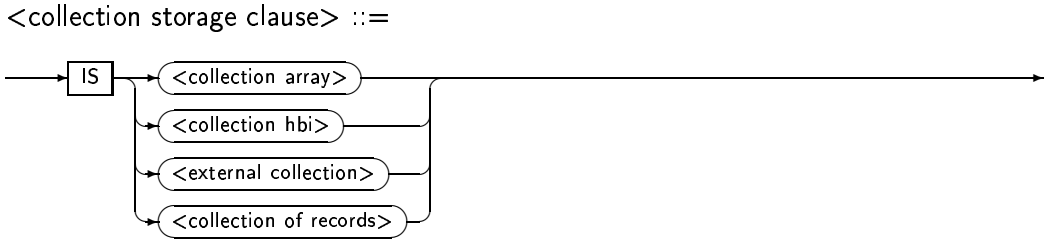
Analog zu den Speicherangaben bei strukturierten Attributen wird ein kollektionswertiges Attribut durch Angabe des Adreßpfades und anschließendem Doppelpunkt ausgewählt. Das Schlüsselwort, das kennzeichnet, daß es sich um eine Kollektion handelt, ist `COLLECTION` gefolgt von der Definition der gewünschten physischen Speicherstruktur mit den Regeln dieses Abschnitts.

Die Definition der gewünschten physischen Speicherstruktur erfolgt mittels einer festen Anzahl sogenannter Konstruktoren. SYNTAXREGEL 33 <collection storage clause> führt diejenigen Kollektionsstrukturen auf, die zur Einlagerung in den übergeordneten Primär- oder Sekundärsatz geeignet sind, sowie die Möglichkeit, die Kollektionsstruktur auszulagern. Die Konstruktoren für ausgelagerte Kollektionen gibt nachfolgend SYNTAXREGEL 34 <collection reference target> an. Die Verbindung beider Produktionen bildet SYNTAXREGEL 45 <external collection>.

Jeder der Konstruktoren der beiden letzten Syntaxregeln steht für eine Art Strukturbaustein, mit deren Hilfe durch Kombination und gegenseitige Einsetzung eine Vielzahl von unterschiedlichsten physischen Speicherstrukturen für Kollektionen entstehen können. Jeder Konstruktor besteht aus einer Liste von Parametern sowie der Möglichkeit, bestimmte andere Konstruktoren in ihn einzusetzen.

Die rekursive Definition der Kollektionsspeicherstruktur beginnt mit der Festlegung des ersten beziehungsweise äußeren Konstruktors über die folgende Syntaxregel.

SYNTAXREGEL 33



ERLÄUTERUNG

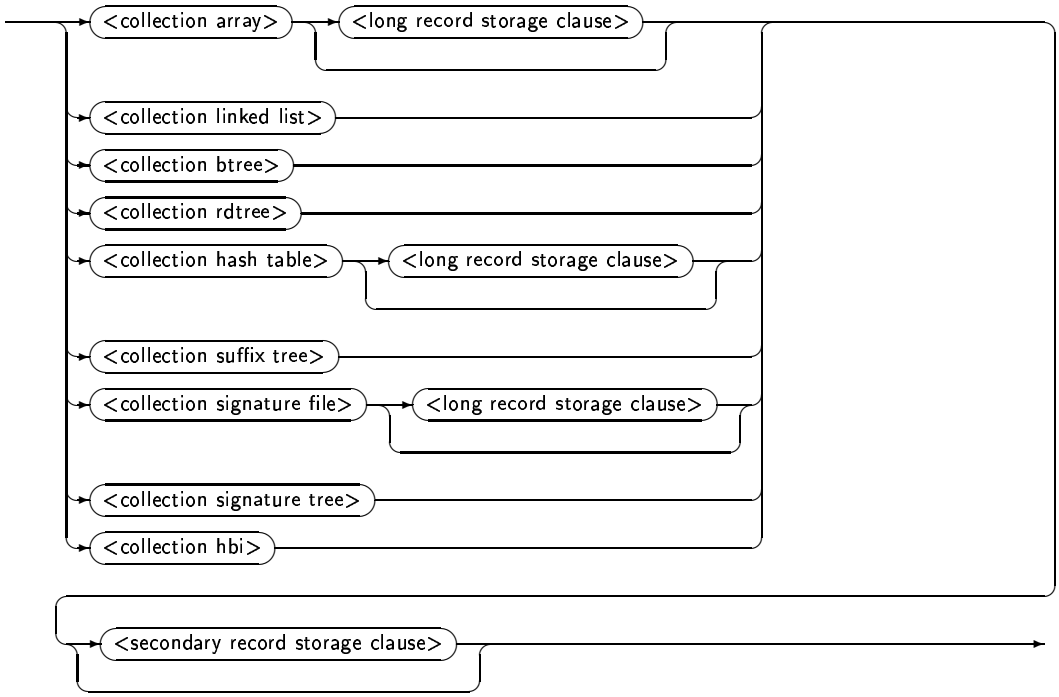
Diese Syntaxregel bietet die Wahl zwischen den Kollektionsstrukturen `ARRAY` und `HIERARCHICAL BITMAP INDEX`, die zur Einlagerung in den übergeordneten Primär- oder Sekundärsatz geeignet sind, sowie die Möglichkeit, mit <external collection> die Kollektionsstruktur oder mit <collection of records> die Kollektionselemente unverbunden auszulagern.

TEILKLAUSELN

- ▷ SYNTAXREGEL 47 *<collection array>*
  Speicherung der Kollektion als Feld.
- ▷ SYNTAXREGEL 63 *<collection hbi>*
  Speicherung der Kollektion als hierarchischer Bitmap-Index.
- ▷ SYNTAXREGEL 45 *<external collection>*
  Speicherung der Kollektion über eine ausgelagerte Kollektionsstruktur.
- ▷ SYNTAXREGEL 44 *<collection of records>*
  Speicherung der Kollektion mit ausgelagerten Elementsätzen ohne Kollektionsstruktur.

SYNTAXREGEL 34

<collection reference target> ::=



ERLÄUTERUNG

Diese Regel faßt die Konstruktoren für ausgelagerte Kollektionen zusammen. Sie erlaubt die Speicherung von Kollektionen als Feld, verkettete Liste, B*-Baum, RD-Baum, Hashtabelle, Suffix-Baum, Signatur-File, Signatur-Baum oder als hierarchischer Bitmap-Index. Nach der Angabe der Kollektionsstruktur kann auch ihr Speicherplatz festgelegt werden. Bei Kollektionsstrukturen, bei denen das sinnvoll ist, kann zudem festgelegt werden, wie diese gespeichert werden sollen, wenn sie die Größe einer Speicherseite überschreiten.

TEILKLAUSELN

- ▷ SYNTAXREGEL 47 *<collection array>*
  Speicherung der Kollektion als Feld.
- ▷ SYNTAXREGEL 50 *<collection linked list>*
  Speicherung der Kollektion als verkettete Liste.
- ▷ SYNTAXREGEL 52 *<collection btree>*
  Speicherung der Kollektion als B*-Baum.
- ▷ SYNTAXREGEL 55 *<collection rdtree>*
  Speicherung der Kollektion als RD-Baum.
- ▷ SYNTAXREGEL 56 *<collection hash table>*
  Speicherung der Kollektion als Hashtabelle.
- ▷ SYNTAXREGEL 59 *<collection suffix tree>*
  Speicherung der Kollektion als Suffix-Baum.
- ▷ SYNTAXREGEL 65 *<collection signature file>*
  Speicherung der Kollektion als Signatur-File.
- ▷ SYNTAXREGEL 67 *<collection signature tree>*

TABELLE 5.1: Klassifizierung der zur Verfügung stehenden Konstruktoren

Konstruktor	Einordnung als			
	Hilfs-	Kollektionskonstruktor		
	konstruktor	baum-basiert	sequenz-basiert	signatur-basiert
EXTERNAL COLLECTION	⊕	—	—	—
COLLECTION OF RECORDS	⊕	—	—	—
EXTERNAL ELEMENT	⊕	—	—	—
B-TREE	—	⊕	—	—
HASH TABLE	—	—	—	⊕
ARRAY	—	—	⊕	—
LINKED LIST	—	—	⊕	—
SIGNATURE FILE	—	—	—	⊕
SIGNATURE TREE	—	⊕	—	⊕
SUFFIX TREE	—	⊕	—	—
RD-TREE	—	⊕	—	—
HIERARCH. BITMAP INDEX	—	—	—	⊕

Speicherung der Kollektion als Signatur-Baum.

▷ SYNTAXREGEL 63 *<collection hbi>*
Speicherung der Kollektion als hierarchischer Bitmap-Index.

▷ SYNTAXREGEL 14 *<long record storage clause>*
Angaben zur Behandlung langer Sätze.

▷ SYNTAXREGEL 30 *<secondary record storage clause>*
Festlegung des Speicherplatzes für die Kollektionsstruktur.

5.2.6.1 Klassifizierung und Aufbau der Konstruktoren

1. Klassifizierung der Konstruktorarten

Alle zur Verfügung stehenden Konstruktoren sind in TABELLE 5.1 aufgeführt. Sie können zwei Hauptgruppen zugeordnet werden: Kollektionskonstruktoren und Hilfskonstruktoren. Beispielsweise ist `LINKED LIST` ein Kollektionskonstruktor und `EXTERNAL COLLECTION` ein Hilfskonstruktor.

Innerhalb eines Ausdrucks zur Definition einer Speicherstruktur erzeugen Kollektionskonstruktoren den Teil der Struktur, in dem sich die 1:n-Beziehung zwischen übergeordneter Struktur und Kollektionselementen widerspiegelt. Daher muß pro Ausdruck auch immer mindestens ein Kollektionskonstruktor vorkommen.

Kollektionskonstruktoren können in drei Gruppen eingeordnet werden. Die erste Gruppe bilden Konstruktore, die auf einem Suchbaum basieren. Dazu gehört zum Beispiel der klassische B*-Baum. Konstruktoren, die Elemente als Sequenz ablegen, heißen sequenzbasierte Konstruktoren. Ein Vertreter dieser Klasse ist zum Beispiel `LINKED LIST`. Konstruktoren wie `SIGNATURE FILE`, die auf der Verwendung von Signaturen aufbauen, werden auch signaturbasierte Konstruktoren genannt.

Konstruktoren sind ineinander verschachtelbar. Dadurch können kompliziertere Zugriffsstrukturen ‚zusammengebaut' werden. So kann beispielsweise eine Kollektionsstruktur, die eine verkettete Liste von physischen Sätzen mit jeweils mehreren Elementen aufweist,

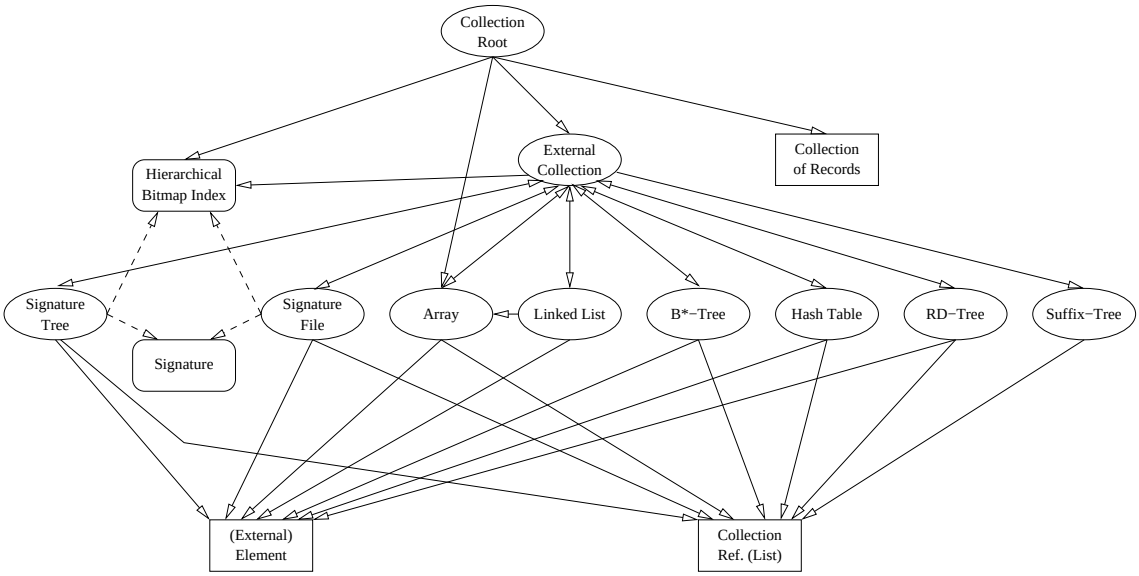
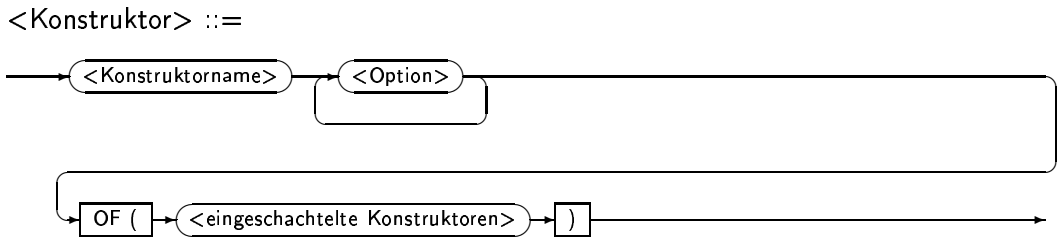


ABBILDUNG 5.4: Schachtelungsmöglichkeiten der Konstruktoren

durch die Einschachtelung eines Feldes in eine verkettete Liste erzeugt werden. Die möglichen Verschachtelungen zeigt ABBILDUNG 5.4. Die durchgezogenen Linien verdeutlichen die Einschachtelung von Kollektionskonstruktoren auf der obersten Ebene in die Wurzel der Kollektion (Collection Root) und die rekursiven Schachtelungsmöglichkeiten. Hier ist auch das eben angesprochene Auftreten eines Feldes als Element einer verketteten Liste wiederzufinden. Weiterhin ist zu erkennen, daß die rekursive Schachtelung von Hashtabellen, B*-Bäumen und verketteten Listen ineinander und in Felder über Referenzen erfolgen muß, da eine direkte Einlagerung keinen Sinn hat. Mit den gestrichelten Pfeilen auf die Kästen mit abgerundeten Ecken wird die Verwendung von Signatur- und Bitmap-Strukturen dargestellt. Die gepunkteten Linien verdeutlichen die Einschachtelung der zwei Hilfskonstruktoren (eckig umrahmt), deren Funktion weiter unten erklärt wird. Auch die einzelenen Schachtelungsmöglichkeiten und -restriktionen werden im folgenden bei der Beschreibung der einzelenen Konstruktoren und Syntaxelemente genauer erklärt.

2. *Aufbau der Konstruktoren*

Alle Konstruktoren sind nach einem bestimmten Schema aufgebaut:



Eine Ausnahme von diesem Schema bilden lediglich `COLLECTION OF RECORDS`, `EXTERNAL COLLECTION` und `COLLECTION ELEMENT`, wie bei der Vorstellung dieser Konstruktoren deutlich werden wird.

Für den Optionsteil eines Konstruktors gibt es eine Reihe von wiederkehrenden Teilklauseln. Jedoch wird das später verwendbare Wort *National* zur Steigerung des Durch-

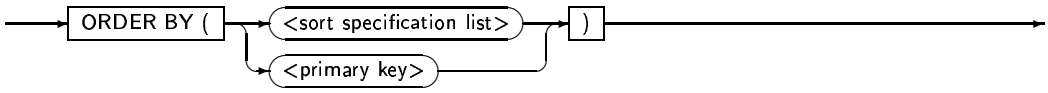
haltewillens beim Lesen mitgeteilt, bevor die Klauseln in den folgenden Unterabschnitten vorgestellt werden.

2.1. Angabe des Indexschlüssels

Folgende Klauseln werden benötigt, um die (Sortier-)Schlüssel einer Zugriffsstruktur deklarieren zu können.

SYNTAXREGEL 35

<order by clause> ::=



ERLÄUTERUNG

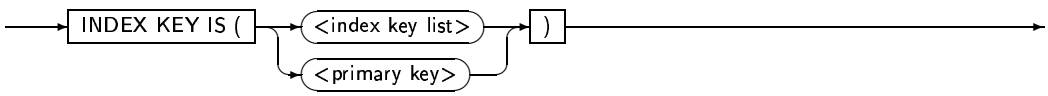
Mit `ORDER BY` können bei Strukturkonstruktoren Einträge ‚künstlich' geordnet gespeichert werden. Die Sortierung kann dabei auf der Grundlage von Attributen in einer <sort specification list> oder auf der Grundlage eines Primärschlüssels, so vorhanden, erfolgen.

TEILKLAUSELN

- ▷ *<sort specification list>*
  Festlegung der Sortierordnung über eine Attributliste nach Norm-SQL.
- ▷ SYNTAXREGEL 37 *<primary key>*
  Festlegung der Sortierordnung über den Primärschlüssel, wenn dieser vorhanden ist.

SYNTAXREGEL 36

<index key clause> ::=



ERLÄUTERUNG

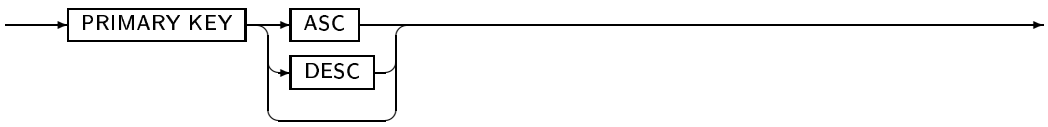
Zur Angabe eines Indexschlüssels kann in dieser Klausel eine Liste von Attributen und Ausdrücken oder, falls vorhanden, der Primärschlüssel herangezogen werden.

TEILKLAUSELN

- ▷ SYNTAXREGEL 38 *<index key list>*
  Festlegung des Indexschlüssels über eine Liste von Attributen oder Ausdrücken.
- ▷ SYNTAXREGEL 37 *<primary key>*
  Festlegung des Indexschlüssels über den Primärschlüssel, falls dieser vorhanden ist.

SYNTAXREGEL 37

<primary key> ::=

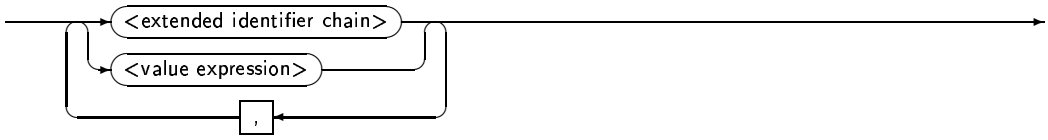


ERLÄUTERUNG

Mit dieser Syntaxregel wird die Nutzung des Primärschlüssels in ab- oder aufsteigender Reihenfolge als Indexschlüssel oder Sortierkriterium definiert. Entfällt die Angabe der Reihenfolge, so wird standardmäßig eine aufsteigende Sortierung angenommen.

SYNTAXREGEL 38

<index key list> ::=



ERLÄUTERUNG

Die Festlegung des Indexschlüssels erfolgt hier über eine Liste von Attributpfaden oder Ausdrücken. Wie schon in SQL99 [ISO99] sind dabei allgemeine Ausdrücke als Indexschlüssel erlaubt. Dies beinhaltet explizit auch Unterabfragen. SQL99 ist dahingehend konzipiert, daß zur Laufzeit eine Fehlermeldung erzeugt wird, falls das Ergebnis einer solchen Unterabfrage nicht skalar ist. Bis auf den Fall, das es sich um einen eindeutigen Index handelt, gilt diese Einschränkung hier explizit nicht mehr: Bei einem mengenwertigen Ergebnis werden entsprechend viele Einträge in der Zugriffsstruktur abgelegt.

TEILKLAUSELN

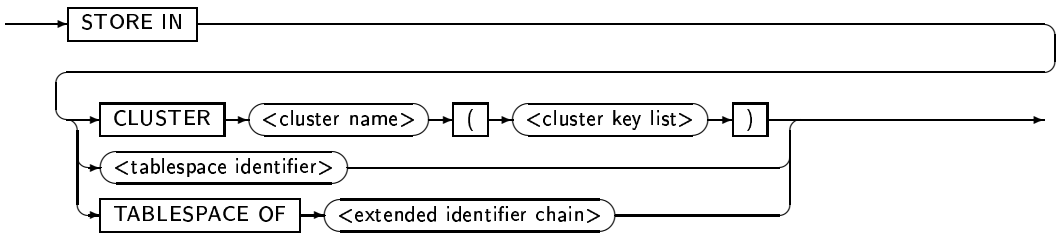
- ▷ SYNTAXREGEL 41 *<extended identifier chain>*
  Erweiterter Pfadausdruck.
- ▷ *<value expression>*
  Ausdruck nach Norm-SQL als Indexschlüsselbestandteil.

2.2. Angabe eines Speicherorts

An verschiedenen Stellen ist es notwendig, auftretenden Satztypen einen Speicherplatz zuzuweisen. Dazu dienen die folgenden Syntaxregeln.

SYNTAXREGEL 39

<record storage option> ::=



ERLÄUTERUNG

Mit dieser Regel kann der Speicherort für physische Sätze angegeben werden. Satztypen können dazu entweder einem bestehenden Cluster zugewiesen oder in Tablespaces abgelegt werden. Tablespaces können dabei entweder direkt benannt oder indirekt über ein (Teil-)Objekt, das in ihnen abgelegt wird, bezeichnet werden.

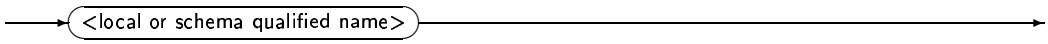
Der Standardspeicherplatz ist immer der datenbankweite Standard-Tablespace beziehungsweise – falls vorhanden – der Speicherplatz des übergeordneten Objekts. Letzteres entspricht in PRDL ausgedrückt: `STORE IN TABLESPACE OF SUPERIOR`.

TEILKLAUSELN

- ▷ SYNTAXREGEL 40 *<cluster name>*
  Identifikation eines benannten Clustertyps.
- ▷ SYNTAXREGEL 42 *<cluster key list>*
  Angabe der Attribute zur Nutzung als Clusterschlüssel.
- ▷ SYNTAXREGEL 9 *<tablespace identifier>*
  Identifikation eines Tablespaces.
- ▷ SYNTAXREGEL 41 *<extended identifier chain>*
  Erweiterter Pfadausdruck.

SYNTAXREGEL 40

<cluster name> ::=



ERLÄUTERUNG

Diese Regel dient der Benennung von Clustern.

TEILKLAUSELN

- ▷ *<local or schema qualified name>*
  Bezeichnung laut Norm-SQL.

SYNTAXREGEL 41

<extended identifier chain> ::=

ERLÄUTERUNG

Hier wird der Aufbau eines Pfadausdrucks zur Bezeichnung von Objekten über erweiterte Identifikatoren definiert. Werden Pfadausdrücke beispielsweise zur Deklaration von Clusterschlüsseln eingesetzt, dann reicht es – im Gegensatz zur Angabe eines Indexschlüssels – nicht mehr aus, in der Hierarchie kollektionswertiger Attribute nur tiefer hineinnavigieren zu können. Um auch höhere Ebenen erreichen zu können, stehen als Identifikatoren die Schlüsselwörter `ROOT` und `SUPERIOR` zur Verfügung. `ROOT` navigiert zu dem Tupel der Relation, also in der Hierarchie ganz nach oben, während `SUPERIOR` nur eine Ebene nach oben navigiert. Beginnt ein Pfadausdruck mit `ROOT`, ist es möglich, durch die Angabe der entsprechenden Attribute in der Hierarchie hinabzusteigen. Mit Hilfe von `ELEMENT` ist es dann bei der Navigation in ein kollektionswertiges Attribut hinein möglich, zu unterscheiden, ob die Kollektion als Ganzes oder nur das einzelne, konkrete Element auf dem Pfad zu dem Ausgangspunkt gemeint ist.

Der Ausgangspunkt für Pfadausdrücke bei der Angabe des Clusterschlüssels hängt von der Art des Satzes ab, für den die Speicherangabe gedacht ist. Werden in dem Satz mehrere Elemente einer Kollektion gespeichert (beispielsweise bei einem Überlaufsatz für ein eingebettetes Feld), so bezieht sich das erste Element eines Pfadausdrucks auf dieselbe Ebene, in der auch die Kollektion gespeichert ist. Ist der Satz genau einem Element einer Kollektion (auch eine Relation kann hier als Kollektion gesehen werden) zugeordnet, befindet sich der Ausgangspunkt für Pfadausdrücke in der obersten Ebene dieses Elements.

Zur Illustration soll aus dem Beispielszenario der Typ `Werkstückart_t` mit dem Listenattribut `Herstellungsablauf` dienen. Jeder Fertigungsschritt eines Herstellungsablaufs wiederum enthält ein Mengenattribut `Bearbeitungskonfiguration` mit Einrichtungsparametern. Um die Einrichtungsparameter in separaten Sätzen zu speichern, wobei alle Einrichtungsparameter eines Fertigungsschritts zusammen geclustert abgelegt werden sollen, muß bei der Deklaration ein entsprechender Bezug auf den Clusterschlüssel möglich sein.

Vorausgesetzt, es wurde bereits der Clustertyp mit Namen Fertigungscluster angelegt, gäbe es zwei Varianten zum Erreichen des beschriebenen Ziels:

- `STORE IN CLUSTER Fertigungscluster (SUPERIOR)`
- `STORE IN CLUSTER Fertigungscluster (ROOT.Herstellungsablauf.ELEMENT)`

In der ersten Form navigiert man innerhalb der Hierarchie um eine Stufe in Richtung Wurzel. Da die Klausel im Kontext der Speicherung der Einrichtungsparameter eingesetzt wurde, ist klar, daß `SUPERIOR` auf die Ebene der Fertigungsschritte verweist. Da kein explizites Attribut in dem Pfadausdruck angegeben wird, wird als Clusterschlüssel die physische Adresse des Fertigungsschritts benutzt. Damit haben also alle Einrichtungsparameter eines bestimmten Fertigungsschritts ein eindeutiges gemeinsames Merkmal, anhand dessen sie geclustert gespeichert werden können.

Die zweite Variante navigiert erst durch `ROOT` auf die Ebene der Wurzel und verweist anschließend durch Angabe des Attributnamens auf das kollektionswertige Attribut `Herstellungsablauf`. Mit dem Schlüsselwort `ELEMENT` wird klargestellt, daß nicht die Kollektion als Ganzes der Clusterschlüssel sein soll, sondern das einzelne Element dieser Kollektion, welches den jeweiligen Fertigungsschritt beinhaltet.

Zu beachten ist, daß ein Auf- und Absteigen in der Hierarchie immer dem aktuellen Pfad folgen muß, das heißt, den entsprechenden Instanzen der Attribute zwischen Wurzel und betrachtetem Teilobjekt, denn die Semantik allgemeiner Pfadausdrücke ist nicht eindeutig, wenn in eine andere Kollektion als die vom Kontext vorgegebene hinein navigiert wird. Bildet zum Beispiel in ABBILDUNG 5.5 das Attribut `Herstellungsablauf.ELEMENT.Id_Bearbeitungsvorgang` von `Werkstückart_t` den Kontext, dann ist nicht eindeutig, welcher Einrichtungsparametername durch die Ausdrücke

- `SUPERIOR.SUPERIOR.Bearbeitungskonfiguration.ELEMENT.Parametername`
- `ROOT.Herstellungsablauf.ELEMENT.Bearbeitungskonfiguration.ELEMENT.Parametername`

ausgehend von `Id_Bearbeitungsvorgang = V111` referenziert wird, `Parametername = P111` oder `P112` oder `P121` oder `P122`. Deshalb ist es sinnvoll, allgemeine Pfadausdrücke nicht zuzulassen und auf den aktuellen Pfad zu beschränken.

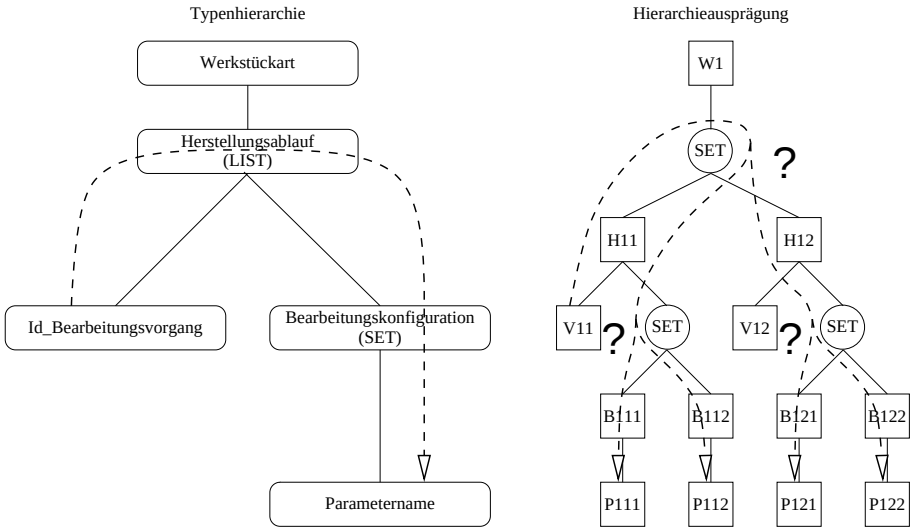


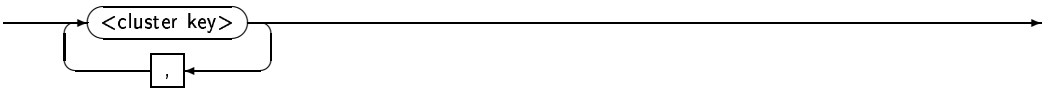
ABBILDUNG 5.5: Mehrdeutigkeit von allgemeinen Pfadausdrücken

TEILKLAUSELN

▷ *<identifier>*
Identifikator nach Norm-SQL.

SYNTAXREGEL 42

<cluster key list> ::=



ERLÄUTERUNG

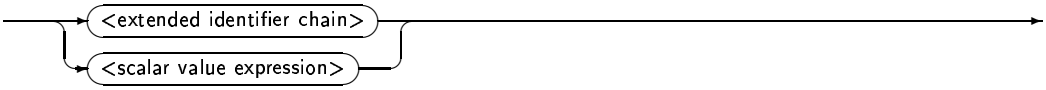
Die Festlegung eines Clusterschlüssels erfolgt hier über eine Liste von Attributen oder Ausdrücken.

TEILKLAUSELN

▷ SYNTAXREGEL 43 *<cluster key>*
Definition eines Attributs oder Ausdrucks als Indexschlüsselbestandteil.

SYNTAXREGEL 43

<cluster key> ::=



ERLÄUTERUNG

Diese Regel legt für Clusterschlüssel fest, daß für sie, wie auch schon bei den Indexschlüsseln, <extended identifier chain>-Klauseln oder allgemeine Ausdrücke zugelassen sind. Im Gegensatz zu Indexschlüsseln müssen diese allerdings linear geordnet beziehungsweise mindestens auf Gleichheit prüfbar sein.

TEILKLAUSELN

- ▷ SYNTAXREGEL 41 *<extended identifier chain>*
  Erweiterter Pfadausdruck.
- ▷ *<scalar value expression>*
  Skalarer Ausdruck nach Norm-SQL als Indexschlüsselbestandteil.

2.3. Referenzen

Referenzen werden sowohl bei der Deklaration der Speicherungsform von einfachen, strukturierten und Kollektionsobjekten als auch bei der Definition von Zugriffsstrukturen benötigt. Werden Daten in separate Sätze ausgelagert, ist es in der Regel möglich, Rückwärtsreferenzen auf Sätze übergeordneter Objekte anzulegen. Beide Möglichkeiten wurden bereits in SYNTAXREGEL 27 <reference clause> vorgestellt.

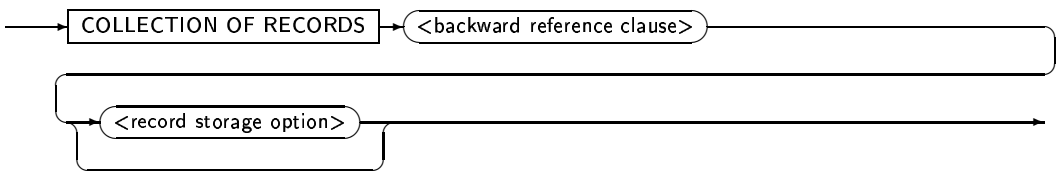
5.2.6.2 Syntax der Kollektionskonstruktoren

Dieser Abschnitt erläutert die Syntaxregeln für die Kollektionskonstruktoren:

- ▷ *COLLECTION OF RECORDS*
  für die Ausgelagerung von unverbundenen Elementsätzen von Kollektionen,
- ▷ *EXTERNAL COLLECTION*
  für die Auslagerung von Kollktionsstrukturen,
- ▷ *ELEMENT*
  für eingelagerte und ausgelagerte, verbundene Kollktionselementsätze,
- ▷ *ARRAY*
  für die Kollektionspeicherung als Feld,
- ▷ *LINKED LIST*
  für die Kollektionspeicherung als verkettete Liste von Sätzen,
- ▷ *B-TREE*
  für die Kollektionspeicherung als B*-Baum,
- ▷ *RD-TREE*
  für die Kollektionspeicherung als RD-Baum,
- ▷ *HASH TABLE*
  für die Kollektionspeicherung als Hashtabelle,
- ▷ *SUFFIX TREE*
  für die Kollektionspeicherung als Suffix-Baum,
- ▷ *SIGNATURE*
  für die Definition von Signaturen,
- ▷ *HIERARCHICAL BITMAP INDEX*
  für die Definition von hierarchischen Bitmap-Indexen,
- ▷ *SIGNATURE FILE*
  für die Kollektionspeicherung als Signatur-File und
- ▷ *SIGNATURE TREE*
  für die Kollektionspeicherung als Signatur-Baum.

SYNTAXREGEL 44

<collection of records> ::=



ERLÄUTERUNG

Der Konstruktor `COLLECTION OF RECORDS` spielt eine gewisse Sonderrolle innerhalb der kollektionswertigen Konstruktoren. Er kann nur als oberster Konstruktor der Hierarchie aufgerufen werden und legt pro Kollektionselement einen Satz mit einer Rückwärtsreferenz auf das übergeordnete Attribut an.

Um die Elemente einer Kollektion mit der Kollektion selber zu verknüpfen, muß in <backward reference clause> eine Rückwärtsreferenz auf das übergeordnete Objekt angegeben werden. Eine Rückwärtsreferenz ist für diese Speicherungsform obligatorisch. Wenn nicht anders angegeben, werden physische Referenzen eingesetzt.

Mit der Option <record storage option> wird der Speicherort der Elementsätze festgelegt. Als Standard wird derselbe Speicherort gewählt, in dem sich der übergeordnete Satz befindet.

TEILKLAUSELN

- ▷ SYNTAXREGEL 28 *<backward reference clause>*
  Rückwärtsreferenzen auf den übergeordneten Kollektionsatz.
- ▷ SYNTAXREGEL 39 *<record storage option>*
  Speicherort der Elementsätze der Kollketion.

BEISPIEL

Der PRDL-Ausdruck

```
COLLECTION OF RECORDS
    BACKWARD REFERENCE
        IS LOGICAL USE PRIMARY KEY WITHOUT PPP
        TO ( SUPERIOR )
    STORE IN TABLESPACE OF SUPERIOR
```

spezifiziert eine Kollektionsspeicherungsform, bei der die Elemente des kollektionswertigen Attributs in einem eigenen physischen Satz im gleichen Tablespace wie der übergeordnete Satz gespeichert werden und mit einer logischen Rückwärtsreferenz (per Fremdschlüssel) auf den entsprechenden übergeordneten Satz zeigen. Diese Speicherungsform ist unter anderem in Oracle bei Nested Tables zu finden.

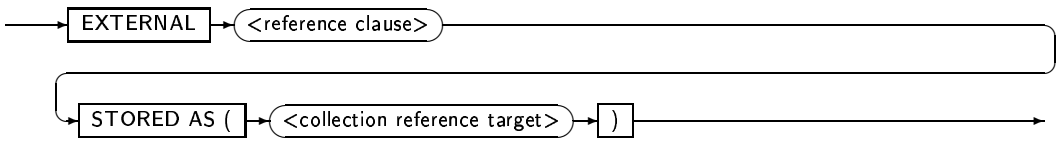
Bezogen auf das Mengenattribut `Maschinenlogbuch` der Fertigungsmaschinentabelle im Beispielszenario würde die Speicherdefinition so aussehen:

```
CREATE TABLE Fertigungsmaschine AS Fertigungsmaschine_t
BEGIN PRDL
    Maschinenlogbuch : COLLECTION IS
        COLLECTION OF RECORDS
            BACKWARD REFERENCE
```

```
                    IS LOGICAL USE PRIMARY KEY WITHOUT PPP
                    TO ( SUPERIOR )
                STORE IN TABLESPACE OF SUPERIOR
    END PRDL
```

Syntaxregel 45

<external collection> ::=



Erläuterung

Der Konstruktor `EXTERNAL COLLECTION` erzeugt einen ausgelagerten Satz zur Aufnahme eines Kollektionsattributs. Nach `EXTERNAL` läßt sich die Verbindung zwischen ihm und dem übergeordneten Satz über Referenzen definieren. Und nach `STORED AS` wird zwischen den Klammern die Repräsentationsform für die extern zu speichernde Kollektion angegeben.

Teilklauseln

- ▷ Syntaxregel 27 *<reference clause>*
  Verbindung zwischen Kollektion und übergeordnetem Satz.
- ▷ Syntaxregel 34 *<collection reference target>*
  Kollektionsrepräsentation.

Beispiel

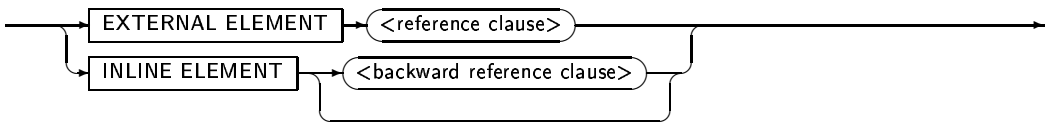
In diesem Beispiel wird eine physisch referenzierte externe Kollektion ohne Rückreferenz angelegt und in Form einer verketteten Liste abgelegt:

```
EXTERNAL
    REFERENCE IS PHYSICAL
STORED AS ( LINKED LIST ... )
```

Der `LINKED-LIST`-Konstruktor wird weiter hinten in Syntaxregel 50 <collection linked list> definiert.

Syntaxregel 46

<collection element> ::=



Erläuterung

Dieser Konstruktor definiert, wie die Elemente einer Kollektion gespeichert werden sollen. Mit `EXTERNAL ELEMENT` erlaubt er die Referenzierung der Elemente aus der Zugriffsstruktur heraus. Mit `INLINE ELEMENT` werden sie direkt in der Kollektionsstruktur (bei Baumen in den Blättern, bei Feldern in den Feldelementen et cetera) gespeichert. Die Art der Referenzierung ausgelagerter Elementsätze wird nach `EXTERNAL`

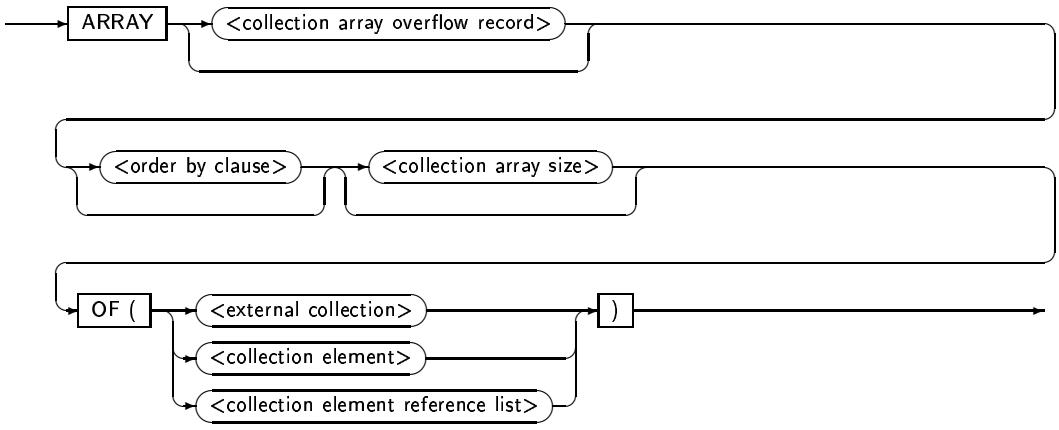
`ELEMENT` festgelegt. Damit auch eingelagerte Elemente auf übergeordnete Sätze verweisen können, wenn zum Beispiel die Kollektionsstruktur ausgelagert ist, können nach `INLINE ELEMENT` Rückwärtsreferenzen spezifiziert werden.

TEILKLAUSELN

- ▷ SYNTAXREGEL 27 *<reference clause>*
  Verbindung zum ausgelagerten Kollektionselementesatz.
- ▷ SYNTAXREGEL 28 *<backward reference clause>*
  Rückreferenzen vom eingelagerten Kollektionselement zu übergeordneten Sätzen.

SYNTAXREGEL 47

<collection array> ::=



ERLÄUTERUNG

Mit diesem Kollektionskonstruktor wird ein eingebettetes Feld angelegt. Elemente des Feldes können entweder mit <external collection> weitere eingeschachtelte Kollektionskonstruktoren oder mit <collection element> Kollektionselemente sein.

Wird der Feldkonstruktor in der Definition einer Zugriffsstruktur benutzt, so kann das Feld mit <collection element reference list> Referenzen auf die indexierten Elemente enthalten.

Die Nutzung von Überlaufsätzen für Felder, die größer als eine Seite sind, kann mit <collection array overflow record> festgelegt werden. Ohne Angabe der Option werden keine Überlaufsätze genutzt.

Mit <order by clause> kann auf den Elementen des Arrays eine Sortierordung festgelegt werden. Wird jedoch mit dieser physischen Struktur die logische Kollektionsart `ARRAY` implementiert, so ist die Angabe eines Ordnungskriteriums nicht zulässig, da der Zugriff und das Einfügen von Elementen über die Feldposition erfolgt.

Mit <collection array size> wird optional die Größe des Feldes festgelegt.

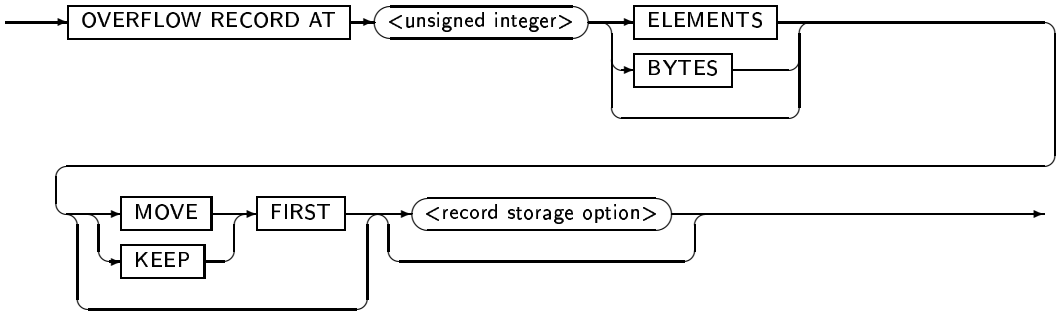
TEILKLAUSELN

- ▷ SYNTAXREGEL 48 *<collection array overflow record>*
  Festlegungen für Überlaufsätze.
- ▷ SYNTAXREGEL 35 *<order by clause>*
  Sortierreihenfolge auf dem Feld.
- ▷ SYNTAXREGEL 49 *<collection array size>*
  Größenfestlegungen für das Feld.

- ▷ SYNTAXREGEL 45 *<external collection>*
  Verweis auf eine weitere Kollektionsstruktur.
- ▷ SYNTAXREGEL 46 *<collection element>*
  Ein- oder ausgelagertes Kollektionselement.
- ▷ SYNTAXREGEL 53 *<collection element reference list>*
  Verweise auf die indexierten Elemente bei der Definition einer Zugriffsstruktur.

SYNTAXREGEL 48

<collection array overflow record> ::=



ERLÄUTERUNG

Mit dieser Regel kann die Auslagerung von Feldelementen ab einer bestimmten Anzahl beziehungsweise Größe deklariert werden. `MOVE FIRST` legt fest, daß beim Überschreiten der angegeben Anzahl die bereits vorhandenen Elemente mit ausgelagert werden. `KEEP FIRST` beläßt die vorderen Elemente in dem Feld und ist die Standardeinstellung.

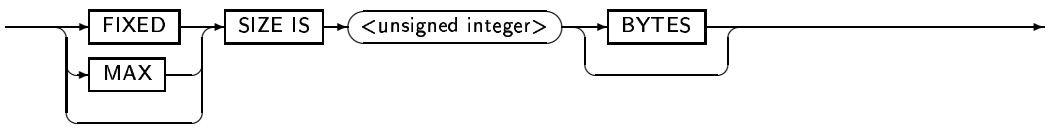
Die ausgelagerten Elemente werden in einem einzigen separaten Satz gespeichert. Mit <record storage option> kann der Speicherort dieses Satzes festgelegt werden. Ohne diese Angabe werden sie im selben Tablespace abgelegt, in dem sich bereits das Feld befindet. Eine sehr sinnvolle Möglichkeit besteht hier in der Angabe eines LOB-Tablespace, so daß sehr große Felder (Arrays) enstehen können.

TEILKLAUSELN

- ▷ *<unsigned integer>*
  Größenangabe als nicht negative ganze Zahl nach Norm-SQL.
- ▷ SYNTAXREGEL 39 *<record storage option>*
  Speicherort der Überlaufsätze.

SYNTAXREGEL 49

<collection array size> ::=



ERLÄUTERUNG

Mit dieser Regel wird die Größe des Feldes festgelegt. Mit `FIXED` beziehungsweise `MAX` kann bestimmt werden, ob es sich um die Angabe einer festen oder maximalen Größe handeln soll. Durch die Angabe von `BYTES` wird die Größe nicht mehr über die Elementanzahl, sondern über die reale physische Länge bestimmt.

