

5. Speicherungsstrukturen

- **Ziel:** Entwurf von
 - Speicherungsstrukturen für Sätze und komplexe Objekte
 - Hilfsstrukturen wie Freispeicherverwaltung, Adressierung usw.
- **Freispeicherverwaltung**
- **Externspeicherbasierte Satzadressierung**
 - TID, Zuordnungstabelle
 - Indexierung von Tabellen (Satzmengen)
- **Hauptspeicherbasierte Satzadressierung**
 - Klassifikation der Lösungskonzepte
 - *Pointer-Swizzling*-Verfahren
- **Abbildung von Sätzen**
 - feste/variable Felder, Partitionierung
- **Speicherungsstrukturen für komplexe Objekte**
 - Listen- und Mengenkonstruktoren
 - Tupelkonstruktoren
- **Probleme großer Objekte**
- **Speicherungsstrukturen für LOBs**
 - Segmente fester und variabler Größe
 - Zugriff über B*-Baum, Zeigerliste, . . .
- **DB-Anbindung externer Daten**
 - DataLinks-Konzept
 - Referentielle Integrität, Zugriffskontrolle, Transaktionskonsistenz, Koordiniertes Backup und Recovery

Speicherungsstrukturen

- **Operationen**

insert <record> at <location> with <database-key>

retrieve <record> with <database-key>

add <entry> to <B*-tree>

retrieve <address-list> from <B*-tree> for <value>

Abbildungsfunktionen

- Satz-Identifikator \leftrightarrow address
- Attributwert \leftrightarrow record-id.list
- Satz-Identifikator \leftrightarrow record-id.list
- Adresse \leftrightarrow {occupied, free}

FIX P_i , FIX P_j , UNFIX P_j ,

FIX P_k , UNFIX P_i , ...

- **Eigenschaften der oberen Schnittstelle**

- Nicht-flüchtiger Speicher mit Adressierungshilfen
- Freispeicherverwaltung
- Adressierungsverfahren von und zwischen physischen Sätzen
- Zugriffspfade zur Realisierung von Inhaltsadressierbarkeit

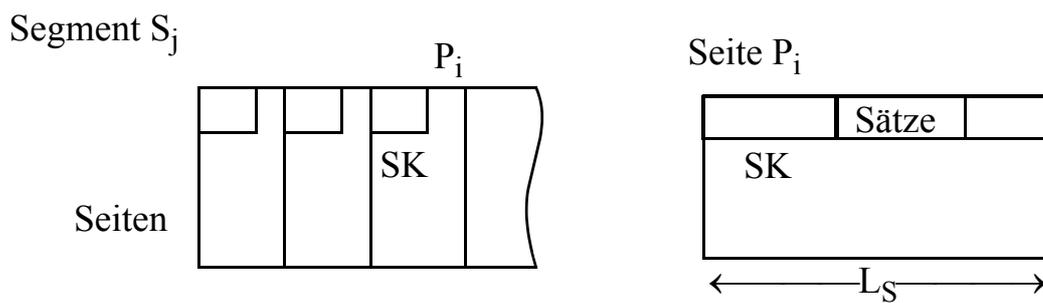
Freispeicherverwaltung

- **Freispeicherverwaltung (FPA) für**

- Externspeicher (Allokation von Dateien)
- Segmente (Allokation von internen Sätzen)
- Seiten (Verwaltung von belegten/freien Einträgen)

- **Für alle Seiten eines Segmentes:**

- Einfügen/Ändern → Suche nach n freien Bytes
- Löschen/Ändern → Freigabe oder Markierung von Speicherplatz
- allgemein: Suche, Belegung und Freigabe von Speicherplatz in S_j



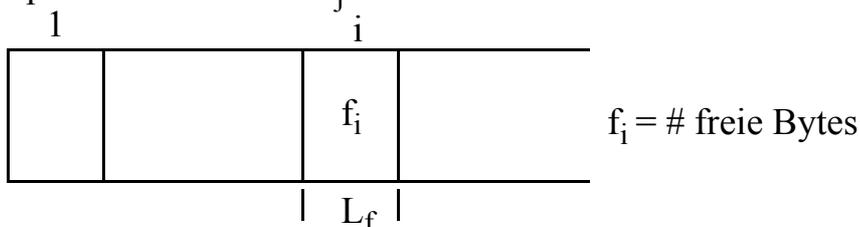
➔ in SK (Seitenkopf):

ID von P_i ,

Freiplatz-Info,

Typ, Org.-Daten

Freispeichertabelle F in S_j



Freispeicherverwaltung (2)

- **Größe von F**

Einträge pro Seite der Länge L_S

$$k = \left\lfloor \frac{L_S - L_{SK}}{L_f} \right\rfloor$$

mit $s = \#$ Seiten im Segment

↳ $n = \left\lceil \frac{s}{k} \right\rceil$ Seiten für F

- **Lage von F**

- Segmentanfang
- äquidistante Verteilung $i \cdot k + 1$ ($i=0,1,2,\dots$)
- Segmentende

- **Art der FPA**

- exakt: $L_f = 2\text{Bytes}$
- unscharf: $L_f = 1\text{Byte}$ (oder weniger)

Einheiten von $f_i \rightarrow \lceil L_S / 256 \rceil$ - Vielfache

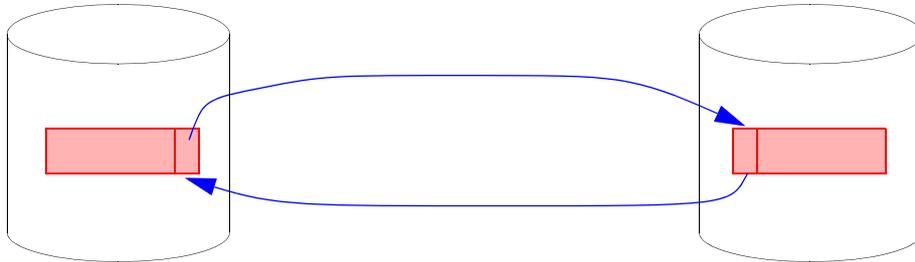
bei $L_S = 4\text{KB} \rightarrow 16\text{Bytes}$

- **FPA innerhalb von P_i**

- exaktes f_i in SK
- zusammenhängende Verwaltung (Verschiebungen!)
- Freispeicherkette (*best-fit / first-fit*)

Externspeicherbasierte Satzadressierung

- **Problemstellung**



- langfristige Speicherung der Datensätze
- Vermeiden von „Technologieabhängigkeiten“
- Unterstützung von Migration u. a.

- **Allgemeine Form einer Satzadresse**

- DBID, SID, TID und ggf. Relationenkennzeichnung (RID)
- Relation vollständig in einem Segment gespeichert: TID
DBID, SID im DB-Katalog
- Relation in mehreren Segmenten: SID, TID

- **Ziele der Adressierungstechnik:**

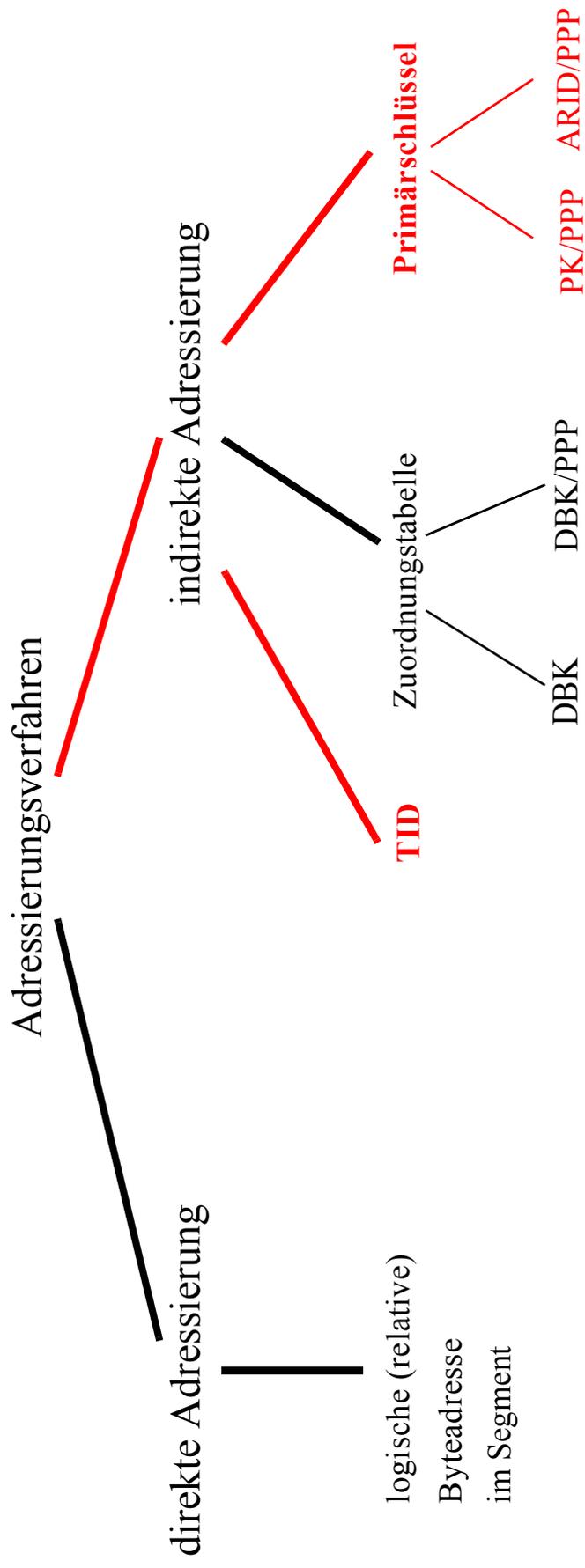
- schneller, möglichst direkter Satzzugriff
- hinreichend stabil gegen geringfügige Verschiebungen
(Verschiebungen innerhalb einer Seite ohne Auswirkungen)
- seltene oder keine Reorganisationen

- **Adressierung in Segmenten**

- logisch zusammenhängender Adreßraum
- direkte Adressierung (logische Byte-Adresse, RBA)
 - ➔ instabil bei Verschiebungen

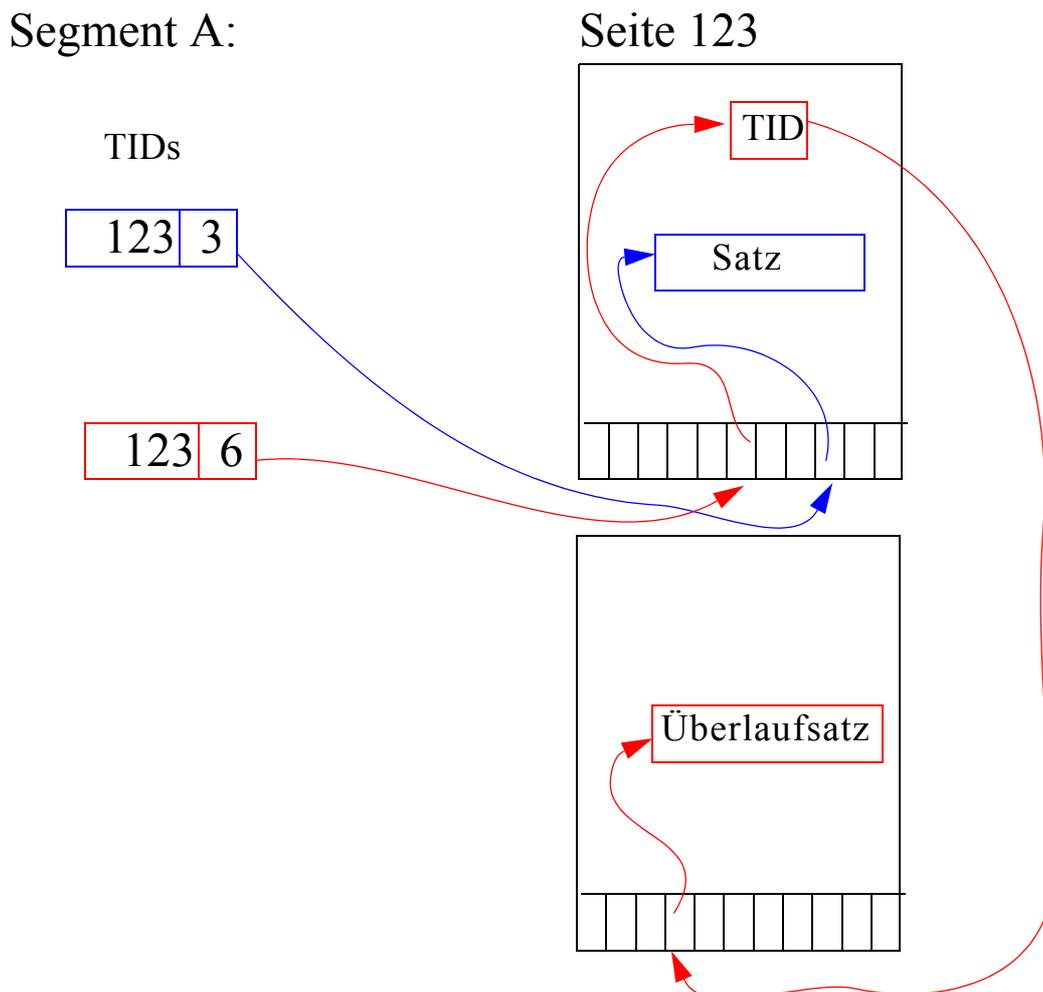
➔ deshalb indirekte Adressierung

Techniken zur externspeicherbasierten Satzadressierung



Satzadressierung: TID-Konzept

- **TID (*tuple identifier*) besteht aus zwei Komponenten:**
 - Seitennummer (3 B)
 - relative Indexposition innerhalb der Seite (1 B)
 - dient zur Adressierung in einem Segment (z. B. SID = A)
- **Migration eines Satzes in andere Seite ohne TID-Änderung möglich**
 - ↳ Einrichten eines Stellvertreter-TID in Primärseite
- **Überlaufkette: Länge ≤ 1**

