4. Speicherungsstrukturen

- **Freispeicherverwaltung**
  - im Segment
  - in der Seite
- **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

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

FIX P_i, FIX P_j, UNFIX P_j,

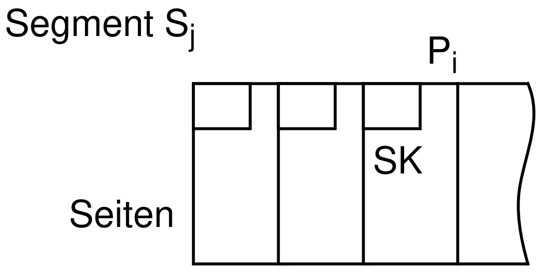
FIX P_k, UNFIX P_i, ...

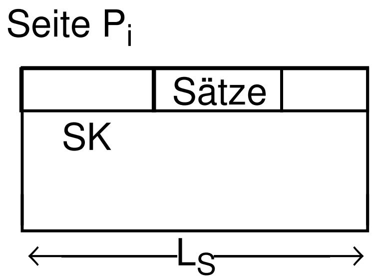
- **Eigenschaften der oberen Schnittstelle**
  - Nicht-flüchtiger Speicher mit Adressierungshilfen
  - Freispeicher-Verwaltung
  - Adressierungsverfahren für physische Sätze
  - Adressierungsverfahren zwischen verschiedenen Sätzen
  - Zugriffspfade zur Realisierung von Inhaltsadressierbarkeit

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



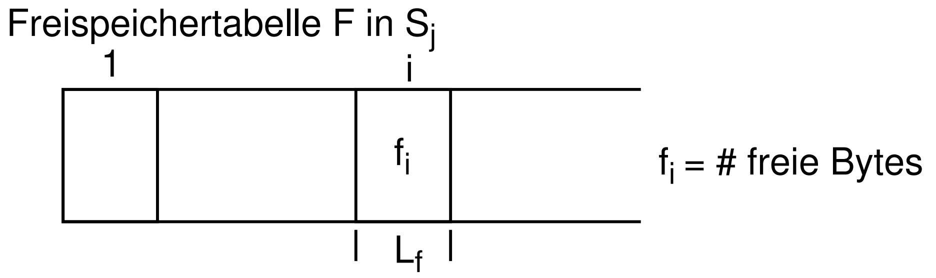


➥ in SK (Seitenkopf):

ID von P_i,

Freiplatz-Info,

Typ, Org.-Daten



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,...)
  - Segmentende

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

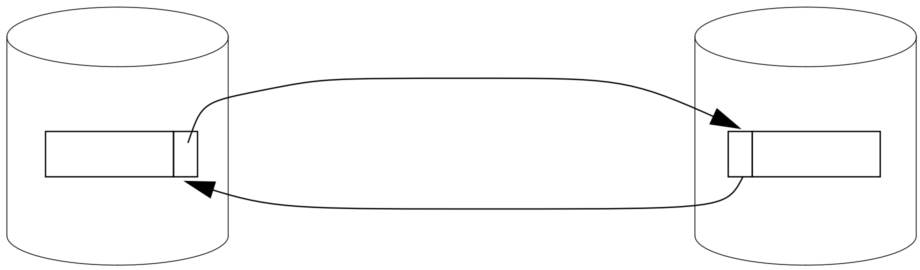
  Einheiten von $f_i \rightarrow \lceil L_S / 256 \rceil$ - Vielfache

  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

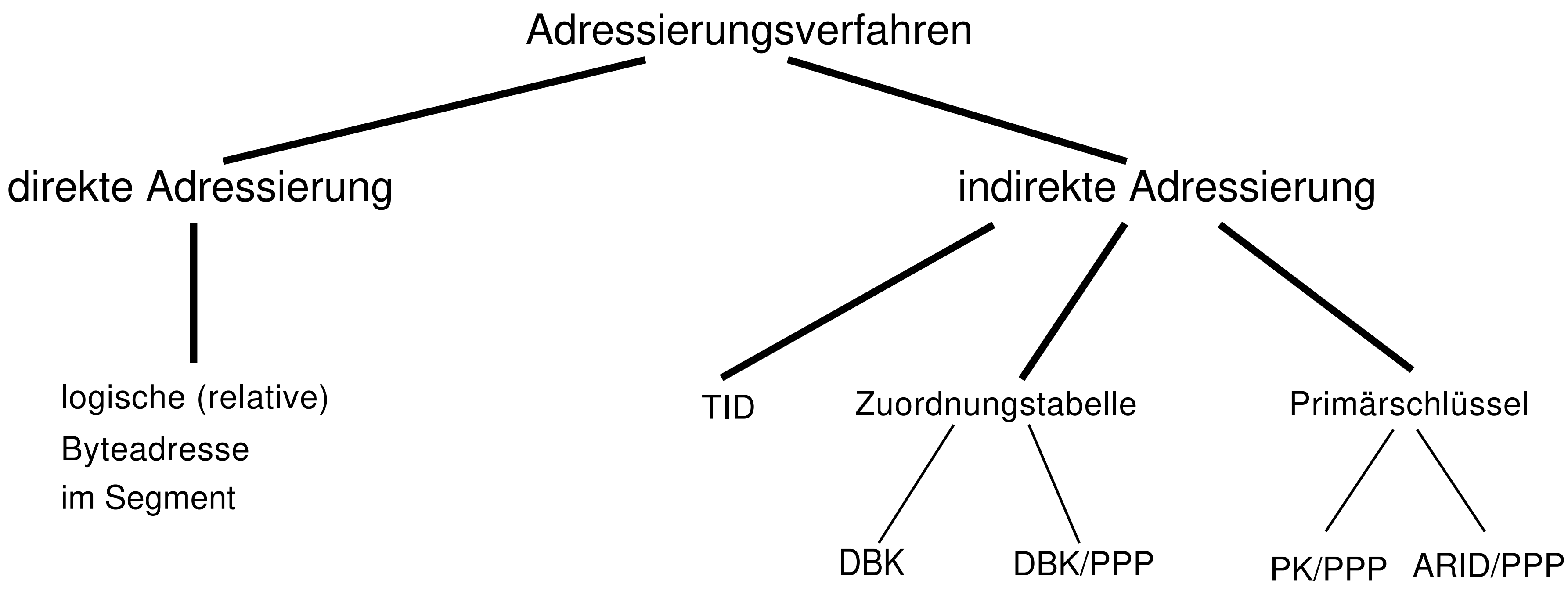
- **Ziele der Adressierungstechnik:**
  - schneller, möglichst direkter Satzzugriff
  - hinreichend stabil gegen geringfügige Verschiebungen
    (Verschiebungen innerhalb einer Seite ohne Auswirkungen)
  - seltene oder keine Reorganisationen

