Objektorientierte Datenbanksysteme

Udo Kelter

18.04.2004

Zusammenfassung dieses Lehrmoduls

Objektorientierte Datenbanksysteme (OODBMS) versuchen, die Leistungen und Vorteile von Datenbanken und objektorientierter Programmierung zu vereinigen. Im Gegensatz zu konventionellen DBMS, bei denen die Datenstrukturen, also Schemata, offenliegen, werden die Daten in OODBMS durch Operationen verkapselt. Motiviert sind OODBMS vor allem bei nichtkonventionellen Anwendungen mit komplex strukturierten Daten. Dieses Lehrmodul stellt die wichtigsten Merkmale von OODBMS und Alternativen für deren Ausprägung vor, u.a. komplexe Objekte, Objektidentität, Kapselung und Verwaltung der Operationen u.a. Ferner werden einige einschlägige Standards skizziert.

Vorausgesetzte Lehrmodule:

obligatorisch:
- Datenverwaltungssysteme
- Architektur von DBMS

Stoffumfang in Vorlesungsdoppelstunden: 1.7

Inhaltsverzeichnis

1 Nichtkonventionelle Anwendungen

Das Hauptcharakteristikum objektorientierter Datenbankmanagementsysteme (**OODBMS**) besteht darin, daß ihr Datenbankmodell objektorientierte Konzepte beinhaltet, namentlich die Kapselung von Datenstrukturen durch Operationen[1], Typhierarchien und Polymorphie. Bei vielen (nicht allen) OODBMS ist nicht nur das Datenbankmodell nichtkonventionell, sondern dies gilt auch für Transaktions- und Verteilungskonzepte und andere Aspekte, so daß derartige Systeme auch nichtkonventionelle DBMS genannt werden. Wir untersuchen i.f. zunächst die Motivation für derartige DBMS und beschreiben danach nur die objektorientierten Datenbankmodelle; auf die anderen erwähnten nichtkonventionellen technischen Merkmale dieser DBMS gehen wir in diesem Lehrmodul nicht ein.

Objektorientierte bzw. nichtkonventionelle DBMS sind motiviert durch die Anforderungen, die nichtkonventionelle Anwendungen an die Datenverwaltung stellen. Wir listen zum Vergleich zunächst einige wesentliche Merkmale konventioneller Anwendungen (also i.w. übliche betriebliche Informationssysteme) auf:

- Es handelt sich um große Mengen relativ einfach und homogen strukturierter Daten, z.B. Buchungen.
- Einzelne Tupel (oder Records) sind relativ klein (wenige 100 Byte); oft werden Sätze fester Länge verwendet.
- Einzelne Datenfelder sind atomar (s. erste Normalform).
- Die Datenbankschemata werden sehr selten geändert.
- Ändernde Zugriffe zur Datenbank betreffen meist nur wenige Datengranulate; dementsprechend sind Transaktionen sehr kurz und dauern in der Größenordnung von 1 Sekunde.

Beispiele für nichtkonventionelle Anwendungen sind:

[1] Dies ist eigentlich kein originär objektorientiertes Konzept, es stammt aus der Konzeptwelt modularer Sprachen wie Modula-2.

- Technische Entwurfsumgebungen (CAD, CASE usw.): Als Datenarten treten hier Quell- und Binär-Programme, Dokumentation, Zeichnungen, Bitmaps usw. auf, die oft als Dokument aufgefaßt werden. Es müssen Versionen der Dokumente verwaltet werden. Dokumente können eine sehr komplexe Struktur, insb. hinsichtlich der Konsistenzbedingungen, haben.
- Multimedia-Datenbanken: Als Datenarten treten hier u.a. Video- und Audio-"Dateien" auf. Deren enorme Größe verbietet ein atomares Lesen oder Schreiben wie bei Tupeln. Stattdessen müssen die Daten mit bestimmten Datenraten über Netzwerkverbindungen abgeliefert werden (sog. *streaming* Formate).
- Büroinformationssysteme: Als Datenarten treten hier insb. Briefe, digitalisierte Papiervorlagen u.ä. auf. Die Suchfunktionen sind i.w. die aus dem Information Retrieval bekannten.
- Expertensystem-Datenbanken: diese speichern Fakten und Regeln; aus diesen können andere Fakten abgeleitet werden.

Konventionelle DBMS sind für diese Anwendungen nicht konzipiert worden und dort mehr oder weniger unbrauchbar, es treten die folgenden gravierenden Probleme auf:

- Die Daten der nichtkonventionellen Anwendungen können nicht auf natürliche Art und Weise modelliert werden. Wenn man z.B. eine Modulspezifikation (für eine Sprache wie Java, C, Modula-2 o.ä.) strukturiert in einer relationalen Datenbank speichern will, benötigt man alleine Relationen für die Module, die exportierten Typen, die Operationen, die Parameter der Operationen, die Typen der Parameter sowie ggf. die darin auftretenden Typkonstruktoren. Will man eine Modulspezifikation aus der Datenbank auslesen, muß man sehr komplexe Verbunde bilden, die i.a. sehr ineffizient sein werden. Es treten oft sogar rekursive Strukturen auf (Verfeinerungshierarchien, Blockschachtelungen), die mit den üblichen Abfragesprachen für relationale DBMS nicht bearbeitet werden können.
- Die unnatürliche Datenmodellierung kann zu erheblichen Performance-Problemen führen.

- Für Rasterbilder, Karten, Videos usw. benötigt man spezielle Operationen bzw. Datentypen; diese werden nicht unterstützt.
- Aus der Datenbank ausgelesene Daten werden i.a. nicht einfach in Tabellen oder Formularen angezeigt, sondern von Programmen weiterverarbeitet und z.B. graphisch angezeigt. Die Typsysteme von Programmiersprachen und Datenbankmodellen unterscheiden sich aber meist deutlich. Dies hat zur Folge, daß die Daten zwischen diesen Typsystemen konvertiert werden müssen; der hierfür erforderliche Code macht oft einen erheblichen Anteil der Programme aus und ist dementsprechend ein erheblicher Kostenfaktor.

  Man charakterisiert die Situation hier mit einem Begriff aus der Elektrotechnik und spricht vom "*impedance mismatch*", also Leistungsverlusten infolge einer Impedanz-Fehlanpassung.
- Bei manchen Anwendungen muß man die Schemata der Datenbank vergleichsweise rasch ändern können, also während der Laufzeit der Applikation und ohne explizite Konversion der gesamten Datenbank.

Transaktions-, Verteilungs- und Zugriffsschutzkonzepte und andere Systemfunktionen konventioneller DBMS können fallweise ebenfalls ungeeignet sein. Aus der obigen Skizzierung nichtkonventioneller Anwendungen sollte deutlich geworden sein, daß nicht allein die Datenbankmodelle, sondern auch diese anderen Merkmale an die jeweiligen Anforderungen angepaßt sein müssen. Objektorientierte DBMS sind übrigens auch gut für konventionelle Anwendungen geeignet, d.h. hier wäre die Kombination aus objektorientiertem Datenbankmodell und konventionellen Transaktions-, Verteilungs- und Zugriffsschutzkonzepten erwünscht.

Die unterschiedlichen Anforderungen haben zu einer Vielzahl von nichtkonventionellen DBMS geführt; fast alle beinhalten in mehr oder weniger großem Umfang objektorientierte Konzepte.

2 Eigenschaften objektorientierter DBMS

Ziel von objektorientierten DBMS ist es, die Vorteile von DBMS und objektorientierten Programmiersprachen zu vereinigen. Das ist nicht

einfach; bei den meisten Systemen diente eine der beiden Seiten als Ausgangsbasis, wurde – i.d.R. mit einigen Abstrichen und Kompromissen – um Merkmale der anderen Seite erweitert, bleibt aber dennoch dominierend für den Gesamteindruck:

- Ausgangsbasis ist eine objektorientierte Programmiersprache: eine Sprache wie C++ oder Smalltalk wird um ein Persistenzkonzept erweitert, das es ermöglicht, daß Objekte das Ende einer Programmausführung überleben, also bei der nächsten Programmausführung wieder ohne explizite Lade- oder Konversionsaktivitäten vorhanden sind. Man spricht hier auch von **persistenten Programmiersprachen**[2]. Diese Systeme beinhalten auch Konzepte wie Transaktionen (Concurrency Control und Recovery), Mehrbenutzerunterstützung und mengenorientierte Abfragen.
- Ausgangsbasis relationales DBMS: Dieses wird z.B. um "lange Felder", benutzerdefinierbare Datentypen usw. erweitert. Man nennt derartige Systeme auch **objektrelationale DBMS**.

Welche Funktionsmerkmale entscheidend für ein OODBMS sind, war lange umstritten. Eine größere Autorengruppe schlägt in [At+89] einen Kompromiß vor. In dieser Liste werden die Merkmale 1 - 12 als unerläßlich eingestuft, Merkmale 13ff sind wünschenswerte, aber je nach Kontext verzichtbare Eigenschaften. Die Merkmale 1 - 7 sind Merkmale der Objektorientierung, Merkmale 8 - 12 sind DBS-Merkmale:

1. Es gibt komplexe Objekte. Attributwerte können strukturiert sein.
2. Objekte haben eine eigene Identität, die unabhängig vom Inhalt ist und sich während der Lebensdauer des Objekts nicht ändert.
3. Datenabstraktion (Kapselung), also Trennung zwischen der exportierten Schnittstelle und der internen Realisierung.
4. Objekte sind typisiert.

[2]Dieser Begriff ist unsauber, aber üblich. Persistent sind nicht die Sprachen, auch nicht die in den Sprachen geschriebenen Programme, sondern die von diesen Programmen erzeugten Objekte. Als bessere Begriffe wurden Persistent Application System und Persistent Object System vorgeschlagen.

5. Es gibt Typhierarchien.
6. Operationen können polymorph sein, also auf Objekte unterschiedlichen Typs angewandt werden, wobei je nach dem Typ der Argumente andere Implementierungen verwendet werden.
7. Es gibt eine algorithmisch vollständige Datenbank-Programmiersprache (also keinen *impedance mismatch*).
8. Daten werden persistent gespeichert.
9. Die Abbildung der Daten auf die Sekundärspeicher ist für Anwendungen transparent modifizierbar (vgl. internes Schema)
10. Es gibt Transaktionen und die damit verbundenen Concurrency-Control- und Recovery-Mechanismen.
11. Es gibt eine mengenorientierte, deklarative Abfragesprache.
12. Das Datenbankschema ist dynamisch erweiterbar (entsprechend hierzu ggf. auch die Zugriffspfade).
13. Objekte sind versionierbar.
14. Die Kooperation von Benutzern und Benutzergruppen wird durch lange Transaktionen unterstützt.
15. Es gibt Trigger und andere Merkmale "aktiver" Datenbanken.
16. Die Datenbank kann verteilt sein.
17. Multimediale Objekte werden unterstützt.

usw. In den folgenden Abschnitten besprechen wir einige der technischen Merkmale detaillierter. Wir betrachten zunächst zwei zentrale Eigenschaften von OODBMS: Datenkapselung und die Vermeidung des *impedance mismatch*.

3 Datenkapselung

3.1 Motivation

Das Prinzip der Datenkapselung ist in Programmiersprachen seit langem eine absolute Selbstverständlichkeit und ein zentrales Konzept von Sprachen wie Modula-2 oder Ada, die in den 70er Jahren entstanden. Umso verblüffender ist, daß konventionelle Datenbanken eindeutig

im *Widerspruch* zu diesem Prinzip stehen: eine Datenbank entspricht in etwa einer globalen Variablen, deren Struktur in Form ihres konzeptuellen Schemas und der externen Schemata offengelegt ist. Die Offenlegung dieser Struktur ist sogar eine unverzichtbare Vorbedingung für ad-hoc-Abfragen in SQL oder anderen Sprachen.

Ist die Datenkapselung bei persistent gespeicherten Daten also unwichtig? An dieser Stelle sei daran erinnert, daß das Hauptziel der Datenkapselung darin besteht, Datenstrukturen bei Bedarf ohne Nebenwirkungen auf andere Module, Klassen oder Teilsysteme austauschen zu können, also durch Kenntnis der Datenstrukturen entstehende Abhängigkeiten zwischen solchen Programmteilen zu vermeiden. Hierzu müssen Datenstrukturen vor unkontrollierten Eingriffen "von außen" geschützt werden. Bei konventionellen Anwendungen liegen indes meist die folgenden Verhältnisse vor:

- Die Daten sind relativ einfach strukturiert, d.h. eine Verkapselung durch je eine Lese- und Schreiboperation pro Attribut "versteckt" nicht wirklich die interne Struktur.
- Die typischen Konsistenzbedingungen (Identifizierungs- und Fremdschlüssel u.ä.) lassen sich einfacher und besser im DBMS-Kern durch generische Algorithmen, die durch geeignete deskriptive Angaben in der Schemadefinition gesteuert werden, behandeln als durch individuelle (also für jedes Relationenschema neu geschriebene) Algorithmen.
- Eine Änderung der Datenstruktur bedingt eine komplette Konversion der Datenbank und ist daher äußerst unwahrscheinlich.

Bei konventionellen Anwendungen ist also die Verkapselung der Datenbank auf der Ebene der externen bzw. konzeptuellen Schemata durch Operationen deutlich weniger motiviert als die Verkapselung von Datenstrukturen in Programmen.

Bei vielen nichtkonventionellen Anwendungen treten deutlich komplexere Schemastrukturen auf, so daß die Kapselung hier wieder stärker motiviert ist. Festzuhalten bleibt, daß dann, wenn ad-hoc-Abfragen möglich sein sollen, zumindest die externen Schemata offengelegt werden müssen.

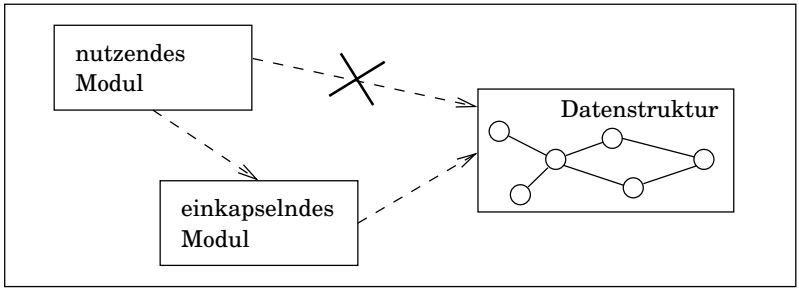

Abbildung 1: Datenkapselung im Programmen

3.2 Implementierungssprachen und Ausführungsort von Operationen

Bei der Kapselung von Datenstrukturen in Programmen wird unterstellt, daß sich normalerweise die Implementierungen der einkapselnden Operationen und der sie benutzenden Programmteile im gleichen Adreßraum befinden, in der gleichen Sprache geschrieben sind und von gleichen Prozessor ausgeführt werden[3]. Bild 1 deutet den Adreßraum durch das äußere Rechteck an. Die inneren Rechtecke sind Teile des Adreßraums, die folgendes enthalten:

- eine eingekapselte Datenstruktur,
- einen Programmteil, der die Datenstruktur einkapselt und Zugriffsoperationen exportiert, und
- einen Programmteil, der diese Zugriffsoperationen nutzt.

