Folien zum Lehrmodul

Objektorientierte Datenbanksysteme

Lernziele:

- OODBMS als Technologie einordnen kennen
- wesentliche Funktionsmerkmale von OODBMS kennen (komplexe Objekte, komplexe Werte, Surrogate, ...)
- Persistenzkonzepte in Programmiersprachen kennen
- Vor- und Nachteile / sinnvolle Anwendungsbereiche von OODBMS kennen

Inhaltsverzeichnis

1 Einleitung

Objektorientierte DBMS (OODBMS) vereinigen

1. Datenbankkonzepte (Abfragesprachen, Transaktionen, ...)
2. Konzepte der objektorientierten Programmierung / Modellierung

Historie:

- 1980er Jahre: sehr viel Forschung
- später 1980er Jahre: erste kommerzielle Systeme
- frühe 1990er Jahre: gewisse Euphorie; mehrere Standards
- heute: wenige Überlebende, primär oo Erweiterungen relationaler Systeme

2 Nichtkonventionelle Anwendungen

Merkmale *konventioneller* Anwendungen:

- relativ einfach und *homogen* strukturierter Daten
- relativ *kleine* Tupel, oft Sätze fester Länge
- *atomare* Datenfelder
- Datenbankschemata werden sehr selten geändert
- *kurze Transaktionen*

Beispiele für nichtkonventionelle Anwendungen:

- Technische Entwurfsumgebungen (CAD, CASE usw.): textuelle und binäre Dokumente, Bitmaps, Versionen, komplexe Struktur
- Multimedia-Datenbanken: u.a. Video- und Audio-"Dateien", *streaming* Formate
- Büroinformationssysteme: Briefe, digitalisierte Papiervorlagen u.ä.; Information Retrieval
- Expertensystem-Datenbanken

Konventionelle DBMS für nichtkonventionelle Anwendungen wenig geeignet, weil:

- *unnatürliche Datenmodellierung*
  z.B. Syntaxbaum → Relationen; beim Lesen sehr komplexe, ineffiziente Verbunde; z.T. rekursive Datenstrukturen
- *spezielle Datentypen* wie Videos, Rasterbilder etc. nicht unterstützt
- *impedance mismatch*: aufwendige Konversion zwischen den Typsystemen von Programmiersprachen und Datenbankmodellen (viel Programmcode)
- oft auch bei Transaktionen, Verteilungs- und Zugriffsschutzkonzepten usw. spezielle Leistungen erforderlich

3 Eigenschaften objektorientierter DBMS

Ziel: Vorteile von DBMS und objektorientierten Programmiersprachen vereinigen; oft Kompromisse erforderlich

- Ausgangsbasis objektorientierte Programmiersprache: erweitert um Persistenzkonzept, Transaktionen, mengenorientierte Abfragen, Mehrbenutzerunterstützung
- Ausgangsbasis relationales DBMS: erweitert um "lange Felder", benutzerdefinierbare Datentypen usw.

  → **objektrelationale DBMS**

Merkmale von OODBMS gemäß dem "*object-oriented database system manifesto*"

[oo Merkmale:]

- komplexe Objekte, strukturierte Attributwerte
- Objektidentität
- Datenabstraktion (Kapselung, Schnittstellen)
- typisierte Objekte
- Typhierarchien
- Polymorphie
- algorithmisch vollständige Datenbank-Programmiersprache (kein *impedance mismatch*)

[DBMS-Merkmale:]

- Persistenz
- internes Schema
- Concurrency-Control- und Recovery-Mechanismen
- mengenorientierte, deklarative Abfragesprache
- dynamisch erweiterbares Datenbankschema

[weitere wünschenswerte Merkmale:]

- versionierbare Objekte
- lange Transaktionen
- Trigger und andere Merkmale “aktiver” Datenbanken
- verteilte Datenbank
- Unterstützung multimedialer Objekte