- **Adressierung in Segmenten**
  - logisch zusammenhängender Adreßraum
  - direkte Adressierung (logische Byte-Adresse, RBA)

    ➥ instabil bei Verschiebungen

➥ deshalb indirekte Adressierung

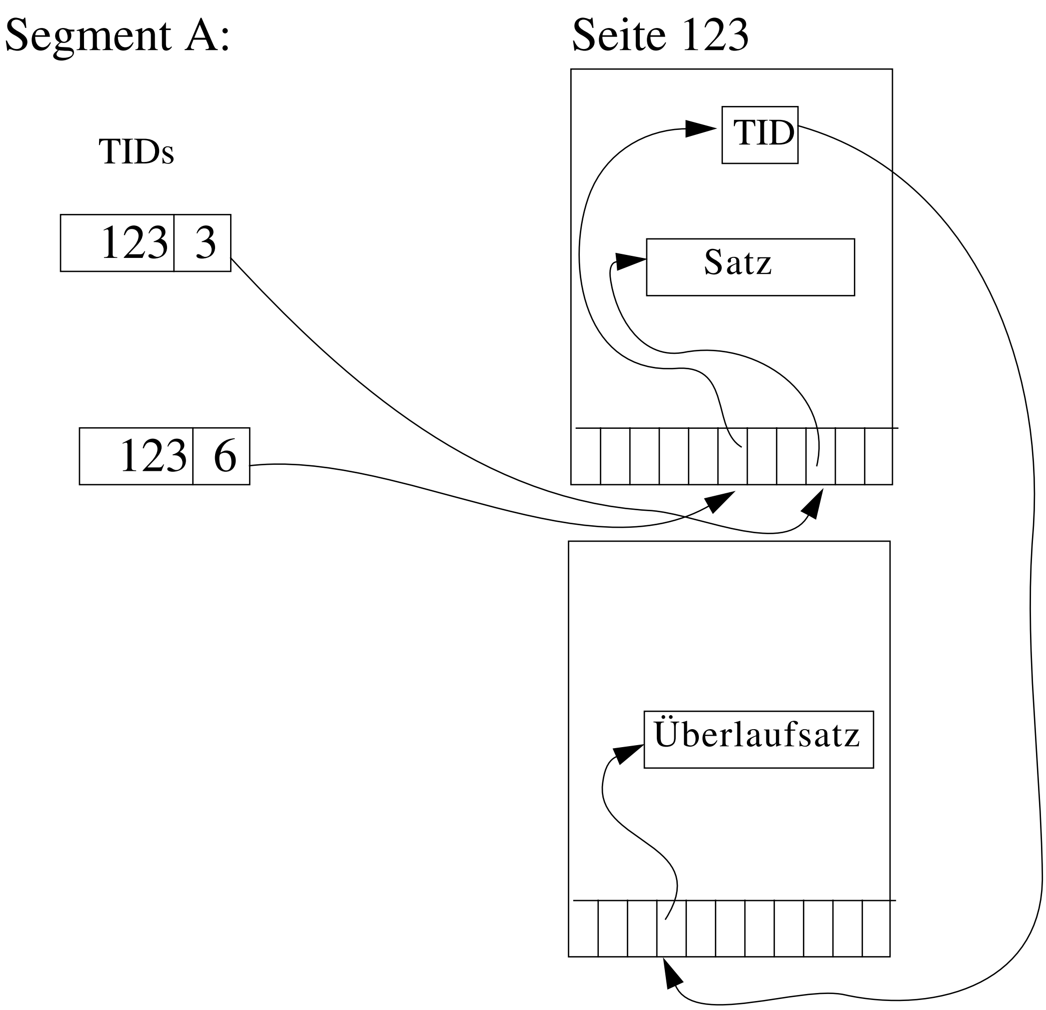
Techniken zur externspeicherbasierten Satzadressierung



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**

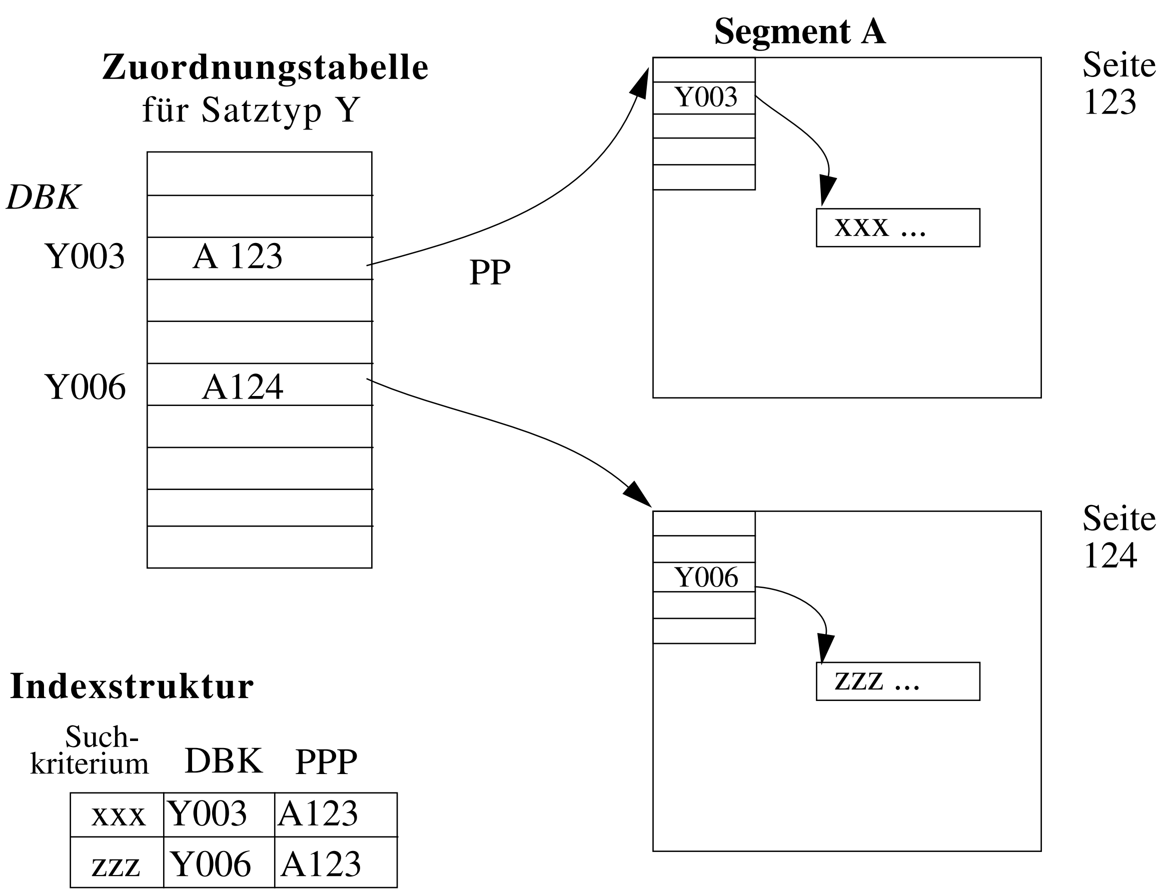


Satzadressierung über Zuordnungstabelle

- **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

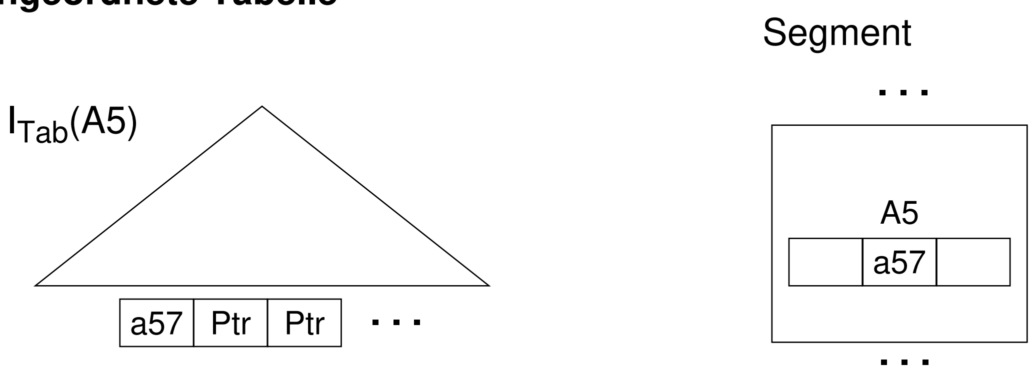
- **Speicherung von Tabellen**
  - **ungeordnete Tabelle**:
    Sätze (Zeilen) sind im Segment verstreut (Heap)
  - **geordnete Tabelle**:
    Sätze sind in B*-Baum eingebettet (key-sequenced table);
    es wird dadurch eine Clusterbildung erzielt

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

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

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

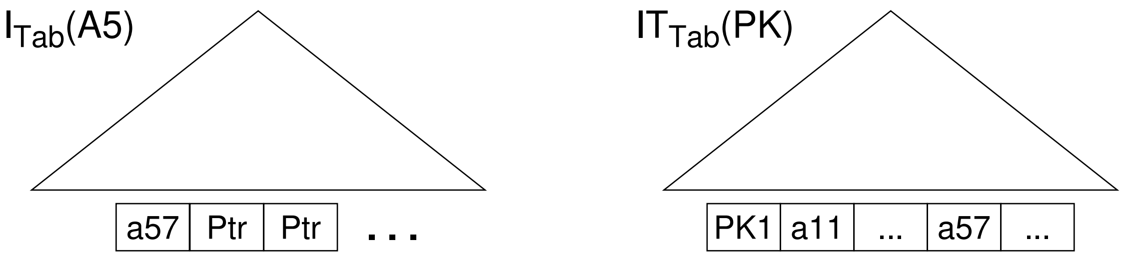
- **Ungeordnete Tabelle**



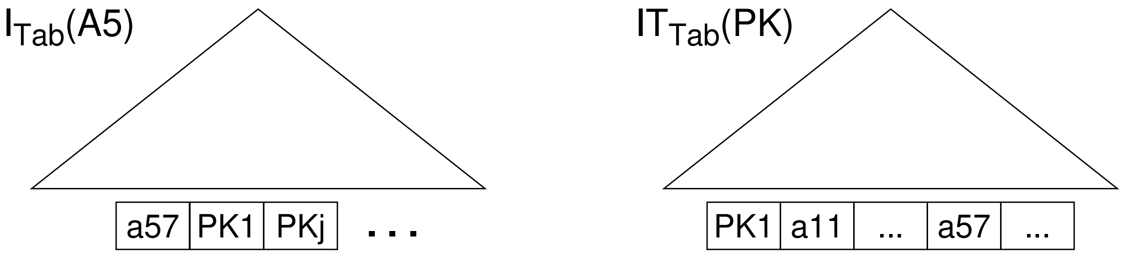
  - Sätze verschieben sich bei Aktualisierung nicht (kaum)
  - Adressierungsverfahren (Ptr):
    Es kommen TID, DBK und DBK/PPP in Frage
  - Indexunterstützung für ungeordnete Tabellen in DB2, Sybase, MS SQL-Server, Oracle, ...

Indexierung von Tabellen (2)

- **Index-organisierte Tabelle**



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

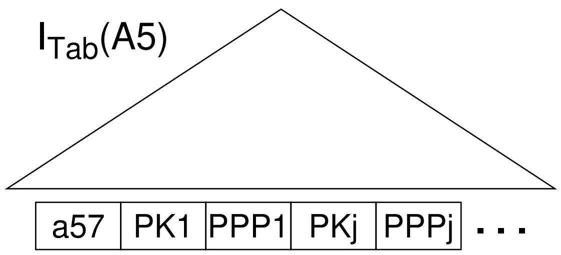


  - keine Wartung der I_{Tab}(Ai) bei Split/Verschiebungen in IT_{Tab} nötig
  - **aber:** höhere Zugriffskosten bei Index-Scan usw.

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

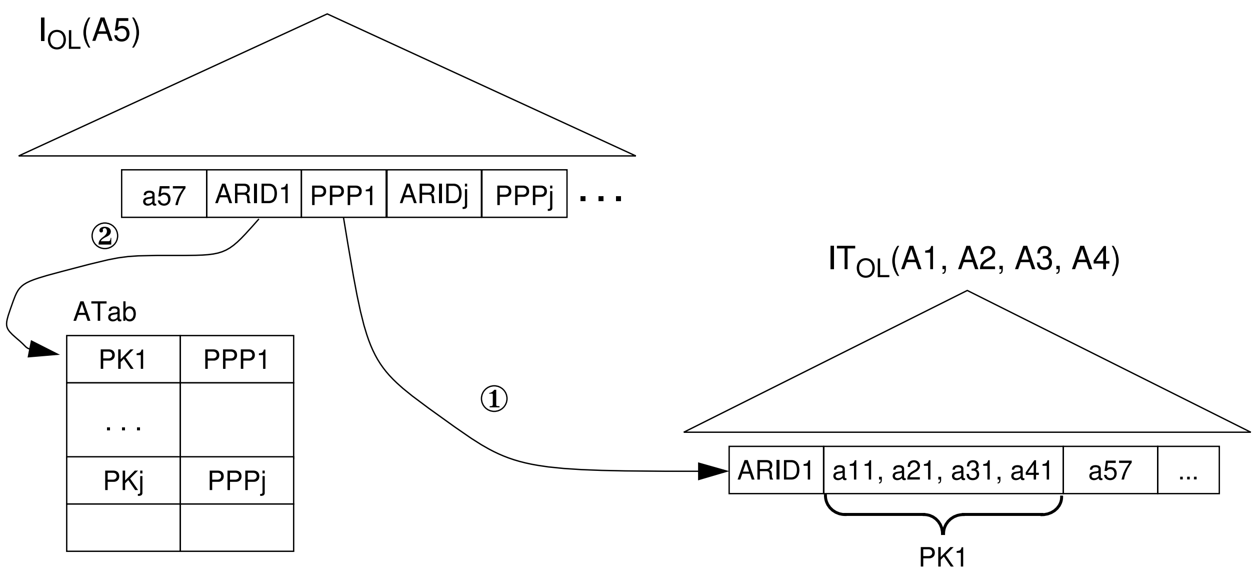
Attributwert	PK	PPP
Indexschlüssel	HRID = (Hybrid Row Identifier)	

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 (<u>ol-o-id, ol-w-id, ol-d-id, ol-number</u>, ol-i-id, ...)
  - Vereinfachte Schreibweise:
    OL (<u>A1, A2, A3, A4</u>, A5, ...)
  - Vermeidung der PK-Speicherung im Index
    - 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}(Ai) benutzen ATab
- Von dort kann über PPP oder über PK auf IT_{OL} zugegriffen werden
- Zugriffsschema entspricht der Oracle-Lösung

Hauptspeicherbasierte Adressierung

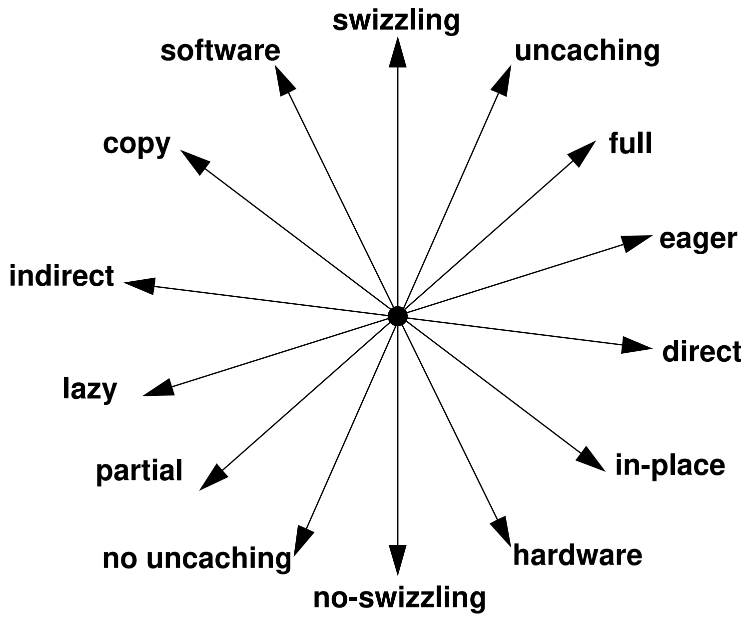
- **Aufgabe**:
  Programme sollen im HSP transparent transiente und persistente Datenobjekte verarbeiten können.
  - Ausschließliche Nutzung von direkten Adressen im HSP (Virtuelle Adressierung), d. h., Zugriff auf persistente Objekte ist im HSP genauso effizient wie auf transiente Objekte.
  - Keine 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-Swizzling**[1]



- **Klassifikation von Swizzling-Verfahren**
  - wichtigste Kriterien: Ort, Zeitpunkt und Art (orthogonal)

- Ort:

- *In-Place Swizzling:* Beibehaltung der Objektformate und der Seiten-strukturen
- *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:

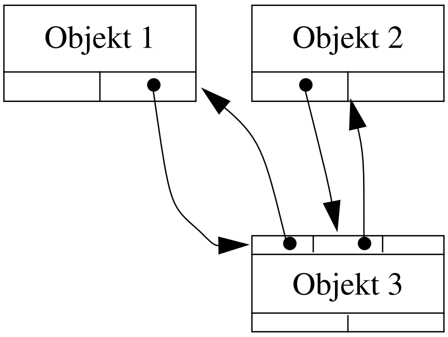
- *Direct Swizzling:* Nutzung der Virtuellen Adresse des Objektes, dadurch kann die Erstetzung von Objekten während der Verarbeitung sehr schwierig oder gar unmöglich werden
- *Indirect Swizzling:* Nutzung der Virtuellen Adresse von Objekt-Deskriptoren

---

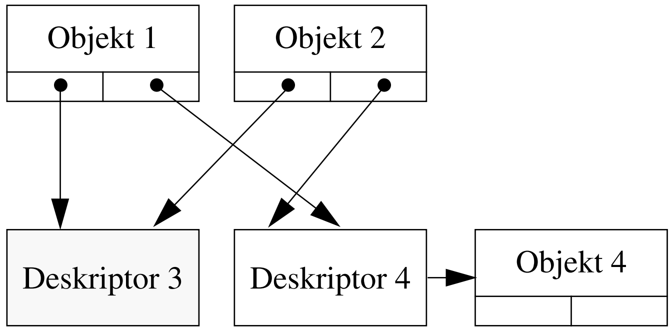
1. White, S.J., DeWitt, D.J.: Quickstore: A High Performance Mapped Object Store, in: The VLDB Journal 4:4, Oct. 1995, pp. 629-674.

Pointer-Swizzling (2)

- **Direktes und indirektes Swizzling – Prinzip**

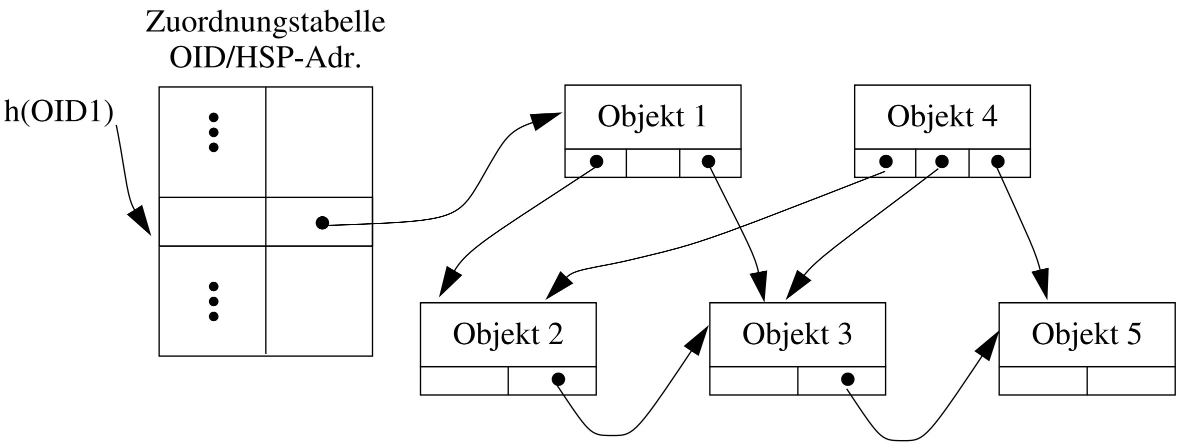


a) Symmetrische Referenzen

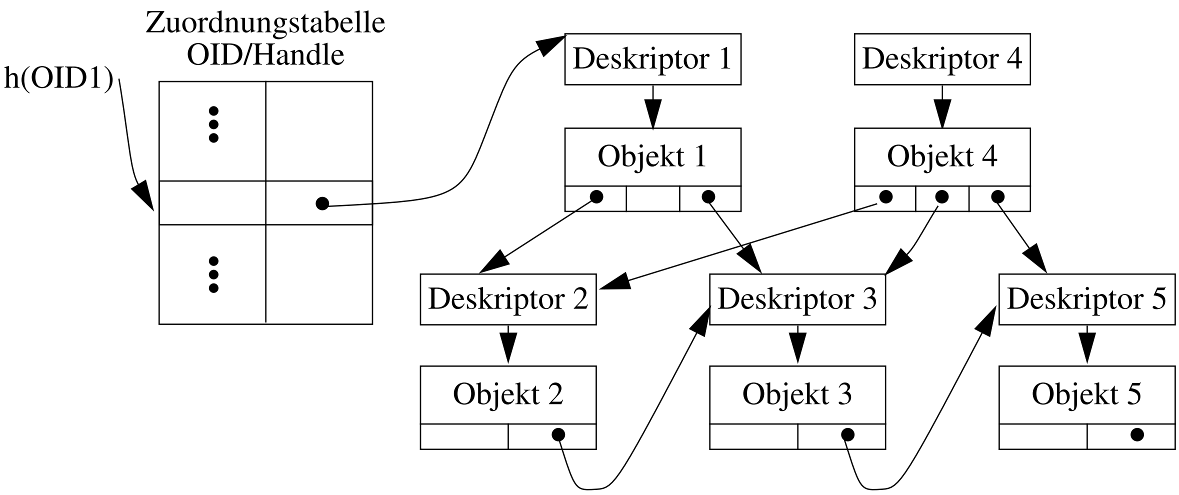


b) Referenzierung von Deskriptoren

- **Direkte und indirekte Variante beim Copy-Swizzling**

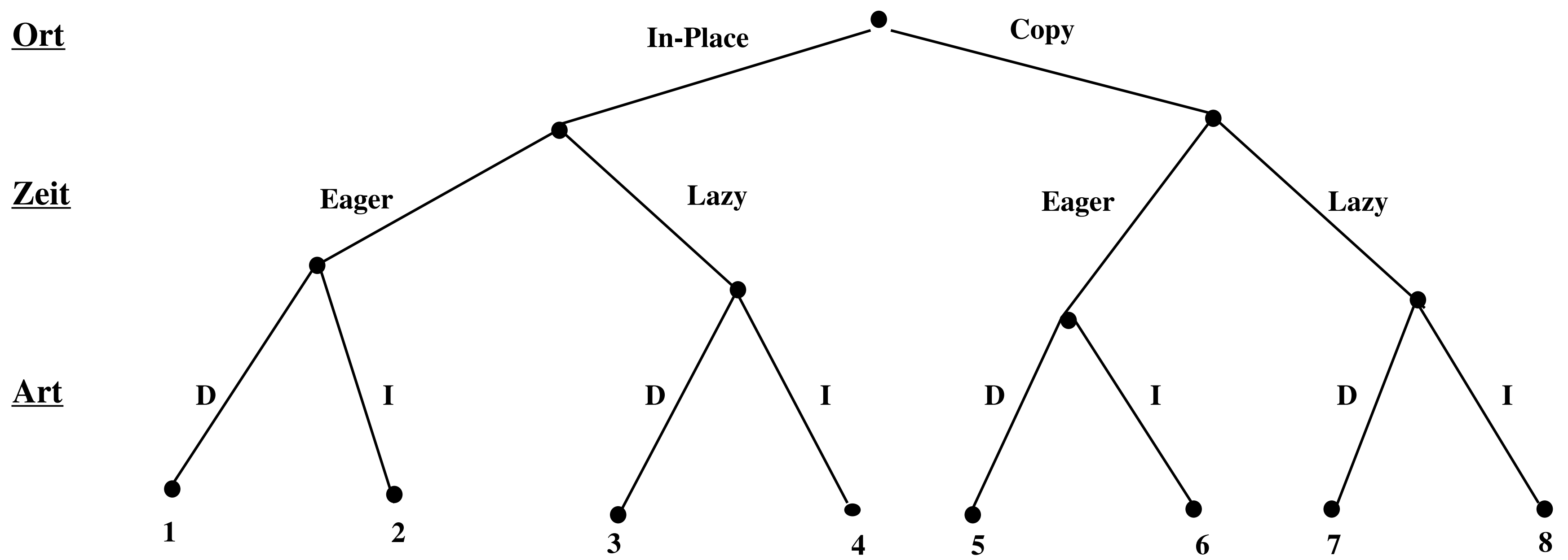


a) Direktes Swizzling in einem Objektpuffer



b) Indirektes Swizzling in einem Objektpuffer

Pointer-Swizzling (3)



Bermerkungen:

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

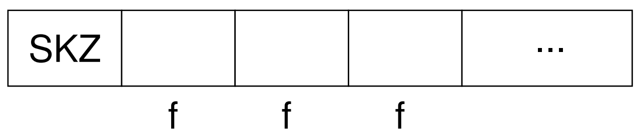
  - Satz- und Zugriffspfadbeschreibung im Katalog
  - besondere Methoden der Speicherung
    - Blank-/Nullunterdrückung
    - Zeichenverdichtung
    - kryptographische Verschlüsselung
    - Symbol für undefinierte Werte
  - Tabellenersetzung für Werte: KL = Kaiserslautern

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

Speicherungsstrukturen für Sätze

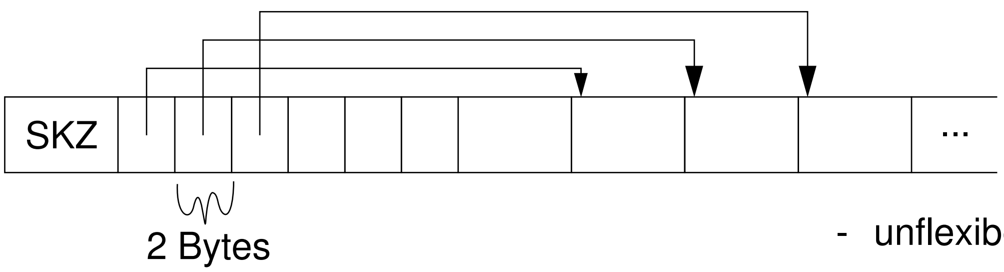
- **Konkatenation von Feldern fester Länge**

  z. B. TID



  - speicheraufwendig
  - unflexibel

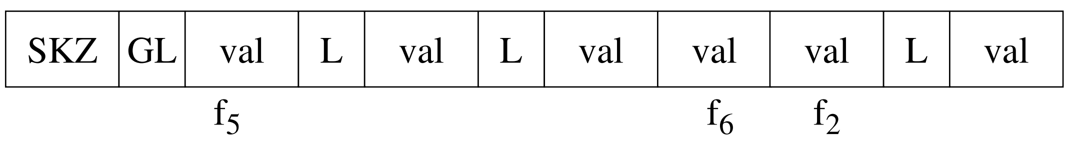
- **Zeiger im Vorspann**



  - unflexibel

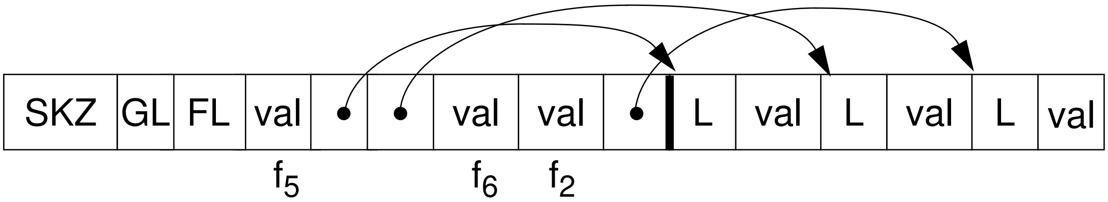
- **Eingebettete Längenfelder**

  Katalog: f_5 | v | v | f_6 | f_2 | v |



  - dynamische Erweiterung möglich

- **Optimierung: eingebettete Längenfelder mit Zeigern**



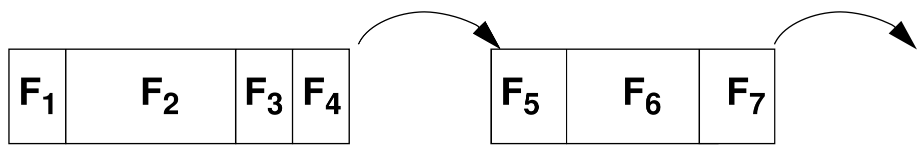
  - Adresse des n-ten Attributs kann berechnet werden
  - Erweiterung für EXPAND ?

Speicherungsstrukturen für Sätze (2)

- **Problem: dynamisches Wachstum/variable Länge**
  - Ausdehnung und Schrumpfung in einer Seite
  - Überlaufschemata
  - Garbage Collection
  - ➥ Die eingeführten 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 Objekte[1]

- **Komplexe Objekte werden gebildet aus**
  - atomaren Werten und darauf
  - rekursiv angewandten Mengen-, Listen- und Tupelkonstruktoren

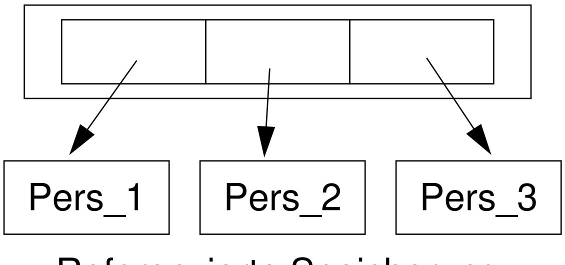
- **Einfaches Beispiel**

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

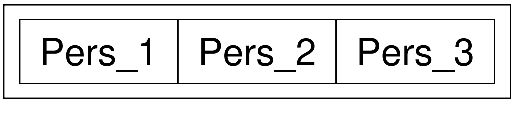
[. . .] kennzeichnet Stelle für Speicherungsstrukturbeschreibung

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



Referenzierte Speicherung



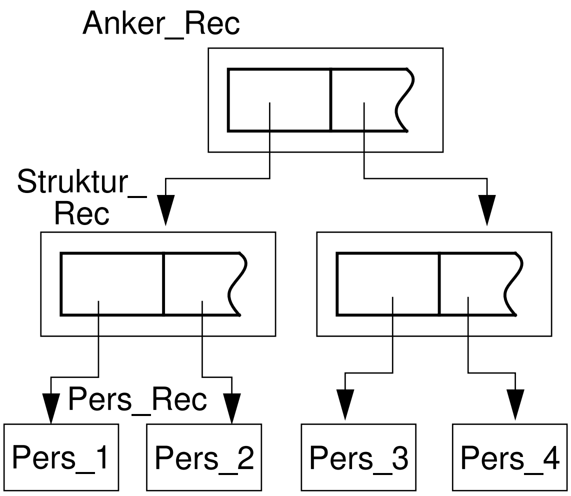
Materialisierte Speicherung

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

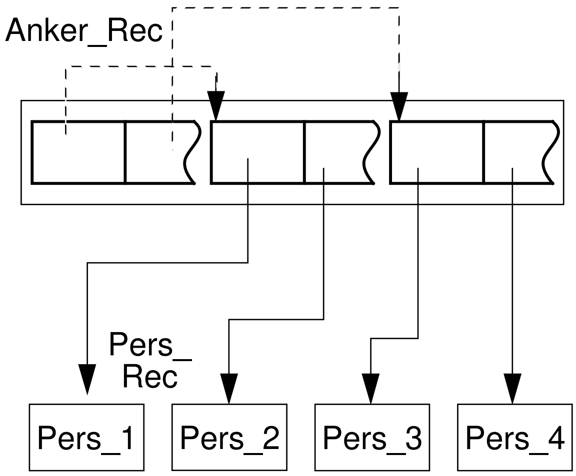
Speicherungsstrukturen für komplexe Objekte (2)

- **Zweimalige Anwendung des Mengenkonstruktors**
  - { {Pers_1 , Pers_2} , {Pers_3 , Pers_4} }
  - Vorgabe variabel langer Arrays als Konstruktordatenstrukturen

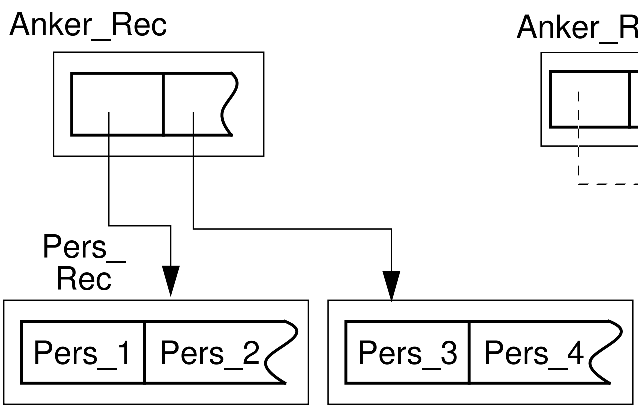
- **Vier Implementierungen**



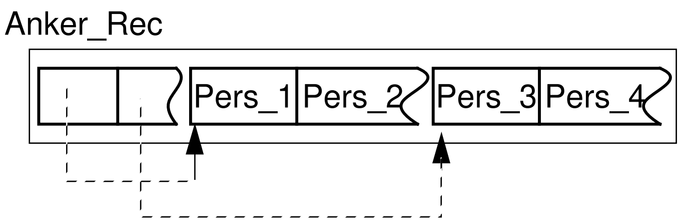
1. Elemente äußere Menge : referenziert
   Elemente innere Menge : referenziert



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 für Tupelkonstruktoren

- **Prinzipiell existieren die gleichen Freiheitsgrade**
  - Wahl einer Konstruktordatenstruktur: Zuordnung des Tupels zu einem Record oder zu mehreren Records
  - materialisierte oder referenzierte Speicherung der Attributwerte

- **Neuer Parameter: „location“**

  attribute_description = attribute_name [location = location_type,
  element_placement = placement_type]

  ⇒ Für jedes Attribut kann „location“ und „element_placement“ separat spezifiziert werden

- **„location“ erlaubt die Optimierung des Satzzugriffs nach den Zugriffshäufigkeiten der einzelnen Attribute**

  location_type = primary | secondary (record_type_name)

  Die Konstruktordatenstruktur eines Tupels kann auf mehrere Records aufgeteilt werden. Mit „primary“ wird das zugehörige Attribut im Primärblock angelegt.

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**

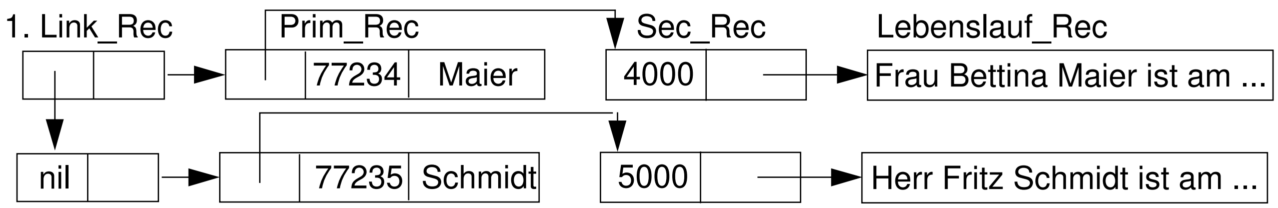
1. referenzierte Prim_Rec

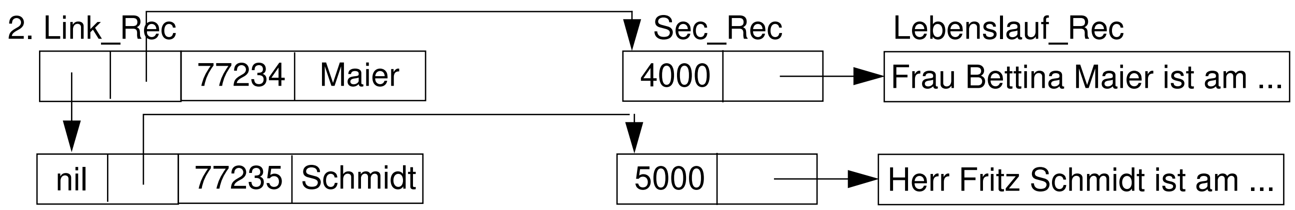
```
complex_object Mitarbeiter [anchor_record_type=Link_Rec]
   set [implementation=linked_list, element_placement=referenced (Prim_Rec)]
     of tuple
     (Pers_Nr      [location=primary, element_placement=inplace] : integer,
     Name          [location=primary, element_placement=inplace] : string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

2. materialisierte Prim_Rec

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

- **Dazugehörige Speicherungsstrukturen für die Mitarbeiterrelation**





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