8. Sekundäre und hierarchische Zugriffspfade

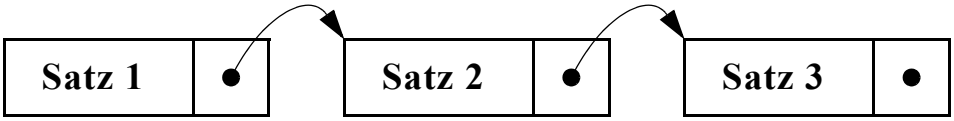
- **Ziele**
  - Entwurfsprinzipien für Zugriffspfade auf alle qualifizierten Sätze einer Tabelle
  - Auswertung von Suchprädikaten durch mengentheoretische Operationen
  - Abbildungsmöglichkeiten für hierarchische Zugriffsanforderungen
- **Zugriff über Sekundärschlüssel**
  - Einstiegs- und Verknüpfungsstruktur
  - Einsatz von Zeigerlisten und komprimierten Bitlisten
- **Bitlistenkomprimierung**
  - Laufkomprimierung (run length compression), Nullfolgen-Komprimierung
  - Mehr-Modus-Komprimierung, Blockkomprimierung
  - Huffman-Codes
- **Implementierung von DeweyIDs**
  - Einsatz als variabel lange Schlüssel und
  - als variabel lange Verweise in Listen
  - Codierungstechniken
- **Hierarchische Zugriffspfade**
- **Verallgemeinerte Zugriffspfadstruktur**
- **Verbund- und Pfadindex**

Verknüpfungsstrukturen für Satzmengen

- **Materialisierte Speicherung**

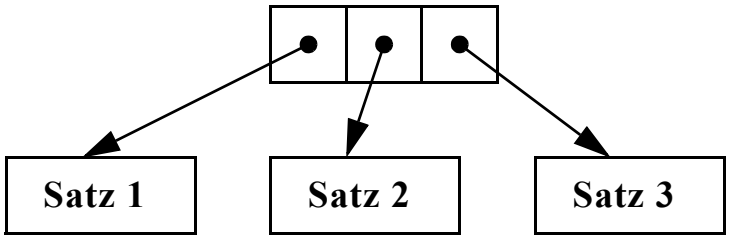
1. Physische Nachbarschaft der Sätze (Cluster-Bildung, Listen)