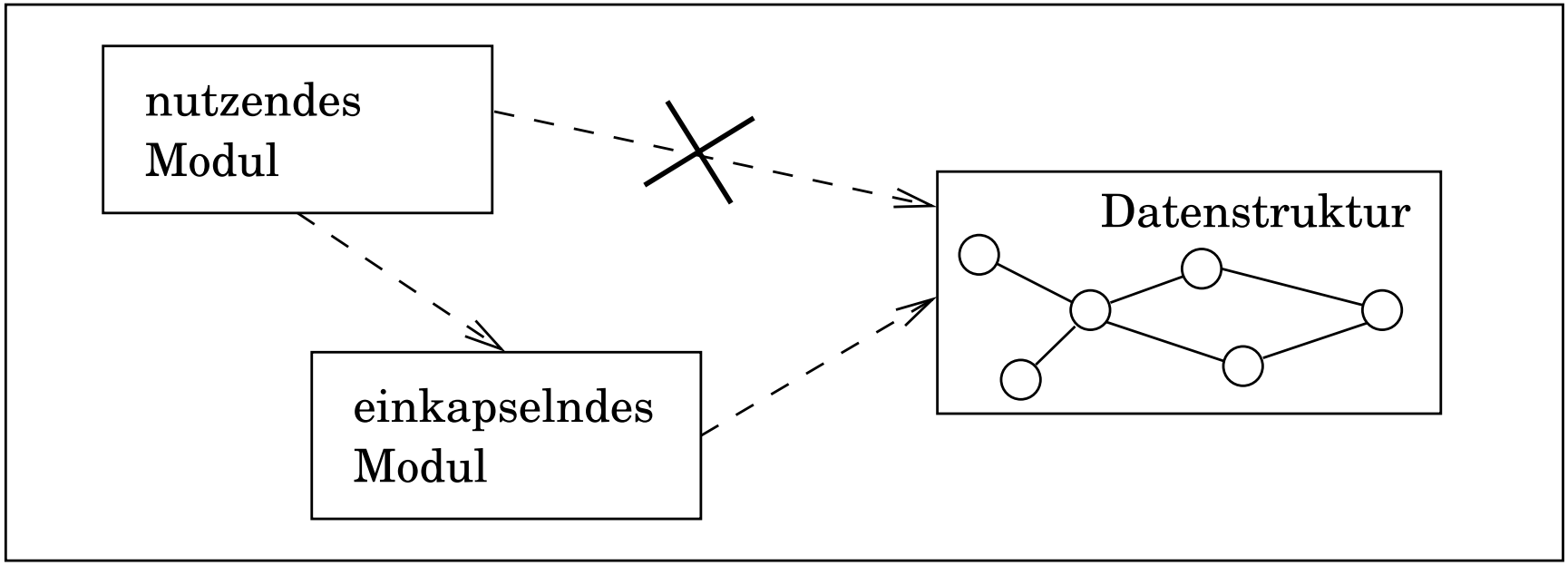
4 Datenkapselung

4.1 Motivation

konventionelle Datenbanken widersprechen dem Prinzip Datenkapselung:
– DB = komplexe Variable
– Schema = Datenstruktur — ist offen!!

Sonst wären keine ad-hoc-Abfragen möglich

Warum überhaupt Datenkapselung? (in konventionellen Programmiersprachen)



Datenkapselung ist bei konventionellen Datenbank-Anwendungen *meistens verzichtbar (!)*:

- Änderung der Datenstruktur unwahrscheinlich

  u.a. weil sehr aufwendig wegen erforderlicher DB-Konversion – ein analoges Problem existiert bei Laufzeitobjekten nicht

- oft nur 1:1-Zuordnung Attribut zu Lese-/Schreiboperation, Datenstruktur wird nicht wirklich versteckt

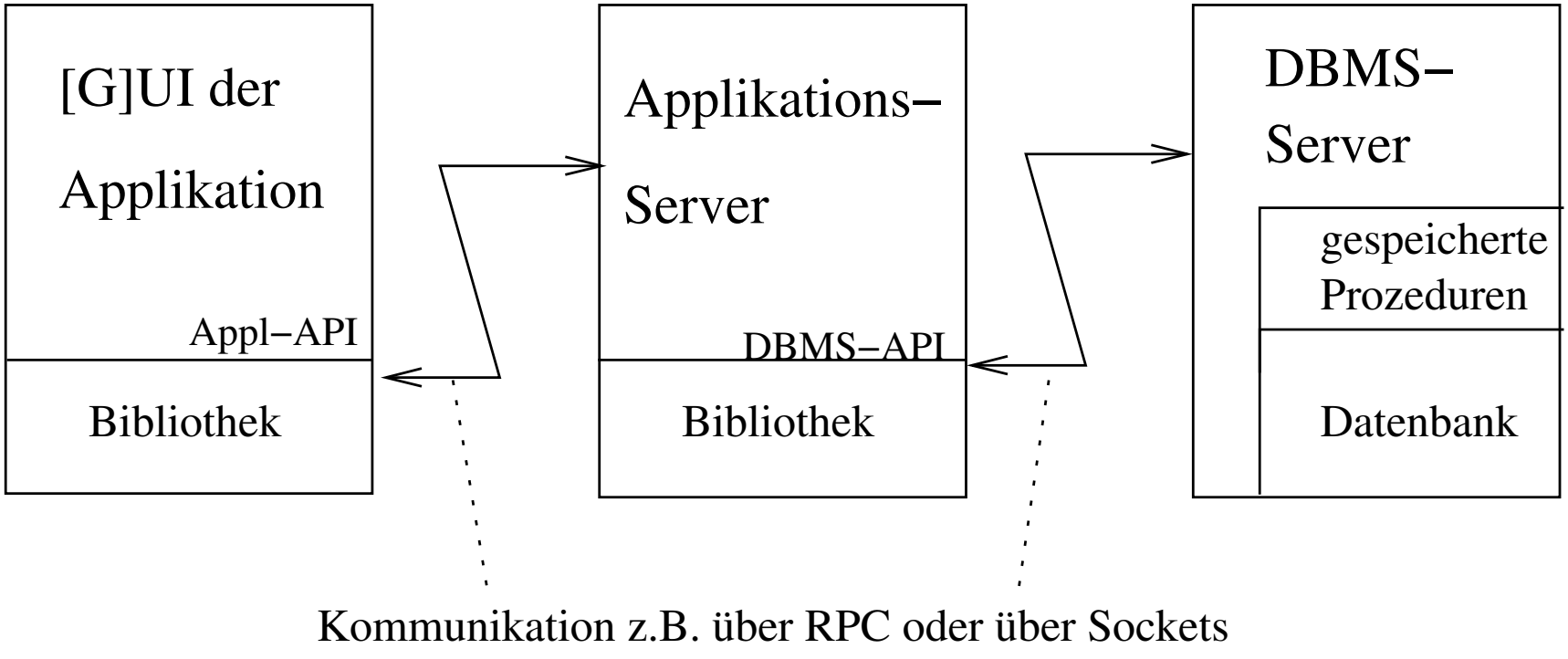
- Schutz der Konsistenz der Daten besser durch Identifizierungs- und Fremdschlüssel u.ä. als durch individuelle Algorithmen

4.2 Implementierungssprachen und Ausführungsort von Operationen

mögliche Ausführungsorte von einkapselnden Operationen:

- GUI / Applikations-Serverprozeß
- DBMS-Serverprozeß (stored procedures)

involvierte Prozesse und deren virtuelle Hauptspeicher:



1. Ausführung von Operationen im DBMS-Serverprozeß:

Vorteile:

- spart aufwendige Kommunikationen und Datentransporte
- effizient bei vielen Zugriffen zu einzelnen Datenelementen,

motiviert *stored procedures* (unabhängig von Datenkapselung)

Probleme:

1. **Sicherheit:**
   einkapselnde Operationen sind von Anwendern geschrieben
   → suspekt, sehr hohes Schadenspotential bei unsicheren Sprachen wie C / C++
   → nur Skriptsprachen u.ä. erlauben (aber ineffizient / eingeschränkte Mächtigkeit)

2. **Arbeitslast / Performance:**
   CPU-Belastung des Rechners, auf dem der DBMS-Serverprozeß läuft (komplexe Algorithmen, Endlosschleife, ...)
   problematisch: Verlagerung der Rechenlast vom Arbeitsplatzrechner / Applikationsserver auf DBMS-Server
   → Ansatz ungeeignet für Systeme mit *vielen* Nutzern

2. Ausführung von Operationen im Applikationsprozeß:

Problem: wer stellt sicher, daß auf einem Objekt nur passende Operationen ausgeführt werden?

DBMS als zentrale Aufsichtsinstanz? (lädt Bibliotheken in die Applikationsprozesse? Für beliebige Plattformen?)

5 Objekte und Beziehungen

i.f. Gestaltungsspielräume / Hauptvarianten von wesentlichen Aspekten von oo Datenbankmodellen

5.1 Homogenität der Typsysteme

Aspekt: Wahl des Typsystems des ooDBMS

impedance mismatch: Differenzen im Datenbankmodell von DBMS und Programmiersprache

Idee: DBMS-Modell identisch wie *eine* bestimmte Programmiersprache wählen

- Probleme mit anderen Programmiersprachen
- Programmiersprache alleine reicht nicht; Bibliotheken usw. $\rightarrow$ sehr hoher Spezifikations- und Implementierungsaufwand
- Typsystem der Programmiersprache ggf. zu komplex als Basis für Abfragesprachen

5.2 Komplexe Objekte

Aspekt: Typkonstruktoren, nichtatomare Objekte

Programmiersprachen haben Typkonstruktoren wie array, set of, record, ... sind Teil-von-Strukturen

DBMS-seitige Nachbildung: **komplexe Objekte** mit **Komponentobjekten**

- Strukturierung der Komponentobjekte mit Typkonstruktoren wie array, set, map, record, file; beliebig schachtelbar
- können mit 1 Operationsaufruf **als Ganzes** bearbeitet werden.

  Operationen: löschen, kopieren, versionieren, sperren, ...

Alternativen für die Struktur komplexer Objekte:

- *baum*- bzw. waldartige Struktur
- *halbgeordnete* (also zyklusfreie) Struktur, gemeinsame Teilobjekte
- *Graph*, Zyklen erlaubt

Alternativen für die Handhabung von Komponentobjekten:

- (spezielle) Attribute, die *Objektreferenzen* oder Mengen von Objektreferenzen enthalten
- Menge der Komponenten eines Objekts als *abstraktes Datenobjekt* mit Interface betrachten (kein explizites Attribut)
- Teil-von-Eigenschaft als *Merkmal von Beziehungstypen*

5.3 Beziehungen

Aggregationen und Assoziationen erforderlich

referentielle Integrität auf Wunsch überprüfbar → paarweise gegenläufige Beziehungen

5.4 Objektidentität

wertbasierte Identität hat diverse Nachteile →

Surrogate; Eigenschaften:

- bei Erzeugung eines Objekt automatisch zugewiesen
- wird nie verändert
- zeitlich und "räumlich" eindeutig

Benutzung für:

- Test, ob zwei Objektreferenzen auf gleiches Objekt verweisen
- teilweise für Direktzugriff

Realisierung: längerer String (externe Darstellung, komprimierbar)
→ "schwergewichtige" Objekte (auch wegen anderer Features)

5.5 Komplexe Objekte vs. komplexe Werte

komplexer Wert in einem Attribut:

- wird auf 1 **Bytefeld** abgebildet, Länge u.U. schon zur Compile-Zeit bestimmbar
- wird nur **als ganzes** zwischen der Datenbank und dem Adreßraum der Anwendung übertragen

  → sehr **effiziente** Verarbeitung
- "Komponenten" in komplexem Wert sind keine Objekte:
  - haben **keine Identität** (kein Surrogat)
  - können **keine Rolle in Beziehungen** spielen
  - können nicht Ziel von Objektreferenzen sein
- Struktur des komplexen Werts ist **offen**

Probleme mit komplexen Werten:

- heterogene Plattformen
- Typsystem von DBMS und Gastsprache müssen kompatibel sein $\rightarrow$ i.w. nur 1 Gastsprache möglich
- Abhängigkeit vom Compiler / Laufzeitsystem

6 Vererbung

Frage zu Bedeutung von Abfragen:
gegeben ein Typ, welche Basismenge an Instanzen gehört dazu?

1. Instanzen nur des **exakten** Objekttyps (ohne Subtypen)
   (Vergleich: horizontale Partitionierung)
2. Instanzen eines Typs und aller seiner direkten und indirekten **Subtypen**
   (Vergleich: vertikale Partitionierung)
3. statt pauschaler Festlegung: Objektmengen völlig unabhängig von der Typstruktur definieren (eher lästig)

7 Persistente Programmiersprachen

Grundidee:

- Typsysteme von Programmiersprache und DBMS identisch
- persistente Objekte: werden automatisch beim Programmende gerettet und beim erneuten Programmstart rekonstruiert

7.1 Konzeptuelle Trennung persistenter und transienter Daten

transiente (normale) Daten: werden bei Programmende gelöscht

Problem: es werden transiente **und** persistente Daten benötigt

transiente Varianten sind anders:

- kein Surrogat
- i.a. Konsistenzbedingungen des DB-Schemas nicht anwendbar
- ggf. andere Rechte

Anforderungen (ideal):

- einfache Spezifikation der Persistenz
- kein Unterschied in der Handhabung persistenter und transienter Daten
- wenig Korrekturaufwand bei Umstellung

Methoden zur Spezifikation, daß Daten persistent sein sollen:

1. **Persistenz als Klasseneigenschaft**:
   vordefinierte Klasse namens `persistent_object` o.ä.;
   hiervon Unterklassen bilden
   Nachteil: transiente und persistente Instanzen der gleichen Klasse schlecht zu trennen

2. **Explizite Markierung**:
   Angabe zu beliebigem Zeitpunkt während Lebensdauer

3. **Persistente Wurzel(n)**:
   alle von dort erreichbaren Objekte implizit persistent
   → wenig Spezifikationsaufwand,
   u.U. schlecht durchschaubare Effekte von Änderungen an Objektstrukturen
   nur implizites Löschen

7.2 Bindung persistenter Objekte an Programmausführungen

gleiches Programm muß mit verschiedenen Objekten ausführbar sein;

Auswahl der DB-Objekte, mit denen ein Programm ausgeführt werden soll, über:

1. Surrogate
2. identifizierende Attribute an Objekten
   (ggf. nur für Wurzelobjekte relevanter Teilbäume)
3. einzelne OODB-Objekte erhalten *explizit* einen Namen (ähnlich Tagging in SVN), explizite Bindung der OODB-Objekte an Programmvariablen
   nur bei kleiner Anzahl von komplexen Objekten praktikabel

7.3 Persistenzmechanismen

bisher offen: Implementierung der Persistenz

irrelevant für Konzepte, sehr relevant für **Performance** (-versprechungen)

Annahme i.f.: Programmteile, die auf die Datenstrukturen von Objekten zugreifen, werden *im Applikationsprozeß* ausgeführt

7.3.1 Grundformen

als persistent markierte Laufzeitobjekte müssen bei Beendigung des Programms in die Datenbank übertragen werden

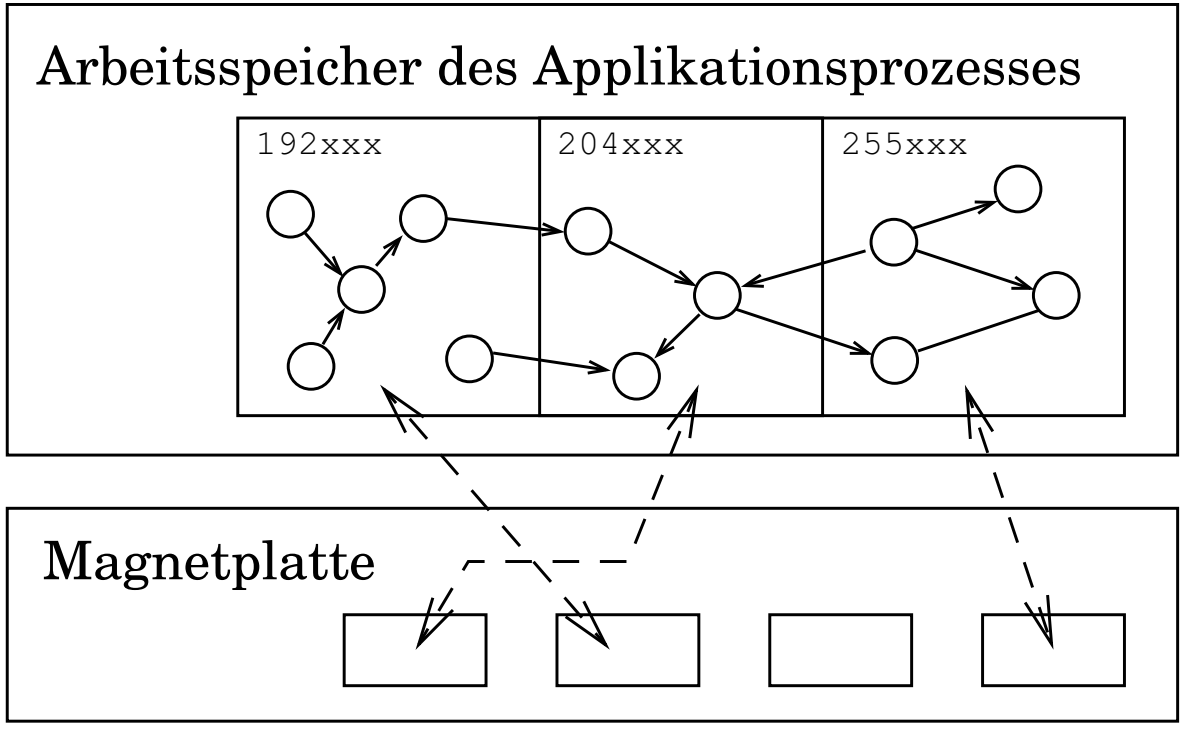
Implementierungsansätze:

1. **Verarbeiten einzelner (atomarer) Objekte**:
   Struktur der Objekte durchlaufen, atomare Objekte einzeln speichern (1 Objekt = 1 Tupel)
   erlaubt Einsatz beliebiger DBMS, automatische Konversionen

2. **Behandlung komplexer Objekte als komplexe Werte**:
   weniger Kommunikationen zum DBMS-Serverprozeß
   Zerlegung des komplexen Werts im DBMS-Serverprozeß, ggf. Konversion in anderes Typsystem;
   Sonderfall hiervon: Paging-Verfahren

7.3.2 Seitenorientierte Persistenzmechanismen

Grundidee: einzelne Seiten des Hauptspeichers, in denen Objekte gespeichert sind, direkt auf Sektoren der Platte speichern



ähnlich Paging / *memory-mapped IO*

Schichtenarchitektur des DBMS-Kerns: Verwaltung von Speichersätzen und Einzeltupeln entfällt!

unter “persistente Programmiersprache” wird oft dieses Implementierungsverfahren verstanden;

wirkt sehr elegant und performant (Hardware-Unterstützung!)

ABER:

1. funktioniert doch nicht so einfach
2. Performance-Vorteile nur unter speziellen günstigen Randbedingungen
3. bedingt Verzicht auf viele übliche Leistungsmerkmale eines DBMS

Pointer Swizzling.

Problem: Referenzen auf andere Objekte werden im Laufzeitsystem durch *Zeiger* (Adreßwerte) realisiert

Zeiger stehen *mitten in komplexen Werten*

Frage: wo stehen die Objekte nach Neuladen im Hauptspeicher???

1. an der gleichen Adresse wie vorher → Zeiger bleiben korrekt
   → Objekte müssen feste Hauptspeicheradresse haben
   → DB-Größe durch virtuellen Adreßraum beschränkt!
   nicht akzeptabel bei 32-Bit-Rechnern

2. andere Adresse als vorher
   → Zeiger müssen angepaßt werden, d.h.:
   - beim Speichern Adressen in Objektidentifizierer umsetzen
     d.h. Seite kann nicht unverändert gespeichert werden!
   - beim Laden Adressen in dann gültige Adressen umsetzen

   Bezeichnung hierfür: **pointer swizzling**

Probleme beim *pointer swizzling*:

- braucht Rechenzeit und/oder Speicherplatz
- Applikation kann (bei "unsicheren" Programmiersprachen wie C++) Inhalt der Seite beschädigt haben
  $\rightarrow$ Analyse der Seite vor dem Speichern notwendig
- nicht anwendbar bei heterogenen Plattformen

Wann verbessern die seitenorientierten Verfahren die Performance?

- 1. Zugriff auf ein Objekt: zeitraubender Transport eines Sektors von der Platte in den Hauptspeicher unverändert notwendig

  2. und folgende Zugriffe zum *gleichen* Objekt: Performance verbessert

- Performance-Verbesserung beim initialen Laden: allenfalls durch günstige Gruppierung (Clusterung):
  - mehrere kleine Objekte auf 1 Seite
  - gemäß Zugriffsverhalten EINER Applikation

  Angabe der Clusterungsstruktur durch Applikation! (erzeuge neues Objekt "in der Nähe" eines vorhandenen)

  ABER: diese Angaben gehören zum "*internen Schema*",
  widersprechen dem Ziel der Datenunabhängigkeit
  (die aber bei nichtkonventionellen Anwendungen sowieso nicht erreicht wird)

Fehlende Leistungen von Datenbanksystemen bei seitenorientierten Verfahren.

1. *Sprachunabhängigkeit*:
   DBMS ist abhängig von Compiler oder sogar Versionen "desselben" Compilers

2. *heterogene Plattformen*

3. *Sichten*:
   Basis für Zugriffskontrollen und Datenunabhängigkeit
   Filterung im DBMS → ggf. völliger Umbau der Seiteninhalte
   Filterung im Laufzeitsystem der Programmiersprache: u.U. nicht sicher, bedingt völlig neue Sprachkonzepte

4. *effiziente Suche durch Indexe*:
   Datenstrukturen in Laufzeitsystemen unterstützen i.a. keine (Primär-) Indexe (sind nicht plattenorientiert)
   Sekundärindexe: verschlechtern Performance
   Problem: Daten, die andere Prozesse frisch erzeugt haben

5. *paralleler Zugriff*:
   gleiche Seite in *mehrere* Adreßräume einspiegeln
   → lokale Änderungen der Kopien; wie mischen??
   → Seiten müssen gleichgehalten werden
   → zeitaufwendige Prozeßkommunikation
   Sperren / Concurrency-Control: nicht dezentral handhabbar

6. *Recovery*:
   hoher Aufwand für Logging; wird nicht reduziert
   Schattenobjekte erfordern völlig andere Handhabung von Objekten als in Laufzeitsystemen

Quintessenz:

- klassische Leistungen eines DBMS haben ihren Preis, nur marginale Performance-Gewinne, wenn klassische DBMS-Leistungen gefragt sind
- "Effizienz" der seitenorientierten Verfahren besteht darin, diese Leistungen nicht zu erbringen.

8 Märkte und Standards

OODBMS sind sehr komplexe Systeme → teuere Implementierung

enge Märkte → lange Reifezeit, teuer; als Produkte kaum überlebensfähig

Standards:

- **SQL3**: Erweiterungen von SQL2 um ADTs, rekursive Verbunde, Trigger und weitere Merkmale
- **IRDS** und **PCTE**: spezielle OODBMS, als Basis von Software-Entwicklungsumgebungen konzipiert
- Standards der **ODMG**

Standards der ODMG (Object Data Management Group; hat sich inzwischen aufgelöst)

- *Objektmodell* (**OM**), das grundlegende Konzepte wie Objekt, Typ. Attribut usw. definiert.
- Objekt[typ]definitionssprache (*object definition language*; **ODL**)

  vergleichbar mit DDL

  definiert auch Signaturen von Operationen
- *object query language* (**OQL**)

  Abfragesprache für Objektbanken, angelehnt an SQL
- Sprachanbindungen für die Sprachen C++, Java und Smalltalk

- *object interchange format* (**OIF**)

  Sprache zur Definition von Objekten (Instanzen von Objekttypen); zum Transport von Datenbankinhalten zwischen Datenbanken

Objektrelationale DBMS (ORDBMS)

Basis: erprobtes relationales DBMS

mit Erweiterungen, die die Anforderungen der Objektorientierung zumindest teilweise erfüllen:

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

Umfang und die konkrete Ausgestaltung der Erweiterungen nicht einheitlich bei verschiedenen Produkten