TEILKLAUSELN

▷ *<unsigned integer>*
Größenangabe als nicht negative ganze Zahl nach Norm-SQL.

BEISPIEL

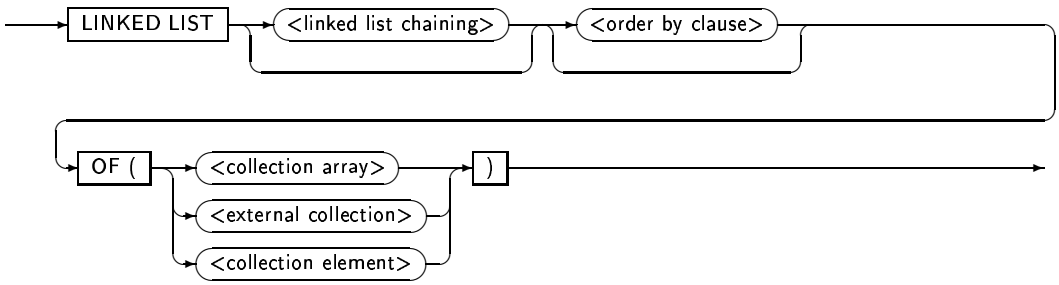
Hier wird der Konstruktor zur Definition eines Feldes fester Länge eingesetzt:

```
ARRAY
    FIXED SIZE IS 2
    OF (...)
```

Diese Klausel könnte beispielsweise in einem `LINKED-LIST`-Konstruktor verwendet werden und würde definieren, daß in jedem Satz der verketteten Liste ein Feld der Länge 2 angelegt wird.

SYNTAXREGEL 50

<collection linked list> ::=



ERLÄUTERUNG

Mit diesem kollektionswertigen Konstruktor wird eine Datenstruktur nach dem Prinzip der verketteten Liste erzeugt. Er muß innerhalb eines `EXTERNAL-COLLECTION`-Konstruktors benutzt werden, durch den unter anderem der Speicherort der verketteten Liste festgelegt wird.

In der Option <linked list chaining> kann die Art der Verkettung festgelegt werden: einfach oder doppelt. Doppelt ist die Voreinstellung.

Mit <order by clause> kann ein Ordnungskriterium für die Liste festgelegt werden. Die Angabe eines Ordungskriteriums ist nur dann erlaubt, wenn der Konstruktor nicht zur Deklaration der Primärspeicherstruktur des logischen Kollektionstyps `LIST` eingesetzt wird, da in diesem Fall die Ordnung von den Operationen auf der Liste bestimmt wird.

Die Elemente der verketteten Liste werden in `OF (...)` angegeben. <collection array> steht dabei für die Definition von in die Listenelemente eingeschachtelten Feldern. Die letzten beiden Klauseln <external collection> und <collection element> haben die gleichen Bedeutungen wie bei SYNTAXREGEL 47 <collection array>.

Verkettete Listen können nicht als Zugriffspfadstruktur verwendet werden.

TEILKLAUSELN

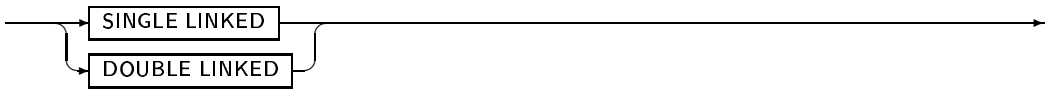
▷ SYNTAXREGEL 51 *<linked list chaining>*
Art der Verkettung.

▷ SYNTAXREGEL 35 *<order by clause>*
Sortierreihenfolge auf der Liste.

▷ SYNTAXREGEL 47 *<collection array>*
In die Listenelemente eingeschachtelte Felder.
▷ SYNTAXREGEL 45 *<external collection>*
Verweis auf eine weitere Kollektionsstruktur.
▷ SYNTAXREGEL 46 *<collection element>*
Ein- oder ausgelagertes Kollektionselement.

SYNTAXREGEL 51

<linked list chaining> ::=



ERLÄUTERUNG

Die Art der Verkettung von Listen wird mit dieser Regel festgelegt. Es kann entschieden werden, ob die Sätze einfach oder doppelt (letzteres ist die Voreinstellung) miteinander verkettet werden sollen.

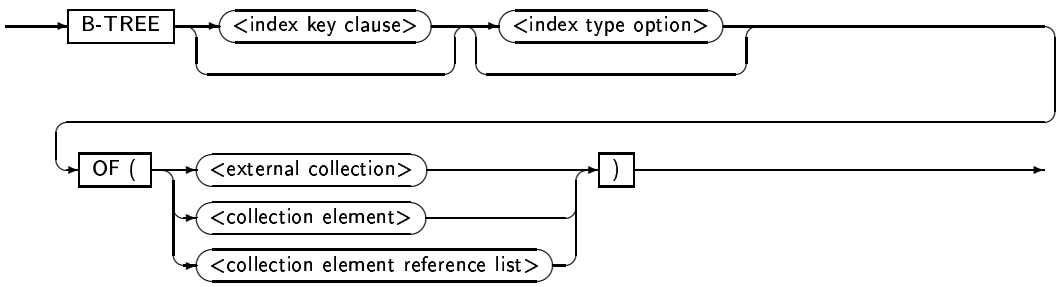
BEISPIEL

In diesem Beispiel wird eine doppelt verkettete Liste von Elementen einer Kollektion erzeugt, die nach der Elementnummer geordnet ist:

```
LINKED LIST
    DOUBLE LINKED
    ORDER BY ( Nr )
OF ( ... )
```

SYNTAXREGEL 52

<collection btree> ::=



ERLÄUTERUNG

Es wird eine Indexstruktur vom Typ B*-Baum angelegt. Je nach Art des eingesetzten Konstruktors innerhalb der Klammern kann mit dieser Klausel sowohl eine einfache Zugriffsstruktur als auch indexorganisierte Speicherung beschrieben werden: Dazu haben <external collection>, <collection element> und <collection element reference list> die gleiche Bedeutung wie bei SYNTAXREGEL 47 <collection array>. Es sei jedoch darauf hingewiesen, daß mit der mehrfachen Spezifikation von Elementreferenzen in <collection element reference list> ein Generalized Access Path erzeugt werden kann.

Mit <index key clause> werden die Indexschlüssel deklariert. Standardmäßig wird der Primärschlüssel des indexierten Elements verwendet, wenn diese Option nicht verwendet wird. Wird ein objektübergreifender Pfadausdruck oder ein Verbundausdruck

zur Schlüsseldeklaration eingesetzt, handelt es sich bei der Indexstruktur um einen Nested Index.

<index type option> gestattet es, einen Indextyp zu spezifizieren, so daß es möglich ist, mit Hilfe eines klassischen B*-Baums oder eines Nested Indexes einen Pfad- beziehungsweise Multiindex zu definieren.

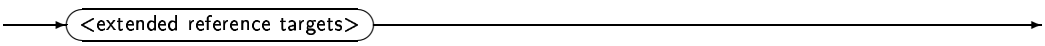
Parameter für B*-Bäume, wie die Knotenauslastung oder die Knotengröße, werden hier nicht weiter thematisiert, da sie stark von der genauen Implementierung eines Baums abhängen. Für diese Arbeit wird im weiteren davon ausgegangen, daß ein Knoten eines B*-Baums immer die Größe einer Seite hat.

TEILKLAUSELN

- ▷ SYNTAXREGEL 36 *<index key clause>*
  Angabe des Indexschlüssels.
- ▷ SYNTAXREGEL 54 *<index type option>*
  Auswahl einer Indexart.
- ▷ SYNTAXREGEL 45 *<external collection>*
  Verweis auf eine weitere Kollektionsstruktur.
- ▷ SYNTAXREGEL 46 *<collection element>*
  Ein- oder ausgelagertes Kollektionselement.
- ▷ SYNTAXREGEL 53 *<collection element reference list>*
  Verweise auf die indexierten Elemente bei der Definition einer Zugriffsstruktur.

SYNTAXREGEL 53

<collection element reference list> ::=



ERLÄUTERUNG

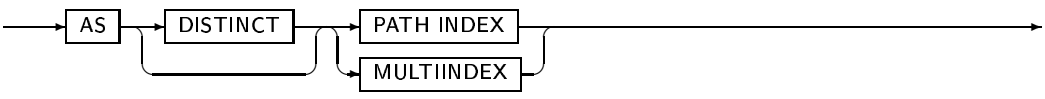
Bei der Definition einer Zugriffsstruktur werden mit dieser Syntaxregel die Verweise auf die indexierten Elemente definiert. Es können mehrfache Elementreferenzen angelegt werden, so daß auch ein Generalized Access Path erzeugt werden kann.

TEILKLAUSELN

- ▷ SYNTAXREGEL 29 *<extended reference targets>*
  Verweise auf die indexierten Elemente.

SYNTAXREGEL 54

<index type option> ::=



ERLÄUTERUNG

Diese Syntaxregel gestattet die Festlegung des Indextyps: Durch die Alternativen `PATHINDEX` und `MULTIINDEX` ist es möglich, anstelle eines Nested Index einen Pfad- beziehungsweise Multiindex zu definieren.

Wird der Indexschlüssel des verschachtelten Index allerdings nicht durch einen einfachen Pfadausdruck, sondern durch komplexe Verbundoperationen definiert, kann es sich für das DBMS als schwierig oder unmöglich erweisen, die einzelnen Stufen des

Nested Index zu identifizieren. Für eine konkrete Implementierung würde es sich daher anbieten, bei der Definition eines Pfad oder Multiindexes die Freiheitsgrade bei Verbundausdrücken einzuschränken und beispielsweise nur eine festgelegte Syntaxvariante zu erlauben.

BEISPIEL

Ausgehend von der Tabelle `Fertigungsmaschine` und ihrem mengenwertigen Attribut `Bearbeitungsfähigkeiten` kann mit folgendem Ausdruck eine Zugriffsstruktur definiert werden, die anhand eines Werksgebäudes alle in den Bearbeitungsfähigkeiten enthaltenen möglichen Bearbeitungsvorgänge findet:

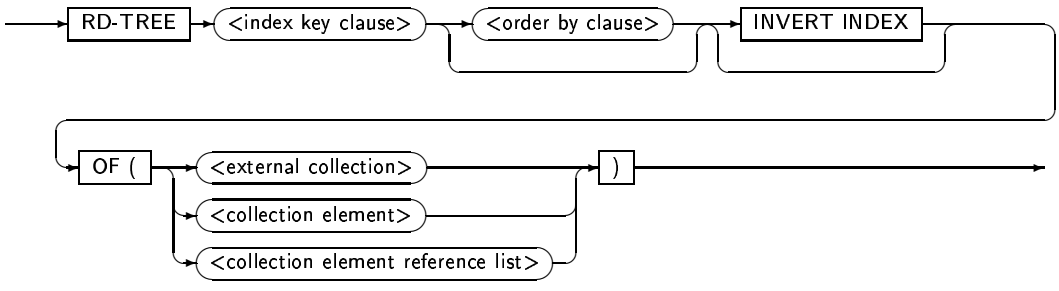
```
CREATE INDEX Werksgebäude_Bearbeitungsfähigkeit
    ON Fertigungsmaschine.Bearbeitungsfähigkeiten
    AS EXTERNAL STORED AS (
        B-TREE
            INDEX KEY IS ( Fertigungsmaschine.Standort
                                        .Werksgebäude )
            AS MULTIINDEX
        OF ( ELEMENT )
    )
```

Das Schlüsselwort `MULTIINDEX` regelt, daß anstelle eines Indexes drei Indexe angelegt werden (siehe dazu Abschnitt 4.1.2.3):

- ▷ Ein Index, der zu einem Werksgebäude alle dort aufgestellten Fertigungsmaschinen findet
- ▷ Ein Index, der zu einer Maschine alle Bearbeitungsvorgänge findet, die in ihren Bearbeitungsfähigkeiten enthalten sind.
- ▷ Außerdem noch ein hier nicht notwendiger (aber für ‚umgekehrte' Anfragen hilfreicher) Index, der zu einer Bearbeitungsfähigkeit alle Maschinen findet, die diese besitzen.

SYNTAXREGEL 55

<collection rdtree> ::=



ERLÄUTERUNG

Mit diesem Konstruktor kann eine Kollektion von eingeschachtelten Kollektionen über einen RD-Baum gespeichert beziehungsweise indexiert werden. Dabei können zwei Fälle unterschieden werden:

- ▷ Die Elemente der äußeren Kollektion bestehen aus eingeschachtelten Mengen, deren Elemente direkt auf ganze Zahlen abgebildet werden können. In diesem Fall ist es möglich, die gesamte Kollektion von Kollektionen innerhalb des RD-Baums

abzuspeichern, und in den Konstruktor kann <collection element> eingesetzt werden.

- ▷ Oder aber der RD-Baum wird nur als Index auf die ausgelagert gespeicherten Elemente genutzt (<external collection>). Voraussetzung für eine Nutzung des RD-Baums ist aber auch hier wieder das Vorhandensein einer weiteren Kollektion in den Elementen der indexierten Kollektion.

Die obligatorische Spezifikation <index key clause> wählt ein eingeschachteltes Kollektionsattribut aus, auf den der RD-Baum aufgebaut wird. Mit <order by clause> wird optional innerhalb der Elemente der indexierten Kollektionen ein (Schlüssel-)Attribut zur Identifizierung der Elemente in den Mengen im RD-Baum angegeben. Entfällt diese Angabe, so muß es entweder einen definierten Primärschlüssel geben, oder die Identität der Elemente wird über die Konkatenation aller ihrer Attribute bestimmt.

Durch die Angabe des Schlüsselworts `INVERT INDEX` ist die Definition eines invertierten RD-Baums möglich.

<external collection>, <collection element> und <collection element reference list> haben die gleiche Bedeutung wie in Syntaxregel 52 <collection btree>.

Teilklauseln

- ▷ Syntaxregel 36 *<index key clause>*
  Angabe einer eingeschachtelten Kollektion über der der RD-Baum aufgebaut wird.
- ▷ Syntaxregel 35 *<order by clause>*
  Identifikation der Kollektionselemente.
- ▷ Syntaxregel 45 *<external collection>*
  Verweis auf eine weitere Kollektionsstruktur.
- ▷ Syntaxregel 46 *<collection element>*
  Ein- oder ausgelagertes Kollektionselement.
- ▷ Syntaxregel 53 *<collection element reference list>*
  Verweise auf die indexierten Elemente bei der Definition einer Zugriffsstruktur.

Beispiel

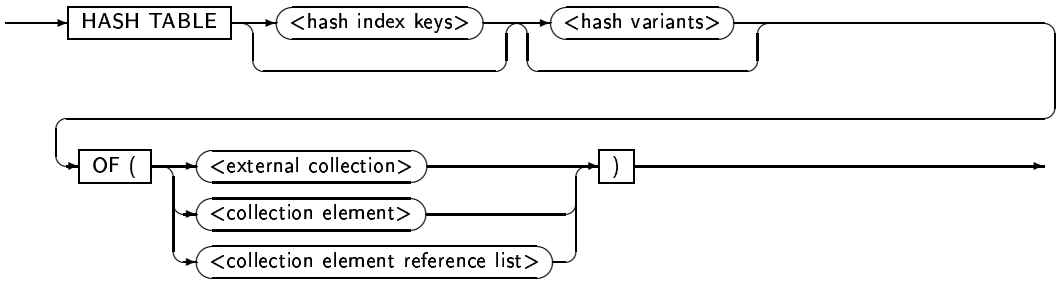
Dieses Beispiel definiert einen RD-Baum für die Werkstücktabelle aus dem Beispielszenario. Der Index wird über den in die Werkstückinstanzen eingeschachtelten Listen von Fertigungsschritten (Listenattribut `Herstellungsablauf`) aufgebaut. Als Identifikator für die Fertigungsschritte dient dabei das Attribut `Id_Bearbeitungsvorgang`.

```
RD-TREE
    INDEX KEY IS ( Herstellungsablauf )
    ORDER BY ( Id_Bearbeitungsvorgang )
    OF (...)
```

Sehr gut eignet sich die so entstandene Struktur beispielsweise zur Suche nach Teilmengen relevanter Bearbeitungsvorgänge in den Herstellungsabläufen der Werkstücke.

SYNTAXREGEL 56

<collection hash table> ::=



ERLÄUTERUNG

Dieser Konstruktor bewirkt das Anlegen einer Hashtabelle zur Indexierung und Referenzierung von Kollektionselementen.

Analog zur Angabe des Indexschlüssels bei B*-Bäumen kann mit <hash index keys> der Schlüssel für die Hashfunktion angegeben werden. Mit der Option <hash variants> wird der genaue Typ des Hashindex festgelegt.

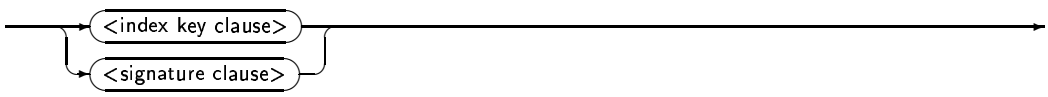
Auch hier haben wieder <external collection>, <collection element> und <collection element reference list> die gleiche Bedeutung wie in SYNTAXREGEL 52 <collection btree>.

TEILKLAUSELN

- ▷ SYNTAXREGEL 57 *<hash index keys>*
  Angabe des Schlüssels für die Hashfunktion.
- ▷ SYNTAXREGEL 58 *<hash variants>*
  Festlegung des Hashverfahrens.
- ▷ SYNTAXREGEL 45 *<external collection>*
  Verweis auf eine weitere Kollektionsstruktur.
- ▷ SYNTAXREGEL 46 *<collection element>*
  Ein- oder ausgelagertes Kollektionselement.
- ▷ SYNTAXREGEL 53 *<collection element reference list>*
  Verweise auf die indexierten Elemente bei der Definition einer Zugriffsstruktur.

SYNTAXREGEL 57

<hash index keys> ::=



ERLÄUTERUNG

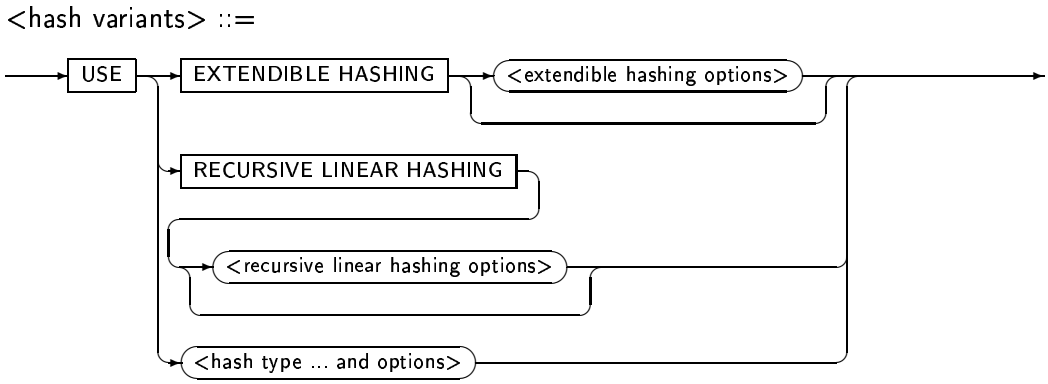
Diese Regel gestattet im Rahmen der Definition von Hashtabellen die Festlegung des Schlüssels für die Hashfunktion. Dies kann per <index key clause> durch die direkte Angabe von Attributen erfolgen oder per <signature clause> über die Erzeugung einer auf Attributen basierenden Signatur.

TEILKLAUSELN

- ▷ SYNTAXREGEL 36 *<index key clause>*
  Festlegung des Schlüssels für die Hashfunktion direkt über Attribute.

▷ SYNTAXREGEL 66 *<signature clause>*
Festlegung des Schlüssels für die Hashfunktion über eine Signatur.

SYNTAXREGEL 58



ERLÄUTERUNG
Mit dieser Option wird der genaue Typ des Hashindex festgelegt. Neben den in Abschnitt 4.1.3.2 vorgestellten beiden Hashverfahren (erweiterbares und rekursiv lineares Hashing) sind beliebige weitere Varianten denkbar. Standardwahl ist die erweiterbare Variante. Spezielle Optionen dieser Verfahren sind stark implementierungsabhängig und werden daher nicht weiter diskutiert.

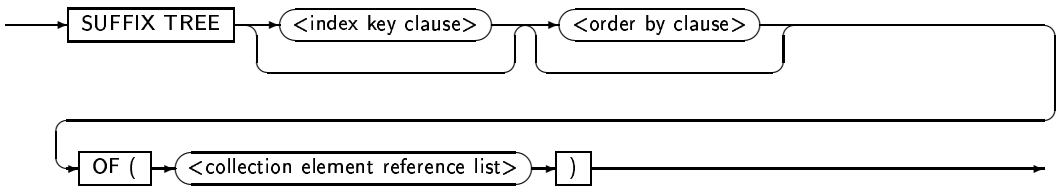
TEILKLAUSELN
- ▷ *<extendible hashing options>*
  Angabe von Optionen für das erweiterbare Hashing. Diese werden im Rahmen dieser Arbeit nicht diskutiert.
- ▷ *<recursive linear hashing options>*
  Angabe von Optionen für das linear rekursive Hashing. Auch Diese Optionen werden im Rahmen dieser Arbeit nicht diskutiert.
- ▷ *<hash type ... and options>*
  Möglichkeit zur Integration und Spezifikation weiterer hier nicht diskutierter Hashverfahren.

BEISPIEL
Im Beispiel wird ein Hashindex auf dem Primärschlüssel einer Kollektion erzeugt. Als Hashvariante kommt erweitertes Hashing zum Einsatz:

```
HASH TABLE
    INDEX KEY IS ( PRIMARY KEY )
    USE EXTENDIBLE HASHING
    OF ( ... )
```

SYNTAXREGEL 59

<collection suffix tree> ::=



ERLÄUTERUNG

Dieser Konstruktor ermöglicht die Definition eines Suffix-Baumes auf einer oder mehreren vorhandenen Kollektionen. Die Struktur kann nur als Sekundärzugriffsstruktur eingesetzt werden.

Mit der Option <index key clause> wird der Schlüssel ausgewählt, nach dem der Suffix-Baum aufgebaut wird. Standardmäßig wird der Primärschlüssel der Elemente angenommen.

Vorausgesetzt, die indexierte logische Kollektionsart ist nicht vom Typ Liste oder Feld, so kann in <order by clause> deklariert werden, nach welchem Schlüssel die Kollektion sortiert gesehen werden soll. Anhand dieser Sortierung wird dann ein Suffix-Baum aufgebaut.

TEILKLAUSELN

- ▷ SYNTAXREGEL 36 *<index key clause>*
  Angabe des Schlüssels für den Suffix-Baum.
- ▷ SYNTAXREGEL 35 *<order by clause>*
  Sortierreihenfolge für die Kollektionselemente.
- ▷ SYNTAXREGEL 53 *<collection element reference list>*
  Verweise auf die indexierten Elemente.

BEISPIEL

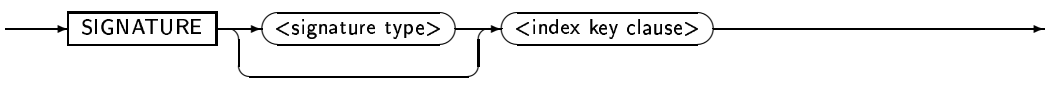
Bezogen auf die Herstellungsabläufe für Werkstücke aus dem Beispielszenario wird hier ein Suffix-Baum über den enthaltenen Folgen von Fertigungsschritten angelegt.

```
SUFFIX TREE
    INDEX KEY IS ( Werkstückart.Herstellungsablauf )
    OF (...)
```

Mit dem entstandenen Index kann dann nach Teilabläufen in der Herstellung von Werkstücken gesucht werden.

SYNTAXREGEL 60

<collection signature> ::=



ERLÄUTERUNG

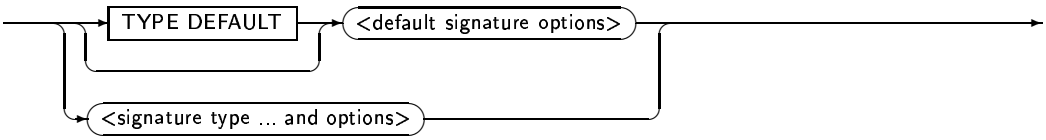
Mit diesem Konstruktor wird eine Signatur anhand der in <index key clause> aufgelisteten Schlüssel erzeugt. Als Schlüssel sind wieder beliebige Ausdrücke erlaubt. Der Signaturtyp wird in <signature type> festgelegt.

TEILKLAUSELN

- ▷ SYNTAXREGEL 61 *<signature type>*
  Festlegung des Signaturverfahrens.
- ▷ SYNTAXREGEL 36 *<index key clause>*
  Angabe der Schlüssel für die Signatur.

SYNTAXREGEL 61

<signature type> ::=



ERLÄUTERUNG

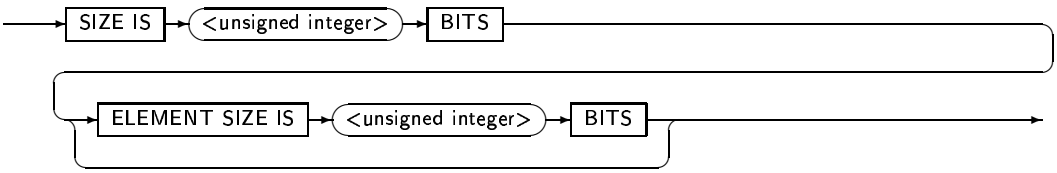
Diese Option legt den genauen Typ der Signatur und spezifische Optionen dazu fest. Auf konkrete Signaturverfahren wird hier nicht weiter eingegangen. Standardmäßig wird für Signaturen auf kollektionswertigen Attributen der in Abschnitt 4.1.3.2 angedeutete Algorithmus herangezogen.

TEILKLAUSELN

- ▷ SYNTAXREGEL 62 *<default signature options>*
  Optionen für den Standardsignaturtyp.
- ▷ *<signature type ... and options>*
  Möglichkeit zur Integration und Spezifikation weiterer hier nicht diskutierter Signaturverfahren.

SYNTAXREGEL 62

<default signature options> ::=



ERLÄUTERUNG

Mit dieser Regel werden beispielhaft Optionen für den in Abschnitt 4.1.3.2 angedeuteten Standardsignaturtyp festgelegt. Dabei geht es nicht um die Vor- oder Nachteile dieses Typs, sondern darum, beispielhaft die Möglichkeiten aufzuzeigen. So könnte mit `SIZE IS` die Länge der Signatur in Bits fest gelegt werden (standardmäßig zum Beispiel die Länge eines Integers). Wird die Signatur für eine ganze Kollektion definiert, könnte mit `ELEMENT SIZE IS` deklariert werden, wieviele Bits dieser Signatur durch ein einzelnes Element gesetzt werden sollen (standardmäßig beispielsweise ein Viertel der Signaturlänge, siehe Abschnitt 4.1.3.1).

TEILKLAUSELN

- ▷ *<unsigned integer>*
  Größenangabe als nicht negative ganze Zahl nach Norm-SQL.

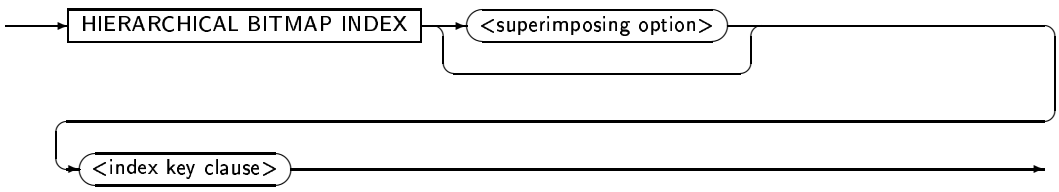
BEISPIEL

Basierend auf dem kollektionswertigen Attribut `ZuverarbeitendeBestandteile` der zusammengesetzten Werkstücke wird hier eine Signatur mit der Länge von 16 Bits erstellt. Jedes Element der Kollektion setzt dabei genau drei Bits der Signatur:

```
SIGNATURE
    SIZE IS 16 BITS
    ELEMENT SIZE IS 3 BITS
    INDEX KEY IS ( Werkstückart.ZuVerarbeitendeBestandteile )
```

SYNTAXREGEL 63

<collection hbi> ::=



ERLÄUTERUNG

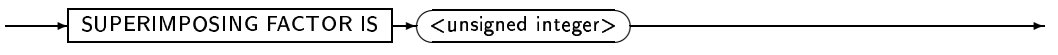
Mit diesem Konstruktor wird ein hierarchischer Bitmap-Index als komprimierte Bitmap-Repräsentation einer Menge erzeugt (siehe Abschnitt 4.1.3.1). In <superimposing option> wird festgelegt, aus wieviel Signaturen sich eine neue Signatur innerhalb der hierarchischen Darstellung zusammensetzen soll. Die zu repräsentierende Menge wird mit <index key clause> festgelegt.

TEILKLAUSELN

- ▷ SYNTAXREGEL 64 *<superimposing option>*
  Kombinationsgrad in der hierarchischen Darstellung.
- ▷ SYNTAXREGEL 36 *<index key clause>*
  Angabe der Schlüssel für die Signatur.

SYNTAXREGEL 64

<superimposing option> ::=



ERLÄUTERUNG

Die Anzahl der Signaturen, die innerhalb der hierarchischen Darstellung (siehe Abschnitt 4.1.3.1) zu einer Signatur der nächst höheren Ebene kombiniert werden sollen, wird mit dieser Syntaxregel festgelegt.

TEILKLAUSELN

- ▷ *<unsigned integer>*
  Größenangabe als nicht negative ganze Zahl nach Norm-SQL.

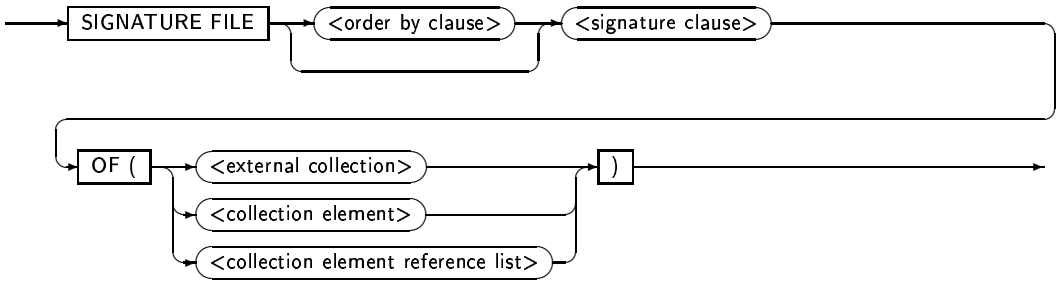
BEISPIEL

Mit folgendem Fragment einer Strukturdefinition wird ein hierarchischer Bitmap-Index angelegt, bei dem von Hierachiestufe zu Hierachiestufe pro Knoten 24 Signaturen miteinander verschmolzen werden:

```
HIERARCHICAL BITMAP INDEX
    SUPERIMPOSING FACTOR IS 24
    INDEX KEY IS (...)
```

SYNTAXREGEL 65

<collection signature file> ::=



ERLÄUTERUNG

Dieser Kollektionskonstruktor erzeugt eine Liste von Indexeinträgen mit Referenzen auf Elemente einer Kollektion. Indexschlüssel ist eine Signatur. Von dieser unabhängig kann eine Ordnung auf dem Index erzeugt werden, vorausgesetzt, mit dem Konstruktor wird nicht die Primärspeicherstruktur einer verketteten Liste oder eines Feldes definiert. Der zu nutzende Signaturtyp wird per <signature clause> ausgewählt.

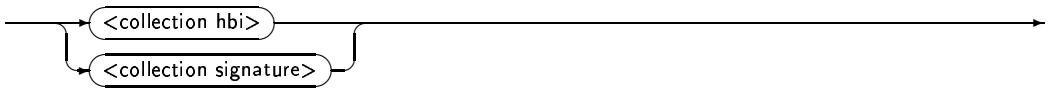
<external collection>, <collection element> und <collection element reference list> haben wieder die gleiche Bedeutung wie in SYNTAXREGEL 52 <collection btree>.

TEILKLAUSELN

- ▷ SYNTAXREGEL 35 *<order by clause>*
  Sortierreihenfolge für die Kollektionselemente.
- ▷ SYNTAXREGEL 66 *<signature clause>*
  Festlegung des Signaturtyps.
- ▷ SYNTAXREGEL 45 *<external collection>*
  Verweis auf eine weitere Kollektionsstruktur.
- ▷ SYNTAXREGEL 46 *<collection element>*
  Ein- oder ausgelagertes Kollektionselement.
- ▷ SYNTAXREGEL 53 *<collection element reference list>*
  Verweise auf die indexierten Elemente bei der Definition einer Zugriffsstruktur.

SYNTAXREGEL 66

<signature clause> ::=



ERLÄUTERUNG

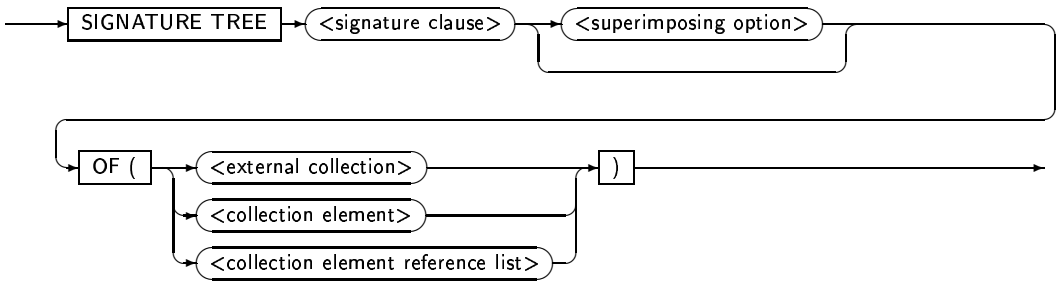
Der zu nutzende Signaturtyp wird mit dieser Regel ausgewählt. Zur Verfügung stehen die hierarchische Bitmap-Kodierung (hierarchischer Bitmap-Index) und die verschiedenen Signaturverfahren.

TEILKLAUSELN

- ▷ SYNTAXREGEL 63 *<collection hbi>*
  Nutzung eines hierarchischen Bitmap-Indexes.
- ▷ SYNTAXREGEL 60 *<collection signature>*
  Nutzung eines Signaturverfahrens.

SYNTAXREGEL 67

<collection signature tree> ::=



ERLÄUTERUNG

Dieser Konstrukor legt eine Struktur vom Typ Signaturbaum an, wie sein Name besagt. Wie auch beim Signatur-File wird mit <signature clause> der dem Signaturbaum zugrunde liegender Signaturtyp ausgewählt. <superimposing option> legt die Anzahl der Signaturen fest, die in einem Knoten des Baums per bitweiser ODER-Verknüpfung zusammengefaßt werden.

<external collection>, <collection element> und <collection element reference list> haben wieder die gleiche Bedeutung wie in SYNTAXREGEL 52 <collection btree>.

TEILKLAUSELN

- ▷ SYNTAXREGEL 66 *<signature clause>*
  Festlegung des Signaturtyps.
- ▷ SYNTAXREGEL 64 *<superimposing option>*
  Kombinationsgrad im Baum.
- ▷ SYNTAXREGEL 45 *<external collection>*
  Verweis auf eine weitere Kollektionsstruktur.
- ▷ SYNTAXREGEL 46 *<collection element>*
  Ein- oder ausgelagertes Kollektionselement.
- ▷ SYNTAXREGEL 53 *<collection element reference list>*
  Verweise auf die indexierten Elemente bei der Definition einer Zugriffsstruktur.

BEISPIEL

Folgendes Fragment erzeugt auf den Elementen eines kollektionswertigen Attributs einen Signaturbaum:

```
SIGNATURE TREE
    SIGNATURE
    SIZE IS 24 BITS
    INDEX KEY IS ( ELEMENT )
    SUPERIMPOSING FACTOR IS 8
    OF (...)
```

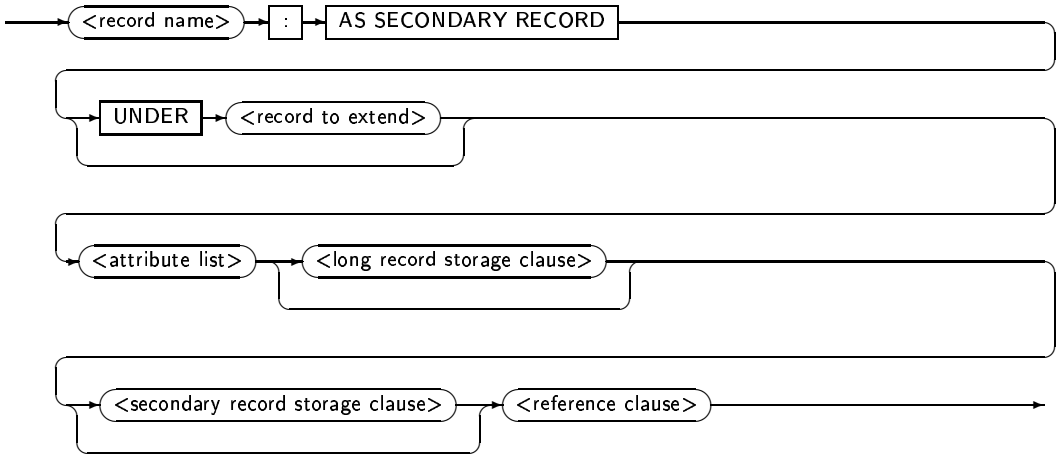
Zur Erstellung der 24 Bit langen Signatur vom vorgegebenen Standardsignaturtyp wird jeweils ein vollständiges Kollektionselement herangezogen. Innerhalb des Baums werden jeweils 8 Signaturen zu einer neuen Signatur verknüpft.

5.2.7 Wahlfreie Zuordnung von Attributen zu Sekundärsätzen

In den bisherigen Überlegungen bestand ein direkter Zusammenhang zwischen dem logischen Aufbau eines Objekts und seiner physischen Satzstruktur. Die wahlfreie Attributauslagerung ermöglicht es, Sekundärsatzstrukturen völlig unabhängig von der Objektstruktur zu schaffen.

Syntaxregel 68

<named secondary record clause> ::=



Erläuterung

Diese Produktion dient der Definition von Sekundärsätzen zur wahlfreien Attributauslagerung.

Sie beginnt mit der Angabe eines Satzidentifikators. Allerdings handelt es sich bei <record name> nicht um einen Pfadausdruck zu einem logischen Attribut, sondern um die Deklaration eines Namens für den neu zu erstellenden Sekundärsatz. Nach einem Doppelpunkt leiten die Schlüsselwörter `AS SECONDARY RECORD` den eigentlichen Definitionsteil ein. Die erste optionale Angabe klärt mit <record to extend> die Frage, unter welchen bereits vorhandenen Satz der neue Sekundärsatz angehängt werden soll. Wie bei der Auslagerung von Feldern strukturierter Attribute, werden die dem Sekundärsatz zugewiesenen Attribute in einer Liste aufgezählt.

Die anderen Angaben für Sekundärsätze, wie die Festlegungen für Überlaufsätze, zum Speicherort und die Deklaration der Referenzierung, entsprechen den Optionen, die auch bei Sekundärsätzen zu strukturierten Attributen möglich sind.

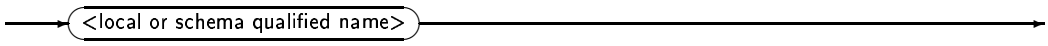
Teilklauseln

- ▷ Syntaxregel 69 *<record name>*
  Vergabe eines Satznamens.
- ▷ Syntaxregel 70 *<record to extend>*
  Bezeichnung des Satzes unter den der neue Sekundärsatz gehängt werden soll.

- ▷ SYNTAXREGEL 26 *<attribute list>*
  Liste der Attribute die dem Sekundärsatz zugewiesenen werden.
- ▷ SYNTAXREGEL 14 *<long record storage clause>*
  Festlegung des Speicherplatzes für Sätze, die zu groß für eine physische Seite sind.
- ▷ SYNTAXREGEL 30 *<secondary record storage clause>*
  Angaben zur Speicherung des neu definierten Sekundärsatzes.
- ▷ SYNTAXREGEL 27 *<reference clause>*
  Anbindung des neu definierten Sekundärsatzes per Referenzierung.

SYNTAXREGEL 69

<record name> ::=



ERLÄUTERUNG

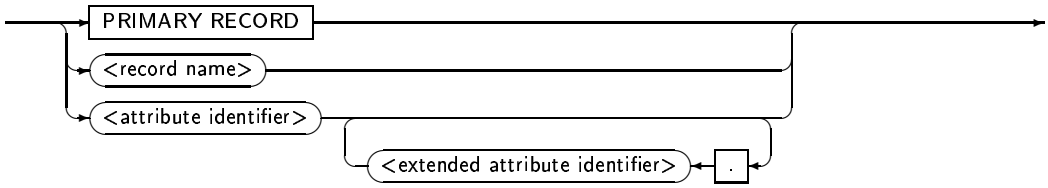
Diese Regel gestattet die Deklaration eines Namens für den neu zu erstellenden Sekundärsatz.

TEILKLAUSELN

- ▷ *<local or schema qualified name>*
  Bezeichnung laut Norm-SQL.

SYNTAXREGEL 70

<record to extend> ::=



ERLÄUTERUNG

Diese Regel dient der Bezeichnung des bereits vorhandenen Satzes, an den der neue Sekundärsatz angehängt werden soll. Es existieren hierfür drei Möglichkeiten: Variante 1 (Standardvariante) erstellt ihn als direktes Kind des Primärsatzes. Variante 2 erlaubt die Nennung eines bereits definierten benannten Sekundärsatzes. Variante 3 gestattet die Angabe eines Pfadausdruckes zur Adressierung eines Attributs und identifizert so den das Attribut enthaltenden Satz. Auf diese Weise ermöglicht diese Regel die Definition von Satzhierarchien.

TEILKLAUSELN

- ▷ SYNTAXREGEL 69 *<record name>*
  Bezeichnung eines Satzes.
- ▷ SYNTAXREGEL 13 *<attribute identifier>*
  Attributbezeichnung für den Pfadausdruck.
- ▷ SYNTAXREGEL 22 *<extended attribute identifier>*
  Erweiterte Attributbezeichnung für den Pfadausdruck.

ABBILDUNG 5.6: PRDL-Beispiel für attributbasierte Definition von Sekundärsätzen

BEISPIEL

Das folgende Beispiel illustriert die wahlfreie Zuordnung von Attributen zu Sekundärsätzen.

```
BEGIN PRDL
    sekSatz1: AS SECONDARY RECORD UNDER PRIMARY RECORD
            (Standort, Maschinenlogbuch)
    sekSatz2: AS SECONDARY RECORD UNDER sekSatz1
            (Maschinenlogbuch.Beschreibung_Wartungsmaßnahme)
    sekSatz3: AS SECONDARY RECORD UNDER PRIMARY RECORD
            (Standort.Werksgebäude)
END PRDL
```

Die so entstehende physische Satzstruktur zeigt ABBILDUNG 5.6.

5.3 PRDL-Beispiel

Die Vorstellung der Speicherbeschreibungssprache PRDL in diesem Kapitel soll durch ein etwas umfassenderes Beispiel abgerundet werden. Dazu wird auf das in Abschnitt 1.4 vorgestellte Szenario zurückgegriffen.

Zur Speicherung der Daten zu Fertigungsmaschinen und Werkstückarten sollen vorbereitend folgende Tablespaces und Cluster definiert sein.

- ▷ Für die Fertigungsmaschinendaten:
  - ◇ *Cluster* `FM_cl` im Tablespace `FM_ts` für die Fertigungsmaschinendaten mit dem Clusterschlüssel `(Werksgebäude VARCHAR(10), Etage INTEGER)`.
  - ◇ *Tablespace* `Log_ts` für die Logbuchdaten der Fertigungsmaschinen.
  - ◇ *Tablespace* `Long_ts` für die Beschreibungstexte in den Fertigungsmaschinen-Logbucheinträgen.
  - ◇ *Index-Tablespace* `BFIdx_B_ts` für den lokalen B*-Baum auf den Bearbeitungsfähigkeiten
  - ◇ *Cluster* `FM_BV_cl` im Tablespace `FM_BV_ts` für die Bearbeitungsvorgangsdaten mit dem Clusterschlüssel `(Id_Bearbeitungsvorgang VARCHAR(20))`.
  - ◇ *Index-Tablespace* `BFIdx_RD_ts` für den globalen RD-Baum auf den Bearbeitungsfähigkeiten

- ▷ Für die Werkstückartendaten:
  - ◇ *Tablespace* `WS_ts` für die Daten der Werkstückarten
  - ◇ *Cluster* `WS_BV_cl` im Tablespace `WS_BV_ts` für die Bearbeitungsvorgangsdaten der Werkstückarten mit dem Clusterschlüssel `(Id_Bearbeitungsvorgang VARCHAR(20))`.
  - ◇ *Index-Tablespace* `WSIdx_ts` für den globalen Suffix-Baum auf den Herstellungsabläufen

Die vollständigen PRDL-Spezifikationen zur Speicherung und Indexierung der Fertigungsmaschinen werden in ABBILDUNG 5.7 zusammengefaßt, die Angaben für die Werkstückarten in ABBILDUNG 5.8. Diese Abbildungen stellen auch die resultierenden Speicherstrukturen graphisch dar. Die einzelnen Spezifikationen werden im folgenden erläutert.

Speicherung und Indexierung der Fertigungsmaschinendaten

Bei den Fertigungsmaschinen bewirkt, wie ABBILDUNG 5.7 zeigt, die Angabe von

```
STORE IN CLUSTER FM_cl (Standort.Werksgebäude, Standort.Etage),
```

daß die Primärsätze im Tablespace `FM_ts` in Clustern vom Typ `FM_cl` gespeichert werden. Dabei werden Daten zu Maschinen, die ihren Standort im gleichen Werksgebäude und in der gleichen Etage haben, geclustert, so daß effizient auf alle bis auf die Abteilungszugehörigkeit gleich plazierten Maschinen schnell gemeinsam zugegriffen werden kann.

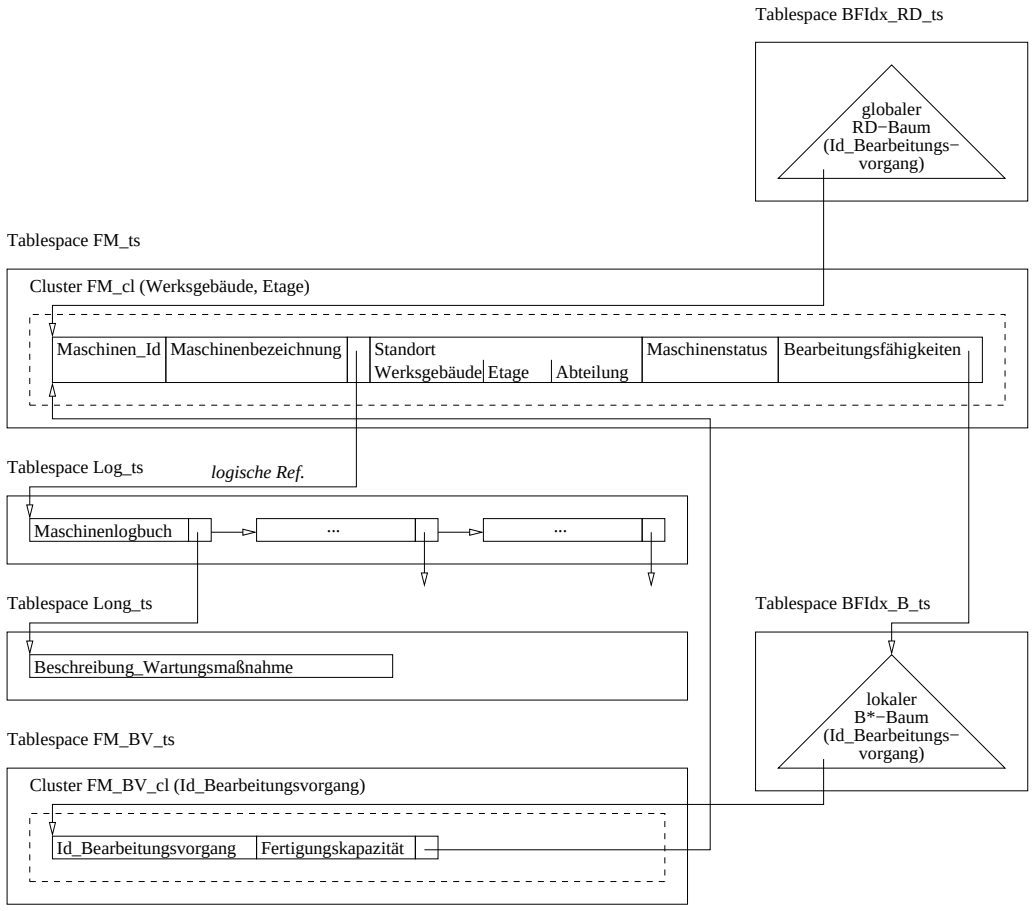
Das Maschinenlogbuch wird mit seinen Einträgen in Sekundärsätze im `Log_ts`-Tablespace ausgelagert:

```
Maschinenlogbuch :
    COLLECTION IS EXTERNAL REFERENCE IS LOGICAL USE SYSTEM GENERATED
                                                        KEY WITH PPP
    STORED AS ( LINKED LIST SINGLE LINKED OF (INLINE ELEMENT) STORE
                                              IN TABLESPACE Log_ts )
```

Die Logbucheinträge werden als einfach verkettete Liste abgelegt, so daß ein nahezu unbeschränktes Wachstum möglich ist. Der Listenkopf wird vom Primärsatz durch künstlich generierte logische Referenzen eingebunden, so daß auch umfangreiche Logbuchdaten Anfragen auf den Maschinendaten nicht behindern. Die logischen Referenzen besitzen jeweils zusätzlich einen physischen Positionshinweis zur Beschleunigung des Zugriffs.

Allerdings werden die potentiell sehr großen Beschreibungstexte zu den einzelen Logbucheinträgen noch einmal getrennt in einem Tablespace für lange Datensätze (`Long_ts`) ausgelagert:

```
Maschinenlogbuch.ELEMENT.Beschreibung_Wartungsmaßnahme :
    STRUCTURE IS SECONDARY RECORD STORE IN TABLESPACE Long_ts
```




```
CREATE TABLE Fertigungsmaschine AS Fertigungsmaschine_t
BEGIN PRDL
    STORE IN CLUSTER FM_cl (Standort.Werksgebäude, Standort.Etage)
    Maschinenlogbuch :
        COLLECTION IS EXTERNAL REFERENCE IS LOGICAL USE SYSTEM GENERATED KEY WITH PPP
        STORED AS ( LINKED LIST SINGLE LINKED OF (INLINE ELEMENT) STORE IN TABLESPACE Log_ts )
    Maschinenlogbuch.ELEMENT.Beschreibung_Wartungsmaßnahme :
        STRUCTURE IS SECONDARY RECORD STORE IN TABLESPACE Long_ts
    Bearbeitungsfähigkeiten :
        COLLECTION IS EXTERNAL STORED AS (
            B-TREE INDEX KEY IS (Id_Bearbeitungsvorgang) OF (EXTERNAL ELEMENT)
            STORE IN TABLESPACE BFIdx_B_ts )
    Bearbeitungsfähigkeiten.ELEMENT :
        STRUCTURE IS SECONDARY RECORD
        STORE IN CLUSTER FM_BV_ts ( Id_Bearbeitungsvorgang )
        BACKWARD REFERENCE IS PHYSICAL TO (ROOT)
END PRDL;

CREATE INDEX Fertigungsmaschine_Bearbeitungsfähigkeiten
    ON Fertigungsmaschine
    AS EXTERNAL STORED AS (
        RD-TREE INDEX KEY IS (Bearbeitungsfähigkeiten) ORDER BY (Id_Bearbeitungsvorgang)
        OF (ELEMENT) STORE IN TABLESPACE BFIdx_RD_ts );
```

ABBILDUNG 5.7: Speicherung von Fertigungsmaschinendaten

Das Mengenattribut `Bearbeitungsfähigkeiten` wird physisch in Form eines lokalen B*-Baums repräsentiert:

```
Bearbeitungsfähigkeiten :
    COLLECTION IS EXTERNAL STORED AS (
        B-TREE INDEX KEY IS (Id_Bearbeitungsvorgang) OF (EXTERNAL
                                                          ELEMENT)
        STORE IN TABLESPACE BFIdx_B_ts )
```

Dieser ist in den Index-Tablespace `BFIdx_B_ts` ausgelagert und basiert auf dem Attribut `Id_Bearbeitungsvorgang` der Mengenelemente. Aus den Blättern des B*-Baums heraus werden die Elemente referenziert. Da nach `EXTERNAL ELEMENT` keine Angaben in einer optionalen <reference clause> gemacht werden, handelt es sich standardmäßig um physische Referenzen.

Die Mengenelemente selbst werden in einem weiteren Tablespace mit Clustertyp `FM_BV_ts` als Sekundärsätze abgelegt:

```
Bearbeitungsfähigkeiten.ELEMENT :
    STRUCTURE IS SECONDARY RECORD
    STORE IN CLUSTER FM_BV_ts ( Id_Bearbeitungsvorgang )
    BACKWARD REFERENCE IS PHYSICAL TO (ROOT)
```

Die Clusterung erfolgt nach `Id_Bearbeitungsvorgang`. Und die Sekundärsätze besitzen jeweils eine physische Referenz auf den jeweiligen Maschinen-Primärsatz, um von einer Bearbeitungsfähigkeit aus schnell auf den zugehörigen Maschinendatensatz zugreifen zu können.

Das schnelle Auffinden einer Menge von unterstützten Bearbeitungsvorgängen in den Bearbeitungsfähigkeiten einer Maschine ermöglicht ein zusätzlich angelegter globaler RD-Baum:

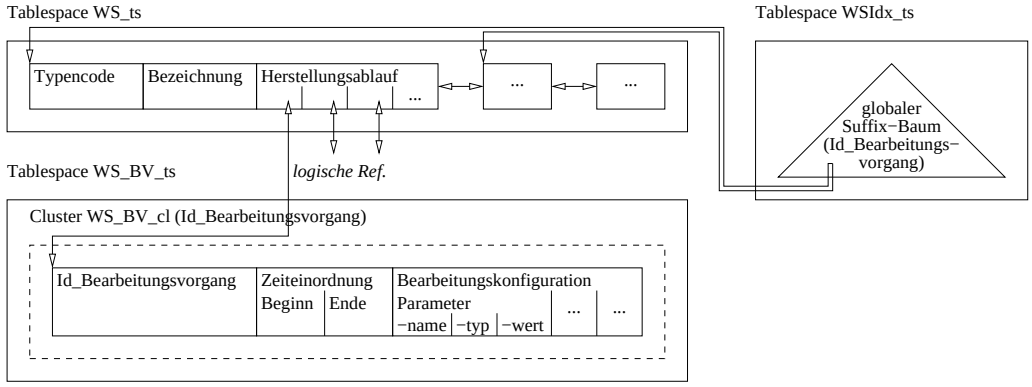
```
CREATE INDEX Fertigungsmaschine_Bearbeitungsfähigkeiten
    ON Fertigungsmaschine
    AS EXTERNAL STORED AS (
        RD-TREE INDEX KEY IS (Bearbeitungsfähigkeiten) ORDER BY
                                                (Id_Bearbeitungsvorgang)
        OF (ELEMENT) STORE IN TABLESPACE BFIdx_RD_ts )
```

Die Indexierung der Bearbeitungsfähigkeiten basiert bei diesem RD-Baum auf ihrem Elementattribut `Id_Bearbeitungsvorgang`, über das hier die (Un-)Gleichheit von Mengenelementen definiert wird. Der Index wird im Index-Tablespace `BFIdx_RD_ts` angelegt.

Speicherung und Indexierung der Werkstückartendaten

Abbildung 5.8 zeigt die Speicherstruktur für die Werkstücksarten. Wie zu erkennen ist, bewirkt die Definition

```
STORE IN TABLESPACE WS_ts FOR LONG RECORDS USE CHAINING,
```




```
CREATE TABLE Werkstückart AS Werkstückart_t
BEGIN PRDL
    STORE IN TABLESPACE WS_ts FOR LONG RECORDS USE CHAINING
    Herstellungsablauf :
        COLLECTION IS ARRAY OF (
            EXTERNAL ELEMENT
            REFERENCE IS LOGICAL USE SYSTEM GENERATED KEY WITH PPP )
    Herstellungsablauf.ELEMENT :
        STRUCTURE IS SECONDARY RECORD
        STORE IN CLUSTER WS_BV_cl ( Id_Bearbeitungsvorgang )
        BACKWARD REFERENCE IS PHYSICAL TO (ROOT)
END PRDL;

CREATE INDEX Werkstückart_Herstellungsablauf
    ON Werkstückart
    AS EXTERNAL STORED AS (
        SUFFIX TREE INDEX KEY IS (Herstellungsablauf) ORDER BY (Id_Bearbeitungsvorgang)
        OF (ELEMENT, ROOT) STORE IN TABLESPACE WSIdx_ts );
```

ABBILDUNG 5.8: Speicherung von Werkstückartendaten

daß der Primärsatz im Tablespace `WS_ts` abgelegt wird. Sollte der Primärsatz durch das eingeschachtelte Kollektionsattribut `Herstellungsablauf` größer als eine Speicherseite werden, so werden weitere Sekundärsätze in Form einer Satzkette angehängt. Auf diese Weise ist dem Größenwachstum der Herstellungsabläufe quasi keine Grenze gesetzt.

Die Repräsentaion des Listenattributs `Herstellungsablauf` wird durch

```
Herstellungsablauf :
    COLLECTION IS ARRAY OF (
        EXTERNAL ELEMENT
        REFERENCE IS LOGICAL USE SYSTEM GENERATED KEY WITH PPP )
```

als nicht größenbeschränktes Zeigerfeld festgelegt. Nach `EXTERNAL ELEMENT` wird die Art der zu nutzenden Referenzen angegeben: logische Referenzen auf die Elemente des jeweiligen Herstellungsablaufs mit physischen Positionshinweisen.

Diese Elemente, die Fertigungsschritte, werden mit folgender Spezifikation in Sekundärsätze ausgelagert:

```
Herstellungsablauf.ELEMENT :
    STRUCTURE IS SECONDARY RECORD
```

```
    STORE IN CLUSTER WS_BV_cl ( Id_Bearbeitungsvorgang )
    BACKWARD REFERENCE IS PHYSICAL TO (ROOT)
```

Diese Sekundärsätze werden im Clustertyp `WS_BV_cl` im Tablespace `WS_BV_ts` abgelegt. Dabei werden alle Fertigungsschritte, die den gleichen Bearbeitungsvorgang erfordern, geclustert. Dadurch kann beispielsweise der Zugriff auf alle Fertigungsschritte von allen Werkstückarten, die den gleichen Bearbeitungsvorgang erfordern, unterstützt werden. Jeder Sekundärsatz enthält zusätzlich zu den Daten des Fertigungsschritts noch eine physische Referenz auf den Primärsatz (`TO (ROOT)`), so daß von einem Fertigungsschritt schnell auf die zugehörige Werkstückart zugegriffen werden kann.

Zuletzt werden die Herstellungsabläufe der Werkstückarten noch mit einem Suffix-Baum indexiert:

```
CREATE INDEX Werkstückart_Herstellungsablauf
    ON Werkstückart
    AS EXTERNAL STORED AS (
        SUFFIX TREE INDEX KEY IS (Herstellungsablauf) ORDER BY
                                                (Id_Bearbeitungsvorgang)
        OF (ELEMENT, ROOT) STORE IN TABLESPACE WSIdx_ts )
```

Dieser erlaubt es, Werkstückarten zu finden, die einen gewissen Teilablauf zur Herstellung benötigen. Wieder dient das Attribut `Id_Bearbeitungsvorgang` zur Festlegung der Ordnung auf beziehungsweise der Gleichheit von Fertigungsschritten, und wieder wird nach dem doch recht umfangreichen Beispiel eine Aufmunterung des Lesers durch den weiter hinten nützlichen Begriff *park* versucht. Der Suffix-Baum wird im Index-Tablespace `WSIdx_ts` aufgebaut. Jeder Indexeintrag verweist, da `(ELEMENT, ROOT)` angegeben wurde, nicht nur auf den jeweiligen physischen Satz mit dem Teilherstellungsablauf, sondern stets auch auf den Primärsatz. Bei Werkstückarten mit kleinen Herstellungsabläufen, die in einen physischen Satz passen, ist das irrelevant. Aber wenn verkettete, seitenübergreifende Sätze benutzt werden müssen, dann kann dadurch sowohl effizient auf den jeweiligen Teilherstellungsablauf als auch auf die Werkstückart als Ganzes zugegriffen werden.

Fazit

Im Beispiel wurde die Verwendung von PRDL an einem kleinen, aber nicht trivialen Szenario verdeutlicht. Dabei wurden die Definitionen und die daraus resultierende Speicher- und Indexierungsstruktur gegenübergestellt. Gleichzeitig konnten wichtige PRDL-Konstukte und interessante physische Strukturen im Anwendungskontext vorgeführt werden.

5.4 Zusammenfassung des Kapitels

In diesem Kapitel wurde die Speicherbeschreibungssprache PRDL für objektrelationale DBMS vorgestellt. Mit ihr ist es möglich, für mit der SQL-DDL erstellte logische Datenstrukturen Spezifikationen zur physischen Speicherung und Indexierung anzugeben. Dabei ist PRDL als eigenständige Sprache von SQL sauber getrennt.

Durch die Trennung der Definitionssprachen und der Phasen für den logischen und den physischen Datenbankentwurf eignet sich PRDL, die Datenunabhängigkeit in objektrelationalen Datenbanksystemen zu gewährleisten. So sollen insbesondere objektorientierte

Datenbankanwendungen und logische Datenstrukturen von Änderungen der physischen Repräsentation entkoppelt werden und im Endeffekt ein Weg zur Optimierung der Speicher- und Indexstrukturen geöffnet werden, wie ihn das nachfolgende Kapitel umreißt.

Den Anfang des Kapitels bildete die Vorstellung der Entwurfskonzepte von PRDL. Danach wurde eine Einordnung in verwandte Ansätze vorgenommen, die vollständige Syntax von PRDL vorgestellt und an Beispielen erläutert.

Dabei wurde deutlich, daß PRDL eine recht komplexe Spezifikationssprache ist und die in Kapitel 3 und Kapitel 4 erarbeiteten umfassenden Möglichkeiten und Freiheitsgrade bietet, um auf die Speicherung und Indexierung komplexer Objekte in objektrelationalen Datenbanken Einfluß zu nehmen.

Kapitel 6

Optimierte Verarbeitung komplexer Objekte

Dieses Kapitel soll Fragen der Verarbeitung komplexer Objekte in Zusammenhang mit der in den letzten Kapiteln besprochenen Speicherung und Indexierung beleuchten. Dabei soll nicht der Anspruch auf eine umfassende und erschöpfende Behandlung dieses Themas erhoben und kein weiteres großes Themengebiet in den Kern dieser Arbeit aufgenommen werden. Es sollen vielmehr einige grundsätzliche Überlegungen zur Anfragebearbeitung und Optimierung als Ergänzung zu den Fragen der Speicherung und Indexierung komplexer Objekte dargelegt, weiterführende Ideen skizziert, offene Fragen aufgeworfen und Perspektiven für weitere Arbeiten aufgezeigt werden.

In Abschnitt 6.1 wird untersucht, inwiefern die Verarbeitung komplexer Objekte mit den in heutigen relationalen DBMS zur Verfügung stehenden Mitteln möglich ist und welche Erweiterungen gegebenenfalls notwendig sind.

Die Frage nach der Verarbeitung komplexer Objekte führt unweigerlich zum Thema Optimierung. Dementsprechend interessieren sowohl Alternativen der Anfrageausführung und ihre kostenmäßige Bewertung als auch alternative Speicherformen und ihre Auswirkungen auf die Anfrageausführung. Zu diesem Zweck setzt sich der darauffolgende Abschnitt 6.2 mit den Ausführungskosten von Operationen auf komplexen Objekten auseinander, deren Kenntnis einerseits Optimierungen bei der Anfrageausführung bei gegebenen Speicherstrukturen und andererseits die Optimierung (Auswahl/Festlegung) der Speicherstrukturen zur Unterstützung einer gegebenen Menge auszuführender Operationen (Workload) erlauben soll.

Abschnitt 6.3 stellt kurz eine im Rahmen dieser Arbeit entwickelte Simulationsumgebung zur Speicherung und Verarbeitung komplexer Objekte vor [Kla03, Lac04, Sch04]. Mit Hilfe dieser Umgebung konnten einige der Speicher- und Indexierungskonzepte umgesetzt und validiert werden. Gleichfalls erlaubt sie eine Überprüfung der Überlegungen zu den Ausführungskosten von Operationen auf komplexen Objekten.

Die Wirksamkeit dieser Optimierungen und das vorhandene Optimierungspotential sollen an Beispielen verdeutlicht werden. Dazu werden in Abschnitt 6.4 einige Optimierungsheuristiken präsentiert und anhand der Simulationsumgebung und der resultierenden Ausführungskosten bewertet.

In Abschnitt 6.5 werden dann abschließend Konzepte zum eventuell gar nicht so ‚fernen' Fernziel einer automatischen Optimierung der Speicherstrukturen durch Datenbank-Management-Systeme selbst, ohne erforderliche Nutzereingriffe, vorgestellt. Akuelle Schlag-

worte wie „Autonomous Computing“ und „Self-Tuning Databases“ symbolisieren derzeitige lebhafte Aktivitäten in diesem Forschungs- und Entwicklungsbereich. Aktivitäten hierzu laufen seit 2005 auch an der Universität Jena gemeinsam mit Industriepartnern (IBM).

6.1 Operationen auf komplexen Objekten und ihre Realisierung mit Planoperatoren

Dieser Abschnitt soll die möglichen Operationen auf komplexen Objekten und ihre Realisierung mittels relationaler Planoperatoren betrachten. Dazu werden die in ORDBMS vorhandenen Modellierungsmittel für Objekte und die damit verbundenen Operationen nacheinander untersucht und jeweils die zur Realisierung nötigen Planoperatoren sowie alternative Ausführungspläne ermittelt.

Ziel dieses Abschnitts ist es zu zeigen, daß alle erweiterten Operationen in ORDBMS weitgehend mit den bekannten relationalen Planoperatoren effizient umsetzbar und nur wenige Ergänzungen beziehungsweise Erweiterungen notwendig sind. Auch andere Arbeiten in der Literatur, wie zum Beispiel [HR01, Mit88, Keß95, JK84], kommen zu ähnlichen Ergebnissen. Auch wenn dort teils andere Operatoren, Abbildungen und Anfragesprachen verwendet werden, so treten immer wieder sehr ähnliche Basiskonzepte und Techniken auf, die mit den hier vorgestellten vergleichbar sind.

Die Anfragebearbeitungstechniken zur Unterstützung objektrelationaler Verarbeitungskonzepte lassen sich prinzipiell folgenden Grundrichtungen zuordnen:

1. *Nutzung vorhandener Anfrageausführungstechniken*
   Vorhandene Mittel, wie Anfragebearbeitungstechniken und Operatoren, werden dabei in der aus relationalen DBMS bekannten Weise für neue Datenstrukturen genutzt.

2. *Erweiterung um neue Anfrageausführungstechniken*
   Die erweiterten Techniken zur Anfrageausführung bilden zwei Unterkategorien:

   2.1 *Modifizierter Einsatz vorhandener Techniken und Konstrukte*
   Durch den modifizierten Einsatz vorhandener Techniken und Konstrukte können eine ganze Anzahl neuer Konzepte zur Verarbeitung komplexer Objekte unterstützt werden. Im Vorgriff auf die nachfolgenden Abschnitte lassen sich folgende Beispiele anführen:
   - Die Indexierungskonzepte lassen sich beispielsweiese durch den Einsatz von B*-Bäumen für verschachtelte Indexe und Pfadindexe an die Erfordernisse verschachtelter Objektstrukturen anpassen (siehe Abschnitt 4.1.2.3).
   - Das Speichersystem kann durch die Nutzung heterogener Seitentypen zur Aufnahme von physischen Sätzen unterschiedlicher Typen den Anforderungen der Speicherung und Clusterung komplexer Objekte entsprechen.
   - Und die Anfrageübersetzung, -optimierung und -ausführung kann durch die Nutzung und ‚Um-Nutzung‘ von Table-Scan-, Index-Scan- und Fetch-Operatoren für den Zugriff auf geclusterte oder ausgelagerte Subobjekte neue objektrelationale Anfragetechniken unterstützen.

2.2 *Integration erweiterter Techniken und Konstrukte*

Gleichzeitig kann die umfassende Unterstützung der Verarbeitung komplexer Objekte auch eine Reihe neu in DBMS zu integrierender Erweiterungen erfordern, so zum Beispiel:

- Notwendig ist etwa eine Einbindung neuer Indexstrukturen, wie Signatur- und Bitmap-Indexe, um Anfragen an komplexe Objekte mit mengenwertigen Attributen wirklich effizient bearbeiten zu können (vergleiche Kapitel 4).
- Auch erfordert die mit der Objektrelationalität verbundene Einführung von Objektstruktur-Polymorphie und Anfrageausführungs-Polymorphie die Schaffung spezieller Planoperatoren, wie zum Beispiel die des Auswahloperators Choice.

6.1.1 Operationen auf Strukturen

Beim Zugriff auf einzelne oder mehrere Attribute von verschachtelten Strukturen im Rahmen von Anfragen kann zunächst einmal zwischen drei grundlegenden Zugriffsmustern unterschieden werden:

▷ *Sequentielles Zugriffsmuster*

Beim sequentiellen Zugriffsmuster wird auf alle Objekte einer Tabelle nacheinander zugegriffen. Alle Objekte können dabei durch ein Suchprädikat überprüft und gegebenenfalls weiterverarbeitet werden. Zur Beschleunigung der Suche wird auf die physischen Sätze der Objekte möglichst in der Reihenfolge ihrer physischen Speicherung zugegriffen. Damit kann auf I/O-Ebene mit sequentiellem Zugriff gearbeitet werden, der im Vergleich zu wahlfreien I/O-Operationen um mindestens eine Größenordnung schneller ist.

▷ *Wahlfreies Zugriffsmuster*

Beim wahlfreien Zugriffsmuster wird gezielt auf einzelne Objekte (eines oder wenige) einer Tabelle über ein Suchprädikat zugegriffen. Da hier nur auf einen sehr kleinen Teil der Objekte zugegriffen wird, das heißt, die Selektivität des Suchprädikates sehr hoch ist (in der Regel ab weniger als etwa 5 % Treffer bei der Suche, genauer entscheidet hierüber der kostenbasierte Optimierer), spart der wahlfreie Direktzugriff im Vergleich zum sequentiellen Zugriff erhebliche I/O-Kosten [FS75, SAC+79, JK84]. Voraussetzung für diese Art des Zugriffs ist jedoch das Vorhandensein eines geeigneten Indexes, der die Auswertung des Suchprädikats unterstützt und somit das direkte Auffinden der physischen Sätze der gesuchten Objekte ermöglicht.

▷ *Sortiertes Zugriffsmuster*

Beim sortierten Zugriffsmuster wird auf die Objekte in einer durch Attributwerte bestimmten Reihenfolge zugegriffen. Dieses Muster kann auf die ersten beiden zurückgeführt werden: Liegen die Objekte auch physisch bereits in der richtigen Reihenfolge vor, was nur bei indexorganisierter Speicherung der Fall ist, so ergeben sich sequentielle I/O-Operationen. Liegt keine solche vorsortierte Speicherreihenfolge vor, dann muß entweder per geeignetem Index in der geforderten Reihenfolge und somit im Resultat wahlfrei zugegriffen werden, oder die Objekte müssen nach dem sequentiellen Lesen, das die Objekte unsortiert liefert, in einem zweiten Schritt durch einen von relationalen Systemen her bekannten Sort-Operator nachsortiert werden.

Als nächstes muß beim Attributzugriff zwischen den im physischen Primärsatz eingelagerten Attributen und den in Sekundärsätze ausgelagerten Attributen unterschieden werden:

1. Zugriff auf eingelagerte Attribute

Bei Attributen, die im Primärsatz eingelagert gespeichert sind, unterscheidet sich der Zugriff sowohl nach dem sequentiellen als auch nach dem wahlfreien Muster nicht von den bekannten Zugriffsoperationen in relationalen DBMS. Dementsprechend können hierfür die bekannten Planoperatoren eingesetzt werden:

- ▷ Table Scan für den sequentiellen Zugriff sowie
- ▷ Index Scan mit jeweils nachfolgendem Fetch für den wahlfreien Zugriff

Fetch bezeichnet dabei den Direktzugriff auf ein Tabellentupel beziehungsweise einen physischen Satz über die Tupel-ID, die vom Index Scan geliefert wird.

Gemeinsam ist beiden Zugriffsmustern, daß alle Attribute eines Objekts in einem einzigen physischen Primärsatz gespeichert sind und somit pro Objekt nur eine einzige I/O-Operation anfällt.

2. Zugriff auf ausgelagerte Attribute

Bei ausgelagerten Attributen ist dagegen der Zugriff auf einen oder mehrere Sekundärsätze notwendig. Je nach der Art, wie die Sekundärsätze mit den Primärsätzen per Referenz verbunden sind, kann der Zugriff unterschiedlich erfolgen:

2.1. Direktzugriff auf Sekundärsätze

Wird auf ausgelagerte Attribute wertebasiert zugegriffen und werden weder Attribute des Primärsatzes noch Attribute von übergeordneten Sekundärsätzen benötigt, so kann direkt auf die entsprechenden Sekundärsätze zugegriffen werden. Eine zeitaufwendige Navigation über den Primärsatz und über andere Sekundärsätze kann auf zweierlei Weisen vermieden werden:

- ▷ *Zugriff per Index Scan*

  Existiert ein geeigneter Index zum direkten Auffinden der gesuchten Sekundärsätze, so kann dieser unter Umgehung der Primärsätze genutzt werden.

- ▷ *Zugriff per Table Scan*

  Ist dagegen kein solcher Index verfügbar, so bleibt nur der Weg zu den Sekundärsätzen über das Durchsuchen aller in Frage kommenden Segmente mit dem oben beschriebenen sequentiellen Zugriff übrig.

Allerdings findet die Zugehörigkeit von Sekundärsätzen zu bestimmten Objekten bei dieser Zugriffsart keine Berücksichtigung, so daß sie nur dementsprechend eingeschränkt eingesetzt werden kann.

2.2. Zugriff von Primärsätzen auf Sekundärsätze über physische Referenzen

In Fällen,

- ▷ bei denen entweder Attribute aus Primär- und Sekundärsatz benötigt werden oder

- ▷ in denen zwar kein Attribut des Primärsatzes benötigt wird, es aber keine andere Einstiegsmöglichkeit für den benötigten Sekundärsatz gibt, wie zum Beispiel durch einen geeigneten Index, oder
- ▷ in denen die Zugehörigkeit von Sekundär- zu Primarsätzen und damit zu bestimmten Objekten von Bedeutung ist,

muß auf beide Sätze zugegriffen werden. Gleiches gilt auch für Hierarchien mit mehreren Sekundärsätzen. Hierfür stehen zwei Alternativen zur Auswahl:

- ▷ *Direktzugriff mittels Fetch-Operator*

  Zuerst wird hier der entsprechende Primärsatz gelesen, wie es bereits für den Zugriff auf eingelagerte Attribute beschrieben wurde. Danach wird mit den in ihm enthaltenen physischen Referenzen direkt auf den oder die benötigten Sekundärsätze zugegriffen. Dieses Vorgehen nutzt den Fetch-Operator und ist sowohl für den wahlfreien als auch den sequentiellen Objektzugriff geeignet. Es ist allerdings durch den wahlfreien Zugriff auf die Sekundärsätze mit hohen I/O-Kosten verbunden, sofern nicht eine geeignete objektbezogene satztypübergreifende Clusterung der physischen Sätze vorliegt.

- ▷ *Verbund mittels Join-Operator*

  Alternativ können auch die Primär- und die zugehörigen Sekundärsätze (oder Sekundärsätze miteinander) per Join-Operator auf der Basis des physischen Referenzwertes (Tupel-ID) verbunden werden. Genau genommen handelt es sich hier um eine ‚interne' Anwendung des Verbunds auf physischen Sätzen und nicht auf logischen Objekten. Der Verbund läßt sich beispielsweise mit einem Sort Merge Join durchführen, indem zuerst auf die Primärsätze zugegriffen wird, diese dann nach der enthaltenen physischen Sekundärsatzreferenz sortiert (Sort-Operator) und abschließend mit den Sekundärsätzen gemischt (Merge-Join-Operator) werden. Auch andere aus RDBMS bekannte Verbundvarianten sind einsetzbar.

2.3. Zugriff von Sekundärsätzen auf Primärsätze über physische Rückwärtsreferenzen

In diesem Fall sollen die benötigten Sekundärsätze nicht über den Primärsatz erreicht werden. In Abhängigkeit von der Existenz eines geeigneten Zugriffspfades kann auf den Sekundärsatz auf die gleichen Arten (Index Scan, Table Scan), die bereits für den Direktzugriff auf Sekundärsätze beschrieben wurden, zugegriffen werden.

Werden zusätzlich zu den Attributen des Sekundärsatzes auch noch Attribute aus dem Primärsatz benötigt, so muß zum Primärsatz ‚navigiert' werden. Dies ist, wie bereits beim Zugriff über physische Referenzen von Primär- auf Sekundärsätze besprochen wurde, auch in entgegengesetzter Richtung auf zwei Arten möglich:

- ▷ Direktzugriff mittels Fetch-Operator oder
- ▷ Verbund mittels Join-Operator.

2.4. Zugriff von Primärsätzen auf Sekundärsätze über logische Referenzen

Bei der logischen Referenzierung der Sekundärsätze auf der Basis eines Attributwertes gibt es nicht die Möglichkeit des Direktzugriffs, da ja die physische Adresse des Satzes, das heißt, seine Tupel-ID, nicht bekannt ist. Deshalb kann auf die Sekundärsätze auf nachfolgende Arten zugegriffen werden:

- ▷ *Zugriff per Join*
  Primärsatz und Sekundärsatz können auf der Basis des logischen Referenzwertes verbunden werden: Alle bekannten Verbundvarianten kommen dafür in Frage.

- ▷ *Zugriff per Index Scan*
  Falls logische Referenzen durch geeignete Zugriffspfade (Pfadindex oder traditionelle Indexe auf Sekundär- und/oder Primärsätzen) unterstützt werden, kann der Zugriff vom Primär- auf den Sekundärsatz oder umgekehrt beschleunigt werden. Die entstehenden I/O-Kosten können dabei insbesondere durch eine geeignete physische Clusterung der Sätze gesenkt werden.

Dabei können wieder Fälle berücksichtigt werden,

- ▷ in denen entweder Attribute sowohl aus dem Primär- als auch aus dem Sekundärsatz benötigt werden oder
- ▷ in denen es keinen anderen geeigneten Einstieg zu den Sekundärsätzen gibt oder aber
- ▷ in denen die Objektzugehörigkeit der Sekundärsätze eine Rolle spielt.

Die Überlegungen treffen natürlich nicht nur auf den Zugriff über logische Referenzen zwischen Primär- auf Sekundärsätzen, sondern auch auf einander logisch referenzierende Sekundärsätze zu.

6.1.2 Operationen bei Vererbungshierarchien

Bei Operationen auf Objekten, deren Typ auf einer Vererbungshierarchie basiert, muß man zunächst zwischen Spaltentypobjekten (Column Types), die wie jedes andere ‚normale' Attribut eines Tupels gespeichert und genutzt werden, und Tabellentypobjekten (Row Types), die als Tabellentupel gespeichert und genutzt werden, unterscheiden. Operationen auf Spaltentypobjekten werden in Abschnitt 6.1.2.1 und Operationen auf Tabellentypobjekten werden in Abschnitt 6.1.2.2 behandelt.

Wie die nachfolgend ausgeführte Unterscheidung ergibt, ist eine Realisierung von Operationen auf Vererbungshierarchien mittels folgender, zum Teil bereits für Strukturen einsetzbarer Operatoren möglich:

- ▷ Direktzugriff mittels Choice- und Fetch-Operator, beispielsweise eingebettet in einen Table Scan
- ▷ Verbund mittels Join-Operator
- ▷ Vereinigung mittels Union-Operator
- ▷ Zugriff per Index Scan, speziell Hierarchy Index Scan [MMNM03]

Im Falle des Direktzugriffs auf polymorphe Objekte in Vererbungshierarchien erlaubt der Choice-Operator (siehe Abschnitt 6.1.2.1) den Zugriff auf jeweils unterschiedliche Speicherformen für die verschiedenen Subtypen. Er ermöglicht die Auswahl unterschiedlicher Teilausführungspläne zur Laufzeit der SQL-Anweisung anhand eines dynamisch auszuwertenden Kriteriums. Dieses dynamische ‚Umschalten' ist mit den traditionellen, aus relationalen DBMS bekannten Ausführungsplantechniken nicht möglich und muß durch diesen neuen Planoperator realisiert werden. Allerdings wird die neue Flexibilität der Laufzeitentscheidungen mit einer Verschlechterung der Vorhersagbarkeit bei der Anfrageausführung (‚Überraschungen' zur Ausführungszeit) und mit einer Verkomplizierung der für die Optimierung wichtigen Anfragekostenschätzung erkauft.

6.1.2.1 Operationen auf polymorphen Spaltentypen

Spaltentypobjekte unterscheiden sich hinsichtlich der auf ihnen auszuführenden Operationen nicht wesentlich von Strukturtypen. Einzig die mögliche Polymorphie bringt zusätzliche Komplexität, die den eben erwähnten, in der relationalen Welt nicht üblichen und nicht benötigten Auswahloperator (Choice) erfordert. Dieser ermöglicht hier das dynamische ‚Umschalten' zwischen entsprechenden Teilausführungsplänen für die unterschiedlichen dynamisch vorliegenden Subtypen, die ja neue eingelagerte und ausgelagerte Attribute besitzen können. Jeder dieser Subtypen stellt aber prinzipiell einen verschachtelten Strukturtyp dar, so daß die bereits besprochenen Umsetzungsmöglichkeiten für Operationen auch hier anwendbar sind.

Als Alternative zum dynamischen ‚Umschalten' für unterschiedliche Subtypen pro zugegriffenem Objekt können auch von vornherein alle möglichen Subtypen im Ausführungsplan berücksichtigt werden, indem beispielsweise für alle potentiell auftretenden ausgelagerten Attribute die entsprechenden Verbundoperationen (Join) eingebunden werden, unabhängig davon, ob diese dann auch benötigt werden oder nicht.

Eine weitere Variante des Zugriffs kommt bei der horizontal partitionierten Speicherung zum Einsatz, bei der nicht nur die hinzukommenden Attribute von Subtypobjekten, sondern das gesamte Subtypobjekt in ein eigenes Segment ausgelagert werden. In diesem Fall müssen je nach Operation Sub- und Supertypobjekte per Union-Operator vereinigt werden.

6.1.2.2 Operationen auf Tabellenhierarchien

Für Objekte in Tabellenhierarchien gelten die gleichen operationalen Alternativen wie für Spaltentypobjekte. Dies gilt insbesondere auch für die Speicherung der Objekte aller Subtypen in einer einzigen Hierarchietabelle, bei der ein Objekt stets in einem einzigen physischen Satz abgelegt wird. Auch bei der vertikal partitionierten Speicherung, bei der Subtypattribute in eigene Segmente ausgelagert werden, können die bekannten Strategien des dynamischen ‚Umschaltens' (Choice) beim Direktzugriff beziehungsweise der Nutzung von Verbundoperationen (Join) eingesetzt werden. Ebenso können bei der horizontal partitionierten Speicherung mittels Vereinigung (Union) von Super- und Subtypobjekten Anfragen auf Supertypobjekte realisiert werden. Auch hier bieten spezielle Hierarchieindexe Unterstützung beim wahlfreien und wertebasierten Zugriff, so daß Index-Scan-Operatoren Verwendung finden können. Dies sind jedoch nicht die Index Scans, die aus relationalen Systemen bekannt sind, sondern spezielle Adaptionen (Hierarchy Index Scans) an die Erfordernisse und Möglichkeiten der Hierarchieindexe, die für den Einsatz in ORDBMS in Abschnitt 4.1.2.5 vorgeschlagen wurden.

6.1.2.3 Ausführung von Methoden

Methoden von Objekten besitzen eine Implementierung, für die es unterschiedliche Möglichkeiten gibt: So können sie in einer der üblichen Programmiersprachen, wie C, C++, Java und so weiter, implementiert sein oder in SQL mit oder ohne prozedurale Erweiterung (wie SQL/PSM und PL/SQL). Wenn die Objektdefinition in eine Vererbungshierarchie eingebettet ist, dann ergibt sich die Möglichkeit von Polymorphie: eine Methode kann für unterschiedliche Typen in der Hierarchie unterschiedliche Implementierungen besitzen. Daher müssen bei der Ausführung von Methoden innerhalb von SQL-Anweisungen folgende zwei Fälle unterschieden werden:

- ▷ *Ausführung nicht überschriebener Methoden*
  Dieser Fall ist insofern einfach, als daß der Aufruf der Methodenimplementierung fest in den Ausführungsplan der umgebenden SQL-Anweisung eingebaut werden kann. Für alle Objekte der von der Anfrage betroffenen Spalte beziehungsweise Tabelle wird ein und dieselbe Methodenimplementierung aufgerufen.

- ▷ *Ausführung überschriebener Methoden*
  In diesem Fall müssen in Abhängigkeit vom jeweils vorliegenden dynamischen Objekttyp unterschiedliche Methodenimplementierungen aufgerufen werden. Das heißt, in den Ausführungsplan ist erstens die Ermittlung des dynamischen Typs und zweitens das dynamische ‚Umschalten' zwischen entsprechenden Teilausführungsplänen per Choice-Operator einzubinden. Dieses ‚Umschalten' entspricht dem aus Programmiersprachen bekannten „Late Binding". Für die Objekte der von der Anfrage betroffenen Spalte beziehungsweise Tabelle können so jeweils unterschiedliche Methodenimplementierungen aufgerufen werden.

6.1.3 Operationen auf referenzierten Objekten

Operationen auf ausgelagerten Subobjekten und anderen per Referenz angebundenen Objekten setzen im allgemeinen das Erreichen und Auslesen dieser Objekte voraus. In dieser Beziehung sind also referenzierte Objekte genauso zu behandeln, wie es bereits bei ausgelagerten Strukturen besprochen wurde. Insofern kann eine Abbildung der Operationen in Abhängigkeit von der Art der Objektreferenzierung (logisch, physisch oder hybrid) und dem Zugriffsmuster (sequentiell, wahlfrei oder sortiert) auf folgende Ausführungsplanoperatoren erfolgen:

- ▷ *Index Scan*
  für die sequentielle, wahlfreie oder sortierte Verarbeitung logisch referenzierter Objekte

- ▷ *Fetch*
  für den Direktzugriff auf physisch referenzierte Objekte

- ▷ *Join mit Table oder Index Scan*
  bei der mengenwertigen Verarbeitung logisch und physisch referenzierter Objekte

6.1.4 Operationen auf Kollektionen

Bei der relationalen Umsetzung von Kollektionsoperationen muß in erster Linie nach der Speicherform unterschieden werden: die Kollektionselemente können entweder in die übergeordnete Struktur, das heißt, im selben physischen Satz eingelagert oder aber in andere physische Sätze ausgelagert gespeichert werden. Entsprechend unterschiedlich gestalten sich die Kollektionsoperationen.

1. Operationen auf eingelagerten Kollektionselementen

Auf Kollektionen, deren Elemente im selben physischen Satz wie ihre übergeordnete Attributstruktur gespeichert sind, wird implizit zugegriffen. Das heißt, mit dem Zugriff auf die übergeordnete Struktur wird auch auf die im gleichen physischen Satz enthaltenen Kollektionselemente zugegriffen, und die Kollektionsoperationen können unmittelbar auf diesem physischen Satz mit relationalen Standardtechniken ausgeführt werden.

2. Operationen auf ausgelagerten Kollektionselementen

Hier muß auf die Sekundärsätze mit den ausgelagerten Kollektionselementen zugegriffen werden. Dabei lassen sich die Operationen in solche, die für alle Kollektionsarten gleich sind, und in spezielle Operationen für die einzelnen Kollektionsarten unterscheiden.

Operationen, die für alle Kollektionen anwendbar sind und sich hinsichtlich ihrer relationalen Umsetzung nicht unterschieden, sind die Aggregatfunktionenen, wie die Bestimmung der Kardinalität, der Extrema (Minimum, Maximum) und so weiter, die Umwandlung von einem Kollektionstyp in einen anderen (Liste in Menge und so weiter) sowie ‚einfache' lesende und schreibende Elementzugriffe. Diese werden nachfolgend besprochen.

2.1. Kardinalität und Aggregationen

Eine sehr effiziente Möglichkeit zur Umsetzung einiger Aggregatfunktionen ist die redundante Speicherung (‚Materialisierung') der entsprechenden aggregierten Werte im Kollektionskopf, das heißt innerhalb der der Kollektion übergeordneten Struktur. So ist es zum Beispiel möglich, die Kardinalität und andere Aggregate vorzuberechnen und bei entsprechenden Abfragen diese Werte ohne Zugriff auf die Kollektionselemente zu nutzen. Der hierfür erforderliche Aufwand ist bei geschickter Wahl der redundant abzuspeichernden Werte sowohl hinsichtlich der zusätzlichen Zugriffe bei Änderungen als auch hinsichtlich des Speicheraufwands in bestimmten Fällen gering: zum Beispiel bei der Erhöhung oder Reduzierung von Kardinalität und Summe beim Einfügen oder Löschen und bei der Darstellung des Durchschnitts als Summe und Kardinaliät. Problematischer beziehungsweise ungeeignet ist diese Technik jedoch beispielsweise für die Extremwerte wegen des hohen Aktualisierungsaufwands. Auch muß die redundante Speicherung durch entsprechend häufige Zugriffe und jeweils eingesparten Berechnungsaufwand gerechtfertigt sein.

Sofern das zu berechnende Aggregationsergebnis jedoch nicht bereits redundant in einer übergeordneten Struktur abgelegt ist, muß entweder auf alle Kollektionselemente oder, wenn vorhanden, zumindest auf einen entsprechenden Index zugegriffen werden. Gleiches gilt auch bei der im folgenden erörterten Umwandlung von Kollektionen von einem Typ in einen anderen.

2.2. Lesende und schreibende Elementzugriffe sowie Kollektionstypumwandlung

Bei der Umwandlung von Kollektionen in einen anderen Kollektionstyp müssen alle Kollektionselemente gelesen und in die neue Kollektion übernommen werden. Hier können die bereits bei den Operationen auf Strukturen diskutierten relationalen Zugriffsvarianten zum Zugriff auf die Sekundärsätze mit ausgelagerten Kollektionselementen eingesetzt werden. Gleiches gilt auch beim lesenden und schreibenden Elementzugriff:

- ▷ Per Table Scan, der hier auch als Subtable Scan bezeichnet werden kann.
- ▷ Per Direktzugriff mittels Fetch bei physischer Referenzierung.
- ▷ Per Index Scan mit nachfolgendem Fetch, falls die Kollektionselemente über einen lokalen Index referenziert sind. Einsetzbar sind auch globale Indexe, wenn die Kollektion über einen Präfix des Suchschlüssels angegeben werden kann.
- ▷ Per Verbundoperation.

Es ist auch ein Zugriff über spezielle Kollektionszugriffsstrukturen, wie sie in Abschnitt 4.1.3 und Abschnitt 5.2.6 besprochen wurden, denkbar. Allerdings sind solche Zugriffsstrukturen noch wesentlich geeigneter für spezielle Kollektionsoperationen, wie sie nachfolgend beschrieben werden.

2.3. Spezielle Operationen auf Kollektionen

Gegliedert nach der Kollektionsart, sind folgende relationale Operationsumsetzungen möglich:

- ▷ *Operationen auf Feldern*

  Für das Suchen, Lesen und Zuweisen von Teilfeldern können, basierend auf dem Feldindex, relationale Standardoperationen zur Anwendung kommen. Sie können allerdings durch spezielle Zugriffsstrukturen, wie zum Beispiel Suffix-Bäume, stark beschleunigt werden. Gleiches gilt für den Test auf Gleichheit zwischen Feldern und Teilfeldern.

- ▷ *Operationen auf Mengen und Multimengen*

  Für den Enthaltenseinstest, die Teilmengenoperationen, die Mengenoperationen Vereinigung, Durchschnitt und Differenz, für den Gleichheitstest und den Test auf die leere Menge sowie für die Duplikateliminierung und die Bestimmung der Kardinalität ohne Duplikate können über die relationalen Standardtechniken hinaus spezielle Kollektionsrepräsentationen, wie Signaturen und Bitmaps, und spezielle Zugriffspfade, wie Signaturindexe, eingesetzt werden (siehe Abschnitt 4.1.3.2).

  Diese Techniken erfordern jedoch eine Erweiterung der in relationalen Datenbank-Management-Systemen in der Regel anzutreffenden Fähigkeiten: Erstens muß die Erzeugung und Verwaltung dieser speziellen Strukturen unterstützt werden, zweitens muß ihre Nutzung bei der Anfragebearbeitung ermöglicht werden, und drittens sind entsprechende Erweiterungen bei der Anfrageoptimierung nötig.

  Allerdings versprechen solche Erweiterungen auch wesentliche Leistungssteigerungen, wie zum Beispiel bei der Verwendung von Signaturbäumen oder hierarchischen Bitmap-Indexen zur Beschleunigung von Operationen auf verschachtelten Mengen (Mengen von Mengen) nachgewiesen werden konnte [Dep86, MMNM03].

- ▷ *Operationen auf Listen*

  Für Listen ist im Gegensatz zu Mengen die Reihenfolge ihrer Elemente wesentlich. Dieses Konzept wird von den auf Mengenverarbeitung ausgerichteten RDBMS und ORDBMS nicht optimal unterstützt, da hier die Navigation auf den Elementen eine wichtige Rolle spielt und die sequentielle Reihung der Listenelemente durch Positionsnummern, Verkettungen oder Zugriffstrukturen teilweise umständlich und wartungsaufwändig realisiert werden muß. Nichtsdestoweniger lassen sich, wenn auch mit erheblichem Aufwand, die Listenoperationen, wie das Einfügen, das Ausschneiden und das Kopieren von Elementen beziehungsweise von Sublisten an gegebenen Positionen, mit relationalen Standardtechniken unter intensiver Nutzung von Sortierungen realisieren. Gleiches gilt für die Verkettung, das Aufspalten und das Abtrennen des ersten und letzten Elementes von Listen sowie für den Test auf Gleichheit und die leere Menge und das Suchen nach Elementen und Sublisten.

  Jedoch können hier sehr große Leistungsgewinne durch den Einsatz spezieller Listenspeicherstrukturen und Listenzugriffsstrukturen erzielt werden. Allerding erfordern auch diese neuen Strukturen objektrelationale Ausführungsplanoperatoren, die

einerseits den navigierenden Zugriff auf sequentielle beziehungsweise verkettete Speicherstrukturen und andererseits die Nutzung spezieller Listenzugriffsstrukturen effizient erlauben.

Speicher- und Zugriffsstrukturen, die diese Operationen auf Feldern, Mengen, Multimengen und Listen unterstützen, wurden in den Kapiteln 3 und 4 vorgestellt. Dementsprechend sind sie auch mit PRDL definierbar und tauchen so auch in Kapitel 5 auf.

6.1.5 Fazit

Bei der Betrachtung der Operationen auf komplexen Objekten läßt sich also feststellen, daß eine weitgehende Realisierbarkeit mit den in relationalen Datenbank-Management-Systemen vorhandenen Mitteln gegeben ist. Allerdings erfordern einige Konzepte, wie spezielle Zugriffspfade, die Erweiterung der zur Verfügung stehenden Ausführungsplanoperatoren und die Integration entsprechender Speicherstrukturen. Diese notwendigen Erweiterungen können jedoch auf den vorhandenen Konstrukten des Speichersystems aufbauen.

Insgesamt wird deutlich, daß die erweiterten Operationen in ORDBMS prinzipiell mit den bekannten relationalen Planoperatoren umsetzbar sind. Effizienzsteigerungen lassen sich dabei durch die Integration spezieller, auf die objektrelationalen Erfordernisse ausgerichteter Operatoren und Anfragebearbeitungskonzepte erreichen.

6.2 Verarbeitungskosten bei komplexen Objekten

Um Aussagen über die Leistungsfähigkeit einzelner physischer Speicherstrukturen hinsichtlich verschiedener Zugriffsalgorithmen tätigen zu können, sind neben detaillierten Kenntnissen physischer Kenngrößen zu Cluster-Eigenschaften, Freiplatzanteilen, Auslagerungsgraden und Pufferungseffekten auch verschiedenste Indexstrukturen zu berücksichtigen. In der Literatur existieren hierzu vielfältige Vorschläge für die Schätzung der Operationskosten, die auf der Basis bestimmter Annahmen und Einschränkungen Vorhersagen über zu erwartende Externspeicherzugriffe treffen [SAC+79, Ges97, Chr84, ML89, TRSB93, NY96].

Wie in Abschnitt 6.1 erörtert wurde, können die Operationen auf komplexen Objekten im wesentlichen auf die bereits aus relationalen DBMS bekannten und implementierten grundlegenden Ausführungsplanoperatoren mit wenigen Erweiterungen und Modifikationen abgebildet werden. Daher kann sich die Betrachtung der Ausführungskosten auf diese Operationen konzentrieren [Lac04]. Auch die anderen Ansätze in der Literatur lassen sich im wesentlichen auf Basisoperationen und -konzepte mit ganz ähnlichen Techniken zurückführen [Lac04, HMWMS87, Mit89, HMS91, GGH+92].

6.2.1 Operatorklassen

Die bei der Betrachtung der Ausführungskosten berücksichtigten Ausführungsplanoperatoren werden der Übersichtlichkeit halber in vier Klassen unterteilt:

- ▷ *Basisoperatoren*

  Diese Klasse deckt Operationen ab, die unmittelbar dem Zugriff auf einzelne Sätze und Indexe sowie ihrer direkten Manipulation dienen. Sie umfaßt die Operatoren Table Scan, Fetch, Index Access beziehungsweise Index Scan, Projektion und Selektion.

▷ *Tupelstromoperatoren*

Diese Klasse umfaßt Operatoren, die nicht einzelne Tupel, sondern den zu verarbeitenden Strom an Tupeln betreffen. Er muß zwischengespeichert (Temporary Table), sortiert (Sort), gruppiert sowie ein- und ausgeschachtelt (Group oder Nest und Unnest) werden können.

▷ *Verknüpfungsoperatoren*

In dieser Klasse sind die gängigen Implementierungen von Verbundoperatoren wie Nested Loop Join, Merge Join und Hash Join zusammengefaßt.

▷ *High-Level-Operatoren*

Die ‚höheren' Operationen auf komplexen Objekten werden schließlich in dieser Klasse zusammengefaßt. Im allgemeinen lassen sich High-Level-Operatoren aus den Operationen der anderen Klassen, das heißt aus Basisoperatoren, Tupelstromoperatoren und Verknüpfungsoperatoren, zusammensetzen und werden nur wegen ihrer wiederkehrenden Verwendung in stets gleichen oder ähnlichen Konstellationen selbst als Operatoren bezeichnet.

Charakteristisch für diese Operationen ist, daß sie auf verschiedenste Art und Weise, je nach physischer Speicherform und Indexierung, in Teilanfragen und Teilausführungspläne umzusetzen sind. Damit eröffnen sie erheblich größere Möglichkeiten zur Optimierung als die anderen Operatoren und stellen einen Übergangsbereich zwischen den vordefinierten Operatoren und den komplexen Anfragen dar. Gleichzeitig bedeutet das jedoch, daß es schwierig ist, feststehende Kostenformeln für sie anzugeben.

Die High-Level-Operatoren umfassen Operationen auf den verschiedenen Arten von Kollektionen: die Mengenoperationen Union, Difference und Intersect, den Enthaltenseins- beziehungsweise Elementtest sowie die Berechnung von Kardinalität und weiteren Aggregationen auf Mengen, Listen und Feldern. Ebenfalls fallen spezielle Listen- und Feldoperationen, wie Einfügen, Anhängen, Verketten und die Positionssuche, in diese Kategorie. Auch Subobjektzugriffe und -operationen fallen in diese Klasse.

Im Vergleich zu den Basisoperatoren zeichnen sich die Tupelstrom- und insbesondere die Verknüpfungsoperatoren durch eine erhöhte Komplexität aus. Sie erlauben höherstufige Strukturveränderungen, die das Zusammenführen ebenso wie das Auflösen von Tupeln und Referenzen betreffen.

Die Mengen-, Listen- und Feldoperationen liegen schließlich in ihrer Komplexität zwar teilweise nicht deutlich höher als manche Tupelstrom- und Verknüpfungsoperatoren, können aber bezüglich ihrer Funktionalität zum großen Teil auf diese abgebildet werden.

Von Interesse ist bei den beiden höchsten Operationsklassen insbesondere, daß einige semantisch äquivalente Operationen unterschiedliche Laufzeitcharakteristika aufweisen. Angepaßte Varianten der Operationen, die der Unterstützung spezieller Optimierungen dienen (wie beispielsweise Signaturen), werden hier jedoch nicht eingehender betrachtet.

6.2.2 Entwurfskriterien

Der Betrachtung der Ausführungskosten wurden eine Reihe von Kriterien zugrunde gelegt. Diese Kriterien werden zunächst kurz aufgelistet und vorgestellt. Ihre detailliertere Diskussion findet dann in den folgenden Unterabschnitten statt.

▷ *Einfache Kostenfunktionen*

Der Berechnungsaufwand für die Kostenfunktionen sollte höchstens polynomiale Komplexität aufweisen. Idealerweise werden jedoch lineare Approximationen eingesetzt.

▷ *Flexible Parametrisierbarkeit*

Falls detailliertere Kenntnisse über Auslagerungsgrade, Pufferungsgrade und Freiplatzanteile vorhanden sein sollten, müssen sie unkompliziert in die Kostenfunktionen einbettbar sein, um genauere Resultate konstruieren zu können.

▷ *Modularität*

Die Kostenfunktionen sollen so beschaffen sein, daß sich komplexere Operationen durch unkomplizierte Kombination der Baiskostenfunktionen ergeben. Die verschiedenen Kostenfunktionen sollen also orthogonal miteinander verwendbar sein.

▷ *Atomarität und Begrenztheit*

Die Basiskostenfunktionen sollen die meisten einfachen Einsatzfälle abdecken. Außerdem sollen nicht unnötig viele Fallunterscheidungen und spezielle Kostenfunktionen eingeführt werden, von denen abzusehen ist, daß sie nur sehr eingeschränkt anwendbar sind.

1. Einfachheit der Kostenfunktionen

In der Literatur werden zahlreiche Kostenmodelle vorgeschlagen und Untersuchungen zu Operationskosten vorgestellt, unter anderem: [Yao77a, SAC+79, Bat86a, HFLP89, OL90, CG94, GGT95, Ioa97, GN93, Böh00, HB04]. Die Mehrzahl dieser Kostenmodelle basiert auf komplexen mathematischen Modellen, denen gewisse Annahmen über die Verteilung, Größe und Zugriffswahrscheinlichkeit der betrachteten Datenbankobjekte zugrunde liegen. Ein sehr häufig zu findender Ansatz sind Kostenmodelle, die auf verschiedenen Wahrscheinlichkeitsverteilungen basieren. Solche stochastischen Modelle abstrahieren häufig sehr stark von realen Verhältnissen, indem sie idealisierte Annahmen unterstellen, wie zum Beispiel die Vernachlässigung der Pufferung, die gleichmäßige Zuordnung der internen Sätze auf Seiten und die Gleichverteilung der Satzzugriffswahrscheinlichkeiten.

Dabei werden oft sehr komplexe und für praktische Zwecke relativ ungeeignete Formeln angegeben, die hohe Berechnungskomplexitäten aufweisen und daher den Optimierer auf inakzeptable Weise verkomplizieren und verlangsamen würden [Lac04]. Daher ist es fragwürdig, ob die Resultate aus solchen Formeln, die letztlich primär zur Differenzierung günstiger und ungünstiger Ausführungspläne dienen sollen, in einem vernünftigen Verhältnis zum betriebenen Berechungsaufwand stehen. Im praktischen Einsatz werden aus Leistungsgründen überwiegend einfache (lineare) Modelle und Heuristiken eingesetzt. Nur so kann ein Anfrageoptimierer bei einer hohen Frequenz von Ad-hoc-Anfragen oder häufigen dynamischen Anfragen überhaupt ein akzeptables Antwortzeitverhalten realisieren und verschiedene Ausführungspläne bewerten [Lac04].

Andererseits werden komplexere, wiederholt gestellte Anfragen meist einer intensiveren Optimierung unterzogen (in DB2 gibt es beispielsweise zehn verschiedene sogenannte Optimization Levels). Da in diesem Fall üblicherweise deutlich mehr Alternativen vom Optimierer in Betracht gezogen und diese ‚tiefer' durchgerechnet werden, wird der Einsatz komplexerer und exakterer Kostenfunktionen sinnvoll, um die höhere Vielfalt an Ausführungsplänen genauer miteinander vergleichen zu können. Die auf diese Weise einmalig anfallenden hohen Optimierungskosten werden durch entsprechend häufiges Ausführen einer

derart optimierten Anfrage wieder nivelliert. Dies gilt aber nur, wenn die durch das Modell getroffenen Annahmen hinreichend realistisch sind [Lac04, Ges97, ML89].

Die Bestimmung optimaler Speicherstrukturen und Algorithmen für eine gegebene Anfragelast stellt ein ähnlich komplexes Problem dar, wie das Finden eines optimalen Ausführungsplans für Anfragen. Es liegt nahe, hier Parallelen zu ziehen. Denn tatsächlich ist es im praktischen Einsatz weniger von Interesse, etwa *den* besten oder schnellsten Ausführungsplan zu finden, als vielmehr, ungünstige Pläne mittels heuristischer Verfahren zu verwerfen und den Raum an Alternativen durch gezielte Vereinfachungen zu begrenzen.

Für die folgenden Überlegungen ist es daher von zentraler Bedeutung, mit Hilfe von Kostenfunktionen Aussagen abzuleiten, die sich gegebenenfalls zu Heuristiken verallgemeinern lassen. Das kann dann sinnvoll geschehen, wenn die Kostenfunktionen gewisse Grenzfälle abdecken und asymptotische Aussagen über bestimmte Speicherstrukturen und Algorithmen erlauben. Auf eine geringe Komplexität der Kostenschätzungen wurde auch deshalb besonderer Wert gelegt, damit sie bei der späteren Bewertung ganzer Anfragelasten sinnvoll als Grundlage verwendet werden können.

2. Flexible Parametrisierbarkeit

Beim Entwurf wurde auf die flexible Parametrisierbarkeit der Kostenformeln Wert gelegt, um sie möglichst flexibel an unterschiedliche Situationen anpassen zu können. Dadurch sollte ihre Anwendbarkeit verbessert werden. Gleichfalls wird so die Beschreibung mehrerer spezieller Anfrageverarbeitungsmöglichkeiten mit einer einzigen allgemeineren Formel möglich. Auf diese Weise soll die Kostenmodellierung möglichst kompakt gehalten werden. Folgende Liste führt beispielhaft einige Kostenfaktoren auf, die Ein- und Ausgabevorgänge auf Externspeichermedien maßgeblich beeinflussen.

- ▷ *Parameter zur Beschreibung von Operationsart und Datenorganisation:*
  - ◇ Freiplatz
  - ◇ Verhältnis der Anteile von wahlfreiem und sequentiellem Zugriff
  - ◇ Indexhöhen
  - ◇ Sortierordnungen
  - ◇ Auslagerungen
- ▷ *Parameter zur Beschreibung der Charakteristika des Externspeichermediums:*
  - ◇ Suchzeiten
  - ◇ Latenzzeiten
  - ◇ Übertragungszeiten

Diese Kostenfaktoren sind absichtlich auf relativ hohem Abstraktionsniveau angesiedelt. Technische Maßnahmen zur Leistungssteigerung, wie Externspeicher-Caches, RAID-Arrays und so weiter, bleiben dabei ausgeklammert.

3. Modularität

Damit über die einzelnen Kostenformeln später auf einfache Weise die Kosten für gesamte Pläne oder in einem weiteren Schritt die Kosten von gesamten Workloads bestimmt werden können, ist ein modularer Aufbau unabdingbar. Die orthogonale Kombinierbarkeit der Basisoperationen muß dabei natürlich in analoger Weise in den Kostenfunktionen reflektiert werden. Die Forderung nach Modularität steht dabei in gewissem Widerspruch zur Forderung nach Atomarität und flexibler Parametrisierbarkeit. Je vielfältiger der Parameterraum

ist, desto schwieriger wird es, die Orthogonalität aufrechtzuerhalten, ohne die Schnittstellen der einzelnen Module künstlich aufzublähen. Andererseits würde die Modularität die Definition ähnlicher, aber jeweils spezialisierter Komponenten erlauben, die aber durch die Forderung nach Atomarität weitestgehend verboten wird. Daher wurde hier ein Mittelweg eingeschlagen, der versucht, allen Entwurfskriterien möglichst ausgeglichen zu genügen.

4. Atomarität und Begrenztheit

Es sei bemerkt, daß die hier vorgestellten Kostenformeln nicht alle Fälle abdecken können. In der Literatur existiert eine Vielfalt an Indexierungsverfahren (wie R-Bäume, Gridfiles, Multiindexe und viele andere mehr), die allein schon aus Platzgründen nicht alle im Detail berücksichtigt werden können. Sie würden die Kostenformeln nur überladen und zu keinen wesentlich neuen Kernaussagen beitragen. Statt dessen beschränkt sich die Darlegung auf einige typische Standardindexe, wie B*-Bäume. Zudem konzentrieren sich die hier geführten Untersuchungen auf lesende Operationen (Retrieval), die in der Praxis eine hohe Relevanz aufweisen. Auf Manipulationsoperationen, wie Einfügen, Ändern und Löschen von Sätzen, wird an entsprechender Stelle in begrenztem Umfang eingegangen. Auch die Optimierer von DBMS-Produkten berücksichtigen diese Operationen oftmals weit weniger als reine Leseoperationen.

6.2.3 Operationskosten

Es folgt die Betrachtung der entstehenden Operationskosten gegliedert nach den Operatorklassen: Zunächst werden die einfachen Basisoperationen beschrieben, bevor anschließend auf die Tupelstrom-, Verknüpfungs- und High-Level-Operationen, die jeweils aufeinander aufbauen, eingegangen wird. Dabei sollen jedoch die Operationen und ihre Kostenformeln nicht einzelnen durchdekliniert, sondern lediglich an Beispielen vorgestellt werden. Eine ausführliche Beschreibung und Diskussion der Kostenformeln ist in [Lac04] zu finden.

6.2.3.1 Basisoperatoren

Die Grundlage für die Ausführung aller Operationen auf komplexen Objekten bilden Zugriffsoperationen, die die internen Sätze aus Segmenten vom Externspeicher holen. Hier wird auf bekannte relationale Techniken zurückgegriffen, die sich wie folgt grob klassifizieren lassen:

- ▷ Index Scan zum Zugriff auf interne Sätze über Zugriffspfade, wie:
  - ◇ Hashtabelle
  - ◇ Baumindex
  - ◇ Indexorganisierte Tabelle
- ▷ Table Scan zum sequentiellen Zugriff auf interne Sätze
- ▷ Fetch zum Direktzugriff auf interne Sätze über:
  - ◇ Physische Referenzen (Tupel-ID-Konzept)
  - ◇ Logische Referenzen (Objekt-ID-Konzept)

Allerdings muß der Umstand berücksichtigt werden, daß in unseren Szenarien ein logisches Tupel nicht zwangsläufig mit nur einem internen Satz korreliert.

Neben dem Zugriff auf die internen Sätze umfassen die Basisoperationen natürlich auch die Operationen

- ▷ Selektion und
- ▷ Projektion.

Als Beispiel für die Basisoperationen soll nun der Fetch-Operator beschrieben werden.

Der Operator Fetch

Die Funktion des Fetch-Operators ist es, jede der übergebenen Tupelreferenzen aufzulösen und den an dieser Adresse vorhandenen Satz zurückzuliefern. Ähnlich wie beim Table Scan, können einfache Suchargumente angewendet werden, um qualifizierte Sätze zu selektieren.

Er hat drei Eingabe- und einen Ausgabeparameter:

- ▷ Eingabeparameter: Einen Zielbereich für die Tupelreferenzen (Segment, Cluster oder Datenbank)
- ▷ Eingabestrom: Menge von k Tupelreferenzen T_{Ref}
- ▷ Eingabestrom: Menge von einfachen Suchargumenten S_{Arg}
- ▷ Ausgabestrom: Menge T von Tupeln

Bei Direktzugriffen müssen zwei unterschiedliche Arten differenziert werden: der k-fache wahlfreie Direktzugriff und der k-fache Direktzugriff innerhalb des gleichen Clusters. Die Trennung hat hier insbesondere deswegen Sinn, weil Clustereffekte dadurch explizit berücksichtigt werden können:

- ▷ *Wahlfreier Zugriff*

  Unter Berücksichtigung eines Pufferungsgrades β_R (Anteil gepufferter Seiten der Relation R) und eines Anteils von σ_R auslagerten Tupeln gilt:
  - ◇ Der Anteil ungepufferter und also beim Zugriff vom Externspeicher zu holender Seiten beträgt $(1 - \beta_R)$.
  - ◇ Die Zugriffskosten werden durch Tupelauslagerungen erhöht und belaufen sich daher auf $(1 + \sigma_R)$.

  Somit ergeben sich für k wahlfreie Direktzugriffe Zugriffskosten von:

$$K_{kRandom}(k, \beta_R, \sigma_R) = k(1 - \beta_R)(1 + \sigma_R)$$

- ▷ *Clusterbeschränkter wahlfreier Zugriff*

  Befindet sich die Tupelmenge, auf welche wahlfrei zugegriffen wird, innerhalb eines definierten Clusters mit S_C Seiten, so kommt eine leicht abgewandelte Formel zum Einsatz. Dabei werden Pufferung und Tupelauslagerungen wie im allgemeinen Fall ebenfalls berücksichtigt. Die Besonderheit ist hier lediglich, daß nach einer gewissen Anzahl an Zugriffen mit hoher Wahrscheinlichkeit der gesamte Cluster bereits im Puffer vorhanden ist und daher keine weiteren Externspeicherzugriffe notwendig werden. Als Vereinfachung wird angenommen, daß eine lineare Abhängigkeit zwischen der Anzahl wahlfrei gelesener Tupel k und der Anzahl der aus dem Cluster zu lesenden Seiten besteht. Je kleiner die einzelnen Sätze und je größer der Belegungsgrad der Clusterseiten, desto häufiger können Seitenzugriffe durch einen Pufferzugriff erledigt werden.
  - ◇ Die Wahrscheinlichkeit, daß auf eine von S_C Seiten zugegriffen wird, ist $\frac{1}{S_C}$.
  - ◇ Bei k-fachem Zugriff ist also die Wahrscheinlichkeit, daß mindestens einmal auf eine Seite zugegriffen wird $1 - (1 - \frac{1}{S_C})^k$.

◇ Für k Tupelzugriffe auf ein Cluster mit S_C Seiten sind also $S_C \left(1 - (1 - \frac{1}{S_C})^k\right)$ Seitenzugriffe zu erwarten. Die Clustergröße S_C stellt dabei natürlich eine obere Schranke dar, da $1 - (1 - \frac{1}{S_C})^k \leq 1$.

Für ausgelagerte Tupel wird angenommen, daß sie sich außerhalb des Clusters befinden. Der Anteil σ_R der Seitenzugriffe, der durch Auslagerung verursacht wird, erfährt durch die Clusterung also keine Reduzierung. Der Pufferungsgrad β_R bezieht sich allerdings sowohl auf die Wahrscheinlichkeit, eine Seite des Clusters als auch eine zu einem ausgelagerten Satz gehörende Seite im Puffer vorzufinden. Es ergibt sich also insgesamt:

$$K_{kRandomCluster}(k, \beta_R, \sigma_R, S_C) = S_C \left(1 - (1 - \frac{1}{S_C})^{k\ (1+\sigma_R)}\right)\ (1 - \beta_R)$$

Die CPU-Kosten können in beiden Fällen mit

$$K^{CPU}_{kRandom} = K^{CPU}_{kRandomCluster} = k * K^{CPU}$$

abgeschätzt werden. Dabei bezeichnet K^{CPU} die Kosten für eine durchschnittliche Operation auf einem Tupel, wie das Vergleichen oder die Tupelkombination et cetera. Zwischen den einzelnen Operationen soll dabei jedoch nicht unterschieden werden.

Unter idealen Bedingungen sind sämtliche Seiten eines Clusters, die zum gleichen Clusterschlüssel gehören, in aufeinanderfolgenden Seiten abgelegt. In diesem Sonderfall könnte man noch den sequentiellen Beschleunigungsfaktor S einbeziehen. Dann müßten allerdings auch immer sämtliche Seiten eines Clusters beim Zugriff auf nur einen einzelnen Wert geholt werden, was im Schnitt aber zu erheblichem I/O-Mehraufwand und größerem Pufferbedarf führen würde.

Der Operator Index Scan

Der Index-Scan-Operator wertet einfache Suchargumente auf einer Indexstruktur I aus und liefert Satzadressen von qualifizierten Tupeln zurück. Die Art der Suchargumente ist durch die Struktur des Indexes beschränkt. Es können üblicherweise nur Suchargumente ausgewertet werden, die sich auf ein indexiertes Attribut beziehen. Ist der Indexschlüssel mit weiteren Attributwerten, -präfixen oder -suffixen verkettet, können diese für die Auswertung der Suchargumente ebenfalls (begrenzt) herangezogen werden.

Der Index-Scan-Operator unterstützt zudem das sortierte Auslesen von Relationen und kann als Zugriffsoperator auf vorberechnete Verbunde oder andere indexorientierte Strukturen eingesetzt werden.

Er hat zwei Eingabe- und einen Ausgabeparameter:

▷ Ausgabestrom: Menge von Tupelreferenzen T_{Ref}

▷ Eingabestrom: beliebige Indexstruktur I

▷ Eingabestrom: Menge von einfachen Suchargumenten S_{Arg}

Obwohl in der Literatur vielfältige Spezialindexe, etwa für die schnelle Wortsuche in Texten, existieren, beschränken sich die im folgenden vorgestellten Kostenabschätzungen auf den B*-Baum, die in der Praxis bei weitem am häufigsten angewandte Indexstruktur.

Im B*-Baum enthalten die inneren Knoten lediglich Schlüsselwerte oder Präfixe des indexierten Attributs oder einer Verkettung von Attributen. Die eigentlichen Daten werden nur in den Blattknoten gespeichert, die zudem doppelt miteinander verkettet sind, damit ein vollständiges oder partielles Auslesen des Index (Index Scan, Index Range Scan) effizient erfolgen kann. Im folgenden wird davon ausgegangen, daß, wie in DBMS allgemein

üblich, lediglich Verweise (in Form von Tupel-IDs) in den Blattknoten abgelegt sind. Bei Änderungen wird der B*-Baum automatisch höhenbalanciert und garantiert dadurch bei gegebener Höhe konstante Suchzeiten für das Auffinden eines beliebigen Schlüssels. Der allgemeine Fall schließt die Indexierung von nicht eindeutigen Werten in der Zielrelation ein, wodurch pro Schlüsselwert im Index potentiell mehrere Blattseiten mit referenzierten Tupeln möglich sind. Hier entscheidet insbesondere die Häufigkeitsverteilung der Werte innerhalb der Relation über den relativen Nutzen des Indexzugriffs.

Zur Quantifizierung der Kosten muß dabei zwischen folgenden Fällen unterschieden werden:

▷ *Punktanfragen via B*-Baum*

Bei einer Punktanfrage auf einem B*-Baum müssen sämtliche inneren Knoten entlang des Pfades von der Wurzel zu dem gewünschten Blattknoten gelesen werden. Falls der Indexschlüssel nicht eindeutig (unique) ist, fallen eventuell zusätzliche Lesekosten für weitere Indexseiten an. Die Berücksichtigung des Pufferungsgrades in Form eines konstanten Faktors scheint beim B*-Baum zunächst schwierig, da die Wahrscheinlichkeit für das Vorhandensein einer bestimmter Indexseite im Puffer von der Distanz dieser Seite zur Wurzel des Baumes abhängt. Der hohe Verzweigungsgrad des Baumes rechtfertigt es jedoch, bei entsprechendem Verhältnis von Puffergröße und Indexgröße davon auszugehen, daß ein konstanter Anteil der inneren Knoten im Puffer vorzufinden ist.

Für Punktanfragen berechnet sich die Anzahl der erforderlichen Seitenzugriffe wie folgt:

- ◇ $log\ S_I - 1$ Zugriffe auf die Indexseiten auf dem Pfad von der Wurzel zu den Blattseiten, bei S_I Seiten im Index.
- ◇ Durchschnittlich $max\left(1, \frac{Leafs_I}{Keys_I}\right)$ Zugriffe auf die Blattseiten mit den gefundenen Tupel-IDs, bei insgesamt $Leafs_I$ Blattseiten und $Keys_I$ unterschiedlichen Schlüsselwerten im Baum.
- ◇ Der Anteil ungepufferter Seiten beträgt dabei $(1 - \beta_I)$.
- ◇ Tupelauslagerungen gibt es natürlich nicht, denn die Tupel befinden sich ja nicht im Baum, sondern außerhalb.

Insgesamt ergeben sich also folgende Kosten:

$$K_{BBaumKeySearch_{Idx}}(S_I, Leafs_I, Keys_I, \beta_I) = \left(log\ S_I - 1 + max\left(1, \frac{Leafs_I}{Keys_I}\right)\right)(1 - \beta_I)$$

Neben dem reinen Indexzugriff und dem Lesen der Tupelreferenzen ist es üblicherweise von Interesse, die eigentlichen Tupel selbst zu lesen. Da es sich dabei um wahlfreie Direktzugriffe handelt, kann auf die bereits bei der Besprechung des Fetch-Operators vorgestellten Kostenschätzungen zurückgegriffen werden.

Die CPU-Kosten können durch

$$K^{CPU}_{BBaumKeySearch_{Daten}} = K^{CPU}$$

abgeschätzt werden.

▷ *Bereichsanfragen via B*-Baum-Index*

Entscheidend für die Richtigkeit der Kostenabschätzung ist eine möglichst genaue Kenntnis des Selektivitätsfaktors Sel. Dieser läßt sich für Schlüsselwerte von einfachen, ganzzahligen Wertebereichen unter Gleichverteilungsannahmen noch leicht rechnerisch abschätzen. Für komplexere Wertebereiche sind hierzu Histogrammdaten oder ähnliches erforderlich.

Für Bereichsanfragen berechnet sich die Anzahl der Seitenzugriffe durch:

◇ $log\ S_I - 1$ Indexseiten auf dem Pfad von der Wurzel zu den Blattseiten.

◇ Anzahl zu lesender Blattknoten: $Sel * Leaf_{S_I}$.

◇ Der Anteil ungepufferter Seiten: $(1 - \beta_R)$.

Insgesamt ergeben sich

$$K_{BBaumRangeSearch_Idx}(S_I, Sel, Leafs_I) = (log\ S_I - 1 + Sel * Leafs_I)\,(1 - \beta_I)$$

als Kosten.

Diese Abschätzung gilt natürlich auch im Fall einer indexorganisierten Tabelle. Für den Zugriff auf die eigentlichen Datensätze fallen dann aber keine weiteren Kosten an. Allerdings ist durch die größere Menge von Daten, die in den Indexeinträgen gespeichert werden, das Verhältnis von Indexeinträgen (N_I) zur Anzahl der Blattknoten ($Leafs_I$) deutlich geringer, was die Kosten für jeden Indexzugriff erhöht.

Bei nicht-indexorganisierten Tabellen müssen die referenzierten Tupel wiederum im Anschluß an den Index Scan gelesen werden. Auch hier handelt es sich um wahlfreie Direktzugriffe per Fetch.

Gegebenenfalls können die Referenzen zunächst nach auf- oder absteigenden Adressen sortiert werden, um ein wiederholtes Holen der gleichen Seite vom Hintergrundspeicher zu vermeiden. Bei einer gleichmäßigen Verteilung der Sätze auf den belegten Seiten der Relation ergibt sich dann die Abschätzung:

$$K_{BBaumRangeSearch_Daten}(Sel, S_R) = Sel * S_R(1 + \sigma_R)$$

6.2.3.2 Tupelstromoperatoren

Die Tupelstromoperatoren umfassen Operatoren, die im Gegensatz zu den Basisoperatoren nicht auf einzelnen Tupeln, sondern auf ganzen Mengen beziehungsweise Strömen von Tupeln arbeiten:

▷ Operator Temporary Table

▷ Operator Sort

▷ Operator Nest beziehungsweise Group

▷ Operator Unnest

Diese Operatoren sind ebenfalls, Unnest ausgenommen, aus relationalen DBMS bekannt und bauen in dem Sinne auf den Basisoperatoren auf, daß die Basisoperatoren als ‚Datenlieferanten' für die Tupelstromoperatoren dienen und sie in Anfrageplänen verschachtelt verwendet werden.

Der Sort-Operator soll nun als Beispiel für die Tupelstromoperatoren beschrieben werden.

Der Sort-Operator

Der Sort-Operator ordnet die Elemente der Eingabemenge neu an. Das Sortierkriterium wird durch die Übergabe der zu sortierenden Attributspalten festgelegt. Aus logischer Sicht

wandelt der Sortieroperator eine beliebige Kollektion (Relation, Multimenge, Feld, Liste) in eine sortierte Liste um. Für die Implementierung oder die Struktur der Ausgabemenge hat diese Überlegung aber keine Konsequenzen, außer daß die einem Sortieroperator folgenden Operationen von der Sortierreihenfolge Gebrauch machen können.

Für den Operator Sort wird hier ein typischer, aber sehr einfacher Algorithmus skizziert, obwohl in der Literatur auch Derivate zu finden sind, auf die aber aus Platzgründen nicht eingegangen werden kann und soll [Här77, Knu97, HR01]. Der Operator besitzt zwei Eingabeparameter und einen Ausgabeparameter:

- ▷ Ausgabetupelstrom: T_{Out}
- ▷ Eingabetupelstrom: T_{In}
- ▷ Eingabeparameter: Eine Sequenz von Spalten Col_{Name}, nach denen auf- oder absteigend sortiert werden soll

Für die Bestimmung der Kostenfunktionen für den Sort-Operator wird, wie auch sonst allgemein üblich, angenommen, daß die zu sortierenden Relationen beziehungsweise Tupelströme in der Regel größer als der verfügbare Hauptspeicher sind und deshalb ein Externspeichersortierverfahren zugrunde zu legen ist. Als Verfahrensvariante wird hier das bekannte Mehrphasen-Mischverfahren angenommen, obwohl in der Literatur eine Reihe weiterer relevanter Verfahren angegeben werden, die aber meist Optimierungen darstellen (Mehrwege-Mischen, Replacement-Selection, Radix-Sort und andere mehr). Eine relativ umfassende und detaillierte Analyse der einzelnen Algorithmen findet sich in [Knu97]. Interessante datenbankspezifische Implementierungsaspekte werden u.a. in [HR01] diskutiert.

Die verschiedenen Varianten des Sortierens durch Mehrphasen-Mischen und verwandter Algorithmen laufen in zwei Phasen ab. In der ersten Phase werden unter maximaler Ausnutzung des vorhandenen hierfür vorgesehenen Pufferspeichers (Sort Heap) möglichst lange sortierte Teilfolgen erzeugt (Runs). Diese werden nach einem bestimmten Muster auf zwei oder mehrere Externspeichersegmente verteilt. In der zweiten Phase werden die einzelnen teilsortierten Folgen zu einer vollständig sortierten Ergebnisfolge zusammengemischt. Durch Maximierung der Folgenlänge, geschickte Mischverfahren und adäquate Puffergrößen versucht man, die Anzahl der Durchläufe (Passes) zu minimieren.

In der Kostenermittlung müßte eigentlich berücksichtigt werden, daß das beschriebene Verfahren durch eine zunehmende Anzahl von Segmenten asymptotisch schneller wird. Aus Gründen der Einfachheit wird jedoch angenommen, daß nur zwei Segmente genutzt werden. Es ist intuitiv einleuchtend, daß nach jedem Mischschritt die durchschnittliche Länge der sortierten Teilfolgen in den Ergebnissegmenten verdoppelt wird. Wenn $Runs$ die Anzahl initialer Runs bezeichnet, so wird das Ergebnis nach maximal $log\ Runs$ Mischvorgängen konstruiert. Jeder Mischvorgang bedeutet einen vollständigen Durchlauf aller teilsortierten Folgen. Man kann als obere Grenze die Kosten für einen Durchlauf mit S_R (Seiten der zu sortierenden Relation) annehmen. Es ergibt sich für die I/O-Kosten folglich (Seitenzugriffe):

- ▷ Für die initiale Generierung der Runs: S_R Seitenzugriffe (alle Seiten der Relation)
- ▷ Für die Mischvorgänge: $S_R \lceil log\ Runs \rceil$ Seitenzugriffe

Bei einem Pufferspeicher von P Seiten und einer Größe des Eingabestroms von S_R beträgt die durchschnittliche Folgenlänge $c * P$ (unter Annahme von Replacement Selection nach [HR01] mit einem Speed Up von $c > 1,8$ [LG98]). Sind sämtliche Seiten der Eingaberelation voll belegt, beträgt die Anzahl der Teilfolgen somit näherungsweise (unter Worst-Case-Abschätzung und der konservativen Annahme von $c = 1,5$):

$$Runs = \frac{S_R}{1,5P}$$

Für die Kostenfunktion ergibt sich somit insgesamt:

$$K_{Sort}(S_R, P) = S_R \left(1 + \frac{3}{2} \left\lceil log \left(\frac{S_R}{1,5P}\right)\right\rceil\right)$$

Die Kosten für die Vergleiche, die zu Lasten der CPU gehen, können bei N_R Tupeln in der Tabelle R wie folgt abgeschätzt werden:

$$K^{CPU}_{Sort}(N_R) = N_R \left\lceil log \left(\frac{S_R}{1,5P}\right)\right\rceil K^{CPU}$$

6.2.3.3 Verknüpfungsoperatoren

Verknüpfungsoperatoren arbeiten auch auf Tupelströmen, jedoch verknüpfen sie, wie der Name schon sagt, mehrere Tupelströme zu einem neuen. Unter diese Kategorie fallen alle bekannten Verbundverfahren, wie

- ▷ der Nested Loop Join,
- ▷ der Merge Join und
- ▷ der Hash Join,

und alle ihre unterschiedlichsten Abwandlungen und Varianten.

Beispielhaft sollen hier als Vertreter dieser Operatorenklasse die Kostenfunktionen des Nested Loop Joins besprochen werden.

Der Operator Nested Loop Join

Der Nested Loop Join ist eine einfache Implementierungsvariante von inneren und äußeren Verbunden. Er verbindet die Tupel eines inneren Tupelstroms T_{Inner} mit den Tupeln eines äußeren Tupelstroms T_{Outer}, wenn sie die spezifizierten Prädikate erfüllen.

Er hat fünf Eingabeparameter und zwei Ausgabeparameter:

- ▷ Äußerer Eingabetupelstrom: T_{Outer}.
- ▷ Innerer Eingabetupelstrom: T_{Inner}.
- ▷ Eingabeparameter: Verbundbedingung mit folgenden Untergliederugen:
  - ◇ Menge von einfachen Suchargumenten S^{Outer}_{Arg} für die äußere Verbundrelation.
  - ◇ Menge von einfachen oder von T_{Outer} abhängigen Suchargumenten S^{Inner}_{Arg} für die innere Verbundrelation.
  - ◇ Die nicht zur Suche über der inneren oder äußeren Relation geeigneten Prädikate P, die erst bei der Verbindung der Tupel ausgewertet werden können (Residual Predicates).
- ▷ Ausgabetupelstrom: T_{Out}.

Der Operator liest den äußeren Tupelstrom T_{Outer} sequentiell. Gegebenenfalls kann dabei eine Einschränkung durch die äußeren Suchargumente S^{Outer}_{Arg} und somit eine Reduzierung der Zugriffskosten erfolgen. Für jedes gelesene Tupel wird der innere Tupelstrom T_{Inner} ebenfalls vollständig sequentiell gelesen, so daß alle äußeren Tupel mit allen inneren Tupeln anhand des Prädikates P verglichen und die Tupel des Ausgabetupelstrom T_{Out} konstruiert werden können. Auch auf dem inneren Tupelstrom kann gegebenenfalls eine Zugriffskostenreduktion durch die inneren Suchargumente S^{Inner}_{Arg} erreicht werden.

Bei der Ermittlung der Operationskosten müssen folgende Fälle unterschieden werden:

1. Fall: *Potentiell unbeschränkter n:m-Verbund*

   Dieser Fall ist dadurch charakterisiert, daß sich das innere Suchargument S_{Arg}^{Inner} nicht zur Suche auf dem inneren Tupelstrom eignet, da es nicht vorhanden beziehungsweise stets *true* ist. Somit muß jedes äußere Tupel mit jedem inneren Tupel verglichen und gegebenenfalls verbunden werden.

   Hier bewegen sich die möglichen Fälle zwischen zwei Extremen:

   1.1. *Der innere Tupelstrom paßt vollständig in den Puffer*

   Idealerweise wählt man den kleineren Tupelstrom als T_{Inner}, in der Absicht, diesen vollständig in den Pufferspeicher lesen und dort halten zu können. Dadurch fallen die Bereitstellungskosten K_{Inner} für den inneren und K_{Outer} für den äußeren Tupelstrom jeweils nur ein einziges Mal an. Es ergibt sich:

$$K_{NestedLoop} = K_{Outer} + K_{Inner}$$

   Jedes Tupel der äußeren Relation muß mit jedem der inneren Relation verglichen werden. Insgesamt sind bei N_{Inner} Tupeln in der inneren und K_{Outer} Tupeln in der äußeren Relation also $N_{Inner} * N_{Outer}$ Vergleiche, die zu folgenden CPU-Kosten führen:

$$K_{NestedLoop}^{CPU} = N_{Outer} * N_{Inner} * K^{CPU}$$

   1.2. *Der innere Tupelstrom paßt nicht vollständig in den Puffer*

   Falls der innere Tupelstrom nicht vollständig in den Puffer paßt, fallen die Bereitstellungskosten für jedes aus T_{Outer} gelesene Tupel erneut an. Es ergibt sich folglich:

$$K_{NestedLoop} = K_{Outer} + N_{Outer} * K_{Inner}$$

   Die CPU-Kosten sind identisch zum Fall 1.1:

$$K_{NestedLoop}^{CPU} = N_{Outer} * N_{Inner} * K^{CPU}$$

2. Fall: *Beschränkter n:m-Verbund*

   Charkteristisch ist hier, daß das innere Suchargument S_{Arg}^{Inner} eine Bereichsanfrage darstellt.

   Der beschränkte n:m-Verbund kann im ungünstigsten Fall (bei schwach selektivem S_{Arg}^{Inner})) zum unbeschränkten Verbund (Fall 1) entarten. Er wird hier jedoch als eigener Fall behandelt, um typische Anwendungsfälle, bei denen etwa die Ergebnismengen der jeweiligen Bereichsanfragen innerhalb einzelner Cluster liegen, differenziert behandeln zu können.

   Pro Schleifendurchlauf fallen hier die Bereitstellungskosten $K_{Inner_{Range}}$ an, so daß sich folgende Zugriffskosten ergeben:

$$K_{NestedLoop} = K_{Outer} + N_{Outer} * K_{Inner_{Range}}$$

   Die CPU-Kosten orientieren sich an der mittleren Anzahl von Tupeln $\overline{N_{Inner}}$, die sich pro Bereichsanfrage in der inneren Schleife qualifizieren.

$$K_{NestedLoop}^{CPU} = N_{Outer} * \overline{N_{Inner}} * K^{CPU}$$

3. Fall: *Tupel-Merge*

   Für diesen Fall ist es charakteristisch, daß das innere Suchargument S_{Arg}^{Inner} eine Punktanfrage darstellt. Es wird angenommen, daß die Punktanfrage stets nur ein Ergebnistupel liefert. Die Werte des entsprechenden Attributs wären also demnach *unique*. Punktabfragen auf *non-unique* Attributen verhalten sich ähnlich wie Bereichsanfragen (siehe Fall 2).

   Bei Punktanfragen auf T_{Inner} fallen pro Schleifendurchlauf lediglich die Bereitstellungskosten $K_{Inner_{Point}}$ an. Entsprechend reduzieren sich auch die CPU-Kosten, da hier nur ein Tupel pro Schleifendurchlauf geprüft werden muß. Somit ergeben sich folgende Kosten:

$$K_{NestedLoop} = K_{Outer} + N_{Outer} * K_{Inner_{Point}}$$

   und

$$K_{NestedLoop}^{CPU} = N_{Outer} * K^{CPU}$$

   Ob die Gesamtkosten für Fall 3 tatsächlich kleiner oder größer als die Gesamtkosten für Fall 1 sind, hängt in starkem Maße von den verwendeten Zugriffspfaden ab, die für die entsprechenden Sätze bereitgestellt werden, sowie von der Speicherstruktur, die diesen Sätzen zugrunde liegt.

Eine optimierte Variante des Nested Loop Joins ist der Blocked Nested Loop Join [Kim80]. Wie die Namensgebung bereits zum Ausdruck bringt, werden die Tupel aus dem äußeren und inneren Tupelstrom nicht einzeln, sondern blockorientiert gelesen. Typischerweise sind mehrere Tupel innerhalb des gleichen Blocks (Seite) organisiert, so daß hier durch das Lesen des kompletten Blocks erhebliche Einsparungen bei den Externspeicherzugriffen zu erwarten sind. Auch für diese Variante wurden Kostenformeln entwickelt, auf die hier jedoch nicht eingegangen werden soll. Dazu sei auf [Lac04] verwiesen.

6.2.3.4 High-Level-Operatoren

In dieser Kategorie werden Operationen auf Kollektionen und ihren Elementen sowie Operationen auf Objekten zusammengefaßt. Zu diesen gehören (vergleiche Abschnitt 6.1):

- ▷ *Mengenoperationen*

  wie zum Beispiel die Vereinigung, die Differenz, der Durchschnitt, der Test auf Teilmengenbeziehung, der Enthaltenseinstest für Elemente, die Ermittlung der Kardinalität und alle Aggregationen, wie die Berechnung von Minimum, Maximum und Durchschnitt.

- ▷ *Listenoperationen*

  wie zum Beispiel der Elementzugriff, das Einfügen von Elementen und Listen, das Anhängen von Elementen, das Verketten von Listen, das Entnehmen von Teillisten und das Suchen von Listenelementen nach ihrem Wert oder ihrer Position.

- ▷ *Feldoperationen*

  wie zum Beispiel der Elementzugriff, Suchen von Feldelementen nach ihrem Wert und der lesende und schreibende Zugriff auf Teilfelder.

▷ *Strukturoperationen*
wie zum Beispiel die Konstruktion von Objekten und der Zugriff auf Subobjekte.

Kennzeichnend für diese Operatoren ist, daß sie eine Grauzone zwischen den Operatoren und den Anfragen bilden. Auf der einen Seite stellen sie immer wiederkehrende Operationen in komplexen Anfragen dar und werden durch spezielle SQL-Schlüsselworte und eingebaute Operatorfunktionen ausgedrückt. Andererseits erfordern sie meist komplexe Ausführungspläne, die in verschiedenen Varianten aus den Basis-, Tupelstrom- und Verknüpfungsoperationen zusammengesetzt und damit stark optimierbar sind.

So kann zum Beispiel die Schnittmengenbildung in vielfältiger Weise realisiert werden:

▷ Per Nested Loop Join mit einer Komplexität von $O(n * m)$.

▷ Per Merge Join mit einer Komplexität von $O(n + m)$ und mit einem Sortieraufwand von $O(n\ log\ n + m\ log\ m)$, wenn erforderlich.

▷ Über Bitmap-Indexe mit einer Komplexität von $O(n + m)$ beziehungsweise von $O(log\ n + log\ m)$, falls nur die Existenz einer nicht leeren Schnittmenge getestet werden soll, ohne die Schnittmenge selbst zu bestimmen.

Diese Liste ließe sich noch fortsetzen. Hier soll jedoch nur die mögliche Unterschiedlichkeit der Realisierungen vor Augen geführt werden.

Da die Operatoren dieser Klasse also auf unterschiedlichste Art aus den in den letzten Abschnitten behandelten Basis-, Tupelstrom- und Verknüpfungsoperationen aufgebaut werden, ergeben sich die Kosten aus den jeweiligen Ausführungsplänen, und es soll hier davon abgesehen werden, allgemeingültige Kostenformeln zu entwickeln. Jedoch sei auf eine eingehendere Diskussion der Thematik in [Lac04] verwiesen.

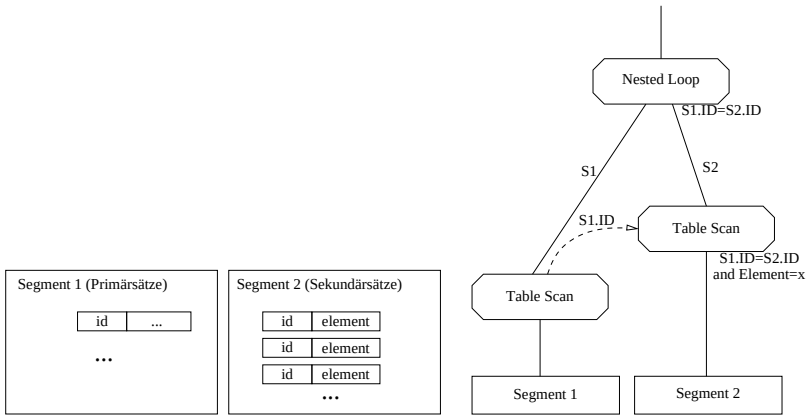
Test auf Enthaltensein eines Elementes in einer Menge

Da hier darauf verzichtet werden soll, allgemeingültige Kostenformeln für High-Level-Operatoren anzugeben, wird im folgenden nur ein Beispiel für die Vielfalt möglicher Ausführungspläne gegeben. Dies soll anhand der Operation zum Test des Enthaltenseins eines Elementes in einer Menge geschehen. Den Anfrageauswertungsplänen liegt dabei folgende SQL-Anfrage zugrunde:

```
SELECT * FROM A WHERE x IN A.Set
```

Es sollen nun verschiedene Anfrageausführungsplanvarianten diskutiert und verglichen werden:

1. Variante: *Auslagerung, logische Referenzierung und Table-Scan-Zugriff*
   Zugrunde gelegt wird eine Speicherstruktur wie in ABBILDUNG 6.1(a) und ein Ausführungsplan wie in ABBILDUNG 6.1(b). Über eine logische Referenz werden der Primärsatz und die Sekundärsätze, die die Kollektionselemente enthalten, in Beziehung gesetzt. Der Ausführungsplan besteht aus einem Nested-Loop-Verbund, bei dem der Table Scan auf Segment 1 ausgeführt wird, wobei von dort aus der Wert der logischen Referenz als Suchprädikat an den Table Scan auf Segment 2 übergeben wird. Die Segmente 1 und 2 müssen nicht notwendigerweise verschieden sein. Der Table Scan in



(a) Speicherstruktur (b) Ausführungsplan

ABBILDUNG 6.1: Variante 1

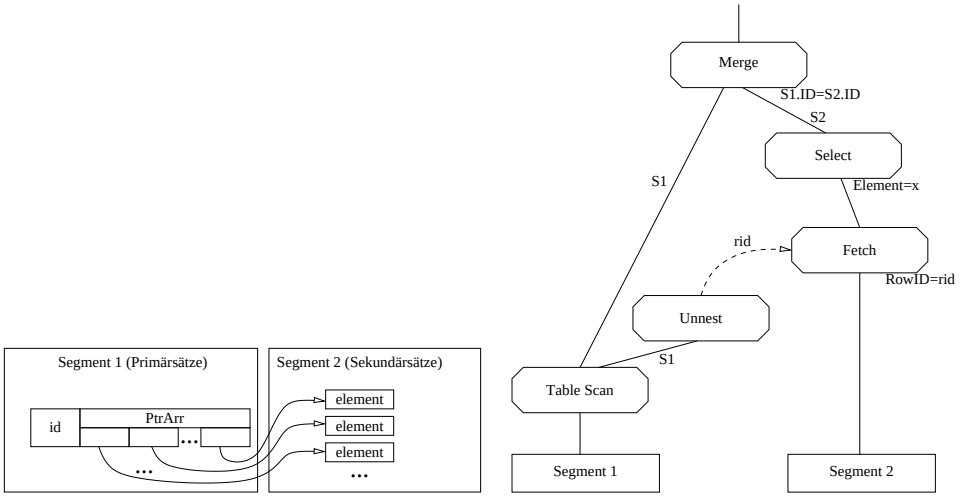
der Unteranfrage wäre in diesem Fall allerdings noch langsamer, da er auch die Primärsätze ein weiteres Mal lesen müßte. Das Prädikat der inneren Selektion „Element = x“ kann, da es ebenfalls als Suchprädikat geeignet ist, durch den Table Scan mit ausgewertet werden.

Eine Verbesserung für diesen Anfragetyp kann durch eine Clusterung innerhalb von Segment 2, optional gemeinsam mit Segment 1, erreicht werden, so daß alle Sekundärsätze einer Kollektion und gegebenenfalls auch der Primärsatz dicht beieinander gespeichert werden. Bei einer idealen Clusterorganisation, bei der jedes Cluster aus einem zusammenhängenden Fragment besteht und alle Cluster nacheinander linear angeordnet sind, würden sich die Kosten für den inneren Table Scan, der dann jeweils clusterbeschränkt wäre, verringern. Allerdings erfordert die Aufrechterhaltung der Clusterung wiederum einen erheblichen Zusatzaufwand bei Änderungsoperationen (Reorganisation, Speichervorreservierung oder ähnliches).

2. Variante: *Auslagerung, physische Referenzierung und Direktzugriff*

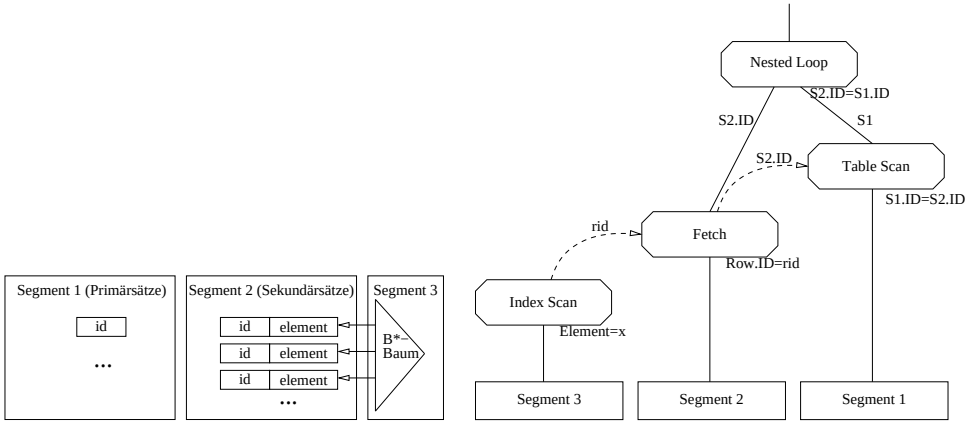
Zugrunde gelegt wird hier eine Speicherstruktur mit Zeigerfeld (Pointer Array) wie in ABBILDUNG 6.2(a) und der Ausführungsplan aus ABBILDUNG 6.2(b). Diese Variante weicht von Variante 1 nur durch das Zeigerfeld ab, das entsprechend entschachtelt werden muß. Über die einzelnen Referenzen werden dann per wiederholtem Fetch die Elemente aus Segment 2 gelesen.

Sinnvoll ist hier die Clusterung auf Subobjektebene, so daß Elemente in Segment 2 dicht beeinander liegen. Als weitere Verbesserung, sofern genügend Pufferspeicher zur Verfügung steht, könnte beim Fetch-Operator zusätzlich Prefetching betrieben werden, so daß jedesmal, wenn eine Referenz aus einem neuen Cluster angefordert wird, das gesamte Cluster sequentiell per Cluster Scan eingelesen wird. Allerdings ist diese Variante nicht immer kostengünstiger. Heuristisch könnte man sagen, daß es sich erst lohnt, wenn durchschnittlich mehr Elemente angefragt werden als jeder einzelne Cluster Seiten hat.



(a) Speicherstruktur (b) Ausführungsplan

ABBILDUNG 6.2: Variante 2

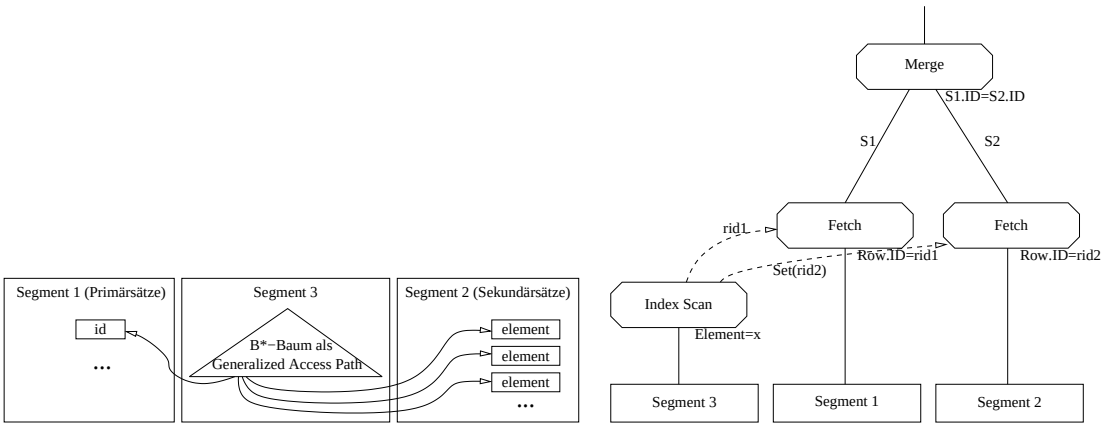


(a) Speicherstruktur (b) Ausführungsplan

ABBILDUNG 6.3: Variante 3

3. Variante: *Auslagerung, Indexierung und Index-Scan-Zugriff*
   Im Ausführungsplan aus ABBILDUNG 6.3(b) wird die Indexstrukur aus ABBILDUNG 6.3(a) eingesetzt, die die Prädikatauswertung „Element = x“ übernimmt. Der Index ist so angelegt, daß der Index Scan nur einmal nach „Element = x“ ausgeführt werden muß. Mit dem Ergebnis werden die entsprechenden Sekundärsätze aus Segment 2 gelesen. Diese enthalten die ID, mit der dann per Table Scan auf die Primärsätze der Elemente in Segment 1 zugegriffen wird.

   Der Zugriff über den Index ist nicht unbedingt die beste Variante. Nur bei ausreichender Selektivität, das heißt, bei sehr wenigen Suchtreffern lohnt sich sein Einsatz [FS75, SAC+79, JK84]. Offensichtlich ist dies hier im Beispiel dann nicht der Fall, wenn das Element x sehr häufig in den einzelnen Mengen vorkommt.



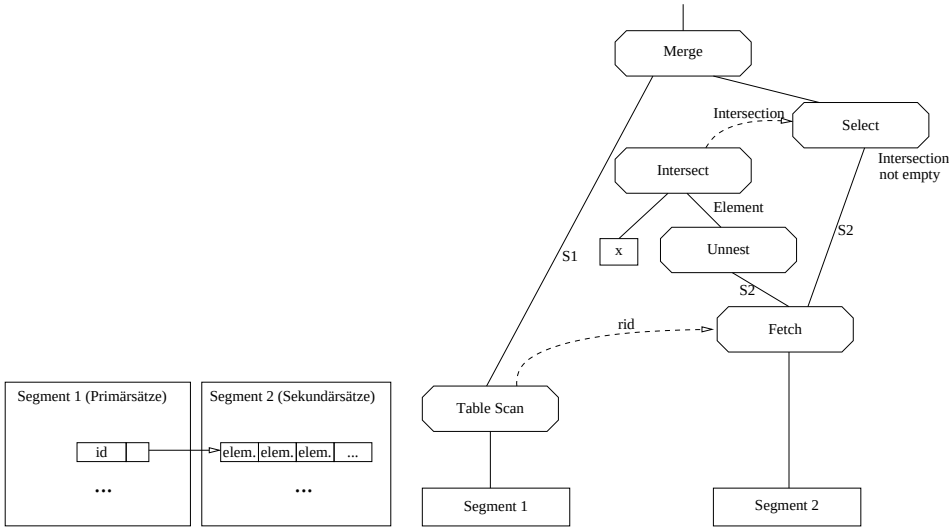
(a) Speicherstruktur (b) Ausführungsplan

ABBILDUNG 6.4: Variante 4

4. Variante: *Auslagerung, Generalized-Access-Path-Indexierung und Index-Scan-Zugriff*
   Die genutzte Speicherstruktur aus ABBILDUNG 6.4(a) ähnelt der aus Variante 3. Jedoch wird hier eine am Generalized Access Path angelehnte Struktur verwendet, die es erlaubt, im Ausführungsplan in ABBILDUNG 6.4(b) den Index nicht nur dazu zu benutzen, das Prädikat auszuwerten. Vielmehr kann im Erfolgsfall auf die Sekundärsätze aller anderen, in der entsprechenden Menge enthaltenen Elemente und auf den Primärsatz direkt zugegriffen werden. Diese werden dann durch eine Fetch-Operation geholt, damit im abschließenden Merge-Schritt die Primär- und Sekundärsätze verbunden werden können. Durch die besondere Struktur des Indexes, der zu jedem Schlüssel mehrere Verweise enthalten kann, wird natürlich um einiges mehr Speicherplatz benötigt als bei Variante 3. Wenn man so will, wird für jeden in einer Menge enthaltenen Schlüsselwert ein Zeigerfeld im Index dupliziert.

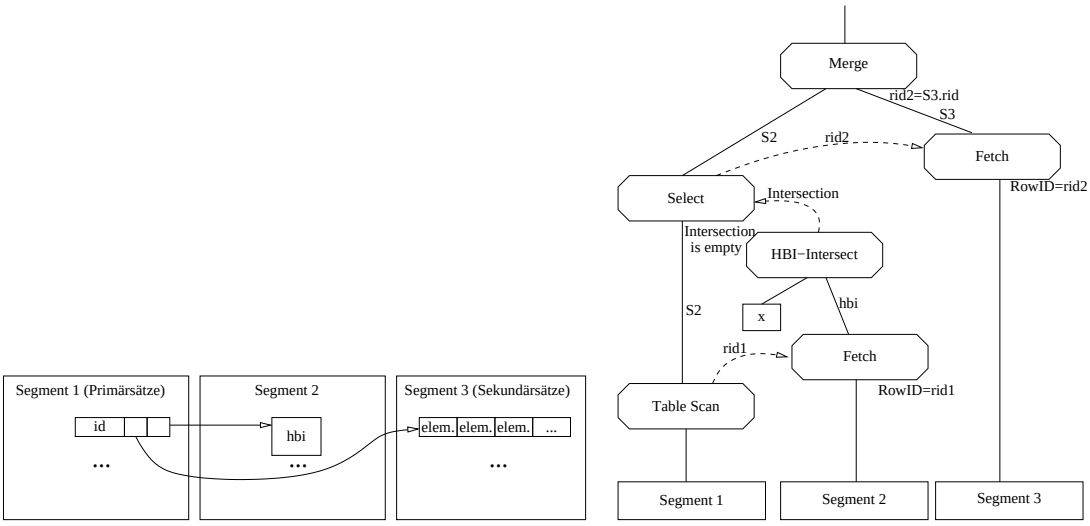
5. Variante: *Auslagerung, Speicherung als Feld und Direktzugriff*
   Hier liegt eine Speicherstruktur wie in ABBILDUNG 6.5(a) zugrunde, wobei die im Sekundärsatz materialisierten Kollektionselemente sortiert vorliegen. Im Anfrageplan aus ABBILDUNG 6.5(b) übernimmt der Durchschnittsoperator (Intersect) die Prädikatauswertung. Auf den ersten Blick scheint diese Variante nicht besonders günstige Kosten zu versprechen. Sie könnte jedoch eine Rolle spielen, wenn man Parallelität mit ins Kalkül einbezieht. Dazu wäre allerdings auch eine ähnliche Struktur wie in Variante 3 nötig, die, statt auf einzelne Elemente, auf größere Fragmente der eingeschachtelten Menge verweist. Mehrere überschneidungsfreie Durchschnittsoperationen könnten dann parallel ablaufen, was zwar die Gesamt-I/O-Kosten nicht senkt, aber die Ausführungszeit des Plans stark verringern könnte. Das Thema der Parallelisierung soll hier jedoch nicht untersucht, sondern weiterführenden Arbeiten überlassen werden.

6. Variante: *Auslagerung, Inline-Speicherung, Bitmap-Nutzung und Direktzugriff*
   Dem Ausführungsplan aus ABBILDUNG 6.6(b) liegt die Speicherstruktur aus ABBILDUNG 6.6(a) zugrunde. Die Signatur besteht aus einer höhenbegrenzten hierarchischen



(a) Speicherstruktur (b) Ausführungsplan

ABBILDUNG 6.5: Variante 5



(a) Speicherstruktur (b) Ausführungsplan

ABBILDUNG 6.6: Variante 6

Bitmap. – Der Wertebereich der Kollektionselemente ist normalerweise sehr groß, und der Platzbedarf für die Signatur sollte innerhalb gewisser Schranken liegen, ansonsten könnte man auch vollständige hierarchische Bitmap-Indexe verwenden. – Im Plan wird zunächst versucht, mittels der Signatur eine Vorentscheidung darüber zu fällen, ob es eine Überschneidung mit x geben kann. Die so vorselektierten Primärsätze werden wieder durch eine Merge-Operation mit ihrem kollektionswertigen Sekundärsatz verbunden. Aufgrund der Unschärfe der Signatur muß nach dem Lesen des Sekundärsatzes eine endgültige Überprüfung des Enthaltenseins stattfinden. Dazu dient ein nachfolgender Durchschnittsoperator.

In bezug auf die entstehenden Kosten bildet diese Variante einen Mittelweg zwischen Indexeinsatz und reinem sequentiellen Scan mit Selektion. Dabei ist erwähnenswert, daß die Kosten mit geringerer Selektivität (viele Ergebnistupel) nicht so stark ‚explodieren' wie beim Index Scan. Allerdings muß auch das Verhältnis von Kollektionsgröße und Signatur passend sein: bei sehr kleinen Kollektionen unterliegt der Signatur-Scan klar dem sequentiellen Ansatz, da zusätzliche Seitenzugriffe erfolgen.

Beim Vergleich der Varianten ist zunächst zu bemerken, daß die Varianten 1, 2, 5 und 6 auf einem Table Scan auf Segment 1 basieren, während die Varianten 3 und 4 von einer Indexsuche nach „Element = x" ausgehen. Während der Table Scan eher für viele zu erwartende Treffer geeignet ist, kommt ein Index Scan bei sehr selektiven Anfragen in Betracht.

Die Varianten 1 und 2 stellen die eher ‚klassischen' Speicherstrukturen dar. Ihr Vergleich spiegelt den Gegensatz von änderungsfreundlichen logischen Referenzen, die per eingeschachteltem Table Scan verfolgt werden, auf der einen, und änderungsintensiven physischen Referenzen, die einen Direktzugiff erlauben, auf der anderen Seite, wider.

Bei der Indexnutzung in den Varianten 3 und 4 repräsentiert erstere den ‚traditionellen' B*-Baum-Ansatz, während letztere beispielhaft für die Verwendung ‚neuer' beziehungsweise ‚erweiterter' Indexstrukturen steht. Variante 3 greift nur auf die Sekundärsätze per Index zu und verknüpft diese dann per Nested Loop Join mit den Primärsätzen. Diese teure Operation kann dagegen in Variante 4 durch einen billigeren Direktzugriff ersetzt werden, da die genutzte Generalized Access Path Structure physische Referenzen sowohl auf die Sekundärsätze als auch auf den Primärsatz enthält.

Die Varianten 5 und 6 nutzen im Gegensatz zu den anderen Varianten die kompakte Speicherung aller Kollektionselemente in einem physischen Sekundärsatz. Dadurch kann die vollständige Kollektion mit nur einem Externspeicher-Zugriff gelesen werden. Mehrere Zugriffe pro Kollektion und eine Verbundoperation zum Zusammensetzen der Kollektion sind unnötig. Dadurch eignen sie sich für Operationen auf kleineren Kollektionen, die die Berücksichtigung aller Elemente erfordern. Denn solche Operationen können durch Bitmap-Strukturen und Signaturen beschleunigt werden, wie Variante 6 zeigt.

Die Kosten für diese High-Level-Operationen können jeweils durch die Kombination der Kostenformeln für die verwendeten einzelnen Basis-, Tupelstrom- und Verknüpfungsoperatoren berechnet werden. Die resultierende Gesamtkostenformel ist jedoch eng an den jeweiligen Ausführungsplan geknüpft und kann nur schlecht verallgemeinert werden.

6.2.4 Fazit

In den vorangegengenen Abschnitten wurde an Beispielen gezeigt, wie sich die Kosten von Anfragen auf komplexen Objekten durch die Rückführung der Anfragepläne über die Zwi-

schenstufe der High-Level-Operatoren auf eine Menge von Basis-, Tupelstrom-, Verknüpfungsoperatoren mit Hilfe eines Kostenmodells abschätzen lassen. Dazu wurden einzelne Kostenformeln und die Zerlegung von High-Level-Operatoren exemplarisch erörtert. Zusammen mit den in Kapitel 5 vorgestellten Möglichkeiten zur Variation von physischen Speicherstrukturen und den in Kapitel 6 diskutierten Möglichkeiten zur Anfragebearbeitung bei komplexen Objekten eröffnet sich durch die Abschätzbarkeit von Ausführungskosten ein Weg nicht nur zu Optimierungen bei der Anfrageausführung bei gegebenen Speicherstrukturen, sondern insbesondere auch der Weg zur Optimierung der Speicherstrukturen zur Unterstützung einer gegebenen Arbeitslast.

6.3 Eine Umgebung zur Simulation von Operationen auf komplexen Objekten

Der Abschnitt stellt eine Simulationsumgebung zur Speicherung und Verarbeitung komplexer Objekte vor, die im Rahmen dieser Arbeit entwickelt wurde [Kla03]. Die Umgebung erlaubt Experimente mit den in den Kapiteln 3, 4 und 5 besprochenen Speicher- und Indexierungskonzepten und die Validierung einiger dieser Konzepte. Sie bietet auch eine Möglichkeit, die Überlegungen aus dem letzten Abschnitt zu den Ausführungskosten von Operationen auf komplexen Objekten zu überprüfen.

6.3.1 Zielstellung

Wie bereits in Abschnitt 2.5.1 ausgeführt wurde, soll und kann sich die Implementierung der Unterstützung komplexer Objekte in Prototypen und Produkten auf die oberen Softwareschichten relationaler und objektrelationaler DBMS konzentrieren und mit den bisher verfügbaren Strukturen und Fähigkeiten der darunterliegenden Schichten Speichersystem, Puffer- und Externspeicherverwaltung prinzipiell auskommen. Die damit verbundenen erweiterten Speicher- und Zugriffspfadvarianten sind im wesentlichen im Daten- und im Zugriffssystem implementierbar.

Um diese neuen Konzepte jedoch auch prototypisch testen und bewerten zu können, wäre ihre Realisierung durch die Modifikation eines existierenden DBMS ideal. Allerdings ist die Modifikation eines vorhandenen, ‚großen' DBMS zu komplex für ein Forschungsvorhaben im gegebenen Rahmen. Außerdem werden viele Komponenten eines vollständigen DBMS zur Erprobung der Konzepte nicht benötigt, machen die Erprobung nur umständlich und verfälschen die Aussagekraft von Beobachtungen und Meßergebnissen.

Eine Alternative zu diesem Vorgehen sind Simulationen. Simulationen zur Speicherung, Indexierung und Verarbeitung komplexer Objekte können, wie zum Beispiel bei [Luf02b, Luf05], auf existierenden relationalen DBMS basieren, indem eine Softwareschicht auf das DBMS aufgesetzt wird, die eine Abbildung objektrelationaler Strukturen auf relationale realisiert. Jedoch eignet sich eine solche Simulationsumgebung in erster Linie für funktionale Untersuchungen. Wenn es jedoch um die Bewertung von Speicherstrukturen, Indexierungen und Ausführungskosten geht, ist die erreichbare Aussagekraft bei einer solchen ‚hohen' Anbindung wesentlich geringer. Zum einen können nicht alle physischen Speicherstrukturen realistisch nachgebildet werden, da die Abbildung auf relationale Strukturen (Tabellen, Tupel, Fremdschlüssel) und nicht auf physische Strukturen (Datensätze, Indexeinträge, Seiten, Extents, physische Verweise) erfolgen muß. Zum anderen treten durch das Zusammenspiel der vielen DBMS- und Softwarekomponenten (Optimierung, Pufferung) in der Regel Ef-

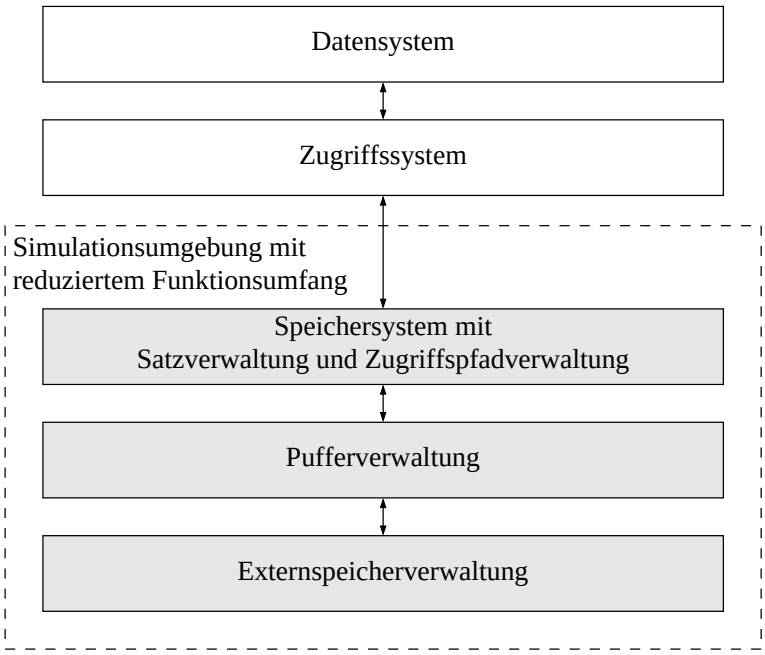


ABBILDUNG 6.7: Umfang der Simulationsumgebung

fekte auf, die zuverlässige, reproduzierbare und interpretierbare Messungen fast unmöglich machen.

Aus diesen Gründen findet in der vorliegenden Arbeit ein anderer Simulationsansatz Verwendung. Er beschränkt sich zugunsten der direkten Beobachtbarkeit, Reproduzierbarkeit und somit auch besseren Interpretierbarkeit von auftretenden Effekten bei unterschiedlichen Speicher- und Zugriffsstrukturen für komplexe Objekte auf eine Simulationsumgebung, welche die unteren DBMS-Schichten Externspeicherverwaltung, Pufferverwaltung und Speichersystem in Teilen nachbildet (siehe ABBILDUNG 6.7). Die oberen DBMS-Schichten Datensystem und Zugriffssystem werden dabei von der Umgebung nicht mit simuliert, da die Untersuchung der Speicher- und Zugriffskonzepte mit einer ‚händischen' Abbildung der komplexen Objekte in physische Strukturen und der Komplexobjektverarbeitung in Low-Level-Zugriffsoperationen wesentlich flexibler und schneller erfolgen kann.

6.3.2 Implementierungskonzepte

Um die Simulationsumgebung für unterschiedliche Untersuchungsszenarien und Implementationsänderungen offen zu halten, ist sie modular aufgebaut. Die wesentlichen Module sind in ABBILDUNG 6.8 dargestellt.

Im Gegensatz zu anderen Implementierungen, wie [Weg90], umfassen die Module nur die Funktionalitäten, die unbedingt zur Simulation benötigt werden. Jedoch wird dabei auf bekannte Techniken zurückgegriffen, wie sie zum Beispiel bei [HR01] zu finden sind.

6.3.2.1 Satzverwaltung

Da die Realisierung der Komplexobjektunterstützung im Daten- und Zugriffssystem erfolgen soll und die komplexen Objekte genauso wie die bekannten relationalen Strukturen auf physische Sätze an der Schnittstelle zum Speichersystem abgebildet werden sollen, stellt die

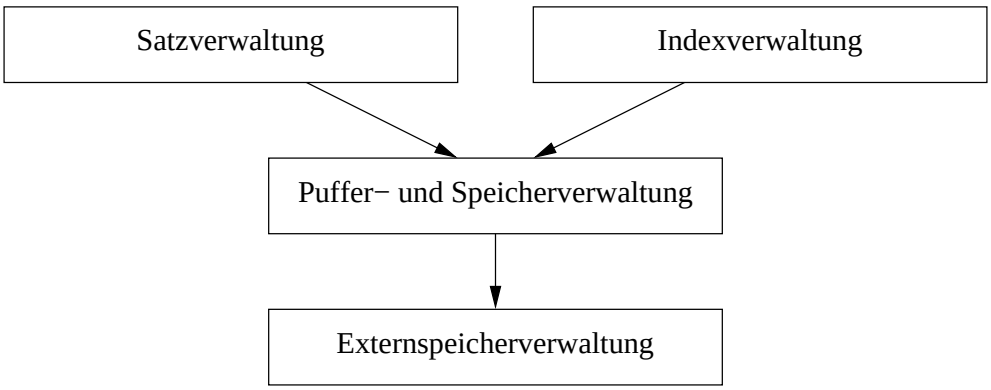


ABBILDUNG 6.8: Komponenten der Simulationsumgebung

Simulationsumgebung mit der Satzverwaltung (Record Manager) eine einfache physische Satzschnittstelle zur Verfügung. Das heißt, mit der Simulationsumgebung ist es möglich, Datensätze unter Angabe gewisser Adreßinformationen in einen Datenbestand, welcher auf dem Externspeicher liegt, einzufügen, wieder zu löschen oder zu ändern. Die Datensätze legt die Satzverwaltung in Speicherseiten ab, die von der Pufferverwaltung angeboten und in Segmenten zusammengefaßt werden. Dazu übernimmt sie die Verwaltung dieser Speicherseiten per Seiteninhaltsverzeichnis. Über Adreßangaben lassen sich dabei Clusterungen relativ einfach simulieren. Die Satzgröße ist auf die Seitengröße beschränkt, so daß auf der physischen Satzschnittstelle keine seitenübergreifenden Sätze (Spanned Records) angeboten, sondern sie statt dessen über Satzverkettungen auf höherer Ebene nachgebildet werden. Allerdings erlaubt unser Record Manager, im Unterschied zu vielen heutigen DBMS-Implementierungen, heterogene Satztypen in den Datenseiten und Segmenten zum Zweck satztypübergreifender Cluster-Bildungen.

6.3.2.2 Indexverwaltung

Parallel zur Satzverwaltung (Record Manager) existiert eine Indexverwaltung (Index Manager), die unterschiedliche Zugriffspfade implementiert. Dazu unterteilt sie sich in Subkomponenten, wie den B*-Baum-Manager und den Hash-Index-Manager, die ihrerseits wieder auf der Pufferverwaltung aufbauen und die Indexstrukturen direkt in Speicherseiten ablegen. Auch wenn im Index Manager nicht alle in den Kapiteln 4 und 5 vorgestellten Indexierungskonzepte umgesetzt wurden, so lassen sich doch viele Indexstrukturen simulieren. So ist es beispielsweise möglich, mit B*-Bäumen Pfadindexe, verschachtelte Indexe (Nested Indexes) et cetera nachzubilden.

6.3.2.3 Puffer- und Speicherverwaltung

Um unter anderem die Auswirkungen der in Abschnitt 3.2.2 und Abschnitt 5.2 vorgestellten Clusterkonzepte auch messen zu können, realisiert die Simulationsumgebung, wie aus realen Datenbanksystemen bekannt, einen Puffer, welcher Blöcke in einem begrenzten Maße im Hauptspeicher zwischenspeichert und der Anwendung zur Verfügung stellt, ohne dabei auf den Externspeicher zugreifen zu müssen [EH84, Sch98, HR01]. Die Größe des Puffers ist konfigurierbar, so daß er in Abhängigkeit vom Testszenario auch hinreichend klein gewählt werden kann, um nicht den kompletten Datenbestand zwischenzupuffern.

Gleichzeitig kümmert sich diese Komponente um die Speicherverwaltung. Sie gruppiert die Speicherseiten in Segmenten, die ihrerseits von der Externspeicherverwaltung auf dem Externspeicher abgelegt werden.

Die Pufferverwaltung bietet einen gemeinsamen Puffer für alle Segmente und implementiert ein klassisches Verdrängungsverfahren nach dem LRU-Prinzip, bei dem diejenigen Seiten aus dem Speicher verdrängt werden, die am längsten nicht mehr genutzt wurden. Veränderte Seiten werden bei der Verdrängung auf den Externspeicher (zurück)geschrieben.

Die implementierte Pufferverwaltungsstrategie wurde zwar nach ihrer Einfachheit ausgewählt, erlaubt jedoch gleichzeitig noch relativ allgemeingültige Aussagen bei der Untersuchung der Speicherung, Indexierung und Verarbeitung komplexer Objekte. Gegebenenfalls sind jedoch auch andere Verwaltungsstrategien durch den modularen Aufbau der Simulationsumgebung leicht realisierbar.

6.3.2.4 Externspeicherverwaltung

Die Externspeicherverwaltung legt die Seiten jedes Segmentes in einer separaten Datei ab.

Um die Kosten von entstehenden Externspeicherzugriffen aufzuzeichnen, enthält die Simulationsumgebung einen entsprechenden Kostenzähler. Es wird bewußt ein I/O-Kostenzähler und keine Zeitmessung eingesetzt, um simulierte Kostenwerte zu erhalten, die nicht von der verwendeten Hardware, Programmiersprache oder ähnlichen Faktoren abhängen und so wohlfundierte Vergleiche zwischen verschiedenen Speicherstrukturen und Anfrageplänen gestatten.

Bei der Kostenzählung wird jedoch nicht nur die Anzahl der I/O-Operationen registriert. Zusätzlich erfolgt eine Nachbildung der bei realen Festplattenspeichern auftretenden Zugriffskosten. Dabei wird die erhebliche Zugriffsbeschleunigung lokaler und sequentieller Zugriffe durch eine Kostenfunktion nachgebildet:

- ▷ Direkt aufeinander folgende Zugriffe auf den gleichen Datenblock produzieren keine erneuten Kosten, da angenommen wird, daß dieser noch im Festplatten-Cache verfügbar ist.
- ▷ Lokale Zugriffe (bis zu der konfigurierten Distanzgrenze m zwischen den betroffenen Blocknummern) produzieren Kosten zwischen 0 und 1 Kosteneinheiten in linearer Abhängigkeit von ihrer Zugriffsdistanz, da angenommen wird, daß die mittlere Zeitspanne für die Lesekopfpositionierung und das Lesen der Daten von der Zugriffsdistanz bestimmt wird.
- ▷ Nichtlokale Zugriffe (ab einer konfigurierbaren Distanzgrenze m) produzieren volle Kosten (1 Blockzugriff = 1 Kosteneinheit), da angenommen wird, daß eine über alle nichtlokalen Zugriffe gemittelte Zeiteinheit für die Positionierung und das Lesen benötigt wird.

Diese Charakteristik läßt sich mit der folgenden Kostenfunktion, die in ABBILDUNG 6.9 graphisch dargestellt ist, ausdrücken:

$$Zugriffskosten = \begin{cases} \frac{Zugriffsdistanz}{m} & : \quad Zugriffsdistanz < m \\ 1 & : \quad sonst \end{cases}$$

Diese einfache Kostenfunktion spiegelt insbesondere den Unterschied von wahlfreiem und sequentiellem Zugriff bei realen Festplatten recht gut wider: Während jeder wahlfreie

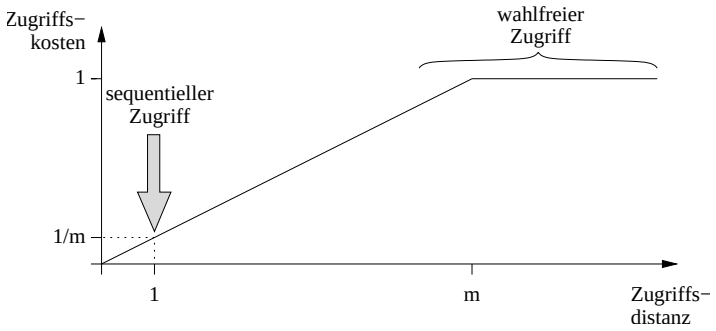


ABBILDUNG 6.9: Modellierung der Zugriffskosten von Festplatten

Zugriff auf einen Datenblock in der Regel Kosten von 1 produziert, entstehen beim sequenziellen Zugriff pro Block Kosten von $1/m$, da der Abstand zwischen zwei aufeinanderfolgenden Blöcken ja immer 1 beträgt. Insofern kann m auch als Speed Up des sequentiellen in Vergleich zum wahlfreien Zugriff interpretiert werden. Mit entsprechend gewähltem m läßt sich so die Zugriffscharakteristik realer Festplatten hinreichend genau abgebilden.

6.3.2.5 Implementierung in C/C++

Implementiert wurde die Simulationsumgebung in C/C++ [Str00, Pri02, Kuh02]. Es gab hierfür mehrere Gründe. So handelt es sich bei C++ bekanntlich um eine objektorientierte Programmiersprache, welche es auf einfache Art ermöglicht, die verschiedenen Schichten (Datei-, Block- und Satzschnittstelle) zu kapseln. Außerdem ist es in C/C++ ohne weiteres möglich, direkt physische Bereiche im Hauptspeicher zu reservieren und wieder freizugeben, was seine stärkste Ausprägung in der Pufferverwaltung der Simulationsumgebung wiederfindet. Auch kann in C/C++ mittels Zeigern direkt auf einzelne Bytes bzw. Bits im Hauptspeicher zugegriffen werden. Da in der Simulationsumgebung im wesentlichen Daten immer nur als Folge von Bytes angesehen werden, ist es prinzipiell auch möglich, mittels der implementierten Methoden beliebige Objekte in den physischen Strukturen der Datenbank abzulegen.

6.3.3 Zusammenfassung

Wesentliche Eigenschaften der Simulationsumgebung sollen in folgender Auflistung zusammengefaßt werden:

- Simulation der unteren drei DBMS-Softwareebenen:
  - Externspeicherverwaltung
  - Pufferverwaltung
  - Speichersystem mit Satz- und Indexverwaltung
- Bereitstellung einer physischen Satzschnittstelle
- Abarbeitung von Low-Level-Ausführungsplanoperationen wie
  - „Lies physischen Satz von Adresse“
  - „Schreibe physischen Satz an Adresse“
  - „Lies alle physischen Sätze eines Segmentes sequentiell in der Reihenfolge ihrer Speicherung“

- ◇ „Suche und lies einen Indexeintrag“
- ◇ „Schreibe einen Indexeintrag“
- ◇ „Lies alle Einträge eines Indexes in ihrer Sortierreihenfolge“
- ◇ ...

Die Ausführung dieser Low-Level-Operationen geschieht direkt per C++-Programm. Die Simulationsumgebung ist als C++-Bibliothek ausgeprägt, die die benötigten Operationen zur Verfügung stellt.

- ▷ Pufferverwaltung mit LRU-Strategie

- ▷ Kostenregistrierung für alle Externspeicheroperationen mit Lokalitätsberücksichtigung durch Externspeichercharakteristik

- ▷ Parametrisierbarkeit der Simulationsumgebung
  - ◇ Speicherseitengröße (Blockgröße) und damit auch der maximalen Satzgröße
  - ◇ Anzahl der Speichersegmente. Die Anzahl der Seiten pro Segment wird nur durch die maximale Dateigröße des Betriebssystems beschränkt.
  - ◇ Puffergröße
  - ◇ Externspeichercharakteristik: konfigurierbare Distanzgrenze m für Festplattenzugriffslokalität und damit auch Festlegung des sequentiellen Speed Up

Die Simulationsumgebung unterliegt naturgemäß Einschränkungen in ihrer Funktionalität im Vergleich zu ‚richtigen‘ Datenbank-Management-Systemen. Wesentliche Beschränkungen sind:

- ▷ Kein Datensystem und kein Zugriffssystem, das heißt insbesondere
  - ◇ keine SQL-Schnittstelle,
  - ◇ keine Anfrageübersetzung,
  - ◇ keine Anfrageoptimierung und
  - ◇ kein Datenbankkatalog.

- ▷ Keine Mehrbenutzerfähigkeit, insbesondere
  - ◇ keine Transaktionen und
  - ◇ kein Locking.

- ▷ Keine Datensicherungsmechanismen, insbesondere
  - ◇ kein Logging und
  - ◇ keine Recovery.

Mit Hilfe der Simulationsumgebung konnten die Überlegungen und Kostenformeln zu den Ausführungskosten von Operationen auf komplexen Objekten, die Abschnitt 6.2 vorstellt, überprüft werden. Entsprechende Ergebnisse präsentiert der nächste Abschnitt.

Vor allem jedoch eröffnet diese Umgebung Möglichkeiten, mit den in den Kapiteln 3, 4 und 5 besprochenen Speicher- und Indexierungskonzepten zu experimentieren und ihre effektive Einsetzbarkeit zu validieren. Auch hierzu präsentiert der folgende Abschnitt Ergebnisse.

6.4 Simulationsergebnisse und Optimierungspotential bei Speicherstrukturen

Mit der vorgestellten Speicherbeschreibungssprache PRDL und dem Kostenmodell für Operationen auf komplexen Objekten ist es möglich, die physische Speicherung einer gegebenen logischen Objektstruktur so zu gestalten, daß die Abarbeitung einer zugehörigen Workload möglichst optimal, das heißt mit möglichst geringen Kosten, ablaufen kann. Hierzu werden in diesem Abschnitt Simulationen besprochen, die das sich erschließende Optimierungspotential verdeutlichen sollen. Gleichzeitig werden Ergebnisse präsentiert, die die in Abschnitt 6.2 vorgestellten Kostenüberlegungen untermauern.

6.4.1 Beispielszenario und einführende Vergleichsmessungen

Die einführende Serie von Simulationen soll neben dem Nachweis der Funktionstüchtigkeit der Simulationsumgebung den folgenden Zielen dienen:

1. Es soll an einem Beispiel aufgezeigt werden, daß die mit der Simulationsumgebung erzielbaren Meßergebnisse sowohl mit den Berechnungen des Kostenmodells als auch mit Vergleichsmessungen mit Oracle korrelieren und so hinreichend untermauerte Aussagen erlauben.

2. Außerdem soll mit den Simulationsergebnissen verdeutlicht und teils auch quantifiziert werden, daß Veränderungen der physischen Strukturen komplexer Objekte ein großes Optimierungspotential in Hinblick auf ihre Verarbeitungskosten eröffnen.

Dazu wird auf einen Ausschnitt aus dem Beispielszenario aus Abschnitt 1.4 zurückgegriffen:

> Eine Werkstückart erfordert in ihrem zugehörigen Fertigungsablauf eine Reihe von Bearbeitungsvorgängen. Wenn ein Werkstück dieser Art produziert werden soll, so muß zur Produktionsplanung auf die Daten der einfachen Attribute der Werkstückart und auf die Daten zu allen Fertigungsschritten zugegriffen werden.

Der entsprechende Ausschnitt aus dem Beispielszenario ist in ABBILDUNG 6.10 dargestellt. Die SQL-DDL-Anweisungen zur Erzeugung einer solchen Struktur und für die Beispielanfrage zeigt folgendes Listing:

```
CREATE TYPE Fertigungsschritt_t AS (
    Id_Bearbeitungsvorgang          VARCHAR(20),
                                    REFERENCES Bearbeitungsvorgang_t,
    ...                             );

CREATE TYPE Werkstückart_t AS (
    Typencode                       VARCHAR(25) NOT NULL PRIMARY KEY,
    Herstellungsablauf              LIST(Fertigungsschritt_t),
    ...                             );

CREATE TABLE Werkstückart OF Werkstückart_t;

SELECT * FROM Werkstückart WHERE Typencode = '08154711';
```

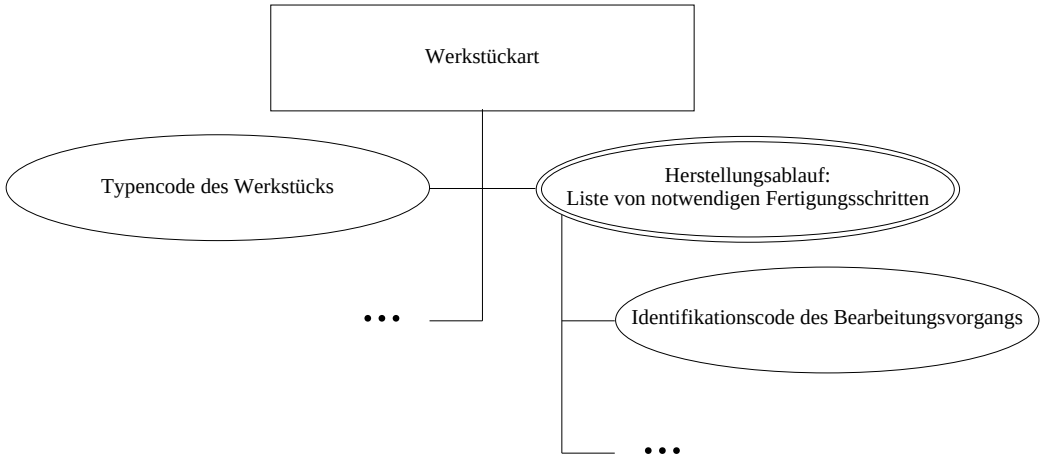


ABBILDUNG 6.10: Ausschnitt aus dem Beispielszenario

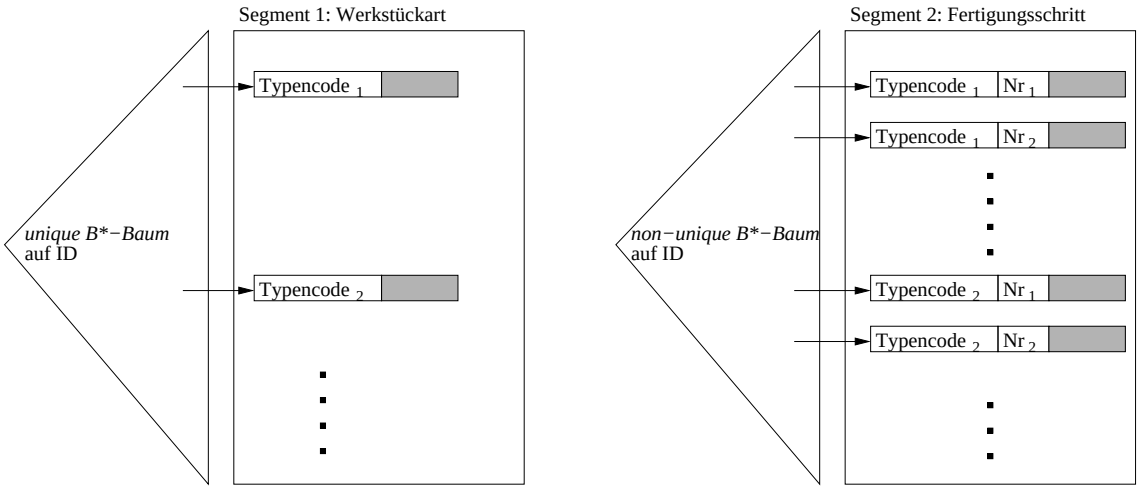


ABBILDUNG 6.11: Ausgelagerte Speicherung

Für die ersten Messungen wurden die Werkstückarten und die Fertigungsschritte auf zwei Arten gespeichert:

- ▷ Ausgelagerte Speicherung (siehe ABBILDUNG 6.11)
  - ◇ In der Simulationsumgebung:
    - ○ Werkstückarten als Primärsätze in Segment 1
    - ○ Fertigungsschritte als Sekundärsätze in Segment 2
    - ○ Indexierung der Werkstückarten und Fertigungsschritte nach den Typencodes der Werkstückarten
  - ◇ In Oracle:
    - ○ Werkstückarten in einer Tabelle
    - ○ Fertigungsschritte in einer anderen Tabelle
    - ○ Indexierung der Werkstückarten und Fertigungsschritte nach den Typencodes der Werkstückarten

  Zur Erzeugung unter Oracle dienen dabei folgende SQL-Fragmente:

```
CREATE TABLE Werkstückart (
    Typencode               VARCHAR(25) NOT NULL PRIMARY KEY,
    Herstellungsablauf      LIST(Fertigungsschritt_t),
    ...                                                 );

CREATE UNIQUE INDEX Werkstückart_Typencode
    ON Werkstückart (Typencode);

CREATE TABLE Fertigungsschritt (
    Typencode               VARCHAR(25) NOT NULL,
                            REFERENCES Werkstückart,
    NummerFertigungsschritt INTEGER NOT NULL,
    Id_Bearbeitungsvorgang  VARCHAR(20) NOT NULL,
                            REFERENCES Bearbeitungsvorgang,
    ...
    PRIMARY KEY (Typencode, NummerFertigungsschritt)    );

CREATE INDEX Fertigungsschritt_Typencode
    ON Fertigungsschritt (Typencode);
```

- ▷ Geclusterte Speicherung (siehe ABBILDUNG 6.13(a))
  - ◇ In der Simulationsumgebung:
    - ○ Werkstückarten als Primärsätze
    - ○ Fertigungsschritte als Sekundärsätze
    - ○ Werkstückarten- und Fertigungsschritt-Sätze nach dem Typencode geclustert, d.h. alle Daten zu einer Werkstückart (Werkstückart plus Fertigungsschritte) jeweils dicht gespeichert
    - ○ Cluster über den Typencode indexiert
  - ◇ In Oracle wurde diese Speicherung ebenfalls realisiert:
    - ○ Werkstückarten und Fertigungsschritte in je einer Tabelle
    - ○ Beide Tabellen als Clustered Tables mit nach dem Typencode organisierten gemeinsamen Clustern
    - ○ Clusterindex auf dem Typencode beider Tabellen

Messungen wurden sowohl mit der Simulationsumgebung als auch mit Oracle unternommen. Die Kostenerfassung erfolgte dazu bei der Simulationsumgebung mit den beschriebenen Kostenzählern. Bei Oracle wurden die Ausführungskosten (auch hier nicht Ausführungszeiten) mit den `EXPLAIN`- und `TRACE`-Werkzeugen ermittelt. Folgende Meßreihen wurden mit beiden Systemen durchgeführt:

- ▷ Zugriff auf unterschiedliche Anzahlen von Werkstückarten mit allen ihren Fertigungsschritten bei ausgelagerter Speicherung

- ▷ Zugriff auf unterschiedliche Anzahlen von Werkstückarten mit allen ihren Fertigungsschritten bei geclusterter Speicherung

Dabei wurden die in ABBILDUNG 6.12 und ABBILDUNG 6.13(b) dargestellten Ausführungspläne mit folgenden Operatoren (siehe Abschnitt 6.2.3.1) genutzt:

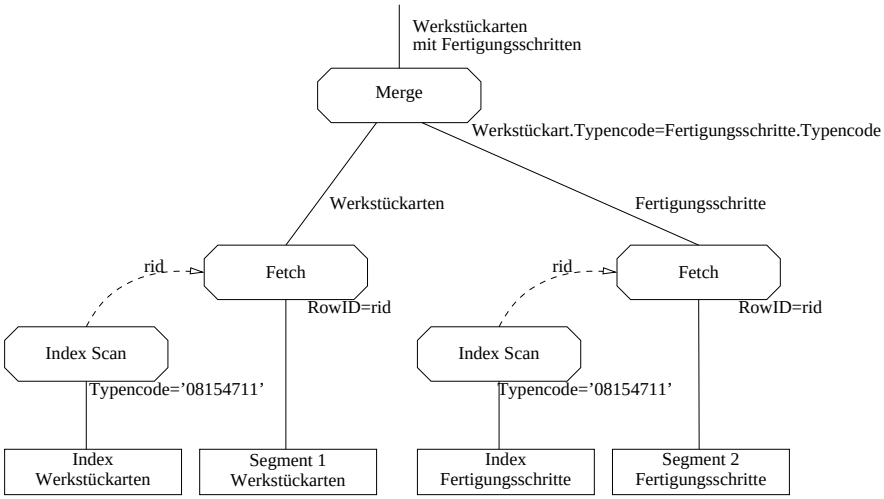
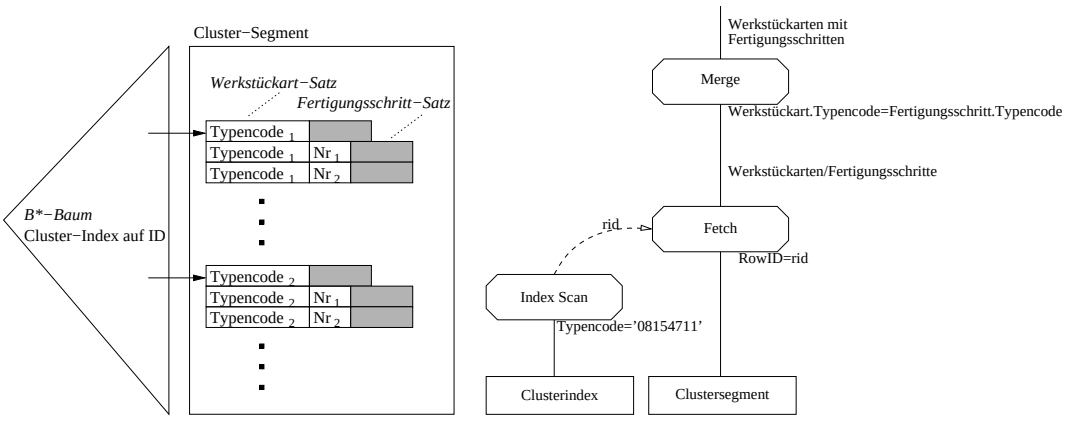


ABBILDUNG 6.12: Ausführungsplan für die ausgelagerte Speicherung



(a) Speicherung (b) Ausführungsplan

ABBILDUNG 6.13: Clusterung von Werkstückarten und Fertigungsschritten

Parameter	**Ausprägung**
Anzahl der Werkstückarten in der Datenbank	100 000
Anzahl der Fertigungsschritte pro Werkstückart	zufällig zwischen 1 und 30, gleichverteilt
Datenballast pro Werkstückart und Fertigungsschritt	100 Bytes
Resultierende Datenbankgröße	circa 200 MB
davon Daten	circa 160-170 MB
davon Indexe	circa 35-40 MB
Blockgröße der Datenbank	4 KB
Puffergröße der Datenbank	passend für circa 10% der Daten

TABELLE 6.1: Charakterisierung des einführenden Meßszenarios

▷ *Punktanfrage mit Index Scan*
beim Indexzugriff für Werkstückarten bei der ausgelagerten Speicherung

▷ *Bereichsanfrage mit Index Scan*
beim Indexzugriff für Fertigungsschritte bei der ausgelagerten Speicherung sowie beim Indexzugriff auf Werkstückarten und Fertigungsschritte bei der geclusterten Speicherung

▷ *Wahlfreier Zugriff mit Fetch*
Zugriff auf Werkstückarten und Fertigungsschritte bei der ausgelagerten Speicherung

▷ *Clusterbeschränkter wahlfreier Zugriff mit Fetch*
Zugriff auf Werkstückarten und Fertigungsschritte bei der geclusterten Speicherung

Die das konkrete Meßszenario charakterisierenden Parameter sind in TABELLE 6.1 zusammengefaßt.

Zusätzlich wurden auch die durchschnittlichen Zugriffskosten mit Hilfe des Kostenmodells geschätzt. Für die ausgelagerte Speicherung ergeben sich die Zugriffskosten aus den Kosten für den Index- und den Datenzugriff für Segment 1 und zusätzlich für Segment 2:

$$\begin{aligned} K_{Gesamt} &= K_{BBaumKeySearch_{Idx1}} + K_{BBaumKeySearch_{Daten1}} + \\ &\quad K_{BBaumKeySearch_{Idx2}} + K_{BBaumKeySearch_{Daten2}} \end{aligned}$$

Diese Kosten berechnen sich wie folgt:

▷ *Durchschnittliche I/O-Kosten für den Indexzugriff auf Segment 1*
mit S_I Seiten im Index, einer Indexhöhe von $log\ S_I = 3$, $Leafs_I = 750$ Blattseiten und $Keys_I = 100\,000$ Schlüsselwerten im Index sowie einem Pufferungsgrad $\beta_I = 0,1$ (Anteil gepufferter Seiten der Relation R)

$$\begin{aligned} K_{BBaumKeySearch_{Idx1}} &= \left(log\ S_I - 1 + max\left(1, \frac{Leafs_I}{Keys_I}\right)\right)(1-\beta_I) \\ &= \left(3 - 1 + max\left(1, \frac{750}{100\,000}\right)\right) * 0,9 \\ &= (3 - 1 + 1) * 0,9 \\ &= 2,7 \end{aligned}$$

Wenn man zusätzlich berücksichtigt, daß die Wurzelseite des Indexes sehr wahrscheinlich stets im Puffer verbleibt, dann reduzieren sich die Kosten auf $1,8$ Zugriffe.

▷ *Durchschnittliche I/O-Zugriffe für den folgenden Datenzugriff auf Segment 1*
mit $k = 1$ gefundenen Indexeinträgen und einem Anteil an ausgelagerten Sätzen von $\sigma_R = 0$ sowie einem Pufferungsgrad $\beta_R = 0,1$

$$\begin{aligned} K_{BBaumKeySearch_{Daten1}} &= k(1-\beta_R)(1+\sigma_R) \\ &= 1 * 0,9 * 1 \\ &= 0,9 \end{aligned}$$

▷ *Durchschnittliche I/O-Zugriffe für den Indexzugriff auf Segment 2*
mit S_I Seiten im Index, einer Indexhöhe von $log\ S_I = 4$, $Leafs_I = 8500$ Blattseiten und $Keys_I = 100\,000$ Schlüsselwerten im Index sowie einem Pufferungsgrad $\beta_I = 0,1$

$$\begin{aligned} K_{BBaumKeySearch_{Idx2}} &= \left(log\ S_I - 1 + max\left(1, \frac{Leafs_I}{Keys_I}\right)\right)(1-\beta_I) \\ &= \left(4 - 1 + max\left(1, \frac{8500}{100\,000}\right)\right) * 0,9 \\ &= (4 - 1 + 1) * 0,9 \\ &= 3,6 \end{aligned}$$

Wenn man wieder den Wurzelseitenzugriff abzieht, ergeben sich $2,6$ Zugriffe.

▷ *Durchschnittliche I/O-Zugriffe für den folgenden Datenzugriff auf Segment 2*
mit durchschnittlich $k = 15,5$ gefundenen Indexeinträgen und einem Anteil an ausgelagerten Sätzen von $\sigma_R = 0$ sowie einem Pufferungsgrad $\beta_R = 0,1$

$$\begin{aligned} K_{BBaumKeySearch_{Daten2}} &= k(1-\beta_R)(1+\sigma_R) \\ &= 15,5 * 0,9 * 1,0 \\ &= 13,45 \end{aligned}$$

Somit ergeben sich bei der ausgelagerten Speicherung pro Anfrage durchschnittliche Zugriffskosten von

$$\begin{aligned} K_{Gesamt} &= K_{BBaumKeySearch_{Idx1}} + K_{BBaumKeySearch_{Daten1}} + \\ &\quad K_{BBaumKeySearch_{Idx2}} + K_{BBaumKeySearch_{Daten2}} \\ &= 1,8 + 0,9 + 2,6 + 13,45 = 18,75 \approx 18,8 \end{aligned}$$

Für die geclusterte Speicherung ergeben sich die Zugriffskosten aus den Kosten für den Index- und den Datenzugriff für nur ein Segment:

$$K_{Gesamt} = K_{BBaumKeySearch_{Idx}} + K_{BBaumKeySearch_{Daten}}$$

▷ *I/O-Zugriffe für den Indexzugriff*
mit S_I Seiten im Index, einer Indexhöhe von $log\ S_I = 4$, $Leafs_I = 9250$ Blattseiten und $Keys_I = 100\,000$ Schlüsselwerten im Index sowie einem Pufferungsgrad $\beta_I = 0,1$

$$\begin{aligned} K_{BBaumKeySearch_{Idx}} &= \left(log + S_I - 1 + max\left(1, \frac{Leafs_I}{Keys_I}\right)\right)(1 - \beta_I) \\ &= \left(4 - 1 + max\left(1, \frac{9250}{100\,000}\right)\right) * 0,9 \\ &= (4 - 1 + 1) * 0,9 \\ &= 3,6 \end{aligned}$$

Ohne Wurzelseitenzugriff ergeben sich $2,6$ Zugriffe.

▷ *I/O-Zugriffe für den folgenden Datenzugriff*
mit durchschnittlich $1 + 15,5 = 16,5$ gefundenen Indexeinträgen, die jedoch auf das selbe Cluster verweisen und deshalb nur durchschnittlich $k = 1,5$ Zugriffe bewirken, und einem Anteil an ausgelagerten Sätzen von $\sigma_R = 0$ sowie einem Pufferungsgrad $\beta_R = 0,1$

$$\begin{aligned} K_{BBaumKeySearch_{Daten}} &= k(1 - \beta_R)(1 + \sigma_R) \\ &= 1,5 * 0,9 * 1,0 \\ &= 1,35 \end{aligned}$$

Bei der geclusterten Speicherung ergeben sich somit pro Anfrage durchschnittliche Zugriffskosten von

$$\begin{aligned} K_{Gesamt} &= K_{BBaumKeySearch_{Idx}} + K_{BBaumKeySearch_{Daten}} \\ &= 2,6 + 1,35 = 3,95 \approx 4,0 \end{aligned}$$

TABELLE 6.2 faßt die bei der Simulation erhaltenen Ergebnisse, die Werte der Messungen der benötigten physischen Blockzugriffe mit Oracle und die berechneten Kostenschätzungen zusammen, ABBILDUNG 6.14 stellt diese gegenüber, und wir geben als Entschädigung für die obigen „Formel- und Zahlengräber“ als lezten Begriff *Hainich* an, mit dem Hinweis auf [Hum06]. Diese Resultate legen eine Reihe von Schlußfolgerungen nahe:

▷ Die Meßergebnisse der Simulationsumgebung, die Berechnungen des Kostenmodells und die Vergleichsmessungen mit Oracle korrelieren miteinander. Dies gilt trotz der unvermeidlichen Abweichungen durch unterschiedliche Implementierungen, Datenrepräsentationen und System-Overheads, die sich aber insgesamt in vertretbaren Bereichen halten, sich stets in gleichem Verhältnis zueinander befinden und sich somit durch einen Korrekturfaktor ausdrücken lassen. Da sich die Vergleichsreihen in Tendenz und Größenordnung entsprechen, erlauben sie vergleichsweise realistische und hinreichend untermauerte Aussagen.

▷ Der deutliche Unterschied zwischen den Ergebnisreihen von geclusterter und ausgelagerter Speicherung läßt erkennen, daß die Anpassung der physischen Strukturen komplexer Objekte an die Workload-Charakteristika ein großes Optimierungspotential in Hinblick auf die Verarbeitungskosten eröffnet.

Dieses Optimierungspotential soll mit den Simulationen und Messungen im nächsten Abschnitt noch genauer untersucht werden.

Ergebnisse	**Anzahl physischer Blockzugriffe bei**	
	ausgelagerter Speicherung	**geclusterter Speicherung**
Simulation		
mit 100 Anfragen	*1778*	*291*
mit 1000 Anfragen	*18958*	*2779*
mit 10000 Anfragen	*192523*	*27510*
⌀ pro Anfrage	18,3	2,8
Oracle-Messung		
mit 100 Anfragen	*1935*	*527*
mit 1000 Anfragen	*18524*	*5054*
mit 10000 Anfragen	*188576*	*50433*
⌀ pro Anfrage	18,9	5,1
Kostenschätzung pro Anfrage	18,8	4,0

TABELLE 6.2: Simulationsergebnisse, Oracle-Meßwerte und Kostenschätzungen

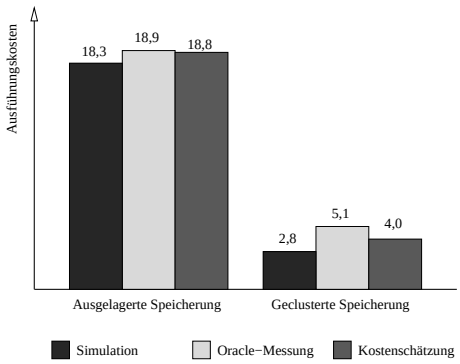


ABBILDUNG 6.14: Vergleich der Simulationsergebnisse, Oracle-Meßwerte und Kostenschätzungen

Parameter	Ausprägung
Anzahl der Werkstückarten in der Datenbank	500
Anzahl der Fertigungsschritte pro Werkstückart	zufällig zwischen 1 und 30, gleichverteilt
Ballast pro Werkstückart und Fertigungsschritt	100 Bytes
Resultierende Datenbankgröße	circa 1300 KB
davon Daten	circa 1100 KB
davon Indexe	circa 160 KB
Blockgröße der Datenbank	4 KB
Puffergröße der Datenbank	passend für circa 10% der Daten

TABELLE 6.3: Charakterisierung des weiterführenden Meßszenarios

6.4.2 Weiterführende Vergleichsmessungen

In der Weiterführung der Versuchsmessungen zu Speicherstrategien und ihrer Tauglichkeit für verschiedene Anforderungen beim Zugriff auf komplexe Objekte führt dieser Abschnitt in die einzelnen Meßreihen ein, präsentiert und bewertet die Meßergebnisse und untersucht das sich daraus ergebende Optimierungspotential. Für die Messungen wurde dabei das Szenario des letzten Abschnitts beibehalten. Allerdings wurde eine andere Anfrage im Stil

> „Gib mir alle Werkstückarten, die Fertigungsschritte mit den Bearbeitungsvorgängen '0815' und '4711' erfordern!"

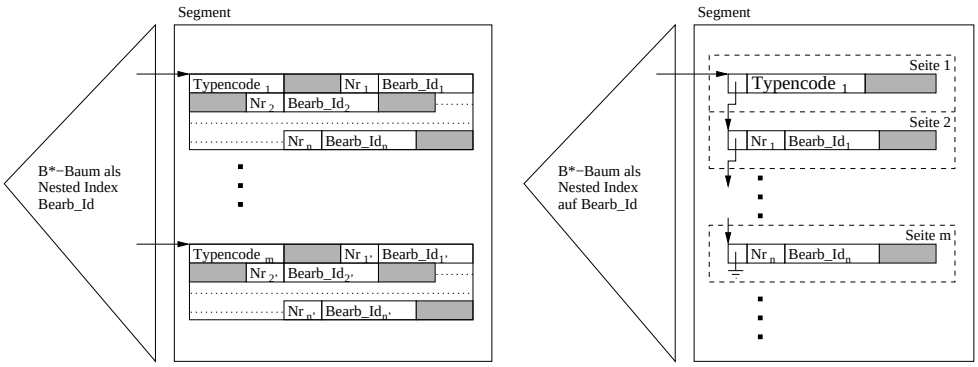
unterstellt. Die SQL-Formulierung lautet dann

```
SELECT * FROM Werkstückart
WHERE ( (0815, 'Drehen'), (4711, 'Fräsen') ) IN Herstellungsablauf
```

Untersucht wurden die beiden Strukturen Inline Array und Pointer Array. Dies erfolgte jeweils mit und ohne Index auf den Werkstückarten. Beim Inline Array wurden die Implementierungsvarianten mit und ohne seitenübergreifende Sätze (Spanned Records) untersucht. Die Struktur Pointer Array wurde dabei in folgenden Varianten implementiert:

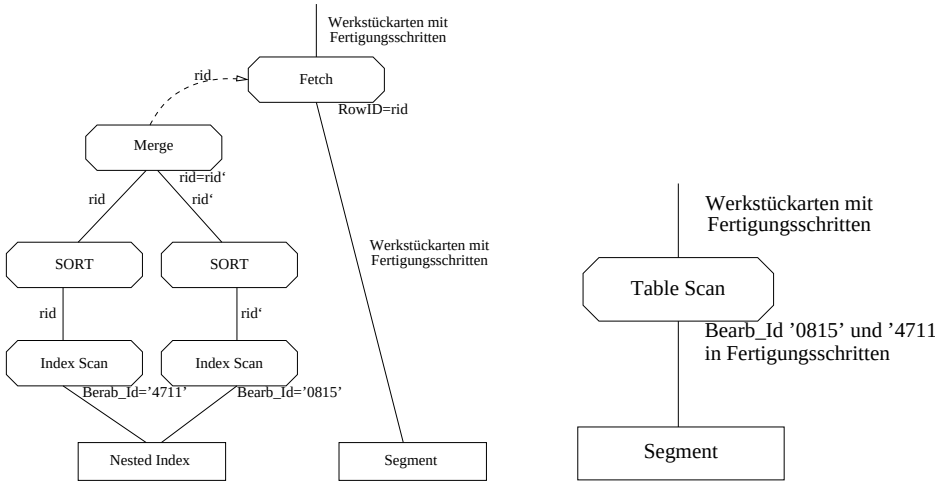
- ▷ Werkstückart- und Fertigungsschritt-Sätze verteilt, Fertigungsschritt-Sätze nach ihrer Werkstückart-Zugehörigkeit geclustert
- ▷ Werkstückart- und Fertigungsschritt-Sätze verteilt, Fertigungsschritt-Sätze nicht geclustert
- ▷ Werkstückart- und Fertigungsschritt-Sätze dicht, Fertigungsschritt-Sätze nach ihrer Werkstückart-Zugehörigkeit geclustert

Die Messungen wurden jeweils mit den in TABELLE 6.3 zusammengefaßten Parametern durchgeführt.



(a) ohne seitenübergreifende Sätze (b) mit seitenübergreifenden Sätzen

ABBILDUNG 6.15: Inline Arrays mit Nested Index



(a) mit Indexnutzung (b) ohne Indexnutzung

ABBILDUNG 6.16: Ausführungspläne für Inline Array

6.4.2.1 Inline Array

Zunächst wurde die Implementierung als Inline Array untersucht (ABBILDUNG 6.15(a)). Dabei wurden die Daten der Werkstückarten und alle ihre Fertigungsschritte in einem Satz gespeichert. Alle Sätze wurden in ein und demselben Segment gespeichert.

Beim Auslesen mit Indexunterstützung wurde ein Nested Index auf den Bearbeitungsvorgangs-IDs aufgebaut. Dieser Nested Index enthält Verweise auf die Werkstückart-Sätze, die einen Fertigungsschritt mit der jeweiligen Bearbeitungsvorgangs-ID enthalten. Bei der Verwendung dieses Indexes wurde zunächst ein Index Scan nach dem Bearbeitungsvorgang '4711' durchgeführt (siehe ABBILDUNG 6.16(a)). Dieser lieferte die RIDs der entsprechenden Sätze, welche in eine Menge eingefügt wurden. Daraufhin erfolgte ein Index Scan nach Bearbeitungsvorgang '0815'. Die Schnittmenge dieser RIDs und derjenigen aus dem ersten Scan wurde über eine Merge-Operation berechnet und

stellte die Ergebnismenge dar. Nachfolgend wurden lediglich jene Datensätze gelesen, deren RIDs in dieser Menge enthalten waren.

Das Auslesen ohne Indexverwendung erfolgte durch einen Segment Scan, wobei für jede Werkstückart alle Fertigungsschritte eingelesen und auf das Vorhandensein der Bearbeitungsvorgänge '4711' und '0815' überprüft wurden. War dies der Fall, wurde das Tupel mit dem zugehörigen Typencode in die Ergebnismenge eingefügt. Den resultierenden Ausführungsplan zeigt ABBILDUNG 6.16(b).

Die Meßergebnisse sind in TABELLE 6.4 unter Szenario 1 zusammengestellt. Bei der Ausführung mit Indexnutzung wurde auf 4 Indexblöcke und 12 Datenblöcke, das heißt in der Summe auf 16 Blöcke zugegriffen. Dabei registrierte die Simulationsumgebung durch die in Abschnitt 6.3.2.4 vorgestellte Modellierung von Externspeicherzugriffskosten 11,45 Kosteneinheiten. Ohne Indexnutzung mußten dagegen natürlich keine Indexblöcke gelesen werden, dafür aber beim vollständigen Scan alle 301 Datenblöcke. Da nach Abschnitt 6.3.2.4 der sequentielle Externspeicherzugriff nur $1/m = 1/20$ der Kosten verursacht (Speed Up m, wobei m die konfigurierte Distanzgrenze für die Kostenreduzierung ist und $m = 20$ angenommen wurde), sind dabei auch nur $301/20 \approx 16$ Kosteneinheiten aufgelaufen.

Die Ergebnisse zeigen, daß diese Art der Speicherung vor allem dann sinnvoll ist, wenn sehr häufig das komplette Werkstückart-Objekt gelesen werden muß. Dazu reicht ein Lesezugriff pro Werkstückart aus, da sie mit allen ihren Fertigungsschritten in einem Satz gespeichert ist.

Nachteilig wirkt sich dies allerdings dann aus, wenn nur einzelne Informationen – etwa die Bezeichnung der Werkstückart – benötigt werden. Es muß auch hier immer das gesamte Objekt teils ‚nutzlos' gelesen werden. Hier kann die Speicherung als Pointer Array Abhilfe schaffen.

Wie oben erwähnt, wurde zusätzlich die Speichervariante mit seitenübergreifenden Sätzen untersucht (ABBILDUNG 6.15(b)). Dazu spielt die maximale Anzahl von Fertigungsschritten, die pro physischem Satz gespeichert werden können, eine wichtige Rolle. Sie ist durch das Verhältnis von Seiten- und Satzgröße bestimmt. Übersteigt die Anzahl der Fertigungsschritte einer Werkstückart die maximale Anzahl der pro Datensatz speicherbaren Fertigungsschritte, werden diese auf mehrere Sätze verteilt. Dabei kann es vorkommen, daß die Teilsätze über mehrere Seiten verteilt gespeichert werden. Somit müßten bei einem Zugriff auf ein einziges Objekt gleich mehrere Seiten gelesen werden. Bis auf diesen Unterschied gelten jedoch die in ABBILDUNG 6.16(a) und ABBILDUNG 6.16(b) dargestellten Ausführungspläne auch für diese Anfragen. Das Szenario 2 in TABELLE 6.4 stellt die Meßergebnisse für den Inline-Array-Fall mit seitenübergreifenden Sätzen dar.

6.4.2.2 Pointer Array

Bei der Implementierung als Pointer Array werden die Informationen über Werkstückarten und Fertigungsschritte getrennt in Primär- und Sekundärsätzen gespeichert. Die Primärsätze nehmen die Werkstückarten auf, in den Sekundärsätzen werden die Fertigungsschritte gespeichert (ABBILDUNG 6.17).

Die Messungen wurden jeweils wieder mit und ohne Index durchgeführt. In ersterem Fall wurde ein Nested Index auf den Werkstückarten aufgebaut, der bei Eingabe einer Bearbeitungsvorgangs-ID eine Referenz auf eine Werkstückart zurückliefert, welche den gesuchten Bearbeitungsvorgang erfordert.

Zunächst wurden die verteilte und die dichte Speicherung der Primär- und Sekundärsätze untersucht. Die verteilte Speicherung, das heißt Ablage der Werkstückarten- und

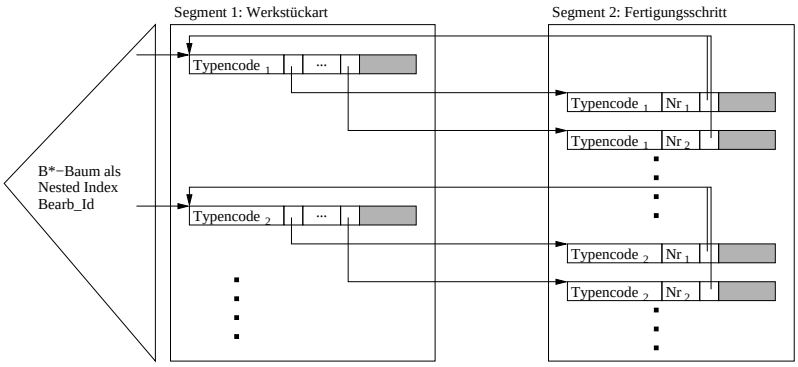


ABBILDUNG 6.17: Pointer Arrays

Fertigungsschrittsätze in zwei verschiedenen Segmenten, verspricht vor allem dann Leistungsvorteile, wenn nicht das gesamte Objekt gefragt ist, sondern beispielsweise nur Auskünfte über einen Fertigungsschritt oder die Werkstückartbezeichnung ermittelt werden sollen. Anders ist das bei dichter Speicherung: Hier werden Werkstückarten- und Fertigungsschrittsätze in ein und demselben Segment gespeichert. Bei einem vollständigen Segment Scan werden somit auch nicht benötigte Sätze – etwa Fertigungsschrittsätze beim Lesen von Werkstückartdaten – mitgelesen.

Im Fall der verteilten Speicherung wurde zudem mit der Option der Clusterung der Sekundärsätze experimentiert. Bei geclusterter Speicherung wurden die Fertigungsschrittsätze einfach direkt nach jedem Werkstückartsatz erzeugt und in Segment 2 geschrieben. Somit stehen alle Fertigungsschritte einer Werkstückart dicht beieinander. Im nicht geclusterten Fall wurden zunächst alle Fertigungsschrittsätze in Segment 2 geschrieben und diese danach zufällig den Werkstückarten zugeordnet. Daraus resultierte eine nichtgeclusterte Speicherung der Sekundärsätze.

Beim Auslesen ohne Index wurde ein Segment Scan auf Segment 1 durchgeführt (siehe ABBILDUNG 6.18). Für jeden Primärsatz (Werkstückart) wurden alle referenzierten Sekundärsätze (Fertigungsschritte) gelesen und auf das Vorhandensein der Bearbeitungsvorgänge '4711' und '0815' überprüft. Die Typencodes derjenigen Werkstückarten, die entsprechende Fertigungsschritte enthielten, wurden in eine Menge geschrieben, die nach Abschluß des Scans das Ergebnis des Anfrage darstellte.

Das Auslesen mit Indexverwendung gestaltete sich folgendermaßen (siehe ABBILDUNG 6.19): Zunächst wurden alle Werkstückarten bestimmt, die den Bearbeitungsvorgang '4711' erfordern, und deren RIDs in eine Menge gespeichert. Anschließend wurde die gleiche Prozedur für den Bearbeitungsvorgang '0815' durchgeführt. Für die in der Schnittmenge enthaltenen RIDs wurden die Typencodes der zugehörigen Werkstückarten ausgelesen. Diese bildete die Ergebnismenge.

In den Szenarien 3, 4 und 5 in TABELLE 6.4 sind die Ergebnisse für Pointer Array dargestellt.

6.4.2.3 Interpretation der Meßergebnisse und Fazit

Die Gegenüberstellung der Meßergebnisse der Szenarien 2 und 4 mit und ohne Indexnutzung aus TABELLE 6.4 in ABBILDUNG 6.20(a) zeigt, daß die eingelagerte Speicherung einer Kollektion als Inline Array für einen Zugriff auf die komplette Kollektion besser geeignet

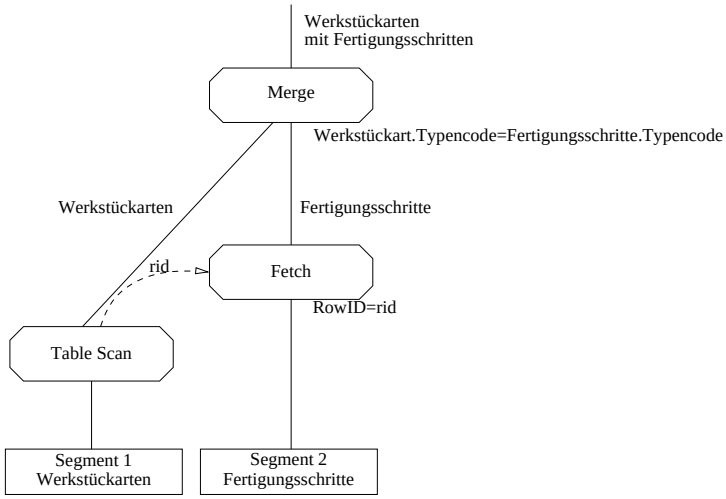


ABBILDUNG 6.18: Ausführungsplan für Pointer Array ohne Indexnutzung

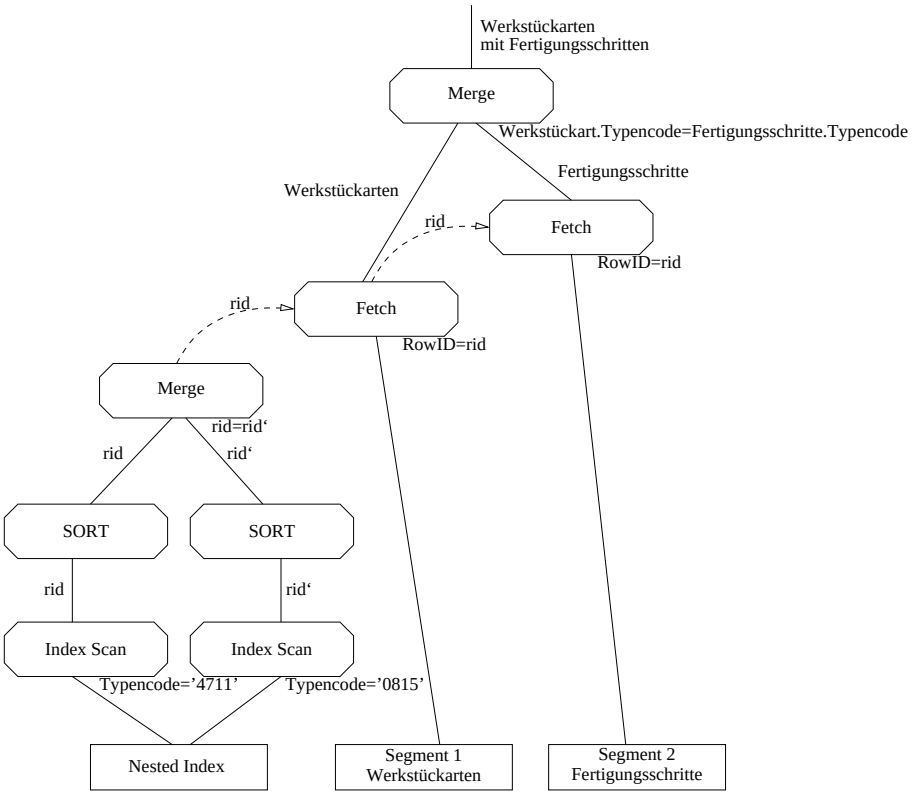
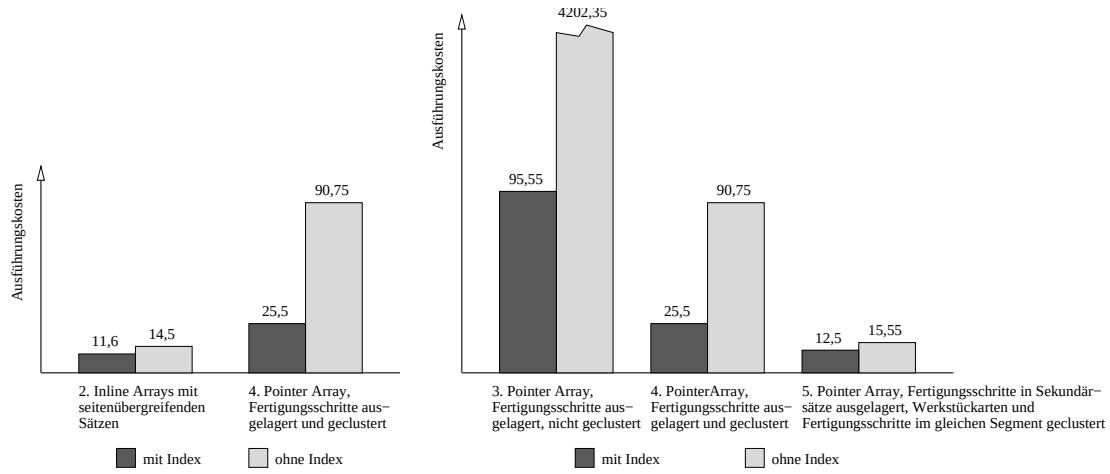


ABBILDUNG 6.19: Ausführungsplan für Pointer Array mit Indexnutzung

Szenario	**mit Index**		**ohne Index**	
Zugriff auf Segment	gelesene Blöcke	Kosten	gelesene Blöcke	Kosten
1. Inline Arrays ohne seitenübergreifende Sätze				
Indexsegment	*4*	*2,1*	-	-
Datensegment Werkstückarten mit Fertigungsschritten	*12*	*9,35*	*301*	*16*
Σ	16	**11,45**	301	**16**
2. Inline Arrays mit seitenübergreifenden Sätzen				
Indexsegment	*4*	*2,1*	-	-
Datensegment Werkstückarten mit Fertigungsschritten	*19*	*9,5*	*271*	*14,5*
Σ	23	**11,6**	271	**14,5**
3. Pointer Array, Fertigungsschritte ausgelagert und nicht geclustert				
Indexsegment	*4*	*2*	-	-
Datensegment Werkstückarten	*10*	*3,5*	*39*	*39*
Datensegment Fertigungsschritte	*156*	*90,05*	*6619*	*4163,35*
Σ	170	**95,55**	6658	**4202,35**
4. Pointer Array, Fertigungsschritte ausgelagert und geclustert				
Indexsegment	*5*	*2,95*	-	-
Datensegment Werkstückarten	*11*	*2,75*	*40*	*40*
Datensegment Fertigungsschritte	*21*	*11,55*	*255*	*50,75*
Σ	37	**25,5**	295	**90,75**
5. Pointer Array, Fertigungsschritte in Sekundärsätze ausgelagert, Werkstückarten und Fertigungsschritte im gleichen Segment geclustert				
Indexsegment	*5*	*2,95*	-	-
Datensegment Werkstückarten und Fertigungsschritte	*21*	*9,55*	*292*	*15,55*
Σ	26	**12,5**	292	**15,55**

TABELLE 6.4: Ergebnisse der weiterführenden Messungen



(a) Ein- und Auslagerung

(b) Clusterung

ABBILDUNG 6.20: Effekte unterschiedlicher physischer Speicherstrukturen für Kollektionselemente

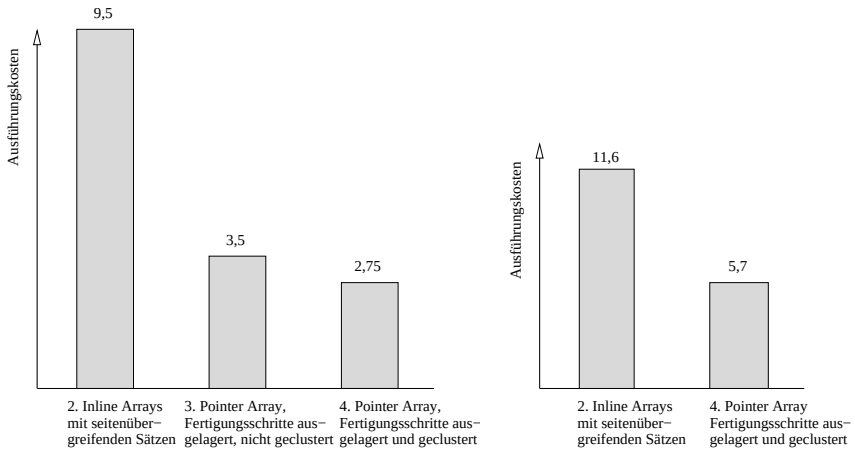
ist als die ausgelagerte Speicherung als Pointer Array. Dies liegt erwartungsgemäß daran, daß bei der ausgelagerten Kollektion extra Zugriffe auf die Kollektionselemente erfolgen.

Beim Pointer Array spielt zudem die Clusterung der Datensätze eine erhebliche Rolle. Dies macht ABBILDUNG 6.20(b) durch die Gegenüberstellung der Szenarien 3, 4 und 5 aus TABELLE 6.4 deutlich. Werden die Fertigungsschritte einer Werkstückart nicht geclustert, sondern wie bei Szenario 3 zufällig über das Sement verteilt, so ergeben sich extreme Ausführungskosten von 4202,35 Kosteneinheiten und 6658 Blockzugiffe. Durch das ständige Nachladen neuer Blöcke kommt es hier zu Verdrängungseffekten, so daß fast jeder Blockzugriff auch zu einer Externspeicheroperation führt. Dagegen halten sich in Szenario 4 die Zugriffskosten durch Clusterung der in ein anderes Segment ausgelagerten Fertigungsschritte in einem vernünftigen Maß.

Diese Kosten können, wie ABBILDUNG 6.20(b) ebenfalls zeigt, in Sezenario 5 durch die Plazierung der Fertigungsschritte im gleichen Segment wie die Werkstückarten und die Clusterung von Fertigungsschritten und Werkstückarten nochmals erheblich gesenkt werden. Dadurch erreichen sie fast das gleiche niedrige Niveau wie die eingelagerte Speicherung mit Inline Arrays, allerdings ohne daß die Fertigungsschritte direkt im Werkstückartensatz gespeichert werden.

Wenn man, wie in ABBILDUNG 6.21(a), die Kosten für den Zugriff auf die Werkstückarten ohne Fertigungsschritte bei der ausgelagerten Speicherung – Zeilen „Datensegment Werkstückarten“ der Szenarien 3 und 4 mit Index – und bei der eingelagerten Speicherung – Zeile „Datensegment Werkstückarten mit Fertigungsschritten“ von Szenario 2 mit Index – gegenüber stellt, dann wird der Vorteil der Kombination von Auslagerung und Clusterung deutlich.

Die erheblichen Unterschiede zwischen den Kosten der Anfrageausführungen mit und ohne Indexunterstützung sind in TABELLE 6.4 im Vergleich der Spalten „mit Index“ und „ohne Index“ erkennbar. Diese sind jedoch insbesondere darauf zurückzuführen, daß ein Nested Index verwendet wird. Dieser erlaubt es, mit den gesuchten Bearbeitungsvorgangs-IDs direkt die sie erfordernden Werkstückarten zu finden, ohne den Umweg über die Fer-



(a) Datenzugriff ohne Kollektionselemente

(b) Index- ohne Datenzugriff

ABBILDUNG 6.21: Effekte bei Auslagerung von Kollektionselementen

tigungsschrittsätze gehen zu müssen. Bei einer Anfrage nicht nach vollständigen Werkstückarten, sondern nur nach Werkstückartdaten ohne Fertigungsschrittdaten, schneiden demzufolge die ausgelagerten Speicherungsformen besser ab als die eingelagerten, wie ABBILDUNG 6.21(b) zeigt. Dort werden nur die Kosten des Zugriffs auf das Indexsegment und das Segment mit den Werkstückarten berücksichtigt.

Insgesamt zeigen diese Überlegungen, daß für das Beispielszenario eine möglichst kompakte Speicherung der Kollektion und ihrer Elemente die beste Speicherungsalternative darstellt, da als Anfrageergebnis vollständige Werkstückarten mit all ihren Fertigungsschritten gefordert waren.

Das vorgestellte Szenario kann zwar keinen Anspruch auf Allgemeingültigkeit erheben und ist auch nicht umfassend, aber es stellt zumindest einen oft wiederkehrenden Ausschnitt komplexer Objektstrukturen und Anfragesituationen dar. Insofern kann den folgenden aus den Meßergebnissen abgeleiteten Ansätzen für Optimierungsheuristiken schon eine gewisse Aussagekraft zugebilligt werden:

- ▷ Geclusterte Speicherung von Speicherobjekten und -subobjekten begünstigt deren gemeinsammen Zugriff.

- ▷ Auslagerung von Speicherobjekten und -subobjekten ist dagegen günstiger, wenn in der Regel nur auf Teilobjekte zugegriffen wird.

- ▷ Der Einsatz von speziellen Indexstrukturen kann, insbesondere für verschachtelte und kollektionswertige Objektstrukturen, ganz erhebliche Leistungsgewinne hervorrufen.

Die konkrete Aussage dieser Heurisitken ist jedoch nicht das Wichtige in diesem Abschnitt. Vielmehr ist es wesentlich festzuhalten, daß die Gegenüberstellungen deutlich auf das Optimierungspotential hinweisen, das in der Adaption von Speicherstrukturen an gegebene Abfrageszenarien und Arbeitslasten liegt.

6.4.3 Zusammenfassung

Die Simulationen dieses Abschnitts machen deutlich, daß in der Adaptation der Speicherstrukturen und Anfragebearbeitungsalgorithmen an die jeweils zu erwartende Arbeitslast (Workload) ein großes Optimierungspotential liegt. Es bietet die Möglichkeit, starke Verbesserungen der Leistungsfähigkeit objektrelationaler Datenbankanwendungen zu bewirken.

Allerdings konnte die Herangehensweise an die Optimierung objektrelationaler Speicher- und Verarbeitungsstrukturen durch Simulationen und die damit verbundenen Möglichkeiten im Rahmen dieser Arbeit nur exemplarisch vorgestellt werden. Jedoch wurde trotz dieser Kürze deutlich, daß Simulationen ein wichtiger Schritt zur Erarbeitung und Validierung von Kostenmodellen für objektrelationale Datenbanken sind und ein vielversprechendes, interessantes Gebiet für weiterführende Arbeiten darstellen.

Gleichzeitig konnten für ein typisches Szenario einige Speicheroptimierungsheuristiken abgeleitet werden. Obwohl diese Heuristiken als Binsenweisheiten erscheinen mögen, zeigen sie jedoch, daß sich schon mit einfachsten Mitteln, gewissermaßen als Vorstufe zur kostenbasierten Optimierung, schon ein erhebliches Verbesserungspotential für objektrelationale Strukturen eröffnet. Gerade wegen der erheblichen Effekte, die mit einfachsten Mitteln erreichbar sind, wäre eine weitere Beschäftigung auf diesem Gebiet von hoher praktischer Relevanz.

6.5 Automatisierte Optimierung von Speicherstrukturen

Nachdem im letzten Abschnitt das Optimierungspotential alternativer physischer Objektspeicherstrukturen anhand von Simulationen verdeutlicht und auf das hohe praktische Potential einfacher, schnell umsetzbarer Optimierungsheuristiken hingewiesen wurde, sollen in diesem Abschnitt einige weiter in die Zukunft weisende und schwieriger zu realisierende Konzepte für eine automatisierte Optimierung objektrelationaler Daten- und Verarbeitungsstrukturen erörtert werden. Dieses Thema spielt gegenwärtig bei kommerziellen Datenbankprodukten eine wichtige Rolle. Schlagworte wie „Self-Tuning Database“, „Autonomous Computing“ und „Zero Administration Strategie“ stammen zwar aus dem Marketingjargon, symbolisieren jedoch eine interessante aktuell sehr lebhafte Entwicklungsrichtung. Allerdings ist es auch ein umfassendes und komplexes Thema, das hier nicht umfassend behandelt werden kann. In diesem Abschnitt sollen deshalb nur weiterführende Gedanken und Visionen, die sich aus dem Schwerpunkt dieser Arbeit ergeben, vorgestellt werden. Zur eingehenden Beschäftigung sei auf die spezielle Literatur verwiesen: [RK84, SD03, Dor03, Dor05].

6.5.1 Zielstellung

Die angestrebte Zielstellung ist die automatische Ermittlung geeigneter physischer Speicherstrukturen bei gegebener logischer Datenstruktur (Schemadefinition), gegebener Datenausprägung und gegebener Arbeitslast (Workload) auf diesen Daten. Dabei bezieht sich das Attribut ‚geeignet‘ auf eine möglichst optimale Unterstützung der Arbeitslast durch die Speicherstrukturen. Jedoch kann diese Zielfunktion unterschiedlich interpretiert und gewichtet werden. Denkbar wären hier beispielsweise

- die Minimierung der Gesamtkosten oder der Ausführungszeit für die Abarbeitung der gesamten Workload oder

- ▷ die Minimierung der Kosten oder der Ausführungszeit für die Abarbeitung der ‚wichtigsten' Einzelanfrage oder ‚wichtigsten' Teilarbeitslast.

- ▷ die Minimierung einer gewichteten Summe der Kosten oder der Ausführungszeit für die Abarbeitung der Einzelanfragen der Arbeitslast.

Dabei kann die Bestimmung der ‚wichtigsten' Einzelanfrage beziehungsweise Teilarbeitslast oder die Gewichtung der Einzelanfragen der Arbeitslast natürlich nach unterschiedlichsten Kriterien erfolgen. So kann beispielsweise auf besonders geschäftskritische Teilarbeitslasten hin optimiert werden oder aber auf die gesamte Arbeitlast hin, oder es kann ein Ausgleich zwischen beiden Zielen angestrebt werden.

6.5.2 Vorgehen

Bei diesen Optimierungen spielen im Gegensatz zur klassischen DBMS-Optimierung zwei Freiheitsgrade eine Rolle: Zur klassischen Optimierung der Anfrageausführungspläne durch das DBMS kommt die Optimierung der physischen Speicherstrukturen als automatisch zu erledigende Aufgabe für das DBMS hinzu. Natürlich stehen beide Freiheitsgrade in einem Wechselverhältnis: Die Anfrageoptimierungsmöglichkeiten hängen einerseits von den vorhandenen physischen Strukturen ab, denn von den Vorteilen einer Auslagerung oder eines Indexes kann ja beispielsweise nur Gebrauch gemacht werden, wenn sie vorhanden sind. Andererseits ist es gerade die Aufgabe der Speicherstrukturoptimierung, solche Optimierungsmöglichkeiten durch die Erzeugung ‚geeigneter' physischer Strukturen zu eröffnen.

Zur Optimierung in Bezug auf diese beiden Freiheitsgrade ist es jedoch wichtig, ihre jeweilige Charakteristik zu beachten:

- ▷ Die Anfrageausführungsplan-Optimierung
  - ◇ wirkt lokal auf eine einzelne Anfrage,
  - ◇ erfordert keine persistenten Änderungen der Datenstrukturen und
  - ◇ erzeugt keinen Zusatzaufwand, da sie ohnehin bei jeder Anfrage implizit vorgesehen ist. (Dieser Aufwand wird außerdem in der Regel durch Anfrage-Caching reduziert.)

- ▷ Die Speicherstruktur-Optimierung dagegen
  - ◇ ist global auf die gesamte Arbeitslast ausgerichtet,
  - ◇ erfordert persistente Änderungen der Datenstrukturen und
  - ◇ erzeugt einen erheblichen Aufwand bei der Reorganisation aller betroffenen Daten.

Aus dieser Charakteristik ergibt sich ein mögliches Vorgehen bei der automatischen Optimierung der Speicherstrukturen, das aus folgenden Phasen besteht:

1. *Beobachtung und Erfassung der Workload*

   In dieser Phase werden die Anfragen auf das DBMS vollständig oder partiell erfaßt und in textueller Form, in codierter Form oder aber in einer auf relevante Informationen reduzierten Form gespeichert.

2. *Berechunung einer optimierten Speicherstruktur der anhand der erfaßten Workload und Kosten-Nutzen-Bewertung der dazu notwendigen Reorganisation*

   Die Optimierungsphase besteht aus den zwei Teilschritten

2.1. Berechnung einer möglichst optimalen Speicherstruktur in Bezug auf die erfaßte Workload

2.2. Ermittlung und Abwägung des zu erwartenden Reorganisationsnutzens und der dazu aufzuwendenden Reorganisationskosten

3. *Reorganisation der Speicherstruktur*

   Wenn ein ausreichendes Optimierungspotential vorhanden ist, wird die Reorganisation der Daten entsprechend der ermittelten Struktur geplant und durchgeführt.

Dieses Vorgehen kann zyklisch auf Anforderung, periodisch oder kontinuierlich erfolgen:

▷ *Optimierung auf Anforderung*

Die Erfassung der Workload und die Optimierung der physischen Speicherstrukturen werden vom DBMS zwar automatisch ausgeführt, jedoch vom Administrator einzeln oder gemeinsam explizit angestoßen. Dies kann geschehen, wenn der Bedarf dazu, bedingt durch Workload-Änderungen, vom Administrator vermutet oder beispielsweise durch verändertes Antwortverhalten des DBS festgestellt wird.

▷ *Periodische Optimierung*

Sowohl die Erfassungsphase als auch die Optimierungsphase könnten periodisch zu festgelegten Zeiten angestoßen werden. So kann die Erfassung etwa in einem, für die Gesamt-Workload repräsentativen aber nicht geschäftskritischen Zeitraum, und die mit einer potentiellen Reorganisation verbundenen Optimierung in lastarmen Zeiten oder nächtlichen Wartungsintervallen erfolgen.

▷ *Kontinuierlich zyklische Optimierung*

Das DBMS überwacht die Workload des DBS kontinuierlich und stößt bei Arbeitslaständerungen, die einen gewissen Schwellwert übersteigen, automatisch die Berechnung möglichst optimaler Speicherstrukturen an. Wird dabei festgestellt, daß damit eine erhebliche Verbesserung der DBS-Leistung erzielt werden kann, so wird eine Reorganisation angestoßen oder für einen lastarmen Zeitraum eingeplant.

Auch Mischformen sind denkbar: So könnte das DBMS die Workload kontinuierlich beobachten, periodisch das vorhandene Verbesserungspotential berechnen und gegebenenfalls dem DBS-Administrator entsprechende Reorganisationsvorschläge unterbreiten, die dieser dann in einem passenden Zeitfenster anstoßen oder verwerfen kann.

6.5.3 Erfassung der Workload

Die Erfassung der Workload kann auf unterschiedlichste Arten erfolgen:

- ▷ Kontinuierliche Aufzeichnung der Anfragen
- ▷ Aufzeichnung der Anfragen über einen größeren Zeitraum
- ▷ Aufzeichnung der Anfragen über einen repräsentativen Zeitraum
- ▷ Künstliche Zusammenstellung von Anfragen
- ▷ Auswahl relevanter Anfragen aus einer Aufzeichung
- ▷ Gewichtung von Anfragen aus einer Aufzeichung

Bei der Aufzeichnung beziehungsweise Speicherung können entweder

- ▷ die Anfragen der Workload vollständig textuell in einer Art Log mitgeschrieben werden oder
- ▷ die Anfragen in einer vollständigen internen Repräsentationsform, beispielsweise als Ablaufplan oder Operatormenge, abgelegt werden oder
- ▷ nur relevante Informationen aus den Anfragen, wie zum Beispiel betroffene Tabellen und Spalten, Anfrageprädikate oder Sortierreihenfolgen et cetera, in Form von Logs oder Statistiken erfaßt werden.

6.5.4 Berechung einer optimalen Speicherstruktur und Kosten-Nutzen-Bewertung der Reorganisation

Für die Berechnung einer möglichst optimalen Speicherstruktur in der Optimierungsphase können sehr viele bekannte Optimierungstechniken eingesetzt werden. Zumindest theoretisch können diese von der Enumerierung und Kostenbewertung aller Möglichkeiten bis zu praktisch anzutreffenden Pruning-, Greedy- und Bottom-Up-Techniken reichen. Die dabei entstehenden Optimierungsergebnisse können von groben Schätzungen über relative Kostenbewertungen bis zu recht präzisen Berechnungen der entstehenden absoluten Kosten reichen. Sie sollten jedoch zumindest eine Aussage darüber ermöglichen, ob eine Reorganisation der Daten eine Leistungssteigerung des DBS verspricht. Im Idealfall könnte sogar die erreichbare Verbesserung absolut oder relativ ermittelt werden.

Gleichfalls sind die zu erwartenden Kosten für die Reorganisation hin zur ermittelten optimierten Speicherstruktur abzuschätzen. Wichtige Faktoren dazu sind

- ▷ die vorhandene und die angestrebte Speicherstruktur und der damit verbundene strukturelle Konvertierungsaufwand,
- ▷ die Anzahl und Größe der insgesamt betroffenen Datenobjekte
- ▷ die geplante Art der Reorganisation: offline/online, vollständig/unvollständig (hierauf wird im nächsten Abschnitt noch eingegangen) und
- ▷ die erwartete Lebensdauer der neuen Speicherstruktur bis zu ihrer Degenerierung und damit erneut notwendigen Optimierung und Reorganisation.

Aus der Gegenüberstellung von Kosten und Nutzen muß dann eine Reorganisationsentscheidung und -strategie bestimmt werden, die für das angestrebte Optimierungsziel, ob dieses nun Durchsatz, Antwortzeit et cetera ist, das größte Verbesserungspotential verspricht. Das ‚Ausreichen' des Verbesserungspotentials kann dabei durch absolute beziehungsweise relative Schwellwerte, durch sein Verhältnis zum erforderlichen Reorganisationsaufwand oder durch die Betrachtung der Gesamtkosten für die Reorganisation zusammen mit der nachfolgenden Workload festgelegt werden.

6.5.5 Reorganisation der Speicherstruktur

Ausgehend von den berechneten Optimierungsergebnissen, erfolgt bei einem ausreichenden Verbesserungspotential die Reorganisation der Daten entsprechend der ermittelten möglichst optimalen Speicherstruktur. Für die Reorganisation gibt es eine Reihe von Freiheitsgraden, deren wichtigste kurz angerissen werden sollen:

- ▷ *Art der Reorganisation*
  - ◇ *Offline-Reorganisation*
    Für die Offline-Reorganisation wird der Datenbankbetrieb unterbrochen, so daß der Nachteil der reduzierten DBMS-Verfügbarkeit besteht. Da jedoch die volle

DBMS-Leistung zur Verfügung steht, kann sie relativ schnell abgeschlossen werden. Ihr wichtigster Vorteil ist jedoch, daß der normale DBS-Betrieb nicht belastet wird, sondern jeweils nur eine Speicherstruktur für die vorhandenen Daten unterstützt werden muß, und die Komplexität der Reorganisation geringer als bei der Online-Reorganisation ist.

- ◇ *Online-Reorganisation*

  Die Online-Reorganisation erfolgt parallel zum normalen Datenbankbetrieb und bietet den Vorteil der erhöhten DBMS-Verfügbarkeit. Da jedoch gleichzeitig berücksichtigt werden muß, daß dabei die DBS-Leistung beeinträchtigt wird, sollte sie in lastarmen Zeiten erfolgen oder bei Lasterhöhungen unterbrochen werden. Außerdem besitzt die Online-Reorganisation eine erhöhte Komplexität und erfordert zusätzlichen Aufwand im DBMS, da über einen gewissen Zeitraum sowohl die alte als auch die neue Speicherstruktur unterstützt werden müssen.

- ▷ *Zeitpunkt der Reorganisation*

  Die Reorganisation kann unmittelbar nach dem Optimierungsschritt oder aber verzögert in einer lastarmen Zeit erfolgen.

- ▷ *Vollständigkeit der Reorganisation*

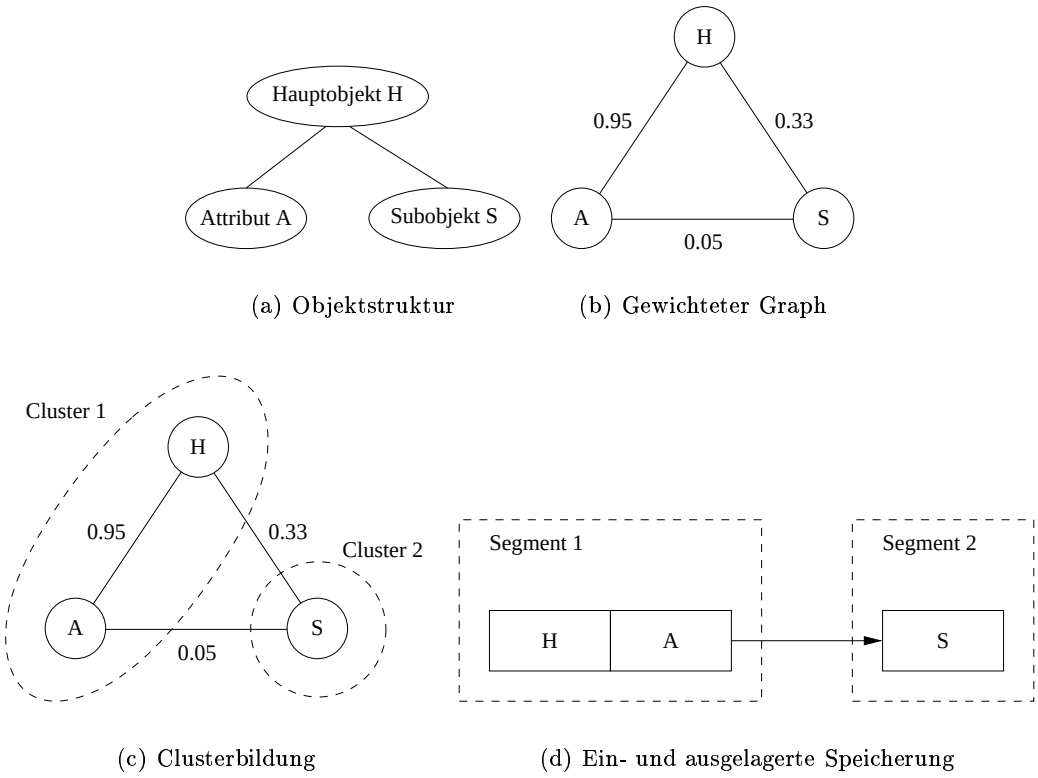
  Die Reorganisation kann entweder
  - ◇ vollständig und sofort für alle Datensätze,
  - ◇ zeitweilig unvollständig nur für Datensätze, auf die zugegriffen wird oder die geändert werden, oder
  - ◇ dauerhaft unvollständig nur für neue Datensätze

  erfolgen.

Eine vollständige Diskussion der Optimierungsthematik kann und soll hier nicht geleistet werden. Dazu sei nochmals auf die Literatur verwiesen: [SD03, Dor03, Dor05]. Jedoch soll zum Abschluß noch ein Optimierungskonzept für die Aus- und Einlagerung von Objektattributen und Subobjekten, die ja im Hauptteil dieser Arbeit eine wichtige Rolle spielt (Abschnitt 3.2.5), im nächsten Abschnitt kurz vorgestellt werden.

6.5.6 Beispiel: Optimierungskonzept für ein- und ausgelagerte Attribute

Für die Entscheidung zwischen Auslagerung und direkter Speicherung von Attributen oder Teilobjekten spielt die Häufigkeit des gemeinsamen Zugriffs eine entscheidende Rolle: Wird häufig auf Objekt und jeweiliges Attribut gemeinsam zugegriffen, so ist es günstig, sie im gleichen physischen Datensatz oder aber zumindest in der gleichen Speicherseite abzulegen, da sie auf diese Weise mit einer einzigen I/O-Operation gelesen oder geschrieben werden können. Wird dagegen auf beide häufig getrennt zugegriffen, indem beispielsweise das Hauptobjekt gelesen wird, ohne daß das Attribut benötigt wird, ist es günstiger, beide nicht im gleichen Datensatz, nicht in der gleichen Speicherseite und evtl. auch nicht im gleichen Speichersegment abzulegen. Auf diese Weise wird beim Zugriff auf das Hauptobjekt auf keinen Fall das Attribut mit gelesen und auch kein Platz auf der Speicherseite des Hauptobjektes für das Attribut belegt. Dieser Speicherplatz kann beispielsweise für weitere Hauptobjekte genutzt werden, auf die ebenfalls zugegriffen werden soll, wodurch mehr Hauptobjekte pro I/O-Operation behandelt werden können.



(a) Objektstruktur (b) Gewichteter Graph

(c) Clusterbildung (d) Ein- und ausgelagerte Speicherung

ABBILDUNG 6.22: Optimierung der Speicherstruktur durch Clusterbildung

Diese Grundüberlegung schlägt sich in folgendem Konzept zur Optimierung von Ein- und Auslagerungen von Attribute und Subobjekten nieder (siehe ABBILDUNG 6.22):

▷ *Workload-Erfassung*

Hier werden die Anfragen der Workload nicht vollständig und auch nicht textuell erfaßt, sondern ihre Eigenschaften bezüglich des gemeinsamen Hauptobjekt-Attribut-Zugriffs in Form einer speziellen Statistik aufgezeichnet. Dazu wird ein potentiell vollständig vermaschter gewichteter Graph genutzt, dessen Knoten die Hauptobjekte, die Objektattribute und die Subobjekte repräsentieren. Die Gewichte der Kanten dieses potentiell vollvermaschten Graphen werden zur Aufzeichnung der Häufigkeit der gemeinsamen Abfrage genutzt. Während des Workload-Erfassungszeitraums werden sie bei jeder Abfrage leicht angepaßt. Auf diese Weise repräsentieren die Gewichte der Kanten am Ende der Workload-Erfassung den Vorteil beziehungsweise den Nachteil der gemeinsamen Speicherung der beiden verbundenen Objekte oder Attribute.

▷ *Optimierung der Speicherstruktur*

Anhand der aufgezeichneten Kantengewichte kann bei der Optimierung mit Hilfe eines Clusterungs-Algorithmus auf dem Graphen eine Speicherstruktur mit entsprechenden Ein- und Auslagerungen berechnet werden, die das Workload-Profil möglichst optimal unterstützt.

6.5.7 Zusammenfassung und Ausblick

Wie die Ausführungen dieses Abschnitts zeigen, bietet das Gebiet der automatischen Optimierung von Speicherstrukturen im speziellen und die automatische Anpassung von Datenbankstrukturen und Verarbeitungsvorgängen an die in konkreten Anwendungsfällen vorliegenden Anforderungen und Arbeitslasten im allgemeinen ein weites Forschungsfeld mit interessanten Fragestellungen. Obwohl auf diesem Gebiet schon einige wissenschaftliche Arbeiten existieren [SD03, Dor03] und auch schon Resultate in Produkten zu beobachten sind, bieten sich für Forschung und Entwicklung hier noch vielfältige Herausforderungen.

6.6 Zusammenfassung des Kapitels

Ziel dieses Kapitels war es, andere, mit dem Hauptthema dieser Arbeit in Verbindung stehende Gebiete kurz anzureißen. Dabei wurden weiterführende Ideen skizziert und auf entsprechende Fragestellungen hingewiesen, um Perspektiven für zukünftige Arbeiten aufzuzeigen.

Es wurden die Möglichkeiten zur Verarbeitung komplexer Objekte mit den Mitteln, die in heutigen relationalen DBMS bereits zur Verfügung stehen, untersucht. Dies erfolgte unter besonderer Berücksichtigung der in den vorausgegangenen Kapiteln vorgestellten Speicherungsalterativen, die mit der Speicherbeschreibungssprache PRDL gesteuert werden können.

Es konnte gezeigt werden, daß die vorhandene relationale DBMS-Technik bereits sehr gut zur Unterstützung der objektrelationalen Verarbeitungskonzepte geeignet ist. Insbesondere bieten das relationale Speichersystem und die relationalen Anfragebearbeitungsoperatoren schon eine sehr gute Basis für die Verarbeitung komplexer Objektstrukturen. Allerdings sind auch Erweiterungen in zwei Grundrichtungen nötig:

1. *Modifizierter Einsatz vorhandener Techniken für den Zugriff auf komplexe Objekte,*wie zum Beispiel die Nutzung und ‚Um-Nutzung' der vorhandenen Table-Scan- und Index-Scan-Operatoren.

2. *Integration neuer Anfragebearbeitungstechniken,*wie beispielsweise die Intergration neuer bitmap- und signaturbasierter Indexstrukturen und des neuen Planoperators Choice zur Polymorphie-Unterstützung.

Nachdem die Frage der prinzipiellen Umsetzbarkeit objektrelationaler Verarbeitungskonzepte mit heutigen DBMS positiv beantwortet werden konnte, beschäftigte sich Kapitel 6 mit den Fragen der Optimierung entsprechender Anfragen und Operationen auf Objektstrukturen. Dazu wurde ein Modell zur Abschätzung der Ausführungskosten von Operationen auf komplexen Objekten vorgestellt. Ziel dieses Modells ist jedoch nicht nur die Anfrageoptimierung, sondern auch die Optimierung der Speicherstrukturen zur Unterstützung einer gegebenen Arbeitslast (Workload).

Es wurde ebenfalls eine im Rahmen dieser Arbeit entwickelte Simulationsumgebung zur Speicherung und Verarbeitung komplexer Objekte präsentiert. Diese gestattet die Erprobung der Speicher- und Indexierungskonzepte, die im Haupteil dieser Arbeit vorgestellt wurden. Gleichzeitig ermöglicht sie die Validierung des in diesem Kapitel vorgestellten Kostenmodells. Auch konnten mit Hilfe der Simulationsumgebung die Plausibilität der Kostenberechungen beispielhaft belegt und das Optimierungspotential alternativer Speicher-

und Verarbeitungsstrategien verdeutlicht werden. An einem Beispielszenario wurden einfache Optimierungsheuristiken herausgearbeitet und ihre praktische Relevanz verdeutlicht.

Die Vison einer automatischen über die Anfrageoptimierung hinausgehenden Optimierung der Speicherstrukturen durch das DBMS wurde abschließend vorgestellt. Dieses an sich ‚alte', seit den vorrelationalen DBMS immer wieder aufgegriffene Thema erlebt aktuell einen lebhaften Aufschwung. „Self-Tuning Databases", „Autonomous Computing", „Zero Administration Strategies", „Index Advisors" und „Storage Wizzards" sind aktuelle Schlagworte, die entsprechende Entwicklungen symbolisieren. Diese Aktualität ist jedoch nicht nur der Entwicklung neuer Datenbankerweiterungen und Einsatzgebiete geschuldet, sondern auch einem sich wandelnden Einsatz von DBMS. Datenbanksysteme werden zunehmend weniger als wartungsintensive Softwaresysteme wahrgenommen, sondern eher als Basiskomponenten heutiger Anwendungsarchitekturen betrachtet, die auch ohne umfangreiche Spezialkenntnisse einsetzbar sein sollen.

Insgesamt zeigt dieses Kapitel, daß das zentrale Thema dieser Arbeit, die Untersuchung und Spezifikation von Speicher- und Indexierungsstrukturen, die Grundlage für eine Reihe weiterer wichtiger und interessanter Arbeitsbereiche liefert. Es gibt mit der Darstellung von Verarbeitungs- und Optimierungskonzepten einen Ausblick auf vielversprechende weitere Arbeitsgebiete und aktuelle Entwicklungsrichtungen. Gemeinsam eröffnen diese ein erhebliches Potential zur Verbesserung der Leistungsfähigkeit von objektrelationalen DBMS, die sich derzeit ja immer noch in der (Fort-)Entwicklung befinden, und von darauf aufbauenden objektorientierten Anwendungen und Softwaresystemlandschaften.

Kapitel 7

Zusammenfassung und Ausblick

7.1 Aufgabenstellung

Die Beschäftigung in dieser Arbeit mit der Speicherung und Indexierung komplexer Objekte in objektrelationalen Datenbank-Management-Systemen hatte das Ziel, einen Beitrag zur Verbesserung eben dieser Kategorie von DBMS zu leisten. Dazu sollte das wichtige Thema der Datenunabhängigkeit angegangen werden, indem ein Weg zur Trennung von logischer und physischer Datenmodellierung vorgeschlagen und diskutiert wurde. Den Grund dafür bildete die in aktuellen relationalen und objektrelationalen Datenbankprodukten und -prototypen vorrangig anzutreffende Abhängigkeit physischer Speicherstrukturen von logischen Datenmodellkonstrukten.

Dazu wurde in der Arbeit die Speicherspezifikationssprache Physical Representation Definition Language (PRDL) vorgeschlagen. Durch die Möglichkeit der expliziten Definition physischer Speicherstrukturen werden diese aus der direkten Abhänigkeit von den logischen Datenstrukturen befreit. Gleichzeitig wird so auch ein Weg aufgezeigt hin zur Anpassung von Speicherstrukturen an die Erfordernisse, die sich aus dem jeweiligen Datenbestand und der jeweiligen Arbeitslast ergeben. Dies ist insbesondere für komplexe Objekte mit tiefverschachtelten, strukturierten und kollektionswertigen Attributen, wie sie SQL:2003 erlaubt, wichtig, da es für diese im Gegensatz zu einfachen Objekten eine große Auswahl alternativer Speicherstrukturen gibt, die zu sehr unterschiedlichen Leistungsfähigkeiten für einzelne Anfragen und Anfragetypen führen.

7.2 Arbeitsschritte, Schwerpunkte und Ergebnisse

Da sich PRDL zur umfassenden Spezifikation von physischen Strukturen eignen soll, muß sie sowohl Speicherungs- als auch Indexierungsaspekte einschließen. In der Literatur, in DBMS-Prototypen und -Produkten sind vielfältige Vorschläge zu Speicher- und Indexstrukturen zu finden. Nach der Einführung in Kapitel 1 und der Darstellung der Grundlagen objektrelationaler DBMS in Kapitel 2 wurden deshalb in Kapitel 3 Speicherstrukturen für komplexe Objekte aus der Literatur untersucht, verglichen und eingeordnet. Aus diesen vielfältigen Ansätzen wurde ein in sich schlüssiges und umfassendes Konzept zur Speicherung komplexer Objekte in ORDBMS zusammengestellt und diskutiert. Desweiteren wurden in Kapitel 4 Indexkonzepte aus der Datenbankforschung untersucht und zu einem Konzept zur Definition von Zugriffspfaden für komplexe Objektstrukturen in ORDBMS zusammengetragen. Sowohl die Speicher- als auch die Indexkonzepte, die in ihren Grund-

lagen sogar teilweise aus der Ära der hierarchischen und Netzwerkdatenbanken stammen, wurden dabei an die speziellen Bedürfnisse objektrelationaler DBMS adaptiert.

Auf diese Weise wurden für die Darstellung der Speicherbeschreibungssprache PRDL in Kapitel 5 die wesentlichen Ansätze zur Speicherung und Indexierung komplexer Objekte in ORDBMS berücksichtigt. Die Darstellung von PRDL erfolgte durch die Angabe von Syntaxregeln und durch die Erläuterung ihrer jeweiligen Semantik, unterstützt von Beispielen.

PRDL bietet durch die Möglichkeiten,

- ▷ Objekte, Subobjekte, Attribute und ganze Attributstrukturen zu physischen Primär- und Sekundärspeichersätzen zuzuordnen,
- ▷ Angaben zur Art und zum Ort der Speicherung zu machen,
- ▷ die Form der Verbindung der Speichersätze über verschiedene Referenzarten festzulegen,
- ▷ lokale und globale Indexstrukturen für Objekt- und Subobjektmengen anzulegen und
- ▷ unterschiedlichste Indexstrukturen über eine Menge von Indexkonstruktoren zu erzeugen,

einen umfassenden Satz an physischen Modellierungskonzepten. Die sehr vielfältigen erzeugbaren Speicher- und Indexstrukturen erlauben es, die physische Repräsentation komplexer Objekte an unterschiedlichste Verarbeitungsanforderungen anzupassen. An Beispielen in den Kapiteln 5 und 6 wurde dies verdeutlicht.

In Kapitel 6 wurden außerdem weitergehende Themen untersucht, die sich aus den Möglichkeiten zur Trennung von logischem und physischem Entwurf beziehungsweise aus der daraus resultierenden Möglichkeit zur Reorganisation der Objektspeicherung ohne Beeinflussung der logischen Modell- und Anwendungsebene ergeben. Bei der Diskussion der Möglichkeiten der Verarbeitung komplexer Objekte zeigte sich, daß heutige relationale DBMS unter anderem in intern verwendeten Operatoren schon eine große Reihe von Fähigkeiten umsetzen, im übrigen beste Voraussetzungen für eine wirklich vollständige Implementation mitbringen und nur wenige Erweiterungen erfordern.

Zur Optimierung der Anfragen auf komplexen Objekten und insbesondere zur Optimierung ihrer Speicherung in Hinblick auf gegebene Arbeitslasten wurde ein auch im Rahmen dieser Arbeit entwickeltes Kostenmodell in Grundzügen vorgestellt. Dieses wurde durch die Gegenüberstellung mit Messungen am DBMS Oracle und mit Simulationsergebnissen validiert. Die Simulationsergebnisse wurden mit Hilfe einer Simulationsumgebung zur Speicherung, Indexierung und Verarbeitung komplexer Objekte ermittelt, die ebenfalls bei den Arbeiten zu dieser Dissertation entstanden ist und gleichfalls präsentiert wurde. Anhand der Simulationen konnte so auch demonstriert werden, welches Potential in der Optimierung der Speicherstrukturen liegt.

Abschließend wurde noch ein Ausblick auf die Möglichkeit der automatischen Optimierung physischer Strukturen gegeben. Durch die Beobachtung und Bewertung der Anfragen und der Speicherstrukturen durch das DBMS selbst und durch die Möglichkeit zur physischen Reorganisation ohne Beeinträchtigung der logischen Modellebene und der Anwendungsebene eröffnet sich ein interessanter Weg hin zu ‚self-tuning' ORDBMS.

7.3 Fazit

Durch die Untersuchung der Speicherungs- und Indexierungskonzepte wurde eine auf dem aktuellen Stand der Forschung aufsetzende, solide Basis für die physische Modellierung komplexer Objektstrukturen erarbeitet.

Weiterhin wurde mit der Definition der darauf aufbauenden Speicherbeschreibungssprache PRDL eine Möglichkeit zur Trennung von logischer und physischer Datenmodellierung und zur umfassenden Spezifikation von Speicherstrukturen in ORDBMS geschaffen. Auf diese Weise konnte eine Chance zur Verbesserung der Datenunabhängigkeit in objektrelationalen DBMS aufgezeigt werden.

Gleichzeitig bietet sich mit PRDL das Potential zur Optimierung physischer Speicherstrukturen durch die Anpassung an spezielle Datenbestände und Arbeitslasten und somit letztendlich zur Verbesserung der Effizienz objektrelationaler DBMS.

Die weiterführenden Untersuchungen zur optimierten Verarbeitung komplexer Objekte weisen einen Weg zu einer zukünftigen automatischen Optimierung und Reorganisation der Speicher- und Indexstrukturen für komplexe Objekte in ORDBMS.

Die Ergebnisse der vorliegenden Arbeit zeigen einen gangbarer Pfad zur Verbesserung der Datenunabhängigkeit, zur Effizienzsteigerung, zur besseren praktischen Benutzbarkeit und damit zur Beseitigung wichtiger Hindernisse beim Einsatz objektrelationaler Datenbank-Management-Systeme auf.

7.4 Ausblick

Eine wesentliche, weiterführende Arbeit stellt die konkrete, wenigstens teilweise Implementierung von PRDL in einem objektrelationalen DBMS-Prototypen oder -Produkt dar. Obwohl sich eine solche Umsetzung im wesentlichen auf die oberen DBMS-Architekturschichten beschränkt, sollte, abgesehen vom sicher ganz erheblichen Aufwand, der hierzu notwendige Eingriff in das DBMS nicht unterschätzt werden. Allerdings verspricht die Umsetzung dafür auch eine erhebliche Verbesserung der Einsetzbarkeit von ORDBMS. Relevante Arbeitsbereiche sind unter anderem

- ▷ Implementierung der vorgestellten erweiterten physischen Speicher- und Indexstrukturen in einem ORDBMS,
- ▷ Realisierung einer Datenbankkatalogerweiterung zur Steuerung der Index- und Speicherstrukturen,
- ▷ Implementierung eines Übersetzers für PRDL und
- ▷ Erweiterung von PRDL um potentiell weitere relevante Speicher- und Indexstrukturen.

Daneben bieten sich auf den in Kapitel 6 angerissenen Gebieten weitere vielversprechende Arbeiten zur Optimierung der Verarbeitung und Speicherung komplexer Objekte und zur automatischen physischen Reorganisation in ORDBMS:

- ▷ Weitere Verfeinerung und Validierung des Kostenmodells zur Verarbeitung komplexer Objekte in ORDBMS
- ▷ Berechnung und Bewertung des physischen Reorganisationspotentials und -bedarfs (Arbeiten hierzu laufen teils in Zusammenhang mit [Dor06]) und
- ▷ Strategien zur Durchführung und zum Scheduling von online und offline Reorganisationen.

Wir glauben, daß Arbeiten auf diesen Gebieten ein erhebliches Forschungspotential bieten und ebenfalls wichtige Beiträge zur praktischen Durchsetzung objektrelationaler Datenbank-Management-Systeme und damit zur verbesserten Verarbeitung komplexer Objekte liefern können.

Anhang A

Beispielszenario

In den Beispielen dieser Arbeit wird in der Regel folgendes Szenario verwendet:

> Ein Unternehmen produziert eine große Anzahl verschiedener Werkstücke, die der heutigen Zeit entsprechend „on demand“ und „just in time“ für Kunden gefertigt werden. Dazu besitzt das Unternehmen eine Menge von Maschinen, die für verschiedene Bearbeitungsvorgänge eingesetzt werden können und unterschiedliche Fertigungskapazitäten besitzen.

Die zugehörigen Objektklassen und ihre Attribute werden zusammen mit einer kurzen Erläuterung nachfolgend aufgelistet:

- Klasse *Fertigungsmaschine*
  Beschreibung von Maschinen durch folgende Attribute:
  - *Maschinenidentifikationsnummer:* Eindeutige Identifikation der Maschine.
  - *Maschinenbezeichnung:* Textuelle Bezeichnung der Maschine.
  - *Bearbeitungsfähigkeiten:* Menge von Bearbeitungsvorgängen, die von der Maschine durchführbar sind. Dieses mengenwertige Attribut enthält eine eingeschachtelte unbenannte Struktur mit folgenden Attributen:
    - *Identifikationscode des Bearbeitungsvorgangs,* den die Maschine beherrscht.
    - *Fertigungskapazität* der Maschine für den jeweiligen Bearbeitungsvorgang.
  - *Standort* der Maschine, angegeben durch die Unterattribute:
    - *Werksgebäude.*
    - *Etage.*
    - *Abteilung.*
  - *Maschinenstatus:* arbeitet die Maschine gerade, ist sie unbeschäftigt, aber bereit oder ist sie wartungsbedingt oder aus sonstigen Gründen außer Betrieb.
  - *Maschinenlogbuch:* Liste von Wartungsmaßnahmen mit folgenden Unterattributen:
    - *Zeitraum der Wartung,* gekennzeichnet durch
      - *Beginn.*
      - *Ende.*
    - *Art der Wartungsmaßnahme:* Kürzel für die Wartungsmaßnahme zur Abrechnung.
    - *Beschreibung der Wartungsmaßnahme:* Freitext für Besonderheiten, Bemerkungen.

- ▷ Klasse *Werkstückart*
  Beschreibung der herstellbaren Produkte und Produktteile.
  - ◇ *Typencode des Werkstücks.*
  - ◇ *Bezeichnung des Werkstücks:* Textuelle Bezeichnung.
  - ◇ *Herstellungsablauf:* Liste von notwendigen Fertigungsschritten:
    - ○ *Identifikationscode des Bearbeitungsvorgangs.*
    - ○ *Zeiteinordnung des Fertigungsschritts,* mit folgenden Attributen:
      - · *Minimaler Zeitabstand zum vorherigen Fertigungsschritt,* der eingehalten werden muß.
      - · *Maximaler Zeitabstand zum vorherigen Fertigungsschritt,* der nicht überschritten werden darf.
    - ○ *Bearbeitungskonfiguration:* Menge von Einrichtungsparametern für die Fertigungsmaschine zur Durchführung des Bearbeitungsvorgangs mit folgenden Unterattributen:
      - · *Parametername.*
      - · *Parametertyp:* numerisch oder textuell.
      - · *Parameterwert:* numerischer Wert oder Zeichenkette.

- ▷ Klasse *Einfache Werkstückart:* abgeleitet von Werkstückart

- ▷ Klasse *Zusammengesetzte Werkstückart:* abgeleitet von Werkstückart mit folgenden zusätzlichen Attributen:
  - ◇ *Zu verarbeitende Bestandteile:* Menge von Bestandteil-Werkstücken mit folgender Angabe:
    - ○ *Typencode des Bestandteil-Werkstücks*: Logische Referenz.
    - ○ *Benötigte Anzahlen in den Fertigungsschritten:* Feld mit Fertigungsschritten als Index und Anzahlen als Werten.

Die Erzeugung der entsprechenden Datentypen sieht in SQL$^+$ [Luf02a] so aus:

```
CREATE TYPE Fertigungsmaschine_t AS (
    Maschinen_Id                    VARCHAR(15) NOT NULL PRIMARY KEY,
    Maschinenbezeichnung            VARCHAR(100),
    Bearbeitungsfähigkeiten         SET(Bearbeitungsfähigkeit_t),
    Standort                        STANDORT_t,
    Maschinenstatus                 VARCHAR(20),
    Maschinenlogbuch                LIST(Maschinenlogbucheintrag_t) );

CREATE TYPE Werkstückart_t AS (
    Typencode                       VARCHAR(25) NOT NULL PRIMARY KEY,
    Bezeichnung                     VARCHAR(100),
    Herstellungsablauf              LIST(Fertigungsschritt_t)       );

CREATE TYPE EinfacheWerkstückart_t UNDER Werkstückart_t;

CREATE TYPE ZusammengesetzteWerkstückart_t UNDER Werkstückart_t AS (
    ZuVerarbeitendeBestandteile     Bestandteil_t                   );
```

Dabei kommen folgende Hilfstypen zum Einsatz:

```
CREATE TYPE Bearbeitungsfähigkeit_t AS (
    Id_Bearbeitungsvorgang          VARCHAR(20),
                                    REFERENCES Bearbeitungsvorgang_t,
    Fertigungskapazität             FLOAT                           );

CREATE TYPE Standort_t AS (
    Werksgebäude                    VARCHAR(10),
    Etage                           INTEGER,
    Abteilung                       VARCHAR(10)                     );

CREATE TYPE Maschinenlogbucheintrag_t AS (
    Zeitraum_Wartung                Zeitraum_t,
    Art_Wartungsmaßnahme            VARCHAR(10),
    Beschreibung_Wartungsmaßnahme   VARCHAR(2000)                   );

CREATE TYPE Einrichtungsparameter_t AS
    Parametername                   VARCHAR(50),
    Parametertyp                    VARCHAR(20),
    Parameterwert_F                 FLOAT,
    Parameterwert_V                 VARCHAR(100)                    );

CREATE TYPE Bestandteil_t AS (
    Typencode                       VARCHAR(25),
                                    REFERENCES Werkstückart_t,
    BenötigteAnzahlen               ARRAY[1..n](INTEGER)            );

CREATE TYPE Zeitraum_t AS (
    Beginn                          DATE,
    Ende                            DATE                            );

CREATE TYPE Fertigungsschritt_t AS (
    Id_Bearbeitungsvorgang          VARCHAR(20),
                                    REFERENCES Bearbeitungsvorgang_t,
    Zeiteinordnung                  Zeiteinordnung_t,
    Bearbeitungskonfiguration       SET(Einrichtungsparameter_t)    );

CREATE TYPE Zeiteinordnung_t AS (
    MinimalerZeitabstand            TIME INTERVAL,
    MaximalerZeitabstand            TIME INTERVAL                   );
```

Das Beispiel soll allerdings nicht vollständig definiert werden. Deshalb wird für den Typ `Bearbeitungsvorgang` nur ein Fragment angegeben:

```
CREATE TYPE Bearbeitungsvorgang_t AS (
    Id_Code                         VARCHAR(20) NOT NULL PRIMARY KEY,
    ...                                                             );
```

Ausgehend von diesen Typen können folgende typisierte Tabellen erzeugt werden:

```
CREATE TABLE Fertigungsmaschine AS Fertigungsmaschine_t

CREATE TABLE Werkstückart AS Werkstückart_t

CREATE TABLE Bearbeitungsvorgang AS Bearbeitungsvorgang_t
```

Anhang B

PRDL-Syntax

Dieser Anhang präsentiert die Syntax der in Abschnitt 5.2 vorgeschlagenen Physical Representation Definition Language (PRDL) mittels einer Grammatik in Backus-Naur-Form (BNF). Die Regeln (auch Produktionen) sind in alphabetischer Reihenfolge aufgeführt.

Die verwendete Notation sei hier noch einmal kurz erläutert: Der Name von Syntaxelementen (Nicht-Terminale) wird in '< >' eingeschlossen, Schlüsselwörter (Terminale) werden groß geschrieben. Mit ‚::=' wird innerhalb einer Produktion das definierte Element (links) von der Definition (rechts) getrennt. Innerhalb der Definitionen spezifizieren die eckigen Klammern ‚[ ]' optionale Elemente, mit ‚{ }' werden Elemente gruppiert. Wiederholungen werden mit ‚{ }*' beziehungsweise mit ‚{ }+' notiert, wobei das eingeschlossene Element beliebig oft auftauchen kann, aber bei ‚+' mindestens einmal auftauchen muß. Alternativen werden durch ‚|' getrennt, sie haben bei der Schachtelung niedrigste Priorität. Bei Bedarf wird die Schachtelung mit runden Klammern ‚()' explizit angegeben.

<attribute based storage options>, SYNTAXREGEL 8, *Seite 156*

```
<attribute based storage options> ::=
      INDEXORGANIZED [ <indexorganized options> ]
    | IN CLUSTER <schema qualified cluster name>
        ( <cluster key list> )
    | IN REFERENCE CLUSTER WITH <attribute identifier>
```

<attribute identifier>, SYNTAXREGEL 13, *Seite 159*

```
<attribute identifier> ::=
      <attribute name> | <attribute position>
```

<attribute list>, SYNTAXREGEL 26, *Seite 168*

```
<attribute list> ::=
    ( <attribute locator> { , <attribute locator> }* )
```

<attribute locator>, SYNTAXREGEL 12, *Seite 159*

```
<attribute locator> ::=
    <attribute identifier> { . <attribute identifier> }*
```

<attribute storage clause>, SYNTAXREGEL 23, *Seite 164*

```
<attribute storage clause> ::=
      REFERENCE <reference storage clause>
    | STRUCTURE <structure storage clause>
    | COLLECTION <collection storage clause>
```

<backward reference clause>, SYNTAXREGEL 28, *Seite 169*

```
<backward reference clause> ::=
    BACKWARD REFERENCE [ <reference storage clause>]
    [ TO ( <extended reference targets> ) ]
```

<clustered storage options>, SYNTAXREGEL 32, *Seite 171*

```
<clustered storage options> ::=
     IN PRIMARY RECORD CLUSTER | IN SUPERIOR RECORD CLUSTER
```

<cluster key list>, SYNTAXREGEL 42, *Seite 180*

```
<cluster key list> ::=
    <cluster key> { , <cluster key> }*
```

<cluster key>, SYNTAXREGEL 43, *Seite 180*

```
<cluster key> ::=
     <extended identifier chain> | <scalar value expression>
```

<cluster name>, SYNTAXREGEL 40, *Seite 178*

```
<cluster name> ::=
    <local or schema qualified name>
```

<collection array overflow record>, SYNTAXREGEL 48, *Seite 185*

```
<collection array overflow record> ::=
    OVERFLOW RECORD
        AT <element number> [ ELEMENTS | BYTES ]
        [ { MOVE | KEEP } FIRST ]
        [ <record storage option> ]
```

<collection array size>, SYNTAXREGEL 49, *Seite 185*

```
<collection array size> ::=
    [ FIXED | MAX ] SIZE IS <unsigned integer> [ BYTES ]
```

<collection array>, SYNTAXREGEL 47, *Seite 184*

```
<collection array> ::=
    ARRAY [ <collection array overflow record> ]
        [ <order by clause> ] [ <collection array size> ]
        OF ( {
          <external collection>
        | <collection element>
        | <collection element reference list> } )
```

<collection btree>, SYNTAXREGEL 52, *Seite 187*

```
<collection btree> ::=
    B-TREE [ <index key clause> ] [ <index type option> ]
        OF ( {
              <external collection>
            | <collection element>
            | <collection element reference list> } )
```

<collection element reference list>, SYNTAXREGEL 53, *Seite 188*

```
<collection element reference list> ::=
    <extended reference targets>
```

<collection element>, SYNTAXREGEL 46, *Seite 183*

```
<collection element> ::=
      EXTERNAL ELEMENT <reference clause>
    | INLINE ELEMENT [ <backward reference clause> ]
```

<collection hash table>, SYNTAXREGEL 56, *Seite 191*

```
<collection hash table> ::=
    HASH TABLE [ <hash index keys> ] [ <hash variants> ]
        OF ( {
              <external collection>
            | <collection element>
            | <collection element reference list> } )
```

<collection hbi>, SYNTAXREGEL 63, *Seite 195*

```
<collection hbi> ::=
    HIERARCHICAL BITMAP INDEX
    [ <superimposing option> ] '<index key clause>'
```

<collection linked list>, SYNTAXREGEL 50, *Seite 186*

```
<collection linked list> ::=
    LINKED LIST [ <linked list chaining> ] [ <order by clause> ]
        OF ( {
          <collection array>
        | <external collection>
        | <collection element> } )
```

<collection of records>, SYNTAXREGEL 44, *Seite 182*

```
<collection of records> ::=
    COLLECTION OF RECORDS <backward reference clause>
        [ <record storage option> ]
```

<collection rdtree>, SYNTAXREGEL 55, *Seite 189*

```
<collection rdtree> ::=
    RD-TREE <index key clause> [ <order by clause> ] [ INVERT INDEX ]
        OF ( {
              <external collection>
            | <collection element>
            | <collection element reference list> } )
```

<collection reference target>, SYNTAXREGEL 34, *Seite 173*

```
<collection reference target> ::=
    {  <collection array> [ <long record storage clause> ]
     | <collection linked list>
     | <collection btree>
     | <collection rdtree>
     | <collection hash table> [ <long record storage clause> ]
     | <collection suffix tree>
     | <collection signature file> [ <long record storage clause> ]
     | <collection signature tree>
     | <collection hbi>
    } [ <secondary record storage clause> ]
```

<collection signature file>, SYNTAXREGEL 65, *Seite 196*

```
<collection signature file> ::=
    SIGNATURE FILE [ <order by clause> ] <signature clause>
        OF ( {
          <external collection>
        | <collection element>
        | <collection element reference list> } )
```

<collection signature>, SYNTAXREGEL 60, *Seite 193*

```
<collection signature> ::=
    SIGNATURE [ <signature type> ] <index key clause>
```

<collection signature tree>, SYNTAXREGEL 67, *Seite 197*

```
<collection signature tree> ::=
    SIGNATURE TREE <signature clause> [ <superimposing option> ]
        OF ( {
          <external collection>
        | <collection element>
        | <collection element reference list> } )
```

<collection storage clause>, SYNTAXREGEL 33, *Seite 172*

```
<collection storage clause> ::=
    IS {  <collection array>
        | <collection hbi>
        | <external collection>
        | <collection of records> }
```

<collection suffix tree>, SYNTAXREGEL 59, *Seite 193*

```
<collection suffix tree> ::=
    SUFFIX TREE [ <index key clause> ] [ <order by clause> ]
        OF ( <collection element reference list> )
```

<create index>, SYNTAXREGEL 4, *Seite 153*

```
<create index> ::=
    CREATE INDEX <index name>
        [ FOR EACH <basic identifier chain> ]
        ON <basic identifier chain>
        {   (<sort specification list>)
          | (<sort specification list>) AS <external collection>
          | AS <external collection> }
```

<default signature options>, SYNTAXREGEL 62, *Seite 194*

```
<default signature options> ::=
    SIZE IS <unsigned integer> BITS
    [ ELEMENT SIZE IS <unsigned integer> BITS ]
```

<extended attribute identifier>, SYNTAXREGEL 22, *Seite 163*

```
<extended attribute identifier> ::=
    <attribute identifier> | ELEMENT
```

<extended identifier chain>, SYNTAXREGEL 41, *Seite 178*

```
<extended identifier chain> ::=
    { ROOT | SUPERIOR | ELEMENT | <identifier> }
    { . { ROOT | SUPERIOR | ELEMENT | <identifier> } }*
```

<extended reference targets>, SYNTAXREGEL 29, *Seite 169*

```
<extended reference targets> ::=
    <extended identifier chain> { , <extended identifier chain> }*
```

<extended tablespace identifier>, SYNTAXREGEL 18, *Seite 162*

```
<extended tablespace identifier> ::=
      <tablespace identifier>
    | SUPERIOR RECORD TABLESPACE
    | TABLESPACE OF <extended identifier chain>
```

<external collection>, SYNTAXREGEL 45, *Seite 183*

```
<external collection> ::=
    EXTERNAL <reference clause>
    STORED AS ( <collection reference target> )
```

<hash index keys>, SYNTAXREGEL 57, *Seite 191*

```
<hash index keys> ::=
    <index key clause> | <signature clause>
```

<hash variants>, SYNTAXREGEL 58, *Seite 192*

```
<hash variants> ::=
    USE
          EXTENDIBLE HASHING [ <extendible hashing options> ]
        | RECURSIVE LINEAR HASHING
            [ <recursive linear hashing options> ]
        | <...>
```

<index key clause>, SYNTAXREGEL 36, *Seite 176*

```
<index key clause> ::=
    INDEX KEY IS ( { <index key list> | <primary key> }
```

<index key list>, SYNTAXREGEL 38, *Seite 177*

```
<index key list> ::=
    { <extended identifier chain> | <value expression> }
    { , { <extended identifier chain> | <value expression> } }*
```

<indexorganized options>, SYNTAXREGEL 10, *Seite 158*

```
<indexorganized options> ::=
      IN <tablespace identifier>
    | IN <tablespace identifier> WITH <sort index key>
    | WITH <sort index key>
```

<index type option>, SYNTAXREGEL 54, *Seite 188*

```
<index type option> ::=
    AS [DISTINCT] { PATH INDEX | MULTIINDEX }
```

<linked list chaining>, SYNTAXREGEL 51, *Seite 187*

```
<linked list chaining> ::=
    SINGLE LINKED | DOUBLE LINKED
```

<long record storage clause>, SYNTAXREGEL 14, *Seite 160*

```
<long record storage clause> ::=
    FOR LONG RECORDS USE {
          CHAINING
        | OVERFLOW RECORD <overflow record options> }
```

<named attribute storage clause>, SYNTAXREGEL 20, *Seite 162*

```
<named attribute storage clause> ::=
    <target attribute locator> : <attribute storage clause>
```

<named secondary record clause>, SYNTAXREGEL 68, *Seite 198*

```
<named secondary record clause> ::=
    <record name> : AS SECONDARY RECORD
        [ UNDER <record to extend> ]
        <attribute list>
        [ <long record storage clause> ]
        [ <secondary record storage clause> ]
        <reference clause>
```

<named storage clause>, SYNTAXREGEL 19, *Seite 162*

```
<named storage clause> ::=
      <named attribute storage clause>
    | <named secondary record clause>
```

<object storage clause>, SYNTAXREGEL 5, *Seite 155*

```
<object storage clause> ::=
      <primary record storage clause>
    | <primary record storage clause> <long record storage clause>
    | <long record storage clause>
```

<order by clause>, SYNTAXREGEL 35, *Seite 176*

<order by clause> ::=
ORDER BY ({ <sort specification list> | <primary key> })

<overflow constraint>, SYNTAXREGEL 16, *Seite 161*

<overflow constraint> ::=
WHEN RECORD SIZE EXCEEDS { <unsigned integer> BYTES
| <unsigned numeric literal> OF PAGE SIZE }

<overflow record options>, SYNTAXREGEL 15, *Seite 160*

<overflow record options> ::=
[<overflow constraint>] [<overflow storage clause>]

<overflow storage clause>, SYNTAXREGEL 17, *Seite 161*

<overflow storage clause> ::=
STORE IN
<extended tablespace identifier>
| SUPERIOR RECORD CLUSTER

<prdl specification>, SYNTAXREGEL 3, *Seite 152*

<prdl specification> ::=
BEGIN PRDL
<object storage clause>
| [<object storage clause>] { <named storage clause> }+
END PRDL

<primary key>, SYNTAXREGEL 37, *Seite 176*

<primary key> ::=
PRIMARY KEY [ASC | DESC]

<primary record storage clause>, SYNTAXREGEL 6, *Seite 155*

<primary record storage clause> ::=
STORE <primary record storage options>

<primary record storage options>, SYNTAXREGEL 7, *Seite 156*

<primary record storage options> ::=
IN <tablespace identifier>
| <attribute based storage options>

<record name>, SYNTAXREGEL 69, *Seite 199*

<record name> ::=
<local or schema qualified name>

<record storage option>, SYNTAXREGEL 39, *Seite 177*

<record storage option> ::=
STORE IN {
CLUSTER <cluster name> (<cluster key list>)
| <tablespace identifier>
| TABLESPACE OF <extended identifier chain> }

<record to extend>, SYNTAXREGEL 70, *Seite 199*

```
<record to extend> ::=
      PRIMARY RECORD
    | <record name>
    | <attribute identifier> { . <extended attribute identifier> }*
```

<reference clause>, SYNTAXREGEL 27, *Seite 168*

```
<reference clause> ::=
    [ REFERENCE [ <reference storage clause> ] ]
    [ <backward reference clause> ]
```

<reference storage clause>, SYNTAXREGEL 24, *Seite 165*

```
<reference storage clause> ::=
      IS PHYSICAL
    | IS LOGICAL
        [ USE { SYSTEM GENERATED | PRIMARY } KEY ]
        [ { WITHOUT | WITH } PPP ]
```

<secondary record storage clause>, SYNTAXREGEL 30, *Seite 170*

```
<secondary record storage clause> ::=
    STORE <secondary record storage options>
```

<secondary record storage options>, SYNTAXREGEL 31, *Seite 170*

```
<secondary record storage options> ::=
      IN <extended tablespace identifier>
    | <attribute based storage options>
    | <clustered storage options>
```

<signature clause>, SYNTAXREGEL 66, *Seite 196*

```
<signature clause> ::=
    <collection hbi> | <collection signature>
```

<signature type>, SYNTAXREGEL 61, *Seite 194*

```
<signature type> ::=
      [ TYPE DEFAULT ] <default signature options>
    | <signature type ... and options>
```

<sort index key>, SYNTAXREGEL 11, *Seite 159*

```
<sort index key> ::=
      PRIMARY KEY
    | ( <attribute locator> { , <attribute locator> }* )
```

<structure storage clause>, SYNTAXREGEL 25, *Seite 166*

```
<structure storage clause> ::=
    {  IS SECONDARY RECORD
     | HAS SECONDARY RECORD <attribute list> }
    [ <long record storage clause> ]
    [ <secondary record storage clause> ]
    <reference clause>
```

<superimposing option>, SYNTAXREGEL 64, *Seite 195*

<superimposing option> ::=
SUPERIMPOSING FACTOR IS <unsigned integer>

<table definition>, SYNTAXREGEL 2, *Seite 151*

<table definition> ::=
<SQL table definition> [<prdl specification>]

<tablespace identifier>, SYNTAXREGEL 9, *Seite 157*

<tablespace identifier> ::=
TABLESPACE <local or schema qualified name>
| DEFAULT TABLESPACE

<target attribute locator>, SYNTAXREGEL 21, *Seite 163*

<target attribute locator> ::=
<attribute identifier> { . <extended attribute identifier> }*
| TABLE

<type definition>, SYNTAXREGEL 1, *Seite 151*

<type definition> ::=
<SQL type definition> [<prdl specification>]

Literaturverzeichnis

[AB84] S. Abiteboul und N. Bidoit. Non First Normal Form Relations to Represent Hierarchical Organized Data. In *Proceedings of the Third ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, Waterloo, Ontario, Canada*, PODS 1984.

[ABC+76] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade und V. Watson. System R: Relational approach to database management. *ACM Transactions on Database Systems*, 1(2), 1976.

[Bat86a] D. S. Batory. Extensible Cost Models and Query Optimization in GENESIS. *IEEE Database Engineering Bulletin*, 9(4), 1986.

[Bat86b] D. S. Batory. GENESIS: A Project to Develop an Extensible Database Management System. In *Proceedings of the International Workshop on Object-Oriented Database Systems, Pacific Grove, California, USA*, OODBS 1986.

[BDH92] V. Benzaken, C. Delobel und G. Harrus. Clustering Strategies in O_2: An Overview. In F. Bancilhon et al. (Hrsg.), *Building an Object-Oriented Database System: The Story of O_2*. 1992.

[BK89] E. Bertino und W. Kim. Indexing Techniques for Queries on Nested Objects. *IEEE Transactions on Knowledge and Data Engineering*, 1(2), 1989.

[Böh00] C. Böhm. A cost model for query processing in high dimensional data spaces. *ACM Transactions on Database Systems*, 25(2), 2000.

[CBBC94] S. Choenni, E. Bertino, H. M. Blanken und T. Chang. On the Selection of Optimal Index Configuration in OO Databases. In *Proceedings of the Tenth International Conference on Data Engineering, Houston, Texas, USA*, ICDE 1994.

[CCN+99] M. J. Carey, D. D. Chamberlin, S. Narayanan, B. Vance, D. Doole, S. Rielau, R. Swagerman und N. M. Mattos. O-O, What Have They Done to DB2? In *Proceedings of 25th International Conference on Very Large Data Bases, Edinburgh, Scotland, UK*, VLDB 1999.

[CDF+82] A. Chan, S. Danberg, S. Fox, W.-T. K. Lin, A. Nori und D. R. Ries. Storage and Access Structures to Support a Semantic Data Model. In *Proceedings of the Eigth International Conference on Very Large Data Bases, Mexico City, Mexico*, VLDB 1982.

[CDV88] M. J. Carey, D. J. DeWitt und S. L. Vandenberg. A Data Model and Query Language for EXODUS. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois*, SIGMOD 1988.

[CG94] R. L. Cole und G. Graefe. Optimization of Dynamic Query Evaluation Plans. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota*, SIGMOD 1994.

[Chr84] S. Christodoulakis. Implications of Certain Assumptions in Database Performance Evaluation. *ACM Transactions on Database Systems*, 9(2), 1984.

[COD70] CODASYL. Task Group of CODASYL System Committee. Storage Structure Definition Language (SSDL). In *Record of the ACM SIGFIDET Workshop on Data Description and Access, Houston, Texas, USA, Second Edition with an Appendix*, SIGMOD 1970.

[COD78] CODASYL. Reports of the Data Description Language Committee. *Information Systems*, 3(4), 1978.

[CS89] W. W. Chang und H.-J. Schek. A Signature Access Method for the Starburst Database System. In *Proceedings of the 15th International Conference on Very Large Data Bases, Amsterdam, Netherlands*, VLDB 1989.

[DCC+01] S. Deßloch, W. Chen, J.-H. Chow, Y.-C. Fuh, J. Grandbois, M. Jou, N. M. Mattos, R. Nitzsche, B. T. Tran und Y. Wang. Extensible Indexing Support in DB2 Universal Database. In K. R. Dittrich et al. (Hrsg.), *Component Database Systems*. 2001.

[Dep86] U. Deppisch. S-Tree: A Dynamic Balanced Signature Index for Office Retrieval. In *Proceedings of the 9th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, Pisa, Italy*, SIGIR 1986.

[Deu90] O. Deux. The Story of O2. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), 1990.

[DKA+86] P. Dadam, K. Küspert, F. Andersen, H. M. Blanken, R. Erbe, J. Günauer, V. Y. Lum, P. Pistor und G. Walch. A DBMS Prototype to Support Extended NF^2 Relations: An Integrated View on Flat Tables and Hierarchies. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Washington, D.C.*, SIGMOD 1986.

[DLM96] D. Dervos, P. Linardis und Y. Manolopoulos. Perfect Encoding: a Signature Method for Text Retrieval. In *Proceedings of the Third International Workshop on Advances in Databases and Information Systems, Moscow, Russia*, ADBIS 1996.

[DLM97] D. Dervos, P. Linardis und Y. Manolopoulos. S-Index: a Hybrid Structure for Text Retrieval. In *Proceedings of the First East-European Symposium on Advances in Databases and Information Systems, St. Petersburg, Russia*, ADBIS 1997.

[Dor03] S. Dorendorf. Reorganisationsbedarfsanalysen bei relationalen Datenbankmanagementsystemen unter Beachtung der Workload. *Datenbank-Spektrum*, 5, 2003.

[Dor05] S. Dorendorf. Quantifizierung des zu erwartenden Nutzens von Datenbankreorganisationen. *Informatik – Forschung und Entwicklung*, 21(1–2), 2005.

[Dor06] S. Dorendorf. *Reorganisationsbedarf bei relationalen Datenbank-Management-Systemen.* Dissertation, Universität Jena, 2006. In Vorbereitung.

[DPS86] U. Deppisch, H.-B. Paul und H.-J. Schek. A Storage System for Complex Objects. In *Proceedings of the International Workshop on Object-Oriented Database Systems, Pacific Grove, California, USA*, OODBS 1986.

[EH84] W. Effelsberg und T. Härder. Principles of Database Buffer Management. *ACM Transactions on Database Systems*, 9(4), 1984.

[FC84] Ch. Faloutsos und S. Christodoulakis. Signature files: an access method for documents and its analytical performance evaluation. *ACM Transactions on Information Systems*, 2(4), 1984.

[FDC+99] Y.-C. Fuh, S. Deßloch, W. Chen, N. M. Mattos, B. T. Tran, B. G. Lindsay, L. DeMichel, S. Rielau und D. Mannhaupt. Implementation of SQL3 Structured Types with Inheritance and Value Substitutability. In *Proceedings of the 25th International Conference on Very Large Data Bases, Edinburgh, Scotland*, VLDB 1999.

[FNPS79] R. Fagin, J. Nievergelt, N. Pippenger und H. R. Strong. Extendible Hashing – A Fast Access Method for Dynamic Files. *ACM Transactions on Database Systems*, 4(3), 1979.

[FS75] G. Farley und S. A. Schuster. Query Execution and Index Selection for Relational Data Bases. In *Proceedings of the International Conference on Very Large Data Bases, Framingham, Massachusetts*, VLDB 1975.

[FSC04] Fujitsu Siemens Computers. *UDS: Entwerfen und Definieren, UDS/SQL V2.4 (BS2000/SQL)*, 2004.

[Ges97] M. Gesmann. A Cost Model for Parallel Navigational Access in Complex-Object DBMSs. In *Proceedings of the Fifth International Conference on Database Systems for Advanced Applications, Melbourne, Australia*, DASFAA 1997.

[GGH+92] M. Gesmann, A. Grasnickel, T. Härder, Ch. Hübel, W. Käfer, B. Mitschang und H. Schöning. PRIMA - A Database System Supporting Dynamically Defined Composite Objects. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, San Diego, California*, SIGMOD 1992.

[GGS96] S. Ganguly, A. Goel und A. Silberschatz. Efficient and Acurate Cost Models for Parallel Query Optimization. In *Proceedings of the Fifteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Montreal, Canada*, PODS 1996.

[GGT95] G. Gardarin, J.-R. Gruser und Z.-H. Tang. A Cost Model for Clustered Object-Oriented Databases. In *Proceedings of the 21st International Conference on Very Large Data Bases, Zurich, Switzerland*, VLDB 1995.

[GN93] D. Gardy und L. Nemirovski. Urn Models and Yao's Formula. In *Proceedings of the 7th International Conference on Database Theory, Jerusalem, Israel*, ICDT 1993.

[Gut84] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts*, SIGMOD 1984.

[Här75a] T. Härder. *Modellierung des Leistungsverhaltens von DBMS-Komponenten.* Dissertation, TH Darmstadt, 1975.

[Här75b] T. Härder. Implementierung von Zugriffspfaden durch Bitlisten. In *5. GI-Jahrestagung, Dortmund, Germany*, GI 1975.

[Här77] T. Härder. A Scan-Driven Sort Facility for a Relational Database System. In *Proceedings of the Third International Conference on Very Large Data Bases, Tokyo, Japan*, VLDB 1977.

[Här78] T. Härder. Implementing a Generalized Access Path Structure for a Relational Database System. *ACM Transactions on Database Systems*, 3(3), 1978.

[HB04] T. Härder und A. Bühmann. Database Caching – Towards a Cost Model for Populating Cache Groups. In *Proceedings of the 8th East-European Conference on Advances in Databases and Information Systems, Budapest, Hungary*, ADBIS 2004.

[Hel97] S. Helmer. *Index structures for databases containing data items with setvalued attributes.* Technischer Bericht, Universität Mannheim, 1997.

[HFLP89] L. M. Haas, J. C. Freytag, G. M. Lohman und H. Pirahesh. Extensible Query Processing in Starburst. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Portland, Oregon*, SIGMOD 1989.

[HM99] S. Helmer und G. Moerkotte. *A Study of Four Index Structures for Set-Valued Attributes of Low Cardinality.* Technischer Bericht, Universität Mannheim, 1999.

[HMS91] T. Härder, B. Mitschang und H. Schöning. Query Processing for Complex Objects. *Data & Knowledge Engineering*, 7, 1991.

[HMWMS87] T. Härder, K. Meyer-Wegener, B. Mitschang und A. Sikeler. PRIMA - a DBMS Prototype Supporting Engineering Applications. In *Proceedings of 13th International Conference on Very Large Data Bases, Brighton, England*, VLDB 1987.

[HP94] J. M. Hellerstein und A. Pfeifer. *The RD- Tree: An Index Structure for Sets.* Technischer Bericht, University of Wisconsin at Madison, 1994.

[HR01] T. Härder und E. Rahm. *Datenbanksysteme, Konzepte und Techniken der Implementierung.* 2001.

[HS93] J. M. Hellerstein und M. Stonebraker. Predicate Migration: Optimizing Queries with Expensive Predicates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Washington, D.C.*, SIGMOD 1993.

[HS00] A. Heuer und G. Saake. *Datenbanken – Konzepte und Sprachen.* 2000.

[Hum06] S. Hummel. Zum Umgang mit den Begriffen. *Die Zusammensetzung und Übermittlung an den Autor zieht für die ersten drei fündigen Leser eine Einladung an eben diesen Ort nach sich*, gültig bis 2006.

[IBM00a] IBM Corporation. *DB2 Universal Database – Administration Guide, Version 7*, 2000.

[IBM00b] IBM Corporation. *DB2 Universal Database – SQL Reference, Version 7*, 2000.

[IBM02a] IBM Corporation. *DB2 Universal Database – Administration Guide – Implementation, Version 8*, 2002.

[IBM02b] IBM Corporation. *DB2 Universal Database – SQL Reference, Version 8*, 2002.

[IBM03a] IBM Corporation. *Informix Dynamic Server – Administratior's Guide, Version 9.4*, 2003.

[IBM03b] IBM Corporation. *Informix Dynamic Server – Administrator's Reference, Version 9.4*, 2003.

[IBM03c] IBM Corporation. *Informix Dynamic Server – Database Design and Implementation Guide, Version 9.4*, 2003.

[IBM03d] IBM Corporation. *Informix Dynamic Server – Guide to SQL – Syntax, Version 9.4*, 2003.

[IBM04] IBM Corporation. *IMS Administration Guide: Database Manager, Version 8*, 2004.

[IKO93] Y. Ishikawa, H. Kitagawa und N. Ohbo. Evaluation of Signature Files as Set Access Facilities in OODBs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Washington, D.C.*, SIGMOD 1993.

[Ing74] J. Inglis. Inverted Indexes and Multi-List Structures. *Computer Journal*, 17(1), 1974.

[INSS97] Y. E. Ioannidis, R. T. Ng, K. Shim und T. K. Sellis. Parametric Query Optimization. *VLDB Journal*, 6(2), 1997.

[Ioa97] Y. E. Ioannidis. Query Optimization. In A. B. Tucker et al. (Hrsg.), *The Computer Science and Engineering Handbook*. 1997.

[ISO99] ISO/IEC. *Information Technology – Database Languages – SQL – Part 2: Foundation (SQL/Foundation)*, 9075-2, 1999.

[JCJÖ95] I. Jacobson, M. Christerson, P. Jonsson und G. Övergaard. *Object-Oriented Software Engineering. A Use Case driven approach*. 1995.

[JK84] M. Jarke und J. Koch. Query Optimization in Database Systems. *ACM Computing Surveys*, 16(2), 1984.

[Kau02] C. Kauhaus. *Architektur einer Transformationsschicht zur Abbildung objektrelationaler Sprachmittel auf reale DBMS*. Diplomarbeit, Universität Jena, 2002.

[KCB87] W. Kim, H.-T. Chou und J. Banerjee. Operations and Implementation of Complex Objects. In *Proceedings of the Third International Conference on Data Engineering, Los Angeles, California, USA*, ICDE 1987.

[Keß95] U. Keßler. *Flexible Speicherstrukturen und Sekundärindexe in Datenbanken für komplexe Objekte.* Dissertation, Universität Ulm, 1995.

[KFIO93] H. Kitagawa, Y. Fukushima, Y. Ishikawa und N. Ohbo. Estimation of False Drops in Set-valued Object Retrieval with Signature Files. In *Proceedings of the 25th International Conference on Foundations of Data Organization ans Algorithms, Chicago, Illinois*, FODO 1993.

[KGBW90] W. Kim, J. F. Garza, N. Ballou und D. Woelk. Architecture of the ORION Next-Generation Database System. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), 1990.

[Kim80] W. Kim. A New Way to Compute the Product and Join of Relations. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Santa Monica, California*, SIGMOD 1980.

[Kis02] F. Kissel. *Physische Speicherung komplexer Objekte mit kollektionswertigen Attributen in ORDBMS.* Studienarbeit, Universität Jena, 2002.

[Kis04] F. Kissel. *Indexstrukturen in ORDBMS.* Diplomarbeit, Universität Jena, 2004.

[KKD87] W. Kim, K.-C. Kim und A. G. Dale. *Indexing Techniques for Object-oriented Databases.* Technischer Bericht, University of Texas at Austin, 1987.

[KKD89] W. Kim, K.-Ch. Kim und A. G. Dale. Indexing Techniques for Object-Oriented Databases. In W. Kim et al. (Hrsg.), *Object-Oriented Concepts, Databases, and Applications.* 1989.

[Kla03] D. Klan. *Entwurf und Implementierung einer Simulationsumgebung zur physischen Speicherung komplexer Objekte in ORDBMS.* Studienarbeit, Universität Jena, 2003.

[KLS02] C. Kauhaus, J. Lufter und S. Skatulla. Eine Transformationsschicht zur Realisierung objektrelationaler Datenbankkonzepte mit erweiterter Kollektionsunterstützung. *Datenbank-Spektrum*, 4, 2002.

[KM90] A. Kemper und G. Moerkotte. Access Support in Object Bases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ*, SIGMOD 1990.

[KM94] Ch. Kilger und G. Moerkotte. Indexing Multiple Sets. In *Proceedings of 20th International Conference on Very Large Data Bases, Santiago de Chile, Chile*, VLDB 1994.

[Knu97] D. E. Knuth. *The Art of Computer Programming.* 1997.

[Kuh02] S. Kuhlins. *Die C++ Standartbibliothek: Einführung und Nachschlagewerk.* 2002.

[KV87] S. Khoshafian und P. Valduriez. Sharing, Persistence, and Object-Orientation: A Database Perspective. In *Proceedings of the Workshop on Advances in Database Programming Languages, Roscoff, France*, DBPL 1987.

[Lac04] N. Lachmann. *Operationen auf komplexen Objekten in ORDBMS: Analyse, Vergleich und Optimierung.* Diplomarbeit, Universität Jena, 2004.

[LG98] P.-Å. Larson und G. Graefe. Memory Management During Run Generation in External Sorting. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Seattle, Washington*, SIGMOD 1998.

[LKD+88] V. Linnemann, K. Küspert, P. Dadam, P. Pistor, R. Erbe, A. Kemper, N. Südkamp, G. Walch und M. Wallrath. Design and Implementation of an Extensible Database Management System Supporting User Defined Data Types and Functions. In *14th International Conference on Very Large Data Bases, Los Angeles, California*, VLDB 1988.

[LL89] D. L. Lee und Ch.-W. Leng. Partitioned signature files: design issues and performance evaluation. *ACM Transactions on Information Systems*, 7(2), 1989.

[LL92] Ch.-W. Roger Leng und D. L. Lee. Optimal Weight Assignment for Signature Generation. *ACM Transactions on Database Systems*, 17(2), 1992.

[LOL92] Ch. Ch. Low, B. Ch. Ooi und H. Lu. H-trees: A Dynamic Associative Search Index for OODB. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, San Diego, California*, SIGMOD 1992.

[LRV88] C. Lécluse, P. Richard und F. Vélez. O_2, an Object-Oriented Data Model. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois*, SIGMOD 1988.

[Luf00] J. Lufter. *Prinzipien für Anfragen an Objekte in objektrelationalen Datenbanken.* Technischer Bericht, Universität Jena, 2000.

[Luf02a] J. Lufter. *Kollektionsunterstützung für SQL:1999.* Technischer Bericht, Universität Jena, 2002.

[Luf02b] J. Lufter. Abbildung normkonformer objektrelationaler Sprachmittel auf reale DBMS. In *14. GI-Workshop, Grundlagen von Datenbanken, Fischland-Daß-Zingst, Meklenburg-Vorpommern, Germany*, GvD 2002.

[Luf05] J. Lufter. *Unterstützung komplexer Datenstrukturen in SQL-Norm und objektrelationalen Datenbanksystemen.* Dissertation, Universität Jena, 2005.

[Mar04] R. Marhold. *Operationale Aspekte der physischen Speicherung komplexer Objekte mit verschachtelten kollektionswertigen Attributen.* Studienarbeit, Universität Jena, 2004.

[Mit88] B. Mitschang. *Ein Molekül-Atom-Datenmodell für Non-Standard-Anwendungen: Anwendungsanalyse, Datenmodellentwurf und Implementierungskonzepte.* 1988.

[Mit89] B. Mitschang. Extending the Relational Algebra to Capture Complex Objects. In *Proceedings of the 15th International Conference on Very Large Data Bases, Amsterdam, Netherlands*, VLDB 1989.

[ML89] L. F. Mackert und G. M. Lohman. Index Scans Using a Finite LRU Buffer: A Validated I/O Model. *ACM Transactions on Database Systems*, 14(3), 1989.

[MMNM03] M. Morzy, T. Morzy, A. Nanopoulos und Y. Manolopoulos. Hierarchical Bitmap Index: An Efficient and Scalable Indexing Technique for Set-Valued Attributes. In *Proceedings of the 7th East-European Conference on Advances in Databases and Information Systems, Dresden, Germany*, ADBIS 2003.

[Mor68] D. R. Morrison. PATRICIA – Practical Algorithm To Retrieve Information Coded in Alphanumeric. *Journal of the ACM*, 15(4), 1968.

[MS86] D. Maier und J. Stein. Indexing in an Object-Oriented DBMS. In *Proceedings of the International Workshop on Object-Oriented Database Systems, Pacific Grove, California, USA*, OODBS 1986.

[NY96] R. T. Ng und J. Yang. An Analysis of Buffer Sharing and Prefetching Techniques for Multimedia Systems. *Multimedia Systems*, 4(2), 1996.

[OL90] K. Ono und G. M. Lohman. Measuring the Complexity of Join Enumeration in Query Optimization. In *Proceedings of the 16th International Conference on Very Large Data Bases, Queensland, Australia*, VLDB 1990.

[Ora01a] Oracle Corporation. *Oracle 9i Administrator's Guide*, 2001.

[Ora01b] Oracle Corporation. *Oracle 9i Application Developer's Guide – Object-Relational Features*, 2001.

[Ora01c] Oracle Corporation. *Oracle 9i Database Concepts*, 2001.

[Ora01d] Oracle Corporation. *Oracle 9i SQL Reference*, 2001.

[Ora03] Oracle Corporation. *Oracle 10g Database Concepts*, 2003.

[Oto84] E. J. Otoo. A Mapping Function for the Directory of a Multidimensional Extendible Hashing. In *Proceedings of the Tenth International Conference on Very Large Data Bases, Singapore*, VLDB 1984.

[PA86] P. Pistor und F. Andersen. Designing A Generalized NF2 Model with an SQL-Type Language Interface. In *Proceedings of the 12th International Conference on Very Large Data Bases, Kyoto, Japan*, VLDB 1986.

[PBC80] J. L. Pfaltz, W. J. Berman und E. M. Cagley. Partial-match retrieval using indexed descriptor files. *Communications of the ACM*, 23(9), 1980.

[Pri02] P. Prinz. *C++ kennen und professionell anwenden.* 2002.

[PSS+87] H.-B. Paul, H.-J. Schek, M. H. Scholl, G. Weikum und U. Deppisch. Architecture and Implementation of the Darmstadt Database Kernel System. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, San Francisco, California*, SIGMOD 1987.

[RK84] A. Reuter und H. Kinzinger. Automatic Design of the Internal Schema for a CODASYL Database System. *IEEE Transactions on Software Engineering*, 10(4), 1984.

[RSD84] K. Ramamohanarao und R. Sacks-Davis. Recursive Linear Hashing. *ACM Transactions on Database Systems*, 9(3), 1984.

[SAC+79] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie und T. G. Price. Access Path Selection in a Relational Database Management System. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts*, SIGMOD 1979.

[Sch98] H. Schöning. The ADABAS Buffer Pool Manager. In *Proceedings of the 24th International Conference on Very Large Data Bases, New York City*, VLDB 1998.

[Sch02] K. Schwarz. *Objektrelationale Erweiterungen von Oracle, DB2 und Informix – Funktionaler Vergleich sowie Einsatz der Erweiterungen zur Verwaltung von Geo-Daten.* Diplomarbeit, Universität Jena, 2002.

[Sch04] R. Schmidt. *Vergleich alternativer physischer Strukturen für komplexe Objekte in ORDBMS.* Studienarbeit, Universität Jena, 2004.

[SD03] S. Skatulla und S. Dorendorf. Optimization of Storage Structures of Complex Types in Object-Relational Database Systems. In *Proceedings of the 7th East-European Conference on Advances in Databases and Information Systems, Dresden, Germany*, ADBIS 2003.

[SH99] G. Saake und A. Heuer. *Datenbanken – Implementierungstechniken.* 1999.

[Ska98] S. Skatulla. *SQL92-normkonforme Katalogsichten für DB2.* Studienarbeit, Universität Jena, 1998.

[Ska99a] S. Skatulla. *Selectivity of User-Defined Predicates in DB2 DataJoiner and Other Object-Relational DBMSs.* Diplomarbeit, Universität Jena, 1999.

[Ska99b] S. Skatulla. Selektivität nutzerdefinierter Prädikate in objektrelationalen DBMS. In *11. GI-Workshop, Grundlagen von Datenbanken, Luisenthal, Thüringen, Germany*, GvD 1999.

[Ska02] S. Skatulla. Storage of Complex Types with Collection-Valued Attributes in Object-Relational Database Systems. In *14. GI-Workshop, Grundlagen von Datenbanken, Fischland-Daß-Zingst, Meklenburg-Vorpommern, Germany*, GvD 2002.

[SS94] B. Sreenath und S. Seshadri. The hcC-tree: An Efficient Index Structure for Object Oriented Databases. In *Proceedings of 20th International Conference on Very Large Data Bases, Santiago de Chile, Chile*, VLDB 1994.

[SSPK00] S. Skatulla, R. Schaarschmidt, P. Pistor und K. Küspert. Entwurf und Implementierung eines SQL-normkonformen Datenbankkatalogs für ein relationales Datenbankmanagementsystem. *Informatik – Forschung und Entwicklung*, 15(3), 2000.

[Str00] B. Stroustrup. *Die C++ Programmiersprache.* 2000.

[STS97] G. Saake, C. Türker und I. Schmitt. *Objektdatenbanken: Konzepte, Sprachen, Architekturen.* 1997.

[TRSB93] W. B. Teeuw, Ch. Rich, M. H. Scholl und H. M. Blanken. An Evaluation of Physical Disk I/Os for Complex Object Processing. In *Proceedings of the Ninth International Conference on Data Engineering, 1993, Vienna, Austria*, ICDE 1993.

[Val87] P. Valduriez. Join Indices. *ACM Transactions on Database Systems*, 12(2), 1987.

[VKC86] P. Valduriez, S. Khoshafian und G. P. Copeland. Implementation Techniques of Complex Objects. In *Proceedings of the 12th International Conference on Very Large Data Bases, Kyoto, Japan*, VLDB 1986.

[Wal04] M. Walther. *Speicherung und Verwaltung komplexer Objekte in ORDBMS-Produkten und ORDBMS-Prototypen.* Studienarbeit, Universität Jena, 2004.

[Weg90] L. Wegner. *A Portable Record Manager.* Technischer Bericht, GH Kassel, 1990.

[Wei73] P. Weiner. Linear Pattern Matching Algorithm. In *Proceedings of the 14th Annual IEEE Symposium on Foundations of Computer Science, Iowa City, Iowa*, FOCS 1973.

[XH94] Z. Xie und J. Han. Join Index Hierarchies for Supporting Efficient Navigations in Object-Oriented Databases. In *Proceedings of 20th International Conference on Very Large Data Bases, Santiago de Chile, Chile*, VLDB 1994.

[Yao77a] S. B. Yao. An Attribute Based Model for Database Access Cost Analysis. *ACM Transactions on Database Systems*, 2(1), 1977.

[Yao77b] S. B. Yao. Approximating the Number of Accesses in Database Organizations. *Communications of the ACM*, 20(4), 1977.

[YM01] N. Yazdani und P. S. Min. Prefix Trees: New Efficient Data Structures for Matching Strings of Different Lengths. In *Proceedings of the International IEEE Database Engineering & Applications Symposium, Grenoble, France*, IDEAS 2001.

[ZRT91] P. Zezula, F. Rabitti und P. Tiberio. Dynamic partitioning of signature files. *ACM Transactions on Information Systems*, 9(4), 1991.

Selbständigkeitserklärung

Ich erkläre, daß ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Hilfsmittel und Literatur angefertigt habe.

Jena, Dezember 2005

Steffen Skatulla

Lebenslauf

Persönliche Daten

Name	Steffen Skatulla, geb. Hummel
Geburtsdatum	2. November 1971
Geburtsort	Jena

Werdegang

1978 – 1988	Polytechnische Oberschule „Dr. Friedrich Wolf“ in Jena
1988 – 1990	Abitur an der Erweiterten Oberschule „Johannes R. Becher“ in Jena
1990 – 1991	Studium der Informatik an der Universität Leipzig
1991 – 1993	Zivildienst
1993 – 1999	Studium der Informatik an der Friedrich-Schiller-Universität Jena, Nebenfach Architektur und Bauwesen, mit Unterbrechung durch Kindererziehungszeit Abschluß: Diplom-Informatiker
1996 – 1998	als Werkstudent kontinuierliche Mitarbeit am Forschungsprojekt ROCOCO der IBM am Wissenschaftlichen Zentrum Heidelberg, Studienarbeit unter Mitbetreuung der IBM
1998 – 1999	Internship am IBM Santa Teresa Lab, San Jose, CA, Diplomarbeit unter Mitbetreuung der IBM
1999 – 2004	Wissenschaftlicher Mitarbeiter am Lehrstuhl für Datenbanken und Informationssysteme der Friedrich-Schiller-Universität Jena, tätig teils in Kooperationsprojekten mit der IBM
seit 2005	Entwickler bei der IBYKUS AG für Informationssysteme, Erfurt

Jena, Dezember 2005

Steffen Skatulla