2. Verkettung der Sätze

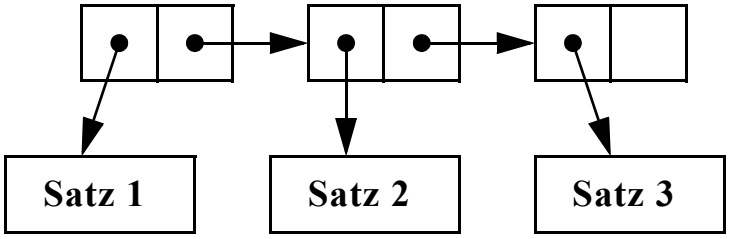


- **Referenzierte Speicherung**

3. Physische Nachbarschaft der Zeiger (Invertierung)



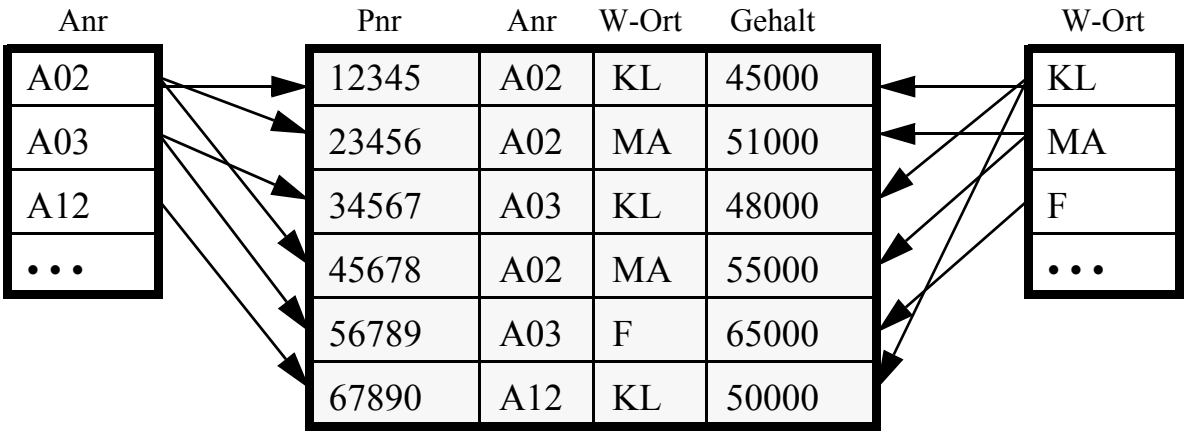
4. Verkettung der Zeiger



Zugriffspfade für Sekundärschlüssel

- **Suche nach Sätzen mit einem vorgegebenen Wert**
  eines nicht-identifizierenden Attributs (*Sekundärschlüssel*)

- **Ergebnis ist Satzmenge**



- **Realisierung: Einstiegsstruktur + Verknüpfungsstruktur**

  - Primärschlüssel-Zugriffspfade als Einstiegsstruktur auf Satzmengen anwendbar
  - Prinzipiell lassen sich alle Verknüpfungsstrukturen für Satzmengen heranziehen

  ➥ vor allem: Verwendung von B*-Bäumen und Invertierungstechniken

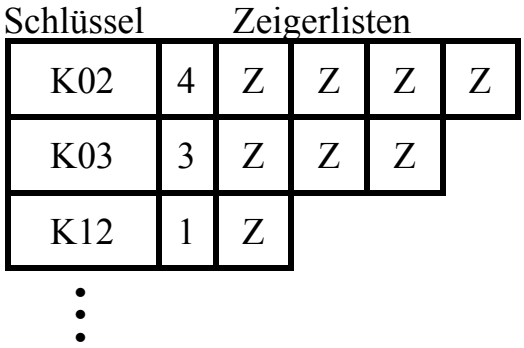
- **Standardlösung bei der Invertierung** sind sequentielle Verweislisten
  (oft OID-Listen oder TID-Listen genannt)

  - effiziente Durchführung von Mengenoperationen
  - kosteneffektive Wartung

Zugriffspfade für Sekundärschlüssel (2)

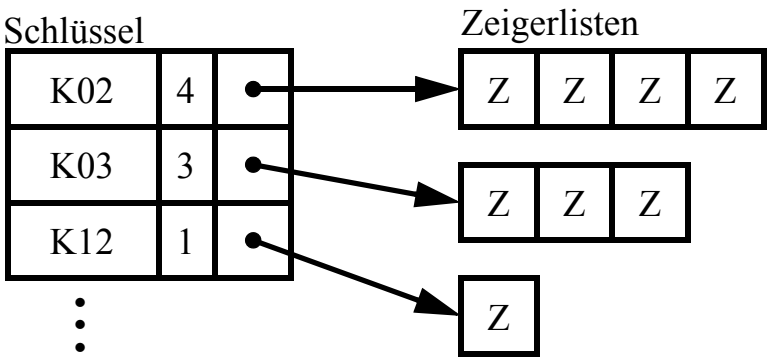
- **Häufige Realisierung: Invertierung**
  - Trennung der Zugriffspfaddaten von den Datensätzen (referenzierte Speicherung)
  - Verweis Z realisiert als TID, DBK/PPP, ...
  - Es kommen zwei Darstellungsmethoden in Betracht:

a) **Gemeinsame Verwaltung** der Suchstruktur und der Zeigerlisten



➥ relativ kurze Zeigerlisten erforderlich!