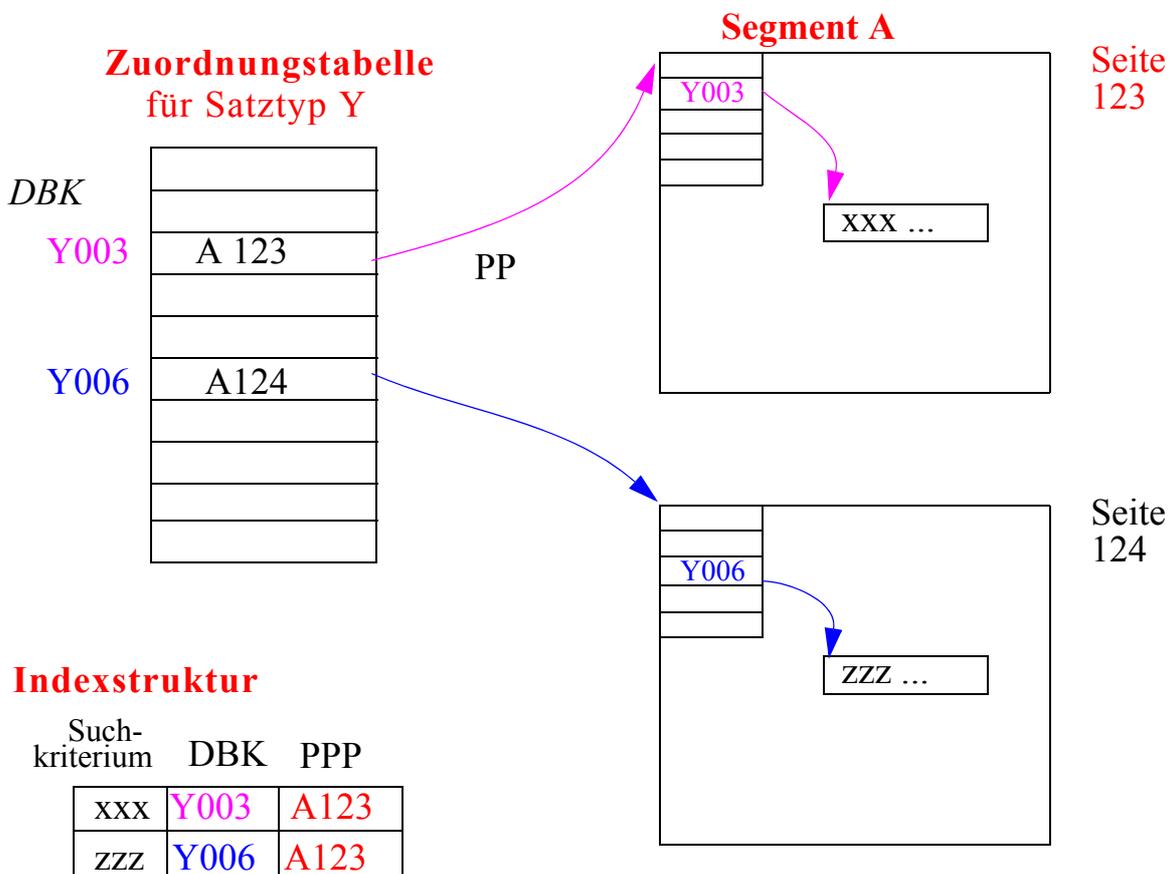


Satzadressierung über Zuordnungstabelle

- **Jeder Satz erhält eindeutigen logischen Identifikator**
 - Datenbankschlüssel (DBK)
 - Vergabe der DBK erfolgt i. allg. durch DBVS
 - systeminterne Verweise auf Sätze erfolgen ausschließlich über den DBK

- **Zuordnungstabelle enthält pro DBK zugehörigen PP**
 - SID (1 B)
 - Seitennummer (3B)

- **Hybrides Verfahren:**
 Verwendung von 'probable page pointers' (PPP) in Zugriffspfaden erspart u. U. Zugriff auf Zuordnungstabelle



Indexierung von Tabellen

- **Speicherung von Tabellen**

- **ungeordnete Tabelle:**
Sätze (Zeilen) sind im Segment verstreut (Heap)
- **geordnete Tabelle:**
Sätze sind in B*-Baum eingebettet (key-sequenced table);
es wird dadurch eine Clusterbildung erzielt

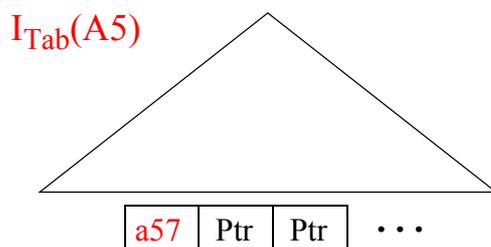
➔ Wir bezeichnen eine solche Tabelle als **Index-organisierte Tabelle (IT)**

- **Indexierung von Tabellen**

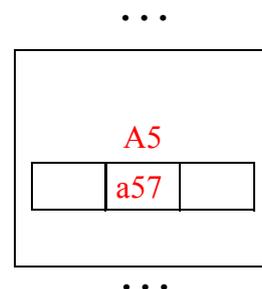
- mit sekundären Indexen für Spalten A_i : $I_{Tab}(A_i)$
- Nutzung verschiedener Adressierungsverfahren
 - TID (physisch)
 - DBK (indirekt: logisch/physisch)
 - PK (Primärschlüssel: logisch)
 - hybride Verfahren

- **Wie spielen Adressierung und Tabellenspeicherung zusammen?**

- **Ungeordnete Tabelle**



Segment

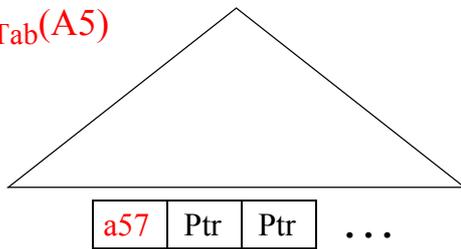


- Sätze verschieben sich bei Aktualisierung nicht (kaum)
- Adressierungsverfahren (Ptr):
Es kommen TID, DBK und DBK/PPP in Frage
- Indexunterstützung für ungeordnete Tabellen in DB2, Sybase, MS SQL-Server, Oracle, ...

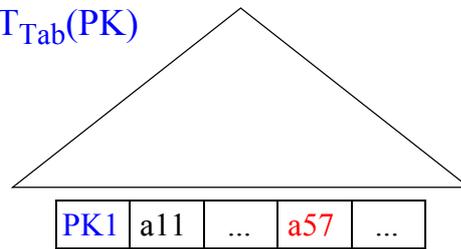
Indexierung von Tabellen (2)

- **Index-organisierte Tabelle**

$I_{Tab}(A5)$

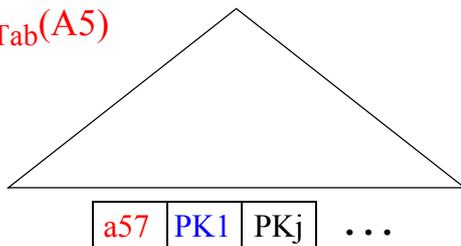


$IT_{Tab}(PK)$

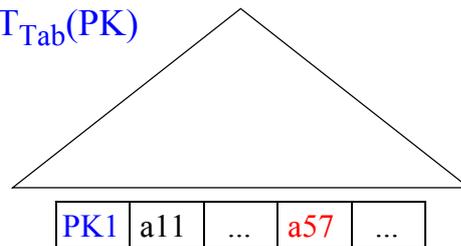


- Split in IT_{Tab} erfordert viele Adressanpassungen in $I_{Tab}(A_i)$
 - bei TID
 - bei DBK
 - bei DBK/PPP
- Verbesserung: logische Adressierung

$I_{Tab}(A5)$



$IT_{Tab}(PK)$



- keine Wartung der $I_{Tab}(A_i)$ bei Split/Verschiebungen in IT_{Tab} nötig
- **aber:** höhere Zugriffskosten bei Index-Scan usw.

- **Nutzung einer hybriden Adressierungstechnik**

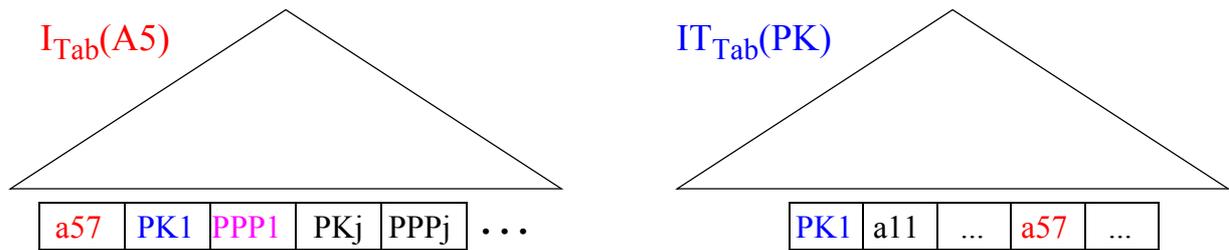
- Verweis hat zwei Komponenten
 - logischer Verweis: PK
 - physischer Verweis: wahrscheinliche DB-Seite (PPP, Guess-DBA)
- Eintrag in Index

Attributwert	PK	PPP
--------------	----	-----

Indexschlüssel

HRID = (Hybrid Row Identifier)

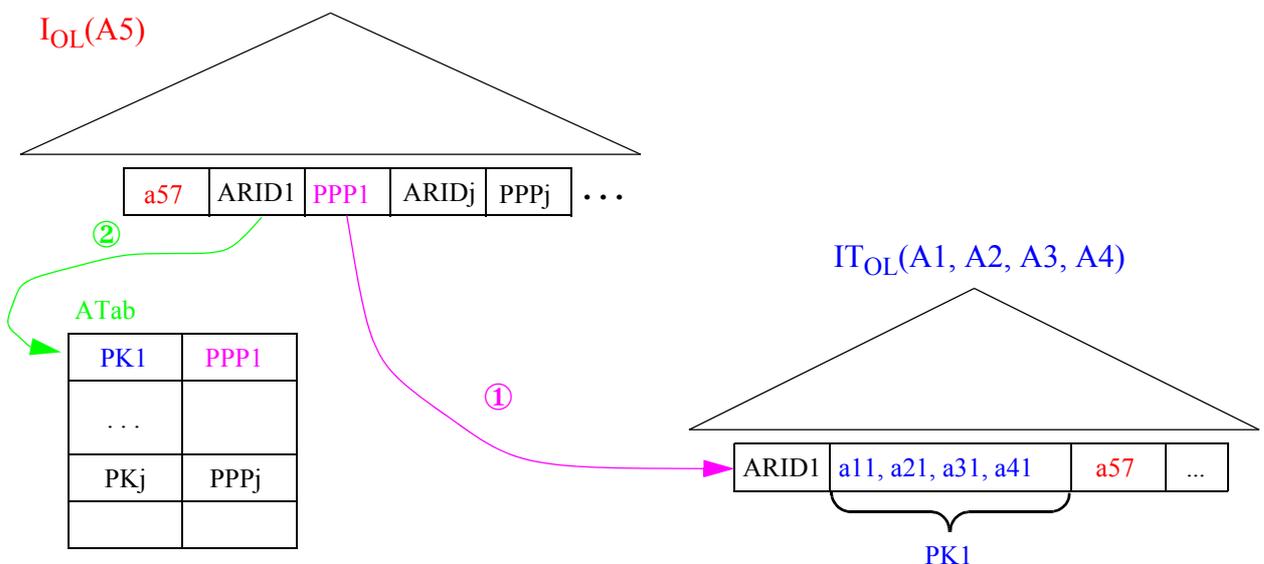
Indexierung von Tabellen (3)



- Vereinigung der Vorteile beider Verfahren
- Was passiert bei langen Primärschlüsseln?

• Optimierung bei langen Primärschlüsseln

- Beispiel: Tabelle Order_Line des TPC-C-Benchmark:
OL (ol-o-id, ol-w-id, ol-d-id, ol-number, ol-i-id, ...)
- Vereinfachte Schreibweise:
OL (A1, A2, A3, A4, A5, ...)
- Vermeidung der PK-Speicherung im Index (Oracle-Lösung)
 - Nutzung einer Abbildungstabelle ATab
 - Verweis auf ATab durch ARID



- Falls der Zugriff über PPP ① fehlschlägt, wird mit Hilfe von $ARID$ ② ATab aufgesucht
- Alle $I_{OL}(A_i)$ benutzen ATab
- Von dort kann über PPP oder über PK auf IT_{OL} zugegriffen werden

Hauptspeicherbasierte Adressierung

- **Aufgabe:**

Programme sollen im HSP **transiente und persistente Datenobjekte transparent** verarbeiten können.

- Ausschließliche Nutzung von direkten Adressen im HSP (Virtuelle Adressierung), d. h., Zugriff auf persistente Objekte ist im HSP genauso effizient wie auf transiente Objekte.
- Keine Mehrkosten für Programme, die nur auf transiente Objekte zugreifen
- Abbildungskosten für persistente Objekte sollen nicht bei jedem Zugriff anfallen

- **Abbildung von persistenten Objekten** auf Externspeichern (ES) auf solche in Virtuellen Speichern (VS)

- Persistente Adressen (z. B. SID, RID, TID) sind lang (z. B. 64 Bit), Virtuelle Adressen dagegen kürzer (z. B. 32 Bit)
- Übersetzung der Zeiger (***pointer swizzling***) vom langen Format mit indirekter Adressierung ins kürzere Format mit möglichst direkter Adressierung

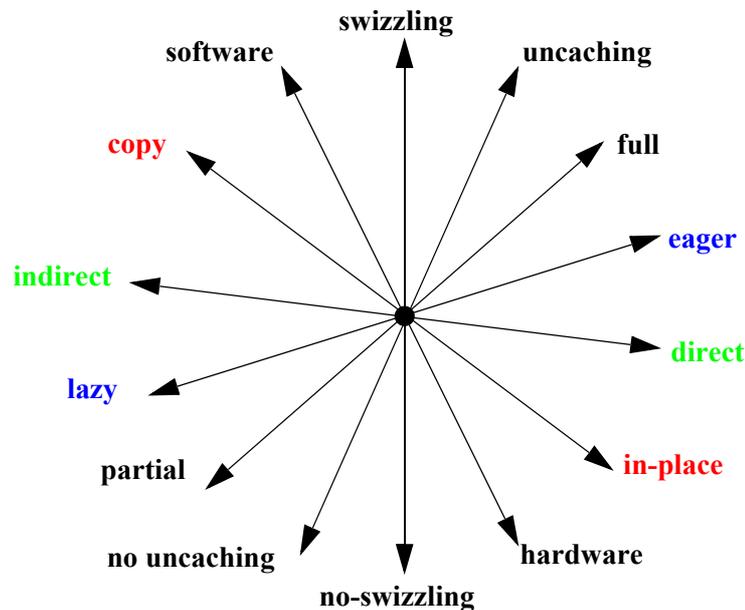
- **Ziel:**

Schnelle Verarbeitung von Pointerfolgen im VS — z. B. 10^5 Refs/sec

- Objektverarbeitung: Traversierung von Referenzfolgen und Navigation in vernetzten Objektstrukturen
- Direkter Zugriff im HSP ist wesentlich billiger als Zugriff über persistente Adresse (Lokalisierung einer Seite im DB-Puffer und Suche des Objektes in der Seite)
- ggf. zusätzliche Zugriffspfade zur Suche im HSP:
B*-Baumzugriff erfordert h+1 direkte Pointerreferenzen

Pointer Swizzling

- Dimensionen von Pointer Swizzling¹



- Klassifikation von Swizzling-Verfahren

- wichtigste Kriterien: Ort, Zeitpunkt und Art (orthogonal)

- **Ort:**

- *In-Place Swizzling*: Beibehaltung der Objektformate und der Seitenstrukturen

- *Copy Swizzling*: Kopieren der Objekte in einen Puffer und Umstellen der Zeiger in den Kopien

- **Zeitpunkt:**

- *Eager Swizzling*: Umstellen aller Zeiger, sobald die Objekte in den Hauptspeicher gebracht werden

- *Lazy Swizzling*: Umstellen der Zeiger bei Erstreferenz oder später (nach beliebigen Kriterien — magische Zahl 3)

- **Art:**

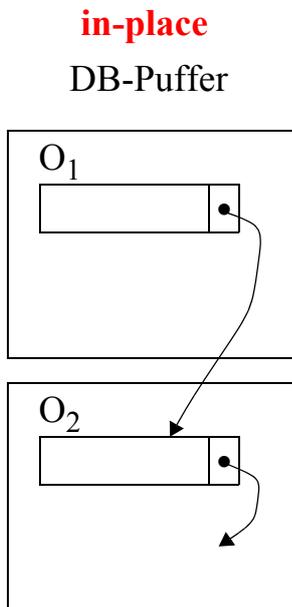
- *Direct Swizzling*: Nutzung der Virtuellen Adresse des Objektes, dadurch kann die Ersetzung von Objekten während der Verarbeitung sehr schwierig oder gar unmöglich werden

- *Indirect Swizzling*: Nutzung der Virtuellen Adresse von Objekt-Deskriptoren

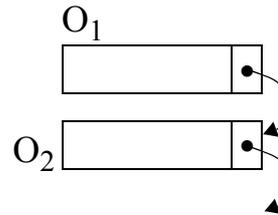
1. White, S.J., DeWitt, D.J.: Quickstore: A High Performance Mapped Object Store, in: The VLDB Journal 4:4, Oct. 1995, pp. 629-674.

Pointer Swizzling (2)

- **Klassifikationskriterium: Ort**



copy
Objektpuffer (Heap) befindet sich typischerweise im Client-Rechner

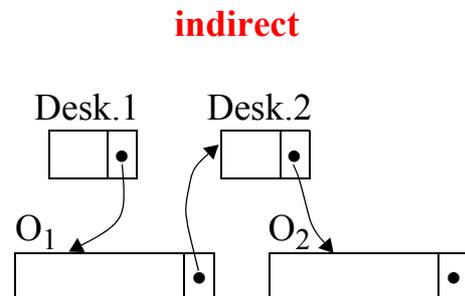
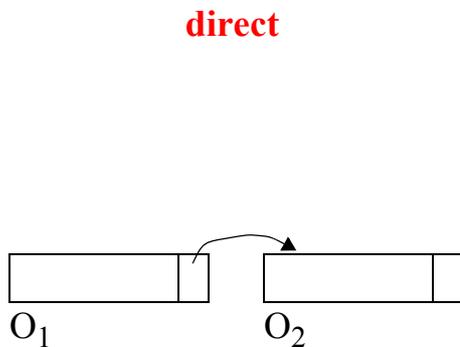


- **Klassifikationskriterium: Zeitpunkt**

eager
sobald im HSP
(Lawineneffekt)

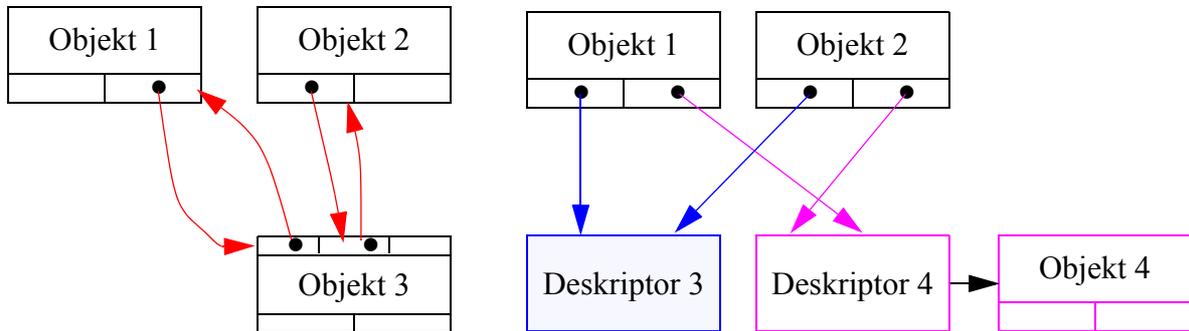
lazy
viele Möglichkeiten

- **Klassifikationskriterium: Art**



Pointer Swizzling (3)

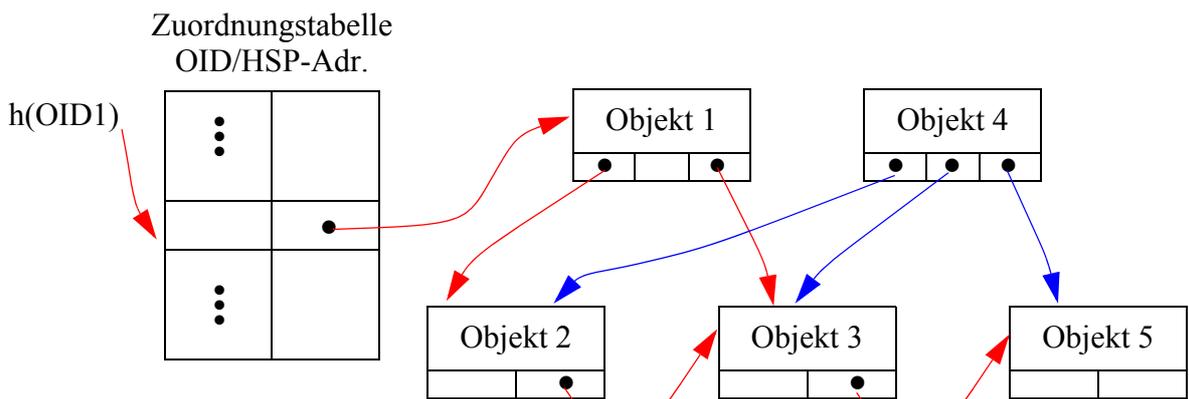
- Direktes und indirektes Swizzling – Prinzip**



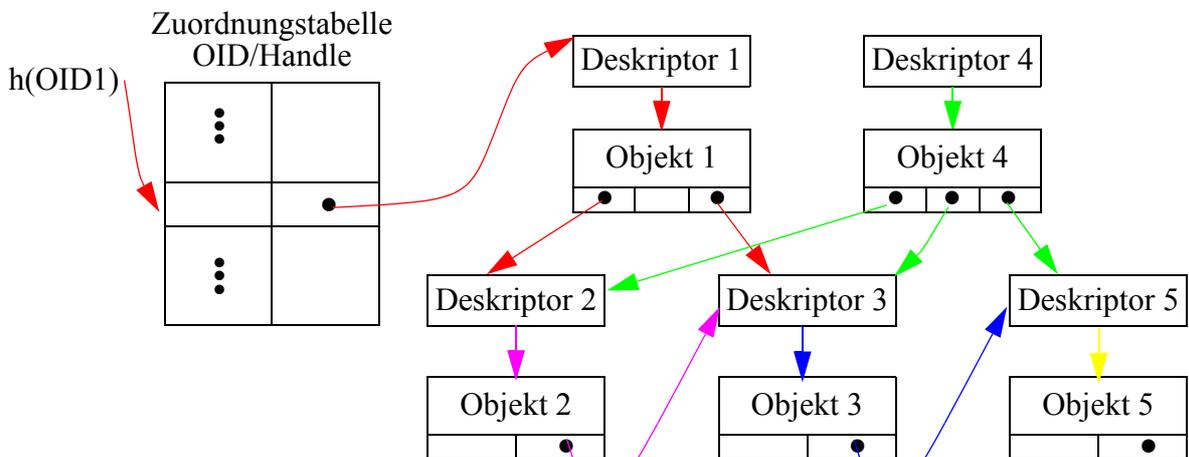
a) Symmetrische Referenzen

b) Referenzierung von Deskriptoren

- Direkte und indirekte Variante beim Copy Swizzling**



a) Direktes Swizzling in einem Objektpuffer



b) Indirektes Swizzling in einem Objektpuffer

Pointer Swizzling (4)

- **Tests**

Lazy: Test, ist Swizzling schon erfolgt?

Eager: kein Test

Indirect: Test, ist Objekt schon/noch da?

Direct: kein Test
(aber kein Ersetzen des Objektes nach Swizzling oder Unswizzling)

- **Kosten**

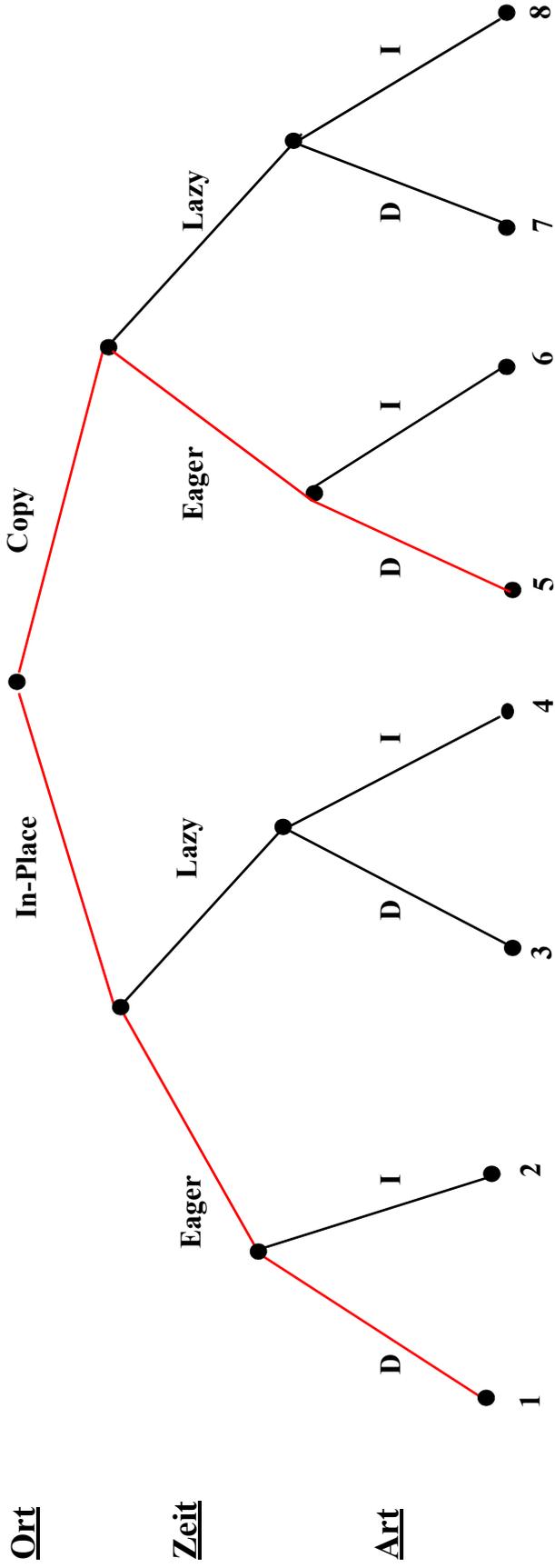
Eager/Direct: 0 Tests (kein Ersetzen)

Eager/Indirect: 1 Test (im Deskriptor)
(→ Ersetzen mit Referenz-Zähler)

Lazy/Direct: 1 Test (+ Heuristik-Kosten)
(→ Ersetzen mit symmetrischen Zeigern)

Lazy/Indirect: 2 Tests (+ Heuristik-Kosten (> 3 Refs))

Pointer Swizzling (5)



Bemerkungen:

- 1 + 5 : Swizzling von allen Seiten/Objekten bei Checkout, kein Ersetzen (*no uncaching*)
- 2 + 4 : Umständliche Organisation

Fragen:

- Welche Verfahren sind bei der Verarbeitung (beim Swizzling) am schnellsten?
- Welche Verfahren erlauben Objektersetzung (*uncaching*)?
- Wie kann Lazy/Direct (3 + 7) realisiert werden?

Abbildung von Sätzen

- **Record-Mgr:**

- physische Abspeicherung von Sätzen in Seiten
- Operationen: Lesen, Einfügen, Modifizieren, Löschen

- **Satzbeschreibung**

- pro Attribut:

Attributname	Typ	...		Länge	Attributwert
Vorname	Char	var	...	5	Xaver

Metadaten im Katalog (DD) Ausprägung im Satz

- Satz- und Zugriffspfadbeschreibung im Katalog
- besondere Methoden der Speicherung
 - Blank-/Nullunterdrückung
 - Zeichenverdichtung
 - kryptographische Verschlüsselung
 - Symbol für undefinierte Werte
- Tabellenersetzung für Werte: KL = Kaiserslautern

- **Organisation**

- n Satztypen pro Segment
- m Sätze verschiedenen Typs pro Seite
- Satzlänge < Seitenlänge: $S_L \leq L_S - L_{SK}$

Speicherungsstrukturen für Sätze

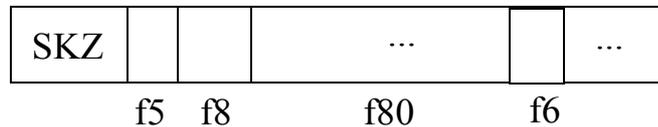
- **Entwurfsziele**

- Speicherplatzökonomie
- schnelles Auffinden des i-ten Feldes
(weitgehende Berechnung aus Kataloginformation)
- dynamische Erweiterung (ALTER TABLE ...)

- **Konkatenation von Feldern fester Länge**

Katalog: f5 | f8 | f80 | f6 | ... |

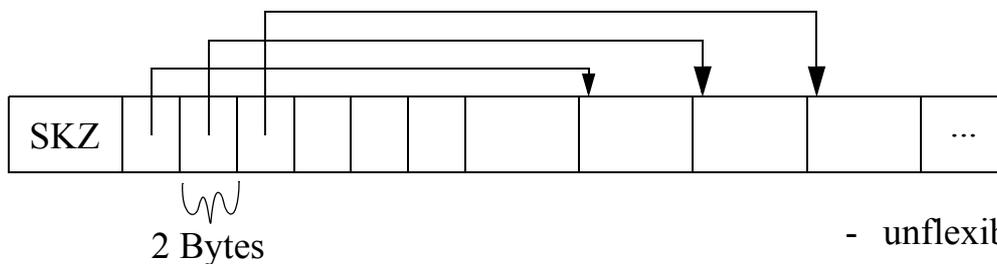
z. B. TID



- speicheraufwendig
- unflexibel

- **Zeiger im Vorspann**

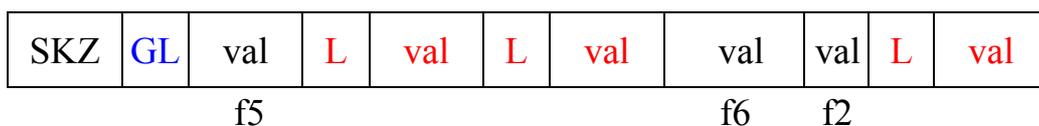
Katalog: f5 | v | v | f6 | ... |



- unflexibel

- **Eingebettete Längfelder**

Katalog: f5 | v | v | f6 | f2 | v |

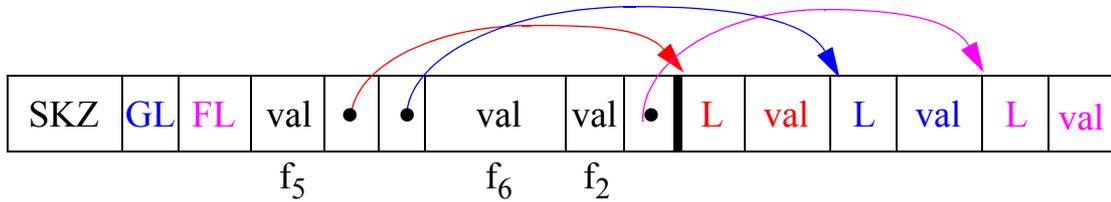


- stärkere Nutzung des Katalogs
- dynamische Erweiterung möglich

Speicherungsstrukturen für Sätze (2)

- Optimierung: eingebettete Längfelder mit Zeigern**

Katalog: f5 | v | v | f6 | f2 | v |



- Adresse des n-ten Attributs kann berechnet werden
- dynamische Erweiterbarkeit

- Satzabbildung: Bewertung der Verfahren**

	Konkatenation von Feldern fester Länge	Zeiger im Vorspann	Eingebettete Längfelder	Eingebettete Längfelder mit Zeigern
Speicherplatz-ökonomie				
Zugriffsgeschwindigkeit innerhalb des Satzes				
Erweiterbarkeit				

Speicherungsstrukturen für Sätze (3)

- Anforderungen

	Vorname	Name	Beruf
OID	Xaver		

- ≥ 200 Attribute/Satz ?
- $S_L \leq L_S - L_{SK}$
- Muss passen für n relationale DBMS
- Indexierung

- Extreme Lösung: AOW

A	OID	W
Vorname	XYZ 0815	Xaver

Abbildung auf n-äre Relation DR

AID	OID	W1	W2	W3	W4	W5
A15	XYZ 0815	∅	∅	∅	Xaver	∅

Typen: INT FLOAT MONEY VARCHAR OWN

- Suche des gesamten Tupels mit 200 Attributen ?

➔ über OID

- Index auf allen Attributen

```

Suche:      Select *
            FROM    Pers(DR)
            WHERE   Vorname = 'Xaver'
            (OR) AND Beruf = 'Prog.'
            (OR) AND Alter > 50
    
```

Speicherungsstrukturen für Sätze (4)

- **Problem: dynamisches Wachstum/variable Länge**

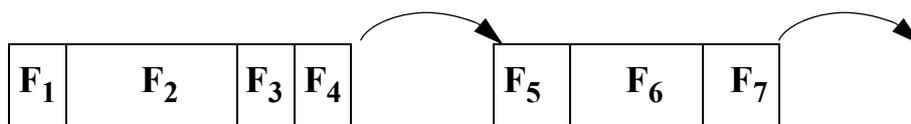
- Ausdehnung und Schrumpfung in einer Seite
- Überlaufschemas, Garbage Collection

↳ Eingeführte Möglichkeiten der Speicherung von Sätzen sind mit weiteren Optionen zu kombinieren

- **Strikt zusammenhängende Speicherung von Sätzen**

- evtl. häufige Umlagerung bei hoher Änderungsfrequenz
- Vorteile für indirekte Adressierungsschemata

- **Aufspaltung des Satzes**



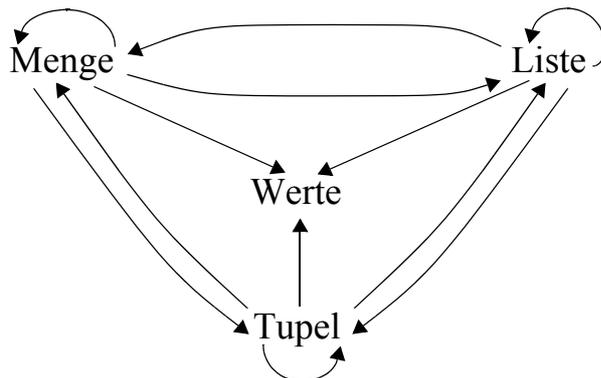
- Ordnung nach Referenzhäufigkeiten
- Verbesserung der Clusterbildung
- Wiederholter Überlauf möglich
- wird unvermeidlich bei der Einbeziehung von Attributen vom Typ TEXT oder BILD

Speicherungsstrukturen für komplexe Objekte¹

- **Komplexe Objekte werden gebildet aus**

- atomaren Werten und darauf
- rekursiv angewandten Mengen-, Listen- und Tupelkonstruktoren

- **Modell für komplexe Objekte (eNF²)**



- **Speicherungsstrategie**

- Orthogonalität ist wichtig: keine Enumeration aller Möglichkeiten
- Ablegen von häufig gemeinsam zugegriffenen Substrukturen in eine oder wenige Speichereinheiten
- selten gemeinsam zugegriffene Substrukturen separieren

➔ **Anwendungswissen!**

- **Leistungsaspekte**

- werden bei komplexen Objekten/Operationen wesentlich bestimmt von den Speicherungsstrukturen
- Minimierung von Ein-/Ausgabe: Clusterbildung
- Berücksichtigung des Wachstums beim Anlegen eines komplexen Objektes

1. Keßler, U., Dadam, P.: Benutzergesteuerte, flexible Speicherungsstrukturen für komplexe Objekte, Proc. BTW'93, Braunschweig, 1993, S. 206-225.

Speicherungsstrukturen für komplexe Objekte (2)

- Einfaches Beispiel

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

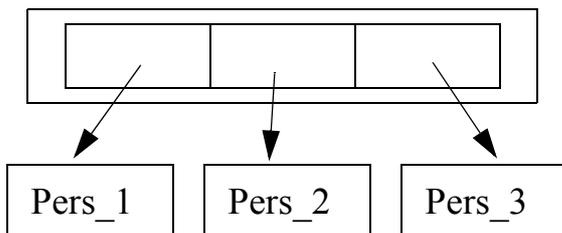
[. . .] kennzeichnet Stelle für Speicherungsstrukturbeschreibung

- Freiheitsgrade für physische Speicherungsstrukturen

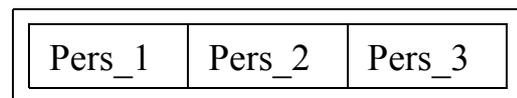
1. Wahl der internen Speicherungsstrukturen zur Implementierung von Mengen, Listen und Tupeln (**Konstruktordatenstruktur**)
2. **Direkte Speicherung oder Referenzierung** der Elemente einer Menge oder Liste bzw. der Attribute eines Tupels **in der Konstruktordatenstruktur**

- Jeder Konstruktor hat eine Konstruktordatenstruktur

- Beispiel: einfache Menge {Pers_1, Pers_2, Pers_3}
- variabel langer Array als Konstruktordatenstruktur



Referenzierte Speicherung



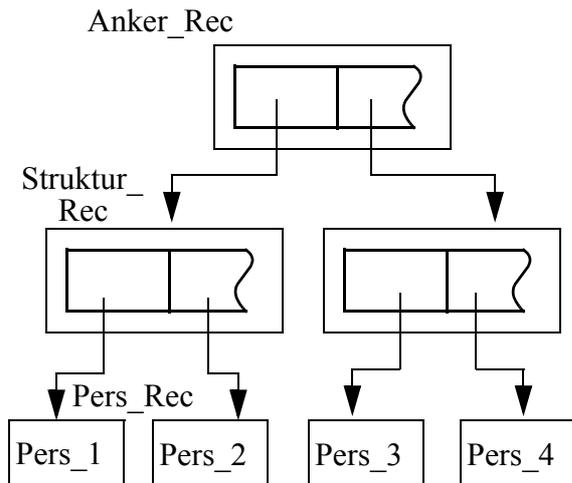
Materialisierte Speicherung

Speicherungsstrukturen für komplexe Objekte (3)

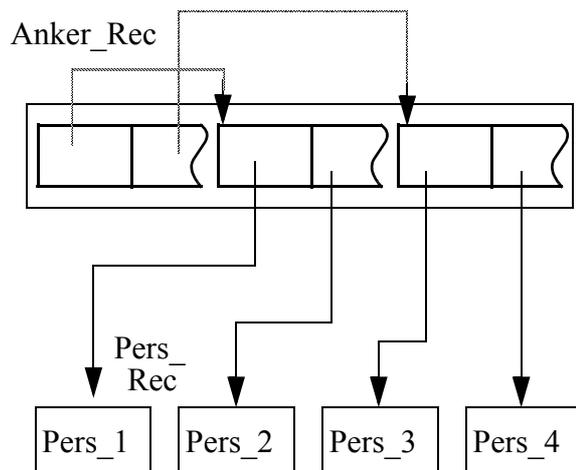
- Zweimalige Anwendung des Mengenkonstruktors**

- { {Pers_1 , Pers_2} , {Pers_3 , Pers_4} }
- Vorgabe variabel langer Arrays als Konstruktordatenstrukturen

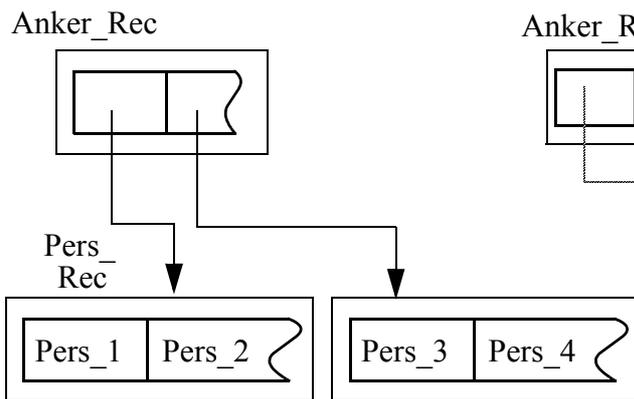
- Vier Implementierungen**



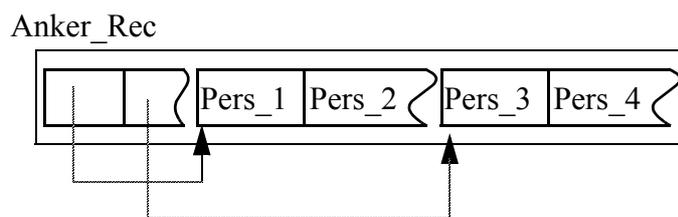
1. Elemente äußere Menge : referenziert
 Elemente innere Menge : referenziert



2. Elemente äußere Menge : materialisiert
 Elemente innere Menge : referenziert



3. Elemente äußere Menge : referenziert
 Elemente innere Menge : materialisiert



4. Elemente äußere Menge : materialisiert
 Elemente innere Menge : materialisiert

- Sind zusätzlich verkettete Listen als Konstruktordatenstrukturen zulässig, so erhält man insgesamt 16 Varianten

Speicherungsstrukturen für Mengen- und Listenkonstruktoren

- **Unabhängige Freiheitsgrade**

- Konstruktordatenstruktur
 - variabel langes Array
 - verkettete Liste
 - ...
- Art der Speicherung der Elemente
 - direkt in Konstruktordatenstruktur
 - Referenzierung der Elemente über Zeiger

- Zur **unabhängigen Spezifikation dieser Freiheitsgrade** sind zwei Parameter (in einer Datendefinitionssprache) erforderlich:

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

- **Parameterwerte**

```
implementation_type = array | linked_list
placement_type      = inplace | referenced (record_type_name)
```

- **Vollständige Definition der Speicherungsstruktur (Fall 1)**

```
complex_object Menge_von_Mengen_von_Pers [anchor_record_type=Anker_Rec]
    set [implementation=array, element_placement=referenced (Struktur_Rec)] of
    set [implementation=array, element_placement=referenced (Pers_Rec)] of
    Pers.
```


Speicherungsstrukturen – Beispiel

- Ausprägung einer Mitarbeiterrelation**

Mitarbeiter			
Pers_Nr	Name	Gehalt	Lebenslauf
77234	Maier	4000	Frau Bettina Maier ist am ...
77235	Schmidt	5000	Herr Fritz Schmidt ist am ...

- Definition einer dazugehörigen Speicherungsstruktur**

- referenzierte Prim_Rec

```

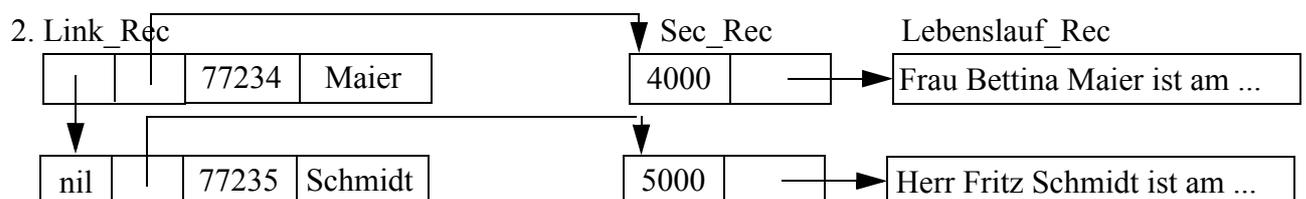
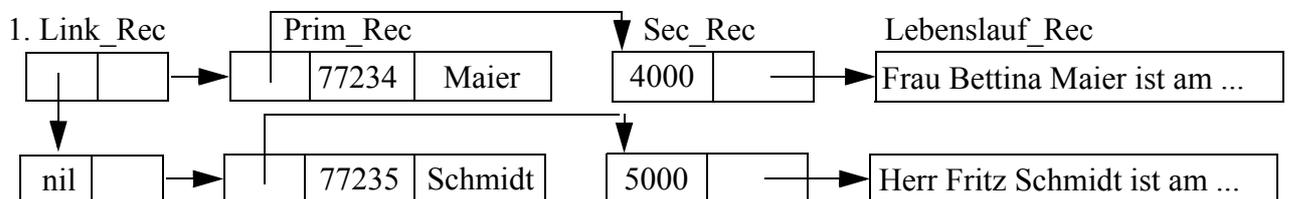
complex_object Mitarbeiter [anchor_record_type=Link_Rec]
  set [implementation=linked_list, element_placement=referenced (Prim_Rec)]
  of tuple
    (Pers_Nr      [location=primary, element_placement=inplace] : integer,
     Name        [location=primary, element_placement=inplace] : string (30),
   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

- materialisierte Prim_Rec

```

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

- Dazugehörige Speicherungsstrukturen für die Mitarbeiterrelation**



Große Objekte

- **Anforderungen**

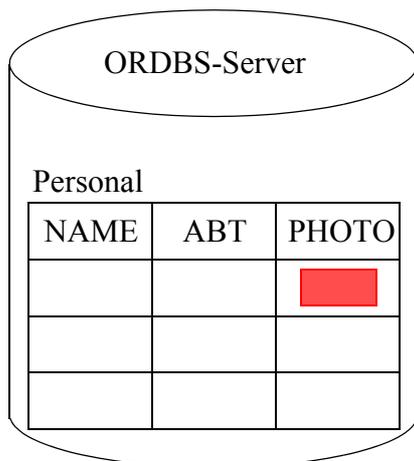
- idealerweise keine Größenbeschränkung
- allgemeine Verwaltungsfunktionen
- zugeschnittene Verarbeitungsfunktionen, . . .

- **Beispiele für große Objekte (heute bis n (=4) TByte)**

- Texte, CAD-Daten
- Bilddaten, Tonfolgen
- Videosequenzen, . . .

- **Prinzipielle Möglichkeiten der DB-Integration**

Speicherung als LOB in der DB (meist indirekte Speicherung)

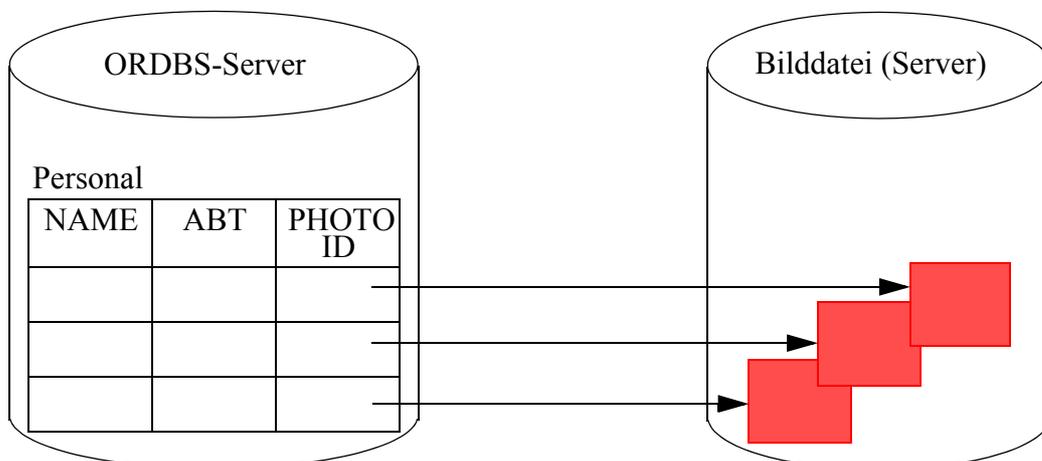


BLOB - Binary Large Object
für Tonfolgen, Bilddaten usw.

CLOB - Character Large Object
für Textdaten

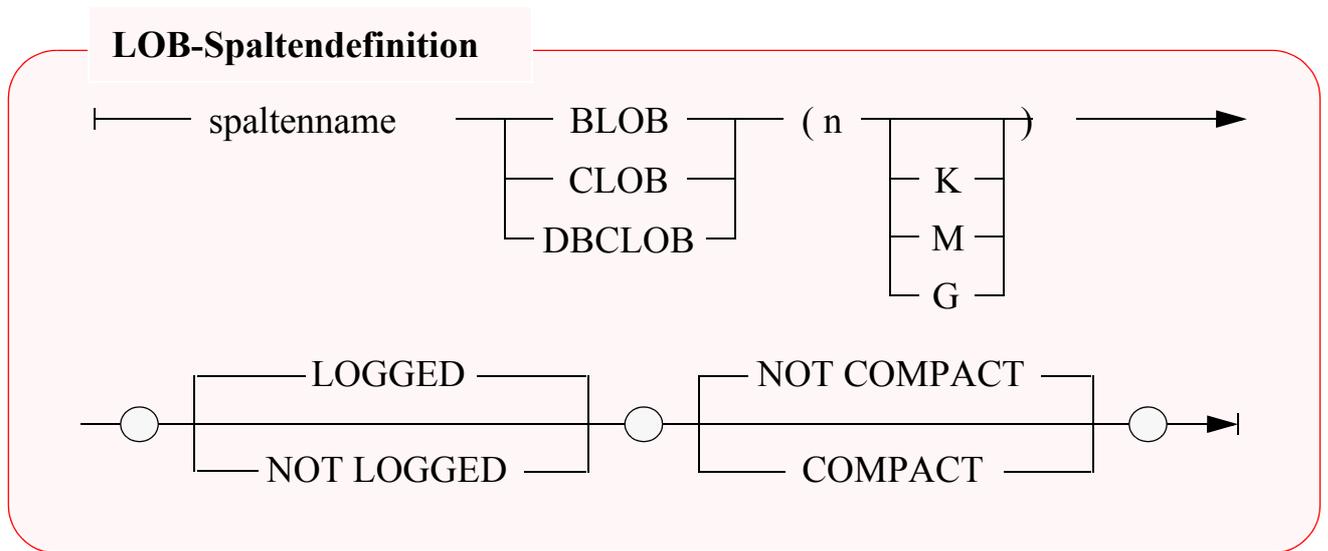
DBCLOB - Double Byte Character
Large Object (DB2)
für spezielle Graphikdaten usw.

Speicherung mit DataLinks-Konzept in externen Datei-Servern



Große Objekte (2)

- Erzeugung von LOB-Spalten¹



- Beispiele

<pre>CREATE TABLE (Lfdnr Name ... Photo Lebenslauf</pre>	<pre>Absolvent Integer, Varchar (50), ... BLOB (5 M) NOT LOGGED COMPACT, CLOB (16 K) LOGGED NOT COMPACT);</pre>	<pre>-- Bild -- Text</pre>
<pre>CREATE TABLE (Teilnr Änderungsstand Geändert_von Zeichnung</pre>	<pre>Entwurf Char (18), Timestamp, Varchar (50) BLOB (2 M) LOGGED NOT COMPACT);</pre>	<pre>-- Graphik</pre>
<pre>ALTER TABLE ADD COLUMN</pre>	<pre>Absolvent Diplomarbeit CLOB (500 K) LOGGED NOT COMPACT;</pre>	

1. Die Realisierungsbeispiele beziehen sich auf DB2 – Universal Database

Große Objekte (3)

- Spezifikation von LOBs erfordert Sorgfalt

- **maximale Länge**

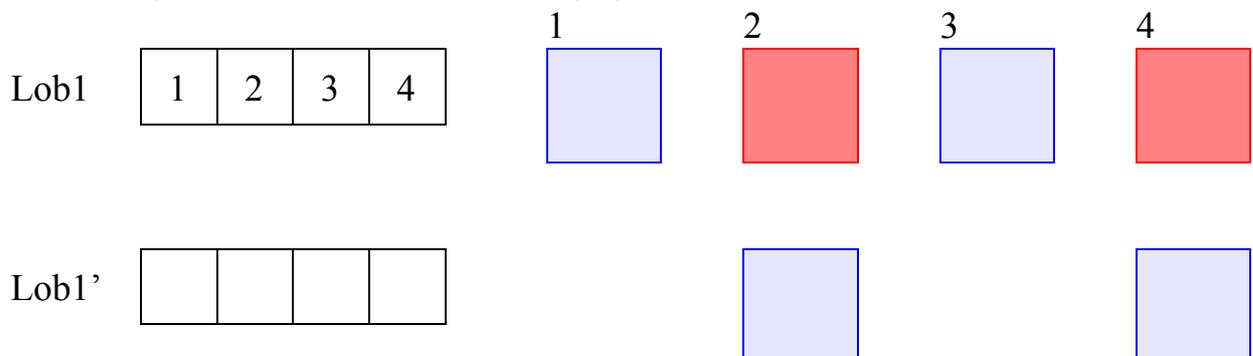
- Reservierung eines Anwendungspuffers
 - Clusterbildung und Optimierung durch indirekte Speicherung; Deskriptor im Tupel ist abhängig von der LOB-Länge (72 Bytes bei <1K bis 316 Bytes bei 2G)
 - bei kleinen LOBs (< Seitengröße) direkte Speicherung möglich

- **kompakte Speicherung**

- **COMPACT** reserviert keinen Speicherplatz für späteres Wachstum
 - ➔ Was passiert bei einer LOB-Aktualisierung?
 - **NOT COMPACT** ist Default

- **Logging**

- **LOGGED**: LOB-Spalte wird bei Änderungen wie alle anderen Spalten behandelt (ACID!)
 - ➔ Was bedeutet das für die Log-Datei?
 - **NOT LOGGED**: Änderungen werden nicht in der Log-Datei protokolliert. Sog. Schattenseiten (shadowing) gewährleisten Atomarität bis zum Commit



➔ Was passiert bei Gerätefehler?

Große Objekte (4)

- **Wie werden große Objekte verarbeitet?**

- BLOB und CLOB sind keine Typen der Wirtssprache

- ➔ **Spezielle Deklaration von BLOB, CLOB, ... durch SQL TYPE ist erforderlich**, da sie die gleichen Wirtssprachentypen benutzen. Außerdem wird sichergestellt, dass die vom DBS erwartete Länge genau eingehalten wird.

- **Vorbereitungen im AWP erforderlich**

- SQL TYPE IS CLOB (2 K) c1 (oder BLOB (2 K))
wird durch C-Precompiler übersetzt in

```
static struct c1_t
{
    unsigned long length;
    char data [2048];
} c1;
```

- Erzeugen eines CLOB

```
c1.data = 'Hello';
c1.length = sizeof('Hello')-1;
```

kann durch Einsatz von Makros (z. B. `c1 = SQL_CLOB_INIT('Hello');`)
verborgen werden

- **Einfügen, Löschen und Ändern**

kann wie bei anderen Typen erfolgen, wenn genügend große AW-Puffer
vorhanden sind

- **Hole die Daten des Absolventen mit Lfdnr. 17 ins AWP**

```
...
SELECT Name, Photo, Lebenslauf
INTO :x, :y :yindik, :z :zindik
FROM Absolvent
WHERE Lfdnr = 17;
```

Große Objekte (5)

- Welche Operationen können auf LOBs angewendet werden?

- Vergleichsprädikate: =, <>, <, <=, >, >=, IN, BETWEEN

- LIKE-Prädikat

- Eindeutigkeit oder Reihenfolge bei LOB-Werten

- PRIMARY KEY, UNIQUE, FOREIGN KEY
- SELECT DISTINCT, . . ., COUNT (DISTINCT)
- GROUP BY, ORDER BY

- Einsatz von Aggregatfunktionen wie MIN, MAX

- Operationen

- UNION, INTERSECT, EXCEPT
- Joins von LOB-Attributen

- Indexstrukturen über LOB-Spalten

- Wie indexiert man LOBs?

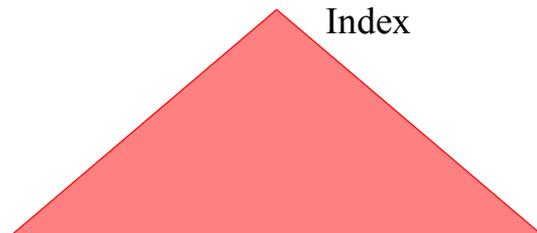
- Benutzerdefinierte Funktion ordnet LOBs Werte zu

- Funktionswert-Indexierung

$f(\text{blob1}) = x$

Blob-
Index

blob1

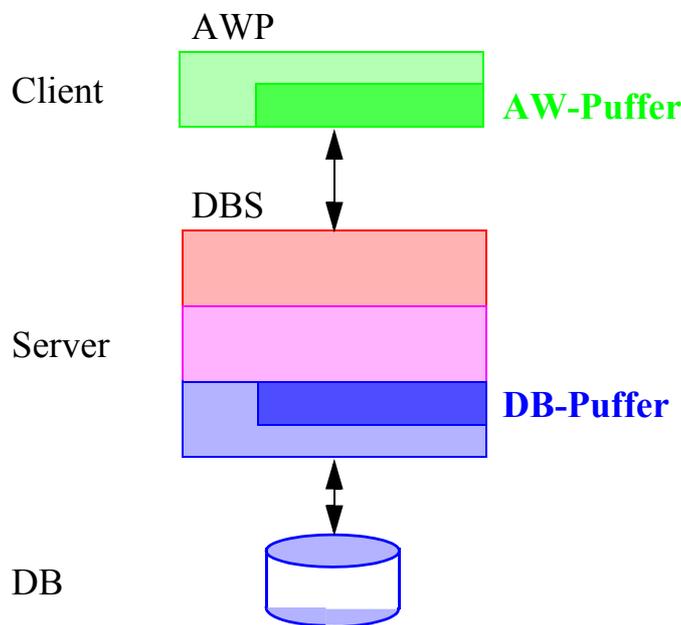


Große Objekte (6)

- Ist die direkte Verarbeitung von LOBs im AWP realistisch?

Bücher		EXEC SQL
(Titel	Varchar (200),	SELECT Kurzfassung, Buchtext, Video
BNR	ISBN,	INTO :kilopuffer, :megapuffer, :gigapuffer
Kurzfassung	CLOB (32 K),	
Buchtext	CLOB (20 M),	FROM Bücher
Video	BLOB (2 G))	WHERE Titel = 'American Beauty'

- Client/Server-Architektur



- Allokation von Puffern?
- Transfer eines ganzen LOB ins AWP?
- Soll Transfer über DBS-Puffer erfolgen?
- „Stückweise“ Verarbeitung von LOBs durch das AWP erforderlich!
- ➔ Lokator-Konzept für den Zugriff auf LOBs

Speicherung großer Objekte¹

- **Darstellung großer Speicherobjekte**

- besteht potentiell aus vielen Seiten oder Segmenten
- ist eine uninterpretierte Bytefolge
- Adresse (OID, *object identifier*) zeigt auf Objektkopf (*header*)
- OID ist Stellvertreter im Satz, zu dem das **lange Feld** gehört
- geforderte Verarbeitungsflexibilität bestimmt Zugriffs- und Speicherungsstruktur

- **Verarbeitungsprobleme**

- Ist Objektgröße vorab bekannt?
- Gibt es während der Lebenszeit des Objektes viele Änderungen?
- Ist schneller sequentieller Zugriff erforderlich?
- . . .

- **Abbildung auf Externspeicher**

- **seitenbasiert**
 - Einheit der Speicherzuordnung: eine Seite
 - „verstreute“ Sammlung von Seiten
- **segmentbasiert (mehrere Seiten)**
 - Segmente fester Größe (Exodus)
 - Segmente mit einem festen Wachstumsmuster (Starburst)
 - Segmente variabler Größe (EOS)
- **Zugriffsstruktur zum Objekt**
 - Kettung der Segmente/Seiten
 - Liste von Einträgen (Deskriptoren)
 - B*-Baum

1. Biliris, A.: *The Performance of Three Database Storage Structures for Managing Large Objects*, Proc. ACM SIGMOD'92 Conf., San Diego, Calif., 1992, pp. 276-285

Lange Felder in Exodus

- **Speicherung langer Felder**

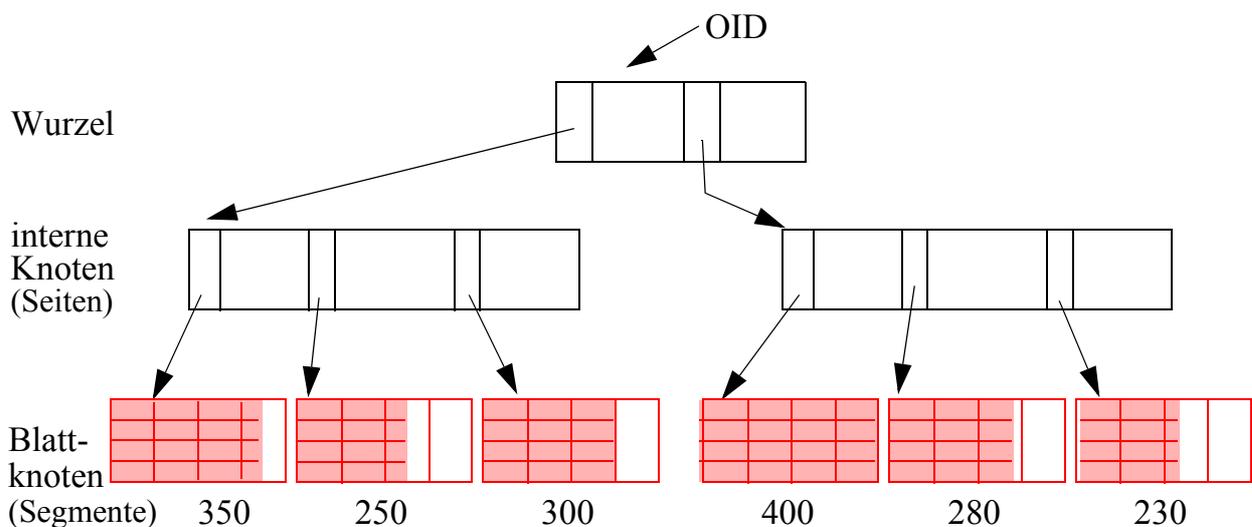
- Daten werden in (kleinen) Segmenten fester Größe abgelegt
- Wahl an Verarbeitungscharakteristika angepasster Segmentgrößen
- Einfügen von Bytefolgen einfach und überall möglich
- schlechteres Verhalten bei sequentiellm Zugriff

- **B*-Baum als Zugriffsstruktur**

- Blätter sind Segmente fester Größe (hier 4 Seiten zu 100 Bytes)
- interne Knoten und Wurzel sind Index für Bytepositionen
- interne Knoten und Wurzel speichern für jeden Kind-Knoten Einträge der Form (Seiten-#, Zähler)
 - Zähler enthält die maximale Bytenummer des jeweiligen Teilbaums (links stehende Seiteneinträge zählen zum Teilbaum).
 - Objektlänge: Zähler im weitesten rechts stehenden Eintrag der Wurzel

- **Repräsentation sehr langer dynamischer Objekte**

- bis zu 1GB mit drei Baumebenen (selbst bei kleinen Segmenten)
- Speicherplatznutzung typischerweise ~ 80%



- Byte 100 in der letzten Seite?

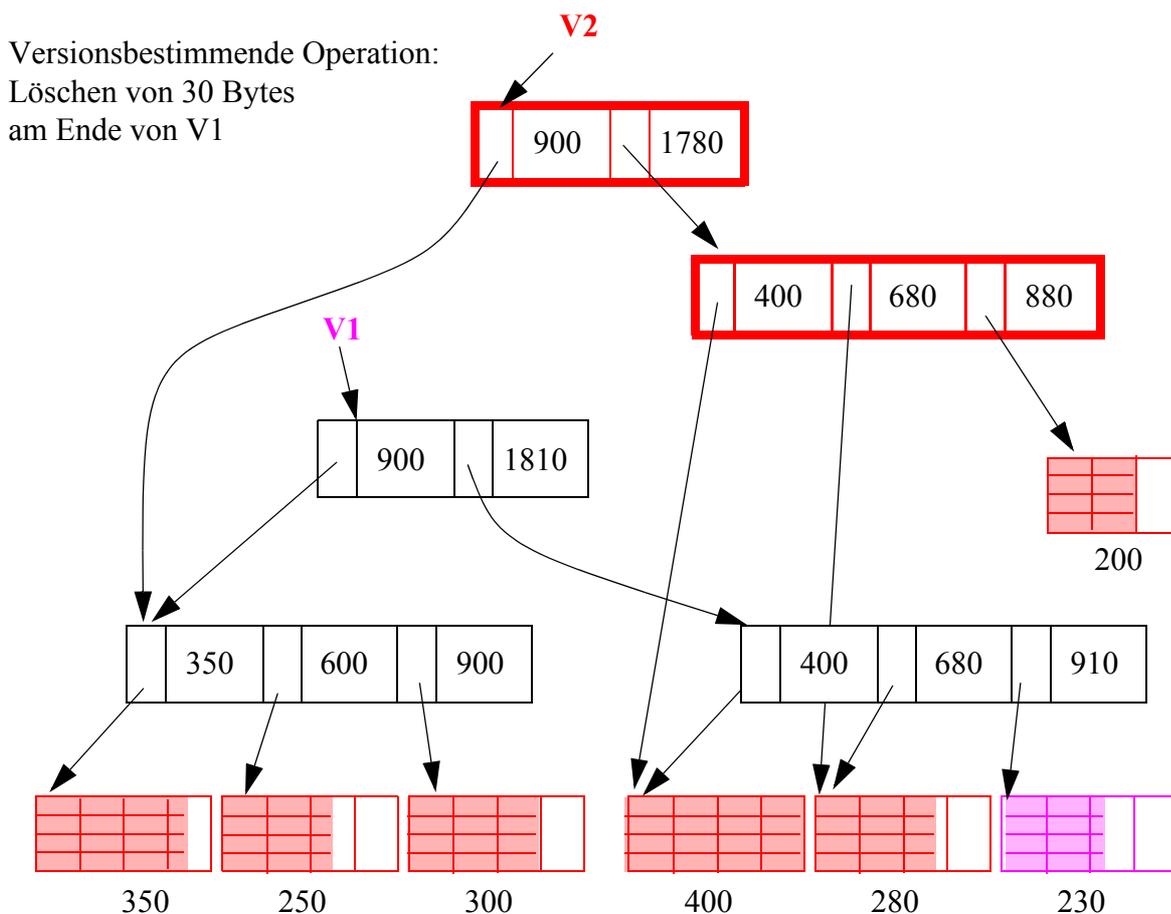
Exodus¹ (2)

- **Spezielle Operationen**

- Suche nach einem Byteintervall
- Einfügen/Löschen einer Bytefolge an/von einer vorgegebenen Position
- Anhängen einer Bytefolge ans Ende des langen Feldes

- **Unterstützung versionierter Speicherobjekte**

- Markierung der Objekt-Header mit Versionsnummer
- Kopieren und Ändern nur der Seiten, die sich in der neuen Version unterscheiden (in Änderungsoperationen, bei denen Versionierung eingeschaltet ist)



1. M.J. Carey, D.J. DeWitt, J.E. Richardson, E.J. Shekita: *Object and File Management in the EXODUS Extensible Database System*. Proc. 12th VLDB Conf., 1986, pp. 91-100

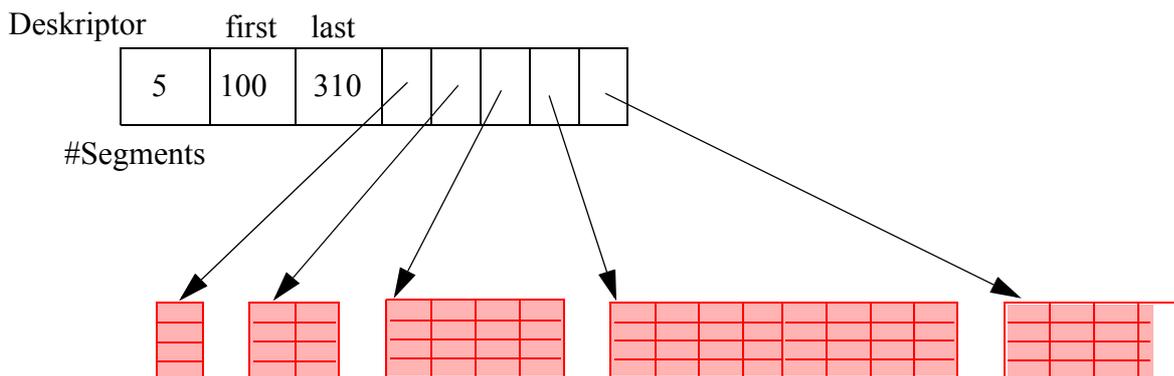
Lange Felder in Starburst

- **Erweiterte Anforderungen**

- Effiziente Speicherallokation und -freigabe für Felder von 100 MB bis 2 GB
- hohe E/A-Leistung:
Schreib- und Lese-Operationen sollen E/A-Raten nahe der Übertragungsgeschwindigkeit der Magnetplatte erreichen

- **Prinzipielle Repräsentation**

- Deskriptor mit Liste der Segmentbeschreibungen
- Langes Feld besteht aus einem oder mehreren Segmenten.
- Segmente, auch als **Buddy-Segmente** bezeichnet, werden nach dem Buddy-Verfahren in großen vordefinierten Bereichen fester Länge auf Externspeicher angelegt.



- **Segmentallokation bei vorab bekannter Objektgröße**

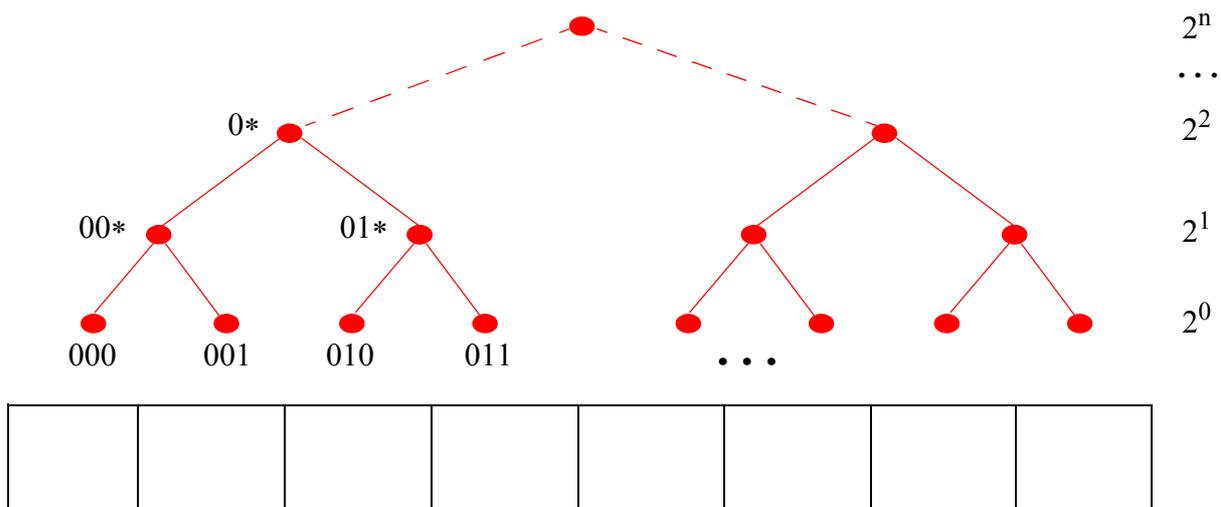
- Objektgröße G (in Seiten)
- $G \leq \text{MaxSeg}$: es wird ein Segment angelegt
- $G > \text{MaxSeg}$: es wird eine Folge maximaler Segmente angelegt
- letztes Segment wird auf verbleibende Objektgröße gekürzt

Lange Felder in Starburst¹(2)

- **Segmentallokation bei unbekannter Objektgröße**

- Wachstumsmuster der Segmentgrößen wie im Beispiel: 1, 2, 4, ..., 2^n Seiten werden jeweils zu einem Buddy-Segment zusammengefasst
- MaxSeg = 2048 für $n = 11$
- Falls MaxSeg erreicht wird, werden weitere Segmente der Größe MaxSeg angelegt
- Letztes Segment wird auf die verbleibende Objektgröße gekürzt

- **Allokation von Buddy-Segmenten** in sequentiellen Buddy-Bereich gemäß binärem Buddy-Verfahren



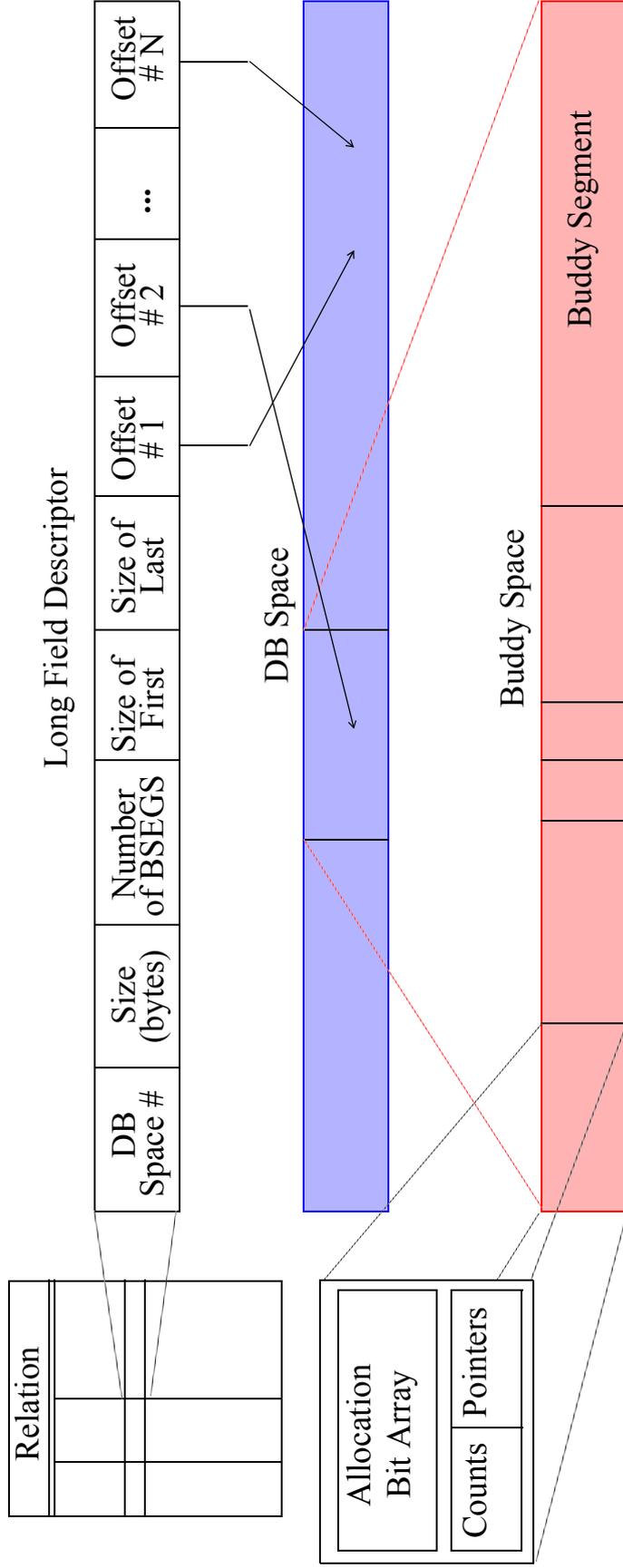
- Zusammenfassung zweier Buddies der Größe $2^n \Rightarrow 2^{n+1}$ ($n \geq 0$)

- **Verarbeitungseigenschaften**

- effiziente Unterstützung von sequentiellen und wahlfreiem Lesen
- einfaches Anhängen und Entfernen von Bytefolgen am Objektende
- schwieriges Einfügen und Löschen von Bytefolgen im Objektinneren

1. T. J. Lehman, B. G. Lindsay: *The Starburst Long Field Manager*. Proc. 15th VLDB Conf., 1989, pp. 375-383

Starburst: Speicherorganisation zur Realisierung Langer Felder



• Aufbau eines Langen Feldes

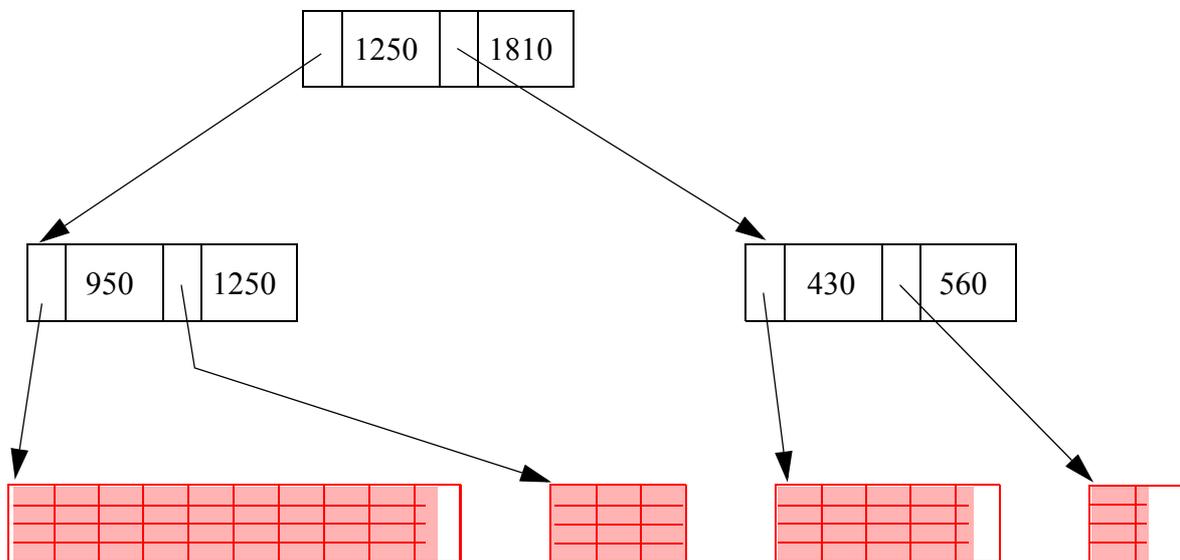
- Deskriptor des Langen Feldes (< 316 Bytes) ist in Relation gespeichert
- Long Field ist aufgebaut aus einem oder mehreren **Buddy-Segmenten**, die in großen vordefinierten **Buddy-Bereichen** fester Länge auf Platte angelegt werden
- Buddy-Segmente enthalten nur Daten und keine Kontrollinformation
- Segment besteht aus 1, 2, 4, 8, ... oder 2048 Seiten (→ max. Segmentgröße 2 MB bei 1 KB-Seiten)
- Buddy-Bereiche sind allokiert in (noch größeren) DB-Dateien (DB Spaces). Sie setzen sich zusammen aus Kontrollseite (Allocation Page) und Datenbereich

Speicherallokation mit variablen Segmenten

- **Verallgemeinerung des Exodus- und Starburst-Ansatzes in Eos**

- Objekt ist gespeichert in einer Folge von Segmenten variabler Größe
- Segment besteht aus Seiten, die physisch zusammenhängend auf Externspeichern angeordnet sind
- nur die letzte Seite eines Segmentes kann freien Platz aufweisen

- **Prinzipielle Repräsentation**



↳ Die Größen der einzelnen Segmente können sehr stark variieren

- **Verarbeitungseigenschaften**

- die guten operationalen Eigenschaften der beiden zugrundeliegenden Ansätze können erzielt werden
- Reorganisation möglich, falls benachbarte Segmente sehr klein (Seite) werden

DB-Anbindung externer Daten

- **Motivation**

- Die meisten Daten in einem Unternehmen sind in Dateien gespeichert.
- Sie werden es auf lange Zeit bleiben und im Umfang zunehmen.
- Da viele Anwendungen mit Dateien arbeiten, ist es erforderlich, auch diese Datenzugriffe zu unterstützen.
(gleichförmiger Zugriff zu DBs und anderen Datenquellen, siehe OLE DB)

- **Eigenschaften**

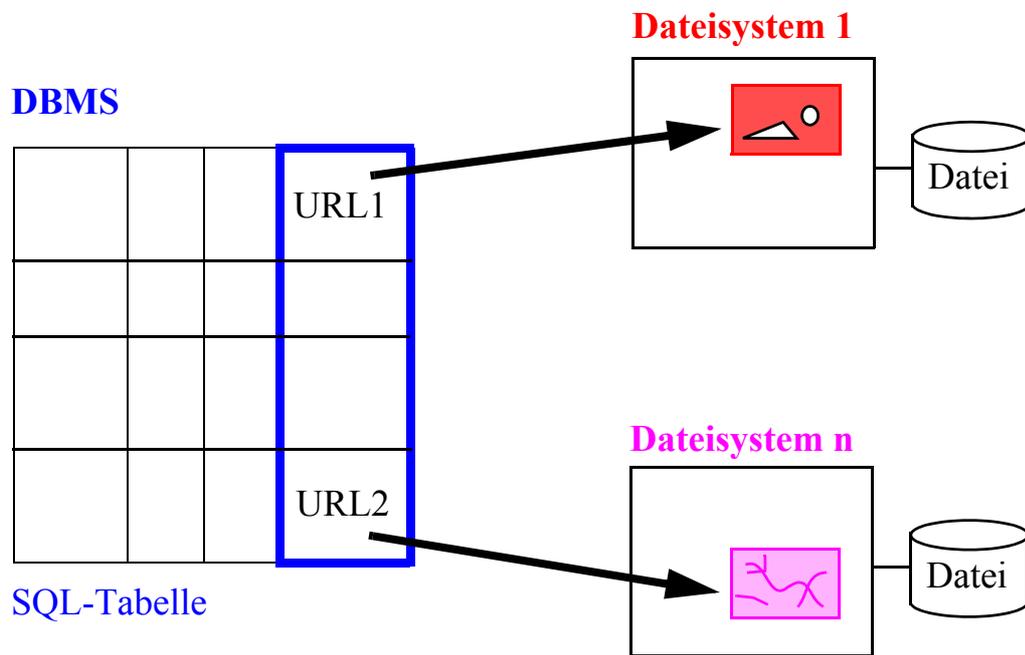
- Dateisysteme bieten **nicht** genügend Metadaten für **Suchfunktionen und Integritätserhaltung**.
 - DBMS unterstützen u. a. ein großes Spektrum an Funktionen, sind momentan aber **nicht** für die Speicherung einer **großen Anzahl von BLOBs** (Multimedia-Typen) optimiert.
 - BLOBs benötigen **hierarchische Speicherverwaltung** von leistungsfähigen Dateisystemen (z. B. Tertiärspeicher), die eine kosteneffektive Verwaltung der Daten für variierende Zugriffsmuster (häufig oder selten) gewährleisten.
- ➔ **Verknüpfung von Dateisystemen und DBMSs soll Vorteile beider Ansätze nutzen!**

- **Anwendungsbeispiele**

- **CAD-Systeme:** Synchronisation von Millionen von Bauteilen (Zeichnungen und Baupläne in einem proprietären Format)
- **Multimedia-Objekte:** Verwaltung von Bibliotheken für Bilder, Programme, Dokumente oder Videos
- **HTML- und XML- Dateien:** DB-Unterstützung für die Funktionalität von Web-Server

DB-Anbindung externer Daten¹ (2)

- Speichermodell für die DB-Anbindung



- Welche Probleme sind zu lösen?

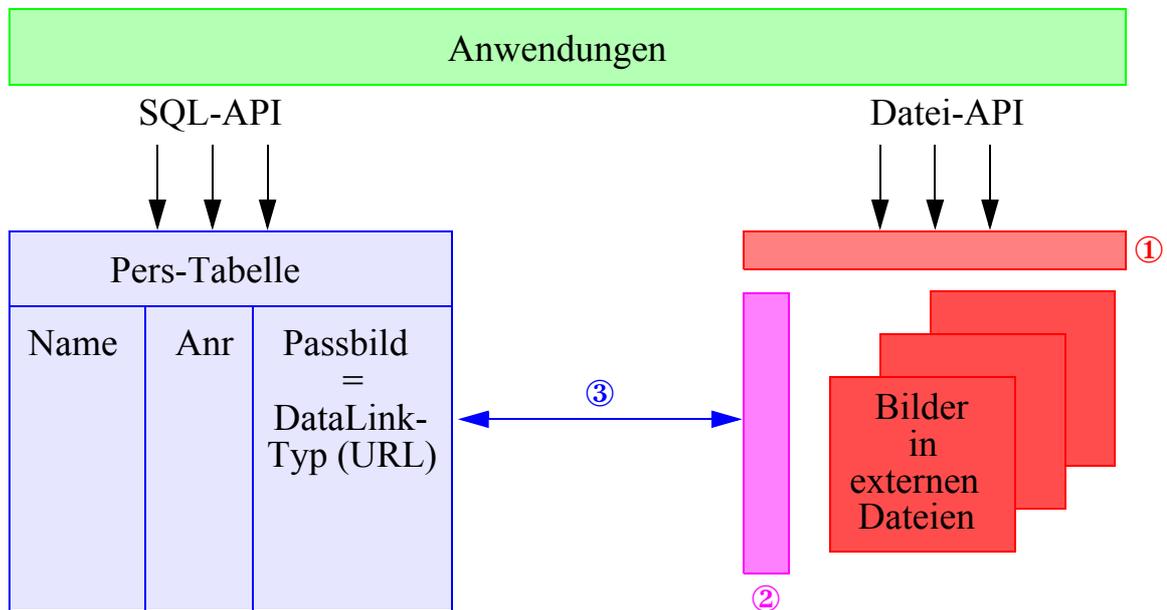
- **Referentielle Integrität**
- **Zugriffskontrolle**
- **Koordiniertes Backup und Recovery**
- **Transaktionskonsistenz**
- zusätzlich: Suche über
 - herkömmliche Datentypen
 - Inhalte externer Daten
- Leistungsaspekte bei DB- und Datei-Anwendungen

➔ **Beteiligte Dateisysteme benötigen zusätzliche Kontrollkomponente, die mit dem DBMS über spezielle Protokolle kooperiert**

1. Information Technology – Database Language SQL - Part 9: Management of External Data, International Standard, May 2001 (www.jtc1sc32.org)

DB-Anbindung externer Daten (3)

- **DataLinks-Konzept zur Verwaltung externer Daten**



① **DataLinks Filesystem Filter (DLFF)**

- erzwingt referentielle Integrität beim Umbenennen und Löschen von Dateien
- erzwingt DB-zentrierte Zugriffskontrolle beim Öffnen einer Datei
- Datei-API bleibt unverändert – keine Änderungen in den Anwendungen
- DLFF liegt nicht im Lese-/Schreib-Pfad für externe Dateien (Performance!)

② **DataLinks File Manager (DLFM)**

- führt Link-/UnLink-Operationen transaktionsgeschützt durch
- gewährleistet referenzielle Integrität
- unterstützt koordiniertes Backup/Recovery

③ **DBMS verwaltet/koordiniert Operationen auf externen Dateien**

- über URL's referenziert
- durch DLFM-API (DataLinks File Manager)

DB-Anbindung externer Daten (4)

- **Verarbeitungsmodell aus der Sicht der Anwendung**

- SQL-Zugriff auf Metadaten-Repository für externe Daten
- Suche ist auch über den Inhalt externer Daten möglich
 - ➔ Funktionswertindexierung
- Liste von Referenzen der gesuchten Objekte
- Anwendung referenziert externe Daten direkt über Datei-API.

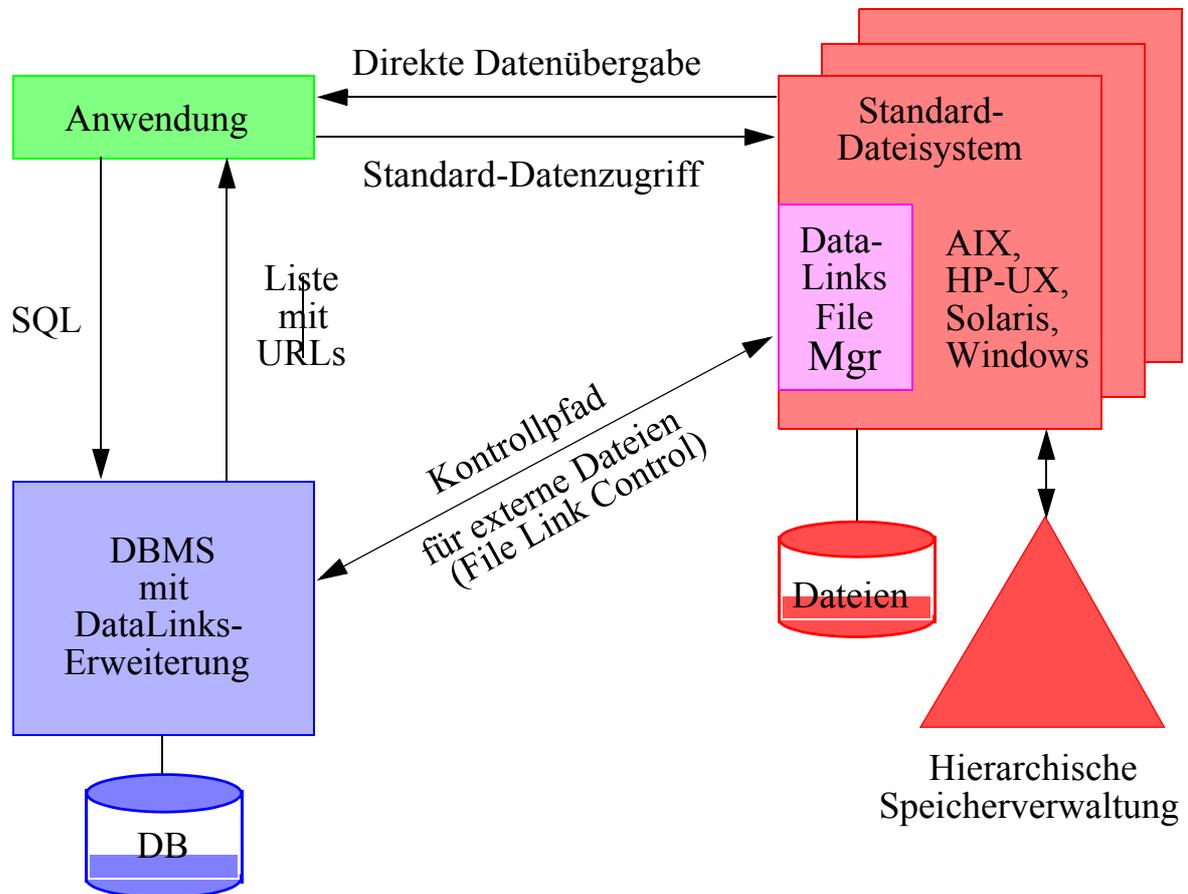
- **DataLinks-Datentyp nach SQL:99 – Beispiel**

```
CREATE TABLE Pers (  
  Name VARCHAR (30);  
  Anr INTEGER,  
  Passbild DATALINK (200)  
    LINKTYPE URL  
    FILE LINK CONTROL  
    INTEGRITY all  
    READ PERMISSION DB  
    WRITE PERMISSION blocked  
    RECOVERY yes  
    ON UNLINK restore  
);
```

- DBMS-Kontrolle lässt sich abgestuft aktivieren.
- URL: http://servername/pathname/filename/
- **Integrity:** URLs als Referenzen werden konsistent gehalten.
- **Read Permission:** bleibt entweder beim Dateisystem oder wird ans DBMS delegiert. Autorisierung wird als Token in die URL eingebettet.
- **Write Permission:** bleibt beim Dateisystem oder wird blockiert
- **Recovery:** Nur bei Option WRITE PERMISSION blocked ist koordiniertes Backup und Recovery möglich.
- **On Unlink:** Datei kann gelöscht oder zur Verwaltung ans Dateisystem zurückgegeben werden.

DB-Anbindung externer Daten (5)

• DataLinks-Architektur



• Typische Anwendung

- Integration von unstrukturierten und semi-strukturierten Daten mit Anwendungen auf DB-Basis
- Reichweite: große Anzahl von Dateien in Rechnernetzwerken
- Bei Funktionswertindexierung: Über URLs referenzierte Dateien bleiben weitgehend unverändert.
- Benutzer extrahiert Features von Bildern oder Videos, speichert sie in der DB zwischen, um Auswertungen zusammen mit Prädikaten auf anderen DB-Daten zu machen.
- *Query By Image Content (QBIC)* unterstützt Extraktion und Suche auf solchen Features.

Zusammenfassung

- **Freispeicherinformation** auf verschiedenen Ebenen erforderlich:
Gerät, Segment (Datei), Seite
- **Ziele bei der externspeicherbasierten Adressierung**
 - Kombination der Geschwindigkeit des direkten Zugriffs mit der Flexibilität einer Indirektion
 - Satzverschiebungen in einer Seite ohne Auswirkungen

↳ **TID, DBK (Zuordnungstabelle) oder Primärschlüssel**
- **Indexierung von Tabellen**
 - physische oder hybride Verfahren bei ungeordneten Tabellen
 - hybride Verfahren kombiniert mit Primärschlüssel bei geordneten Tabellen (Index-organisierte Tabellen)
- **Hauptspeicherbasierte Adressierung (*Pointer Swizzling*)**
 - transparenter Programmzugriff auf persistente und transiente Objekte
 - Abbildung von langen ES-Adressen auf Virtuelle Adressen
 - orthogonale Klassifikationskriterien: **Ort, Zeitpunkt, Art**
- **Abbildung von Sätzen**
 - Speicherung variabel langer Felder
 - dynamische Erweiterungsmöglichkeiten
 - Berechnung von Feldadressen
- **Speicherung komplexer Objekte**
 - Listen-, Mengen- und Tupelkonstruktoren
 - Konstruktor-Anwendung ist orthogonal und rekursiv

Zusammenfassung (2)

- **Spezifikation großer Objekte hat großen Einfluss auf die DB-Verarbeitung**
 - Speicherungsoptionen, Logging
 - Einsatz benutzerdefinierter und systemspezifischer Funktionen
 - Deklarative SQL-Operationen, aber Cursor-basierte und „navigierende“ Verarbeitung von LOB-Werten
- **Spezielle Verarbeitungstechniken und gute Leistungseigenschaften erforderlich**
 - Transport zur Anwendung (Minimierung von Kopiervorgängen)
 - Anfrageoptimierung, Auswertung von LOB-Funktionen
 - Synchronisation, Logging und Recovery
- **Speicherung großer Objekte wird zunehmend wichtiger**
 - **B*-Baum-Technik:** flexible Darstellung, moderate Zugriffsgeschwindigkeit
 - **große variabel lange Segmente (Listen):** hohe E/A-Leistung
 - Auswahl verschiedener, auf Verarbeitungscharakteristika zugeschnittener Techniken
- **DB-Anbindung für externe Dateien**
 - DB-Unterstützung bei der Verwaltung, der Konsistenzerhaltung, und der inhaltsbasierten Suche wünschenswert.
 - **DataLinks-Konzept** bietet Referentielle Integrität, Zugriffskontrolle, Koordiniertes Backup und Recovery sowie Transaktionskonsistenz