Bei Datenbanksystemen kann der einkapselnde Programmteil entweder im Applikationsprozeß oder im DBMS-Prozeß ausführt werden; hierzu betrachten wir erneut die schon in Abschnitt 3 in [DBSA] eingeführte Prozeßarchitektur von Informationssystemen.

[3] Es sind natürlich auch Sprachmischungen möglich und mit entsprechenden Netzwerktechnologien (RPC, CORBA usw.) sind auch verteilte Ausführungen möglich; hierauf gehen wir hier nicht näher ein.

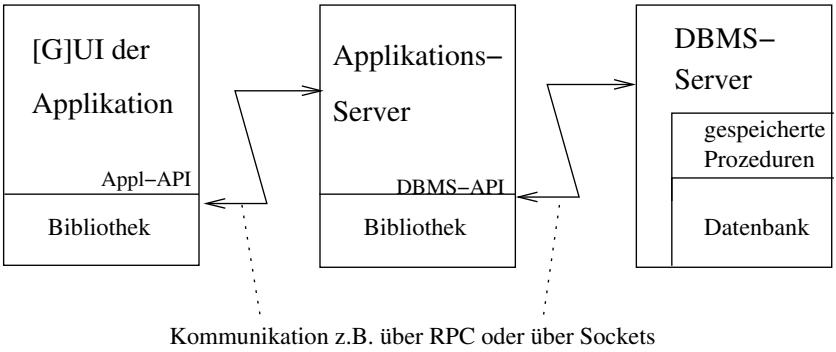

Abbildung 2: Prozeßarchitektur von Informationssystemen mit gespeicherten Prozeduren

Ausführung von Operationen im DBMS-Serverprozeß: Gerade dann, wenn Operationen auf den Daten komplex sind und viele Zugriffe zu einzelnen Datenelementen durchführen, ist es attraktiv, sie im Serverprozeß auszuführen, weil auf diese Weise viele aufwendige Kommunikationen und Datentransporte vermieden werden. Exakt diese Performance-Verbesserung motiviert auch vorkompilierte Statements und gespeicherte Prozeduren (*stored procedures*), die unabhängig von der Datenkapselung motiviert sind.

Nun sind die einkapselnden Operationen allerdings von Anwendern geschrieben und damit aus Sicht des DBMS suspekt; die Trennung der Adreßräume von Applikation und Datenbankkern war ja gerade durch das Mißtrauen gegenüber Anwendersoftware motiviert. Serverseitig auszuführender Code kann daher allenfalls in einer "sicheren" Sprache geschrieben sein, die keine Adreßrechnungen und versehentliches Beschädigen fremder Datenstrukturen erlaubt. Hochgradig unsicher sind Sprachen wie z.B. C / C++. Günstig sind Skriptsprachen, die vom DBMS-Kern selbst interpretiert werden, allerdings sind solche Sprachen ineffizient und für rechenintensive Operationen weniger geeignet. Festzuhalten bleibt, daß dann, wenn die Operationen serverseitig ausgeführt werden, spezielle Programmiersprachen notwendig werden.

Neben der Gefahr unzulässiger Zugriffe besteht das Problem, daß eine Operation infolge eines Programmierfehlers in eine Endlosschleife geraten könnte oder einfach wegen ihrer Komplexität eine sehr hohe Belastung der CPU verursacht; entsprechend weniger Rechenleistung steht für den DBMS-Kern und damit für die Bedienung anderer paralleler Nutzer zur Verfügung.

Prinzipiell wird hier die Rechenlast für die Fachlogik vom Applikationsserver in den DBMS-Server verlagert; bei einer größeren Zahl von parallelen Nutzern kann die CPU-Leistung des DBMS-Server-Rechners leicht zu einem Engpaß werden. Die Ausführung von Operationen im DBMS-Serverprozeß ist daher bei Hochlastsystemen nicht sinnvoll.

Bei der Ausführung von Operationen im DBMS-Serverprozeß sind noch einige technische Probleme zu lösen, u.a. müssen die Operationen in Form von dynamisch bindbaren Bibliotheken an das DBMS übergeben und als Teil der Datenbank gespeichert werden und das DBMS muß sicherstellen, daß auf einem Objekt nur passende Operationen ausgeführt werden.

Ausführung von Operationen im Applikationsprozeß: Ein alternativer Ansatz besteht darin, die einkapselnden Programmteile im Applikationsprozeß auszuführen und die benötigten Teile der Datenbank direkt in den Applikationsprozeß zu laden. Dieser Ansatz ist nur bei persistenten Programmiersprachen sinnvoll und wird in Abschnitt 6 noch ausführlicher diskutiert werden.

Die schon aufgeworfene Frage, wie sichergestellt wird, daß auf die eingekapselten Datenstrukturen nur über die passenden Operationen zugegriffen wird, stellt sich hier verstärkt. Wenn man diese Verantwortung dem DBMS in Sinne einer zentralen Aufsichtsinstanz überträgt, müßten die Implementierungen der Operationen in der Datenbank gespeichert werden und bei Bedarf in den Applikationsprozeß z.B. als dynamisch bindbare Bibliothek eingebunden werden; ein nicht unerheblicher Aufwand. Das Problem unsicherer Sprachen stellt sich auch verstärkt, weil nicht nur die Fachlogik, sondern auch Bedienschnittstellen in dieser Sprache implementiert werden müssen.

4 Objekte und Beziehungen

In diesem und den folgenden Abschnitten stellen wir wesentliche Aspekte vor, die bei der Definition eines objektorientierten Datenbankmodells eine Rolle spielen bzw. Entscheidungsfreiräume bieten. Wir beginnen mit den statischen Datenstrukturen.

4.1 Homogenität der Typsysteme

Der schon erwähnte *impedance mismatch* resultiert daraus, daß die Typsysteme der Programmiersprachen, in denen die Applikationen geschrieben sind, und das Typsystem (also Datenbankmodell) des DBMS differieren. Die Daten müssen also beim Laden bzw. Speichern von einem Typsystem in das andere "übersetzt" werden. Beim Laden müssen neben der reinen Konversion ggf. zusätzliche Prüfungen durchgeführt werden, weil die Datenbank von einem Programm, das in einer anderen Sprache geschrieben worden ist, modifiziert sein kann. Der Umfang dieser Programmteile ist oft erheblich, ein Anteil von 10 - 30% ist nicht ungewöhnlich. Das Erstellen dieser Programmteile verursacht erhebliche Kosten und ist bei Programmierern eher unbeliebt, weil der Code relativ stupide ist[4].

Für *eine* ausgewählte Programmiersprache kann man diese Differenz reduzieren oder ganz aufheben, indem man das Datenbankmodell des DBMS an das Typsystem der Sprache annähert oder mit ihm identisch macht. In diesem Fall sind die Typdefinitionen in einem Programm zugleich Schemadefinitionen. Diesem Vorteil stehen allerdings auch Nachteile gegenüber:

[4]Der Code kann unter bestimmten Bedingungen auch generiert werden, dann reduzieren sich die Kosten erheblich.

Softwaretechnisch gesehen stellen diese Programmteile auch ein Wartungsproblem dar: bei langlebigen Applikationen muß man damit rechnen, daß das unterliegende DBMS mehrfach ausgetauscht wird, im einfachsten Fall durch neue Produktversionen, die aber nicht 100% kompatibel sind. Daher werden diese Programmteile bei der 5-Schichten-Architektur für Informationssysteme in einer eigenen Datenhaltungszugriffsschicht zwischen der Fachlogikschicht und der Datenhaltungsschicht angeordnet.

– Für alle anderen Sprachen vergrößern sich wahrscheinlich die Differenzen zwischen den Typsystemen. Im Extremfall sind diese Sprachen nicht mehr sinnvoll einsetzbar, d.h. man ist auf die ausgewählte Programmiersprache festgelegt.

  Dies steht im Widerspruch dazu, daß Daten in einer Datenbank möglichst für Applikationen, die in unterschiedlichen Sprachen geschrieben sein können, zugreifbar sein sollten. Die Typsysteme konventioneller DBMS sind relativ einfach, deshalb können die Inhalte der Datenbank in den Typsystemen sehr vieler Programmiersprachen repräsentiert werden.

– Für komplexere Applikationen reicht eine reine Programmiersprache meist nicht aus, sondern man benötigt zusätzlich diverse Bibliotheken usw., also letztlich eine komplette Anwendungsentwicklungsumgebung. Der Aufwand für die Entwicklung solcher Umgebungen und der zugehörigen Entwicklungswerkzeuge ist hoch und u.U. nicht zu leisten.
– Da die Typsysteme von Programmiersprachen relativ komplex sind, werden auch Abfragesprachen entsprechend komplex, wodurch sie für ad-hoc-Abfragen weniger brauchbar werden können.

4.2 Komplexe Objekte

Strukturierte Typen bzw. Variablen, z.B. ein Array von Records, die weitere strukturierte Komponenten enthalten können, sind in allen modernen Programmiersprachen üblich. Die grundlegende Motivation für **komplexe Objekte** (auch als zusammengesetzte, molekulare oder aggregierte Objekte bezeichnet) besteht darin, die gleichen Strukturen auch datenbankseitig zu haben. Gemäß den üblichen Modellierungsregeln repräsentieren komplexe Objekte Teil-von-Strukturen. Zentral ist der Begriff der **Komponente**: Ein Objekt kann Komponente eines anderen Objekts sein. Erwünschte Merkmale von komplexen Objekten sind somit:

– Zur Strukturierung der Komponentobjekte sind Typkonstruktoren wie array, set, map, record, file usw. verwendbar. Die Typkonstruk-

toren sollen beliebig schachtelbar sein.

- Typen elementarer Objekte sind die üblichen Basistypen sowie Objektreferenzen.
- Komplexe Objekte können mit diversen "generischen" Operationen (löschen, kopieren, versionieren, sperren u.a.[5]) als Ganzes bearbeitet werden.

Für die Struktur der komplexen Objekte einer Datenbank sind folgende Alternativen denkbar:

- baum- bzw. waldartige Struktur: ausgehend von einem Wurzelobjekt erreicht man alle Komponentobjekte des komplexen Objekts. Jedes Objekt gehört zu höchstens einem äußeren Objekt.
- halbgeordnete (also zyklusfreie) Struktur: hier sind **gemeinsame Teilobjekte** erlaubt. Hiermit können z.B. gemeinsame Module in mehreren Softwaresystemen oder gemeinsame Abschnitte in mehreren Büchern modelliert werden.
- uneingeschränkte Struktur, d.h. es sind Zyklen erlaubt. Zyklen widersprechen eigentlich der Vorstellung von einer Teil-von-Struktur; will man Zyklen ausschließen, muß aber innerhalb jeder Operation, die ein Objekt zur Komponente eines komplexen Objekts macht, ein Zyklustest durchgeführt werden. Derartige Tests sind aufwendig oder in verteilten OODBMS, bei denen auch komplexe Objekte verteilt sein können (z.B. PCTE), nicht immer sofort möglich, weil Teile des komplexen Objekts auf einem Rechner liegen können, der gerade nicht erreichbar ist.

Auch für die Art und Weise, wie die (direkten) Komponentobjekte eines Objekts gehandhabt werden, sind verschiedene Ansätze zu beobachten:

[5] In dieser Liste fehlt bewußt das Erzeugen von komplexen Objekten. Es ist sinnvoller, initiale Objektstrukturen durch individuell programmierte Konstruktor-Operationen zu erzeugen.

– Es gibt Attribute, die Objektreferenzen oder Mengen von Objektreferenzen enthalten.

  Ggf. ist zusätzlich bei solchen Attributen unterscheidbar, ob die Zielobjekte als Komponenten zu behandeln sind oder nicht.
– Bei OODBMS, die auf dem ER-Modell basieren und die explizit Beziehungen unterstützen, kann analog zum vorigen Ansatz einzelnen Beziehungstypen das semantische Merkmal verliehen werden, daß das Zielobjekt als Komponente des Ausgangsobjekts zu behandeln ist.
– Man betrachtet die Menge der Komponenten eines Objekts als abstraktes Datenobjekt und stellt Operationen bereit, durch die Komponentobjekte in die Menge eingefügt oder aus ihr entfernt werden können bzw. mit denen alle Elemente der Menge durchlaufen werden können.

4.3 Beziehungen

Beziehungen zwischen realen Entitäten, die Teil-von-Strukturen sind, können in der Datenbank durch komplexe Objekte nachgebildet werden. Daneben werden aber auch zusätzlich ungerichtete Assoziationen benötigt (für Beispiele wie "ist verheiratet mit"). Das Datenbankmodell muß es also erlauben, Aggregationen und Assoziationen voneinander zu unterscheiden.

Die referentielle Integrität von Beziehungen sollte auf Wunsch überwacht werden. Dies bedingt, beim Löschen eines Objekts herauszufinden, ob es Referenzen auf dieses Objekt gibt und, falls ja, die Löschung abzulehnen. Diese Überprüfung ist ohne Hilfsdaten nicht effizient möglich. Eine Lösung besteht darin, zu einer Referenz, deren referentielle Integrität überwacht werden soll, am Zielobjekt eine zurückführende Referenz anzubringen, und zwar entweder nur intern, also für die Applikation nicht sichtbar, oder extern sichtbar. Der zweite Ansatz bedeutet, daß solche Referenzen aus Sicht der Applikation immer nur paarweise erzeugt und gelöscht werden können.

4.4 Objektidentität

Ein Datenbank-Objekt repräsentiert meist eine Entität in der realen Welt, die eine eigene Identität besitzt. Die interessierenden Attribute der Entitäten sind oft nicht mit Sicherheit eindeutig, können also bei zwei verschiedenen Entitäten komplett übereinstimmen; in solchen nimmt man einen künstlichen Identifizierer hinzu. Dieser kann allerdings durchaus im Laufe der Zeit ausgetauscht werden, d.h. es ist nicht sichergestellt, daß er über die Zeit hinweg immer die gleiche Entität identifiziert. Generell ungeeignet sind aus dem gleichen Argument heraus alle wiederverwendbaren Datenwerte. Als Lösung dieses Problems bieten OODBMS Surrogate an; **Surrogate** haben folgende Eigenschaften:

- Jedes Objekt erhält bei seiner Erzeugung vom DBMS ein Surrogat zugewiesen.
- Das Surrogat bleibt während der ganzen Lebensdauer des Objekts unverändert, auch wenn das Objekt verlagert oder konvertiert wird.
- Das Surrogat ist zeitlich und "räumlich" eindeutig, d.h. jedes Surrogat wird während der Lebensdauer der Datenbank nur einmal an ein Objekt vergeben. Es ist also identifizierend und wird nicht wiederbenutzt.

Mit Hilfe der Surrogate kann sehr einfach geprüft werden, ob zwei Objektreferenzen auf das gleiche Objekt verweisen.

In vielen OODBMS können Surrogate auch für den Direktzugriff zu Objekten benutzt werden; in diesem Fall muß ein Primär- oder Sekundärindex für das Surrogat-Attribut vorhanden sein.

4.5 Objekte vs. Werte

Wir hatten oben erwähnt, daß zur Strukturierung der Komponenten eines komplexen Objekts im Prinzip alle üblichen Typkonstruktoren verfügbar sein sollten. Dies führt zu der Frage, ob man dann noch die üblichen komplexen Werte, die als Inhalt entsprechend getypter Attribute bzw. Variablen im Programmen auftreten können, braucht. Die

Unterschiede zwischen komplexen Werten und komplexen Objekten werden am besten klar, wenn man die typischen Implementierungen vergleicht.

Als Beispiel für einen komplexen Wert betrachten wir einen Array fester Länge von Records, die einige Zahlen und Texte fester Länge enthalten mögen. Dieser Array wird bei Anwendung üblicher Compilerbau-Techniken in einem Speicherbereich fester Länge realisiert, der in einzelne Abschnitte unterteilt ist, die jeweils einzelne Komponenten enthalten und die durch Relativadressen identifiziert werden. Wir nennen dies auch einen **komplexen Wert**.

Dieser Speicherbereich wird nur als ganzer zwischen der Datenbank und dem Adreßraum der Anwendung übertragen; die genaue Struktur dieses Speicherbereichs ist für die Strukturen in der Datenbank weitgehend belanglos; insb. sind keine Referenzen auf Teile dieser Struktur möglich, Referenzen sind nur auf Objekte möglich. Insgesamt treffen folgende Beobachtungen auf komplexe Werte zu:

- Die Werte oder Teile von ihnen haben keine Identität (und natürlich auch kein Surrogat).
- Sie können nicht gemeinsam benutzt werden.
- Sie können keine Rolle in Beziehungen spielen oder Ziel von Objektreferenzen sein.
- Ihre Struktur ist offen, sie sind nicht gekapselt.
- Sie werden als Ganzes in entsprechend getypte Programmvariablen übertragen.
- Über vordefinierte Operationen kann sehr effizient mit ihnen gearbeitet werden. Der mit Objekten verbundene Aufwand wird vermieden.

Insb. das letztgenannte Effizienzargument spricht dafür, neben komplexen Objekten auch komplexe Werte in Attributen anzubieten.

Komplexe Werte sind andererseits nicht ganz unproblematisch. Die oben unterstellte ungeprüfte Übertragung eines Speicherbereichs, der den komplexen Wert enthält, funktioniert leider nicht immer so einfach:

- Wenn auf eine Datenbank von Klienten mit heterogenen Rechnerplattformen zugegriffen wird, sind die Datenformate i.a. nicht kompatibel, d.h. der Speicherbereich muß geeignet konvertiert werden.
- Sofern das Format, wie komplexe Werte auf Speicherbereiche abzubilden sind, nicht durch die Sprache mitdefiniert wird – was eher unwahrscheinlich ist –, kann jeder Konstrukteur eines Compilers diesbezüglich eine andere Entscheidung treffen. Es müßten dann verschiedene Versionen des DBMS für verschiedene Compiler geschaffen werden, schlimmstenfalls sogar für verschiedene Versionen "desselben" Compilers.
- Das Verfahren funktioniert nur, wenn das Typsystem der Gastsprache und des OODBMS kompatibel zueinander sind (vgl. Abschnitt 4.1). Wenn von Programmen in signifikant verschiedenen Gastsprachen (z.B. C++ und Ada) aus auf die gleichen Daten aus zugegriffen werden soll, sind aufwendige Konversionen in Ersatzdarstellungen erforderlich.

5 Vererbung

Für die Vererbung gelten zunächst die gleichen Grundregeln wie in objektorientierten Programmiersprachen, z.B. die Substituierbarkeitsregel, wonach an jeder Stelle, an der eine Instanz eines Typs T1 benötigt wird, auch eine Instanz eines Subtyps T2 von T1 benutzt werden kann. Auf derartige allgemeine Regeln gehen wir hier nicht weiter ein.

Eine DBMS-spezifische Frage stellt sich im Kontext mit der geforderten Abfragesprache. Diese benötigt Basismengen analog zu Relationen in relationalen Systemen. Ansätze hierzu sind:

- Naheliegendenderweise ist die Menge der Instanzen eines Objekttyps (also genau dieses Typs, ohne Instanzen von Subtypen) jeweils eine derartige Basismenge.
- Alternativ kann man, der Substituierbarkeitsregel folgend, die Instanzen eines Typs und aller seiner direkten und indirekten Subtypen als eine Basismenge benutzen.

– Im Prinzip können Objektmengen völlig unabhängig von der Typstruktur definiert werden – wobei natürlich unterstellt ist, daß die Mengen homogen sind –; eine explizite Definition der Objektmengen ist meist wenig sinnvoll und lästig, weil in der Praxis fast immer einer der beiden vorstehenden Fälle zutrifft.

6 Persistente Programmiersprachen

Die grundlegende Idee persistenter Programmiersprachen besteht darin, daß man Objekte als persistent deklarieren kann. Der Zustand solcher Objekte wird automatisch beim Programmende gerettet und beim erneuten Programmstart rekonstruiert, ohne daß seitens des Anwendungsprogramms explizite Lese- bzw. Schreibzugriffe und Konversionen vorgenommen werden müssen[6]. Damit ist implizit festgelegt, daß die Typsysteme von Programmiersprache und DBMS identisch sind (vgl. Abschnitt 4.1). Der *impedance mismatch* verschwindet also völlig.

Auf eventuelle Anpassungen der Programmiersprache sind wir schon in Abschnitt 3.2 eingegangen. In diesem Abschnitt gehen wir auf einige zusätzliche Fragen hinsichtlich der Persistenzkonzepte und deren Implementierung ein.

6.1 Trennung persistenter und transienter Daten

Im allgemeinen sollen nicht alle Objekte, die ein Programm erzeugt, persistent sein, es muß also auch die "normalen" transienten Objekte geben. Es muß also möglich sein, beide Arten von Objekten zu unterscheiden. Eine Anforderung in diesem Zusammenhang ist, daß mit wenig Korrekturaufwand zwischen Persistenz und Nicht-Persistenz umgeschaltet werden kann; dies bedingt, daß im Regelfall kein Unterschied beim Umgang zwischen persistenten und transienten Objekten besteht. Für die Festlegung der Persistenz sind verschiedene Ansätze möglich:

[6] Derartige Persistenzkonzepte sind auch schon lange vor OODBMS für nichtobjektorientierte Sprachen wie z.B. Pascal oder Modula-2 realisiert worden.

1. **Persistenz als Klasseneigenschaft:** Persistenz ist eine Klasseneigenschaft, d.h. alle Instanzen einer solchen Klasse sind persistent. Man kann dies auch erreichen, indem es eine vordefinierte Klasse, die `persistent_object` oder ähnlich heißt, gibt; alle persistenten Klassen müssen als Unterklasse von `persistent_object` definiert werden.

   Nachteil dieses Ansatzes ist, daß man oft sowohl transiente als auch persistente Instanzen einer Klasse braucht und daß hier zu umständlichen Ersatzlösungen gegriffen werden muß.

2. **Explizite Markierung:** Objekte, die persistent sein sollen, müssen also solche markiert werden. Der Zeitpunkt kann entweder beliebig sein oder als Sonderfall bei der Erzeugung des Objekts.

3. **Persistente Wurzel(n):** Es gibt ein oder mehrere spezielle persistente Objekte; diese und alle von dort aus erreichbaren Objekte sind persistent. Um ein Objekt persistent zu machen, muß man eine Referenz darauf von einem bereits persistenten Objekt aus erzeugen. Bei diesem Ansatz ist es sehr einfach, komplette Objektgeflechte persistent zu machen. Das Erzeugen und Löschen von Referenzen kann jetzt allerdings weitreichende Konsequenzen haben.

6.2 Bindung persistenter Objekte an Programmausführungen

Komplexe Objekte modellieren oft Dokumente und entsprechen daher in mancher Hinsicht Dateien, insb. dahingehend, daß man u.U. ein bestimmtes Programm mit verschiedenen Dokumenten ausführen möchte. Einer persistenten Variablen in einem Programm sollen also ohne Veränderung des Programms verschiedene komplexe Objekte in der Datenbank zugeordnet werden können. Hierzu müssen einzelne komplexe Objekte in der Datenbank identifizierbar sein. Hierzu gibt es mehrere Ansätze:

– Die (Wurzeln der) komplexen Objekte haben identifizierende Attribute, und das Objekt kann mithilfe einer Abfragesprache lokalisiert werden.

– Man benutzt die Surrogate der Objekte, sofern der Direktzugriff zu Objekten anhand ihrer Surrogate unterstützt wird. Surrogate sind allerdings i.a. nicht sinnvoll lesbar.
– Objekte können explizit einen Namen zugewiesen bekommen, und es existiert ein expliziter Mechanismus, der Objekte mit einem bestimmten Namen an eine bestimmte Programmvariable bindet. Dieser Ansatz ist nur bei einer kleinen Anzahl von komplexen Objekten praktikabel, ähnlich wie in Dateisystemen.

6.3 Persistenzmechanismen

Die bisherigen Betrachtungen betrafen nur Konzepte (aus Sicht von Applikationsprogrammierern) und ließen die Frage offen, durch welche Algorithmen die Laufzeitobjekte persistent gemacht werden. Für die Konzepte sind die Implementierungstechniken irrelevant, für die Performance spielen sie dagegen eine große Rolle. Eines der wesentlichen Motive für die Einführung von OODBMS war im übrigen die schlechte Performance konventioneller DBMS; die (erhoffte) Performance von OODBMS ist vielfach werblich stark herausgestellt worden und gipfelte in Aussagen, man könne auf Datenbankobjekten genauso schnell arbeiten wie auf Laufzeitobjekten. Nach einer kurzen Übersicht über denkbare Ansätze untersuchen wir in Abschnitt 6.3.2 derartige Möglichkeiten zur Performance-Optimierung.

In der folgenden Diskussion unterstellen wir, daß die Programmteile, die auf die Datenstrukturen von Objekten zugreifen, im Applikationsprozeß ausgeführt werden (vgl. Bilder 1 und 2 in Abschnitt 3.2).

6.3.1 Grundformen

Damit ein Laufzeitobjekt in einem Programm (genauer gesagt in einem Prozeß) persistent wird, muß es spätestens bei Beendigung des Programms in die Datenbank übertragen werden. Hierfür sind zwei Implementierungsansätze denkbar:

1. Verarbeiten einzelner (atomarer) Objekte: Die Struktur der Objekte wird durchlaufen, für jedes einzelne Objekt wird dessen Zustand se-

parat in die Datenbank übertragen. Dies ermöglicht es, auch DBMS zu benutzen, die ein deutlich anderes Datenbankmodell als die Programmiersprache haben, z.B. ein relationales. Man würde hier jedes Objekt (ohne seine Komponentobjekte) als ein Tupel speichern. Die Konversion zwischen den verschiedenen Typsystemen findet hier automatisiert statt.

2. Behandlung komplexer Objekte als komplexe Werte: Man kann ein Objekt auch als einen komplexen Wert betrachten, vgl. das Beispiel in Abschnitt 4.5. Man kann nun einen solchen komplexen Wert auf einmal an das DBMS übergeben. Ein unmittelbarer Vorteil hieraus im Vergleich zur ersten Alternative ist, daß die Zahl der Kommunikationen zwischen dem Anwendungsprozeß und dem Datenbank-Serverprozeß deutlich reduziert wird.

   Im Datenbank-Serverprozeß kann man entweder den komplexen Wert zerlegen und in ein anderes Typsystem konvertieren (s.o.) oder aber diese Struktur i.w. unverändert auf persistenten Medien speichern.

6.3.2 Seitenorientierte Persistenzmechanismen

Unter die o.g. zweite Variante fällt ein Implementierungskonzept, das auf den ersten Blick sehr attraktiv wirkt: Die Grundidee ist, einzelne Seiten des Hauptspeichers, in denen Objekte gespeichert sind, direkt auf Sektoren der Platte speichern[7]. Bild 3 veranschaulicht dies an einem Beispiel: Die Laufzeitobjekte, auf denen eine Applikation arbeitet, sind in dafür reservierten Teilen des Arbeitsspeichers eines Prozesses (der *heap*) angeordnet. Der Arbeitsspeicher ist ein virtueller Arbeitsspeicher und aufgeteilt in Seiten; ungenutzte Seiten können einzeln vom Betriebssystem auf Platte ausgelagert werden (*paging*). Bei neueren Betriebssystemen können Seiten nicht nur auf einen für Applikationen unzugänglichen Teil der Platte ausgelagert werden, sondern auch in eine normale Datei, und von dort später wieder zurückgeladen werden

[7] Ein direkter Zugriff auf die Platte ist tatsächlich nicht möglich, aus Gründen, die später klar werden, muß ein DBMS-Server zwischengeschaltet werden. Die resultierende Architektur nennt man *page-server*-Architektur.

(sogenanntes *memory-mapped IO*).

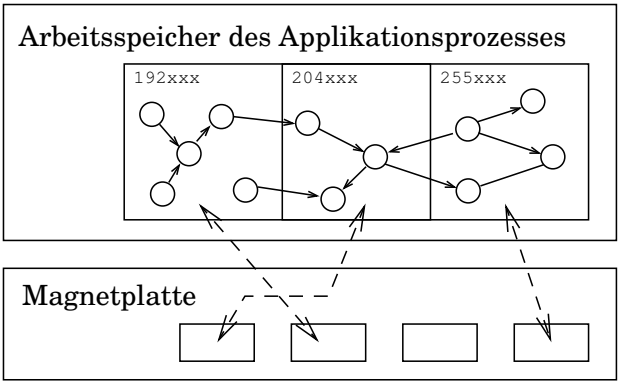

Abbildung 3: Seitenorientierte Verfahren

Im Vergleich zu einer klassischen Schichtenarchitektur eines DBMS-Kerns (vgl. Bild 2) entfällt die Verwaltung von Speichersätzen und Einzeltupeln bzw. diese kann wesentlich vereinfacht werden, ferner entfällt der Aufwand für Konversionen von Datentypen.

Insgesamt wirkt dieses Implementierungsverfahren auf den ersten Blick sehr performant. Oft wird unter dem Begriff "persistente Programmiersprache" nicht nur die Sprache und ihr Persistenzkonzept, sondern zusätzlich dieses Implementierungsverfahren verstanden. Zu einem Konzept kann es aber viele Implementierungen geben, man sollte beides nicht unnötig vermischen.

Der zweite Blick auf dieses Implementierungsverfahren offenbart im übrigen, daß

1. das Verfahren so einfach doch nicht funktioniert,
2. Performance-Vorteile nur unter speziellen günstigen Randbedingungen möglich sind (und dann auch bei konventionellen Datenmodellen erzielbar wären) und
3. man auf viele übliche Leistungsmerkmale eines DBMS verzichten muß; letzteres kann völlig in Ordnung sein, wenn man diese Lei-

stungsmerkmale gar nicht nutzt und man bewußt auf diese Leistungen verzichtet.

Pointer Swizzling. Beziehungen zwischen Objekten werden in Laufzeitsystemen durch Zeiger realisiert. Wir nehmen i.f. vereinfachend an, daß die Attribute eines atomaren Objekts in Form eines komplexen Werts gespeichert werden. Dann sind einzelne Felder innerhalb dieses komplexen Werts Adressen anderer Objekte.

Werden nun Seiten unverändert auf Platte gespeichert und später neu geladen, dann müßte jede Seite wieder an die gleichen Adressen wie früher geladen werden, andernfalls stimmten die Adressen nicht mehr. Hierzu müßte jedem Sektor eine feste Hauptspeicheradresse zugeordnet sein; anders gesehen müßte ein Abschnitt des Adreßraums für den Sektor exklusiv reserviert werden. Bei heute gängigen 32-Bit-Rechnern ist der virtuelle Adreßraum aber nur 1 - 2 GB groß[8], hiervon wird ferner ein großer Teil für die Applikation benötigt. Die Datenbank könnte daher nicht größer als ca. 1 GB sein, was nicht akzeptabel ist.

Die Lösung besteht darin, beim Speichern einer Seite die darin enthaltenen Adressen in geeignete Objektidentifizierer umzusetzen; umgekehrt setzt man beim späteren Laden der Seite die Objektidentifizierer in die dann gültigen Adressen der Objekte um. Diese Vorgänge werden *pointer swizzling* genannt genannt. Für das *pointer swizzling* ist eine Vielzahl von Implementierungen bzw. Optimierungen erdacht worden, die hier nicht diskutiert werden sollen.

Festzuhalten bleibt, daß hierfür Rechenzeit und/oder Speicherplatz verbraucht wird und daß die simple Vorstellung, Hauptspeicherseiten unverändert auf Platte abzulegen, bei heute gängigen 32-Bit-Rechnern nicht realistisch ist.

Beim Schreiben einer Seite auf die Platte tritt bei "unsicheren" Programmiersprachen wie C oder C++ ein zusätzliches Problem auf: die Applikation könnte den Inhalt der Seite und damit auch der Objektbank beschädigt haben, die Datenbank könnte somit nach dem Speichern also physisch inkonsistent werden. Vor dem Speichern muß also ggf. der

[8] Bei 64-Bit-Rechnern ist dies nicht mehr der Fall.

Inhalt der Seite durchleuchtet werden.

Bedingungen für Performance-Verbesserungen. Bei ersten Zugriff auf ein Objekt ist der zeitraubende Transport eines Sektors von der Platte in den Hauptspeicher unverändert notwendig, erst bei den folgenden Zugriffen entfällt dieser Hauptkostenfaktor. Bei Applikationen, die nur einmal zu einem Datenelement zugreifen, ist also keine signifikante Beschleunigung zu erwarten.

Vorteile beim ersten Zugriff sind dann möglich, wenn Objekte klein sind, also viele Objekte in eine Seite passen, und wenn Objekte so angelegt werden, daß sie nicht wild verstreut auf den Seiten liegen, sondern im Zusammenhang benötigte Objekte auf der gleichen Seite; letzteres hängt natürlich vom Verhalten der Applikationen ab, d.h. das OODBMS benötigt von dort aus entsprechende Hinweise. Bei vielen OODBMS kann oder muß man daher in der Operation, die ein Objekt erzeugt, ein anderes Objekt angeben, "in dessen Nähe" das neue Objekt erzeugt werden soll. Mit Hilfe dieser Angaben werden dann Objekte möglichst günstig auf den Seiten gruppiert ("geclustert").

Im Sinne der ANSI/SPARC-Schema-Architektur gehören diese Angaben eigentlich zum internen Schema und widersprechen dem Ziel der Datenunabhängigkeit. Die nichtkonventionellen Anwendungen, an denen OODBMS orientiert sind, sind allerdings so komplex, daß dort Datenunabhängigkeit praktisch nicht erreichbar ist.

Fehlende Leistungen von Datenbanksystemen. Beim simplen Ein- und Auslagern von Seiten werden mehrere übliche und oft wesentliche Leistungen von Datenbanksystemen nicht mehr erzielt, dazu ist dieser Persistenzmechanismus viel zu trivial (ob mit oder ohne *pointer swizzling*, ist hier egal). Will man diese Leistungen trotzdem realisieren, muß entweder der ganze Ansatz aufgegeben werden oder die Performance-Vorteile gehen weitestgehend verloren:

1. *Sprachunabhängigkeit*: Applikationen können nicht mehr in verschiedenen Sprachen geschrieben werden, denn die Typsysteme der

Sprachen sind i.a. zu verschieden, erst recht das Format, in dem die zugehörigen Compiler komplexe Werte ablegen. Letzteres ist sogar dann ein Problem, wenn man zwar nur eine Sprache, aber verschiedene Compiler (von konkurrierenden Anbietern) hat. Es kann notwendig sein, verschiedene Versionen des DBMS für verschiedene Compiler zu bilden, schlimmstenfalls sogar für verschiedene Versionen "desselben" Compilers.

2. *heterogene Plattformen*: Wenn die Datenbank von Klienten benutzt werden soll, die auf heterogenen Hardware-Plattformen laufen, können die elementaren Datenformate verschieden sein, d.h. diesbezüglich ist eine Konversion erforderlich.

3. *Sichten*: Sichten sind das wesentliche technische Mittel, um Zugriffskontrollen und die Datenunabhängigkeit von Applikationen zu realisieren. Werden Seiten unverändert an verschiedene Applikationen geliefert, liegen die Daten auf der Ebene des konzeptuellen Schemas ungefiltert vor.

   Sofern Sichten durch das DBMS realisiert werden sollen, müssen die Seiteninhalte beim Lesen gefiltert werden, die weggefilterten Teile müssen beim Schreiben wieder passend hinzugefügt werden.

   Alternativ könnten Sichten durch das Laufzeitsystem der Programmiersprache realisiert werden; in normalen Programmiersprachen ist ein derartiges Konzept aber völlig unbekannt.

4. *effiziente Suche durch Indexe*: Die in Compilern üblichen Speicherungsstrukturen sind nicht an Sekundärspeichern orientiert und enthalten keine Primärindexe, die die effiziente Suche innerhalb der Menge der Instanzen eines Typs unterstützen können.

   Selbst dann, wenn man nur mit Sekundärindexen – die komplett getrennt von den Primärdaten angelegt werden können – arbeiten würde, müßten diese bei jedem Schreibzugriff korrigiert werden.

5. *paralleler Zugriff*: Eine weitere Frage ist, ob und wie mehrere Anwendungsprozesse parallel auf den gleichen Daten arbeiten können. Beispielsweise könnten zwei Anwendungsprozesse parallel auf ein Objekt, das ein Dokumentverzeichnis repräsentiert, zugreifen wol-

len. Hierzu muß die entsprechende Seiten in *mehrere* Adreßräume eingespiegelt werden.

Wenn nun die beiden Anwendungsprozesse Änderungen vornehmen, z.B. beide fügen ein neues Dokument in das Dokumentverzeichnis ein, entstehen zwei Varianten der Seite; die lokalen Änderungen müßten beim Zurückschreiben gemischt werden. Das DBMS kann dies aber i.a. nicht, weil es die Logik der Applikation nicht kennt. Wenn in unserem Beispiel zufällig beide neuen Dokumente den gleichen Namen bekommen hätten, läge sogar ein inhaltlicher Konflikt vor. Diese Probleme sind durch Mischen nicht lösbar, d.h. die Kopien der Seiten müssen gleichgehalten werden. Hierzu muß ein Prozeß bei einem Schreibzugriff andere Prozesse über die Änderung benachrichtigen, hierzu ist eine zeitaufwendige Prozeßkommunikation erforderlich.

Ferner müssen die Anwendungsprozesse durch Concurrency-Control-Mechanismen, i.d.R. Sperren, voreinander geschützt werden. Die entscheidende Frage ist hier die Granularität der Sperreinheiten. Wählt man diese zu grob (Beispiel: die Datenbank ist nur komplett sperrbar), wird die Parallelität zu sehr reduziert. Wählt man feinkörnigere Sperreinheiten, z.B. einzelne atomare Objekte, so muß vor dem ersten Zugriff zu einem Objekt eine Sperre eingerichtet werden. Die Gesamtmenge aller Sperren muß zentral verwaltet werden, d.h. auch hier sind wieder zeitaufwendige Prozeßkommunikationen erforderlich.

6. *Recovery*: Immer, wenn der Anwendungsprozeß eine Transaktion beendet, müssen die modifizierten Daten sofort zum Datenbank-Serverprozeß übertragen werden, dort müssen Recovery-Daten erzeugt und Sperren freigegeben werden usw. Ferner müssen ggf. Indexe, die von den Änderungen betroffen sind, aktualisiert werden. Zusammen mit der ohnehin langsamen Übertragung der Daten zwischen den Prozessen (vgl. Abschnitt 3 in [DBSA]) sind diese Aufwände dominierend, d.h. die eingesparte Konversion der Daten führt tatsächlich nur zu marginalen Performance-Gewinnen. Dies gilt insb. für viele konventionelle Anwendungen, bei denen in einer Transaktion immer

nur wenige Tupel bzw. Objekte erzeugt oder modifiziert werden.

Wegen des Aufwands für die Datenübertragungen und die Recovery-Mechanismen ist es sinnvoll, das Zurückschreiben der Daten durch die Applikation steuern zu lassen; die Persistenz ist dann nicht mehr völlig transparent, der zusätzliche Programmieraufwand ist aber gering.

Quintessenz der vorstehenden Betrachtungen ist, daß die klassischen Leistungen eines DBMS ihren Preis haben und daß die "Effizienz" der seitenorientierten Verfahren darin besteht, diese Leistungen nicht zu erbringen. Je nach Anwendungskontext können diese Leistungen in der Tat unnütz sein, man sollte sich aber über diesen Verzicht auf bestimmte Leistungen im klaren sein.

7 Märkte und Standards

Die Liste der in Abschnitt 2 gewünschten Merkmale von OODBMS ist schon lang und läßt sich noch beliebig fortsetzen. Die resultierenden Systeme werden immer komplexer, was einige sehr negative Konsequenzen hat:

- Ihre Implementierung immer aufwendiger, also teurer. Es gibt bis heute kein System, das alle o.g. Forderungen komplett erfüllt.
- Die Zeit, bis eine Implementierung ausgereift und stabil ist, wird immer länger, und der Lernaufwand für Anwendungsentwickler wird immer größer. Beides ist nicht gerade akzeptanzfördernd.

Den hohen Kosten steht eine relativ kleine Zahl von Benutzern gegenüber, die diese Systeme wirklich ausnutzen können. Kommerziell überlebensfähig waren daher nur ganz wenige der vielen, besonders Anfang bis Mitte der 80er Jahre entwickelten Systeme. Überlebt haben vor allem objektrelationale Systeme, deren Basis ein erfolgreiches und ausgereiftes DBMS-Produkt ist. Generell kann man sagen, daß sich anfängliche Erwartungen, objektorientierte DBMS würden auf Dauer die konventionellen DBMS ablösen, nicht erfüllt haben.

Für das Entstehen eines Marktes sind natürlich Standards sehr wichtig; einschlägige Standardisierungsvorhaben der ISO (International Standards Organisation) sind:

- **SQL3**: hierbei handelt es sich um Erweiterungen von SQL2 um ADTs, rekursive Verbunde, Trigger und weitere o.g. Merkmale.
- **IRDS** und **PCTE**: hierbei handelt es sich um OODBMS, die speziell als Basis von Software-Entwicklungsumgebungen konzipiert wurden, wobei IRDS an Data-Dictionary-Systemen und Mainframe-Kontexten orientiert war, PCTE an den Verhältnissen in Netzwerken aus UNIX-Workstations. Beide Standards konnten sich nicht durchsetzen.
- Die **ODMG-Standards** [Ca+99]: Die ODMG (Object Data Management Group) ist eine Vereinigung der wichtigsten Hersteller von OODBMS. Dieser Standard wird von einigen Produkten zumindest teilweise implementiert. Die Standards bestehen aus folgenden Teilen:
  - einem *Objektmodell* (**OM**), das grundlegende Konzepte wie Objekt, Typ. Attribut usw. definiert.
  - einer Objekt[typ]definitionssprache (*object definition language*; **ODL**), die vergleichbar ist mit der DDL klassischer DBMS und mit der Schemata notiert werden können. Neben den Datenstrukturen (Objekttypen, Attribute usw.) werden auch Signaturen von Operationen definiert.
  - einer Abfragesprache für Objektbanken (*object query language*; **OQL**), die sich teilweise an die Syntax von SQL anlehnt.
  - Sprachanbindungen für die Sprachen C++, Java und Smalltalk.
  - einer Sprache zur Definition von Objekten im Sinne von Instanzen von Objekttypen (*object interchange format*; **OIF**): Mit Hilfe dieser Sprache können Datenbankinhalte in eine Datei geschrieben und von aus wieder in einer Objektbank eingelesen werden. So können u.a. Datenbankinhalte von einer Datenbank in eine andere transportiert werden.

Objektrelationale DBMS. Die Grundmotivation objektrelationaler DBMS (**ORDBMS**) besteht darin, ein erprobtes relationales DBMS um Merkmale zu erweitern, die die Anforderungen der Objektorientierung zumindest teilweise erfüllen; gleichzeitig soll hierbei konptuelle Klarheit und die präzise mathematische Definition des relationalen Datenbankmodells nicht unterminiert werden.

Ein ORDBMS ist immer auch ein klassisches relationales DBMS; ein Anwender kann also die meist umfangreiche Sammlung von Anwendungen, die auf dem relationalen Modell und den zugehörigen Sprachen (wie SQL) und Schnittstellen basieren, unverändert weiterbenutzen und objektorientierten Erweiterungen punktuell dort einsetzen, wo sie wirkliche Vorteile bringen.

Praktisch alle großen relationalen DBMS-Produkte bieten heute objektorientierte Erweiterungen an und können als ORDBMS bezeichnet werden. Der Umfang und die konkrete Ausgestaltung der Erweiterungen ist nicht einheitlich; Beispiele sind:

- benutzerdefinierte Attributtypen
- abgeleitete Attribute, die durch eine Abfrage definiert sind
- automatisch vergebene Surrogate
- benutzerdefinierte Funktionen
- Typhierarchien
- erweiterte Triggermechanismen
- lange Felder (*binary large objects*)

ORDBMS haben deutlich höhere Chancen als reine OODBMS, zu stabilen Produkten zu werden und am Markt zu überleben.

Literatur

[At+89] Atkinson, M., Bancilhon, F. and DeWitt, D. et al.: The object-oriented database system manifesto; Proc. First Conf. Deductive and Object-Oriented Databases, Kyoto, Japan; December 1989

[Ca+99] Cattel, R.G.G.; et al.: The Object Data Standard, ODMG 3.0; Morgan Kaufmann Publishers; 1999

[DBSA] Kelter, U.: Lehrmodul “Architektur von DBMS”; 2001
[DVS] Kelter, U.: Lehrmodul “Datenverwaltungssysteme”; 2002

Index