b) In der Suchstruktur ist (ähnlich wie bei Zugriffspfaden für Primärschlüssel) nur ein Verweis pro Schlüsselwert vorhanden, der zu einer Liste mit Satzverweisen führt (Zeigerliste)

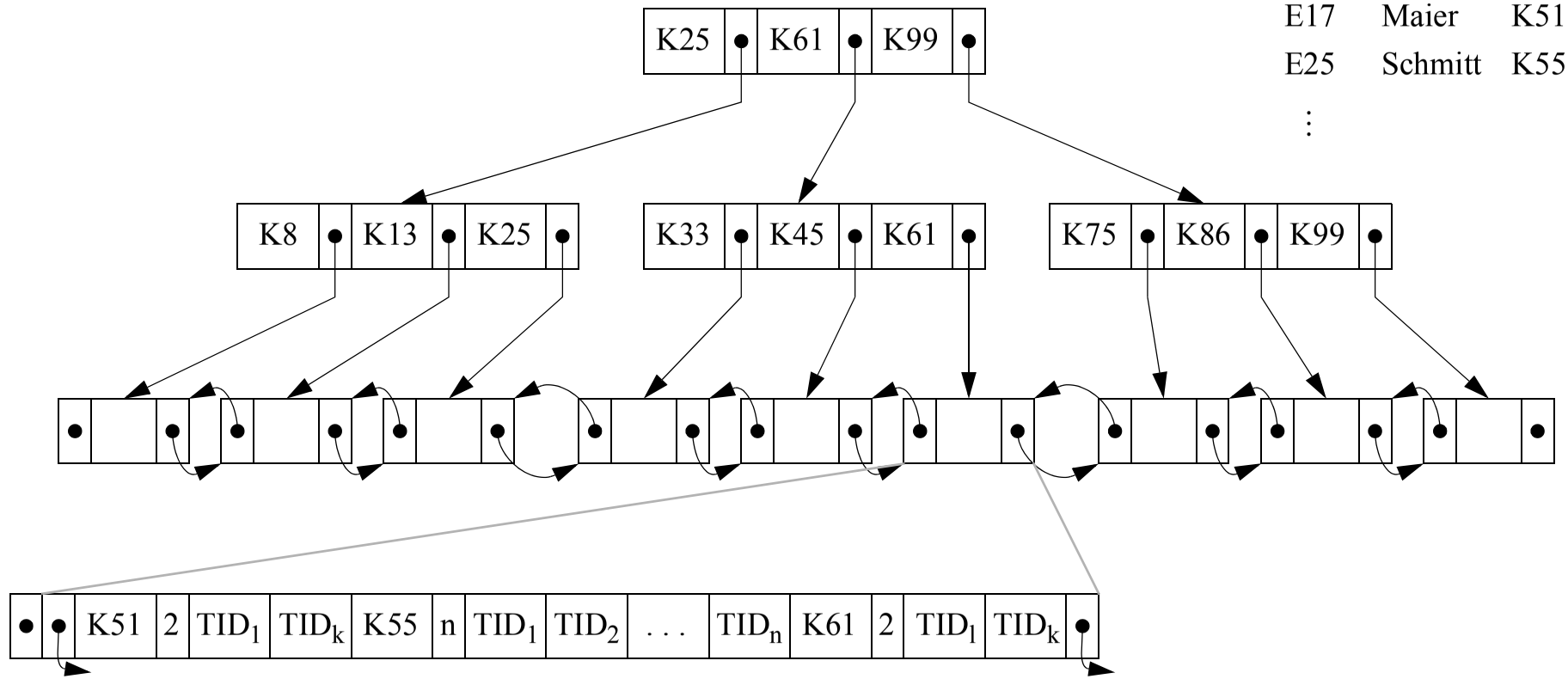


➥ Zeigerlisten können in separaten „Behältern" abgelegt werden

Zugriffspfade für Sekundärschlüssel (3)

Tabelle Pers

Pers (	Pnr,	Name,	Anr,	...)
	E1	Müller	K55	...
	E17	Maier	K51	
	E25	Schmitt	K55	
	⋮			

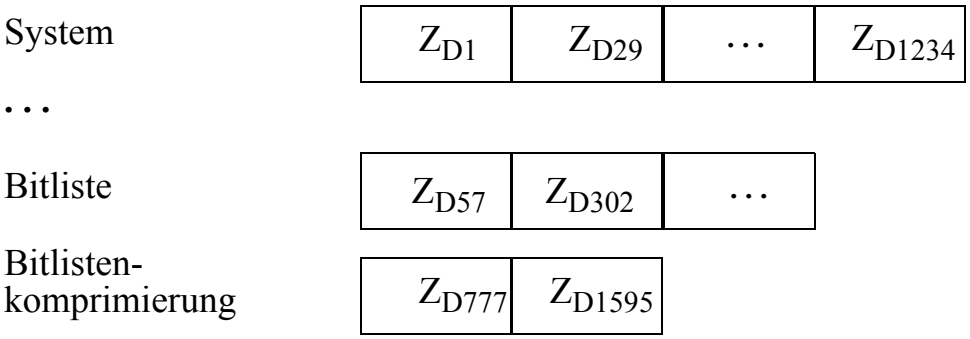


B*-Baum

- als Zugriffspfad für Sekundärschlüssel Anr
- mit Sortierreihenfolge der Sekundärschlüssel (Bereichsfragen!)
  sowie Vorwärts- und Rückwärtsverkettung

Zugriffspfade für Sekundärschlüssel (4)

- **Einsatzmöglichkeit auch beim Information Retrieval**
  - unformatierte Daten: Dokumente
  - Invertierung mit Hilfe von **Deskriptoren** (keine Zuordnung zu Attributen!)



➥ Es sind sehr viele und sehr wenige Verweise möglich

- **Invertierung mit Bitlisten**
  - Adressierung von Datensätzen oder Dokumenten
    - über Zuordnungstabelle ZT
    - direkt bei fester Länge und fortlaufender Speicherung
  - Markierungen der Bitliste entsprechen Einträgen von ZT oder berechenbaren Adressen (b Sätze pro Seite)
  - Attribut A habe j Attributwerte a_1, ..., a_j

- **Bitmatrix für A**

	1 2 3 . . .	**n**
a_1	**0 1 0 0 1 0 0 . . .**	
a_2	**1 0 0 0 0 0 1**	
	. . .	
a_j	**0 0 0 1 0 1 0**	

➥ **Speicherung als vertikale Bitlisten** erlaubt Indexierung von mehrwertigen Attributen (Bsp: Warenkorb mit Produkten)

Zugriffspfade für Sekundärschlüssel (5)

- **Bitlisten fester Länge**
  - j_i Bitlisten von Attribut A_i
  - einfache Änderungsoperationen
  - schneller Vergleich
  - sehr speicherplatzaufwendig
  - nur für kleine j

    ➥ oft lange Nullfolgen: Komprimierung

- **Komprimierte Bitlisten variabler Länge**
  - Speicherplatzeinsparung
  - Reduktion der E/A-Zeit
  - Mehraufwand für Codierung und Decodierung
  - schneller Vergleich
  - umständliche Änderungsoperationen

- **Anwendungsgebiete der Komprimierung**
  - Data Warehouse (Invertierung der Faktentabelle)
  - Übertragung/Speicherung
    - von Multimedia-Objekten (Image, Audio, Video, ...)
    - von dünn besetzten Matrizen
    - von Objekten in Geo-DB, ...

- **Viele Komprimierungstechniken verfügbar**

Komprimierung von Bitlisten

• Laufkomprimierung

Ein „Lauf“ oder ein „Run“ ist eine Bitfolge gleichartig gesetzter Bits. Bei der Laufkomprimierung wird die unkomprimierte Bitliste in aneinanderhängende, alternierende Null- und Einsfolgen aufgeteilt. Die Komprimierungstechnik besteht nun darin, jeden „Lauf“ durch seine Länge in einer Codierfolge darzustellen. Eine Codierfolge kann sich aus mehreren Codiereinheiten fester Länge (k Bits) zusammensetzen. Aus implementierungstechnischen Gründen wird k meist als ein Vielfaches der Bytelänge 8 gewählt. Falls eine Lauflänge größer als (2^k-1) Bits ist, wird zu ihrer Abbildung als Codierfolge mehr als eine Codiereinheit benötigt. Dabei gilt allgemein für die Komprimierung einer Bitfolge der Länge L mit

$$(n-1) * (2^k-1) < L \leq n * (2^k-1), \qquad n = 1, 2, \ldots$$

daß n Codiereinheiten erforderlich sind, wobei die ersten (n-1) Codiereinheiten voll mit Nullen belegt sind. Durch dieses Merkmal kann die Zugehörigkeit aufeinanderfolgender Codiereinheiten zu einer Codierfolge erkannt werden. Die Überprüfung jeder Codiereinheit auf vollständige Nullbelegung ist bei der Dekomprimierung zwar aufwendiger; die Einführung dieser impliziten Kennzeichnung der Fortsetzung einer Folge verhindert aber, daß das Verfahren bei Folgen der Länge > 2^k scheitert.

Ein Beispiel soll diese Technik verdeutlichen (k=6):

Lauflänge	Codierung
1	000001
2	000010
63	111111
64	000000 000001
65	000000 000010

• Nullfolgenkomprimierung

Eine Nullfolge ist eine Folge von 0-Bits zwischen zwei 1-Bits in der unkomprimierten Bitliste. Der Grundgedanke dieser Technik ist es, die Bitliste nur durch aufeinanderfolgende Nullfolgen darzustellen, wobei jeweils ein 1-Bit implizit ausgedrückt wird. Da jetzt auch die Länge L=0 einer Nullfolge auftreten kann, ist folgende Codierung zu wählen (k=6), was einer Addition von Binärzahlen ≤ 2k-1 entspricht:

Komprimierung von Bitlisten (2)

- **Nullfolgenkomprimierung (Forts.)**

Nullfolgenlänge	Codierung
0	000000
1	000001
62	111110
63	111111 000000
64	111111 000001

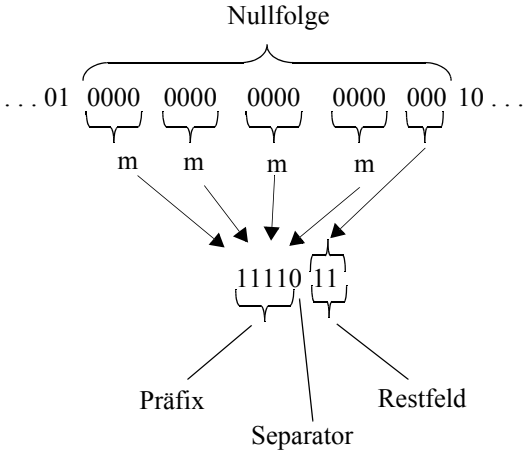
Durch die Möglichkeit, eine Codierfolge wiederum additiv durch mehrere Codiereinheiten zusammenzusetzen, lassen sich beliebig lange Nullfolgen darstellen. Es sind n Codiereinheiten erforderlich, wenn

$$(n-1) * (2^k-1) \leq L < n * (2^k-1), \qquad n = 1, 2, \ldots$$

gilt.

- **Golomb-Codierung**

Eine andere Art der Nullfolgenkomprimierung durch variabel lange Codierfolgen ist unter dem Namen Golomb-Code bekannt. Eine Nullfolge der Länge L wird durch eine Codierfolge bestehend aus einem variabel langen Präfix, einem Separatorbit und einem Restfeld fester Länge mit $\lceil \log_2 m \rceil$ Bit dargestellt. Der Präfix setzt sich aus $\lfloor L/m \rfloor$ 1-Bits gefolgt von einem 0-Bit als Separator zusammen. Das Restfeld beschreibt als Binärzahl die Anzahl der restlich 0-Bits der Nullfolge: $L - m*\lfloor L/m \rfloor$. Dieses Verfahren besitzt den Vorteil, unabhängig von der Wahl der Parameter die Komprimierung beliebig langer Nullfolgen zu gestatten. Wenn p die 0-Bit-Wahrscheinlichkeit in der Bitliste ist, sollte der Parameter m so gewählt werden, daß $p^m \approx 0.5$ ist. Die Komprimierung wird bei diesem Verfahren folgendermaßen durchgeführt (m=4):



Komprimierung von Bitlisten (3)

• Mehr-Modus-Komprimierung

Eine weitere Möglichkeit, Bitlisten zu verdichten, besteht darin, ein oder zwei Bits der Codierfolge fester Länge k als sogenannte Kennbits zu reservieren, um verschiedene Modi der Codierfolge zu kennzeichnen. Mit einem Kennbit lassen sich folgende zwei Modi unterscheiden:

1: k-1 Bits der Folge werden als „Bitmuster“ übernommen;

0: $\leq 2^{k-1}$-1 Bits werden als Nullfolge durch eine Binärzahl ausgedrückt.

Bei dieser Art der Codierung erweist es sich als nachteilig, daß wegen der Beschränkung von k eine Nullfolge oft durch mehrere aufeinanderfolgende Codierfolgen komprimiert werden muß. Weiterhin ist für freistehende Einsen in der Bitliste eine Codierfolge zu „opfern“, um sie als Bitmuster ausdrücken zu können. Durch Reservierung eines weiteren Kennbits sind diese Defekte zu beheben. Es besteht dann zusätzlich die Möglichkeit, lange Einsfolgen als Binärzahl verdichtet zu speichern. Mit zwei Kennbits lassen sich beispielsweise folgende vier Modi unterscheiden:

11: k-2 Bits der Folge werden als Bitmuster übernommen;

10: $\leq 2^{k-2}$ -1 Bits werden als Einsfolge durch eine Binärzahl codiert;

01: $\leq 2^{k-2}$ -1 Bits werden als Nullfolge durch eine Binärzahl codiert;

00: $\leq 2^{2k-2}$ -1 Bits werden in einer verdoppelten Codierfolge als Nullfolge durch eine Binärzahl ausgedrückt.

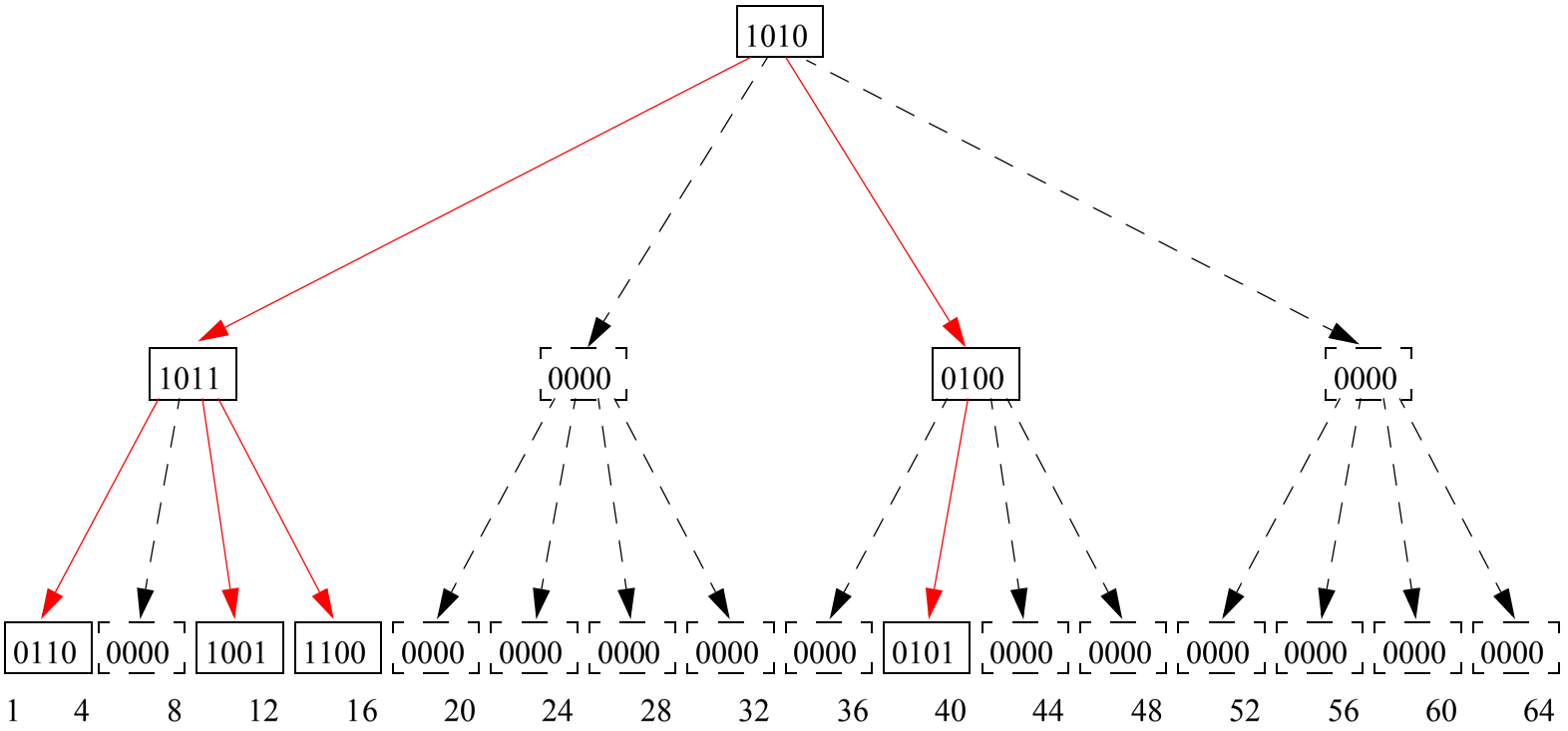
• Blockkomprimierung

Bei dieser Klasse von Verfahren wird die unkomprimierte Bitliste in Blöcke der Länge k eingeteilt. Eine erste Technik besteht darin, die einzelnen Blöcke durch einen Code variabler Länge zu ersetzen. Wenn die Wahrscheinlichkeit des Auftretens von Markierungen bekannt ist oder abgeschätzt werden kann, läßt sich ein sogenannter Huffman-Code anwenden. Bei einer Blocklänge k gibt es 2k verschiedene Belegungen, für die in Abhängigkeit der Wahrscheinlichkeit ihres Auftretens 2k Codeworte variabler Länge zu konstruieren sind. Komprimierung und Dekomprimierung sind bei dieser Technik recht aufwendig, da mit Hilfe einer Umsetztabelle jeder Block durch sein Codewort und umgekehrt zu ersetzen ist.

Bei einer zweiten Technik werden nur solche Blöcke gespeichert, in denen ein oder mehrere Bits gesetzt sind. Zur Kennzeichnung der weggelassenen Blöcke wird eine zweite Bitliste als Directory verwaltet, in der jede Markierung einem gespeicherten Block entspricht. Da im Directory wiederum lange Nullfolgen auftreten können, läßt es sich vorteilhaft mit Methoden der Nullfolgen- oder Mehr-Modus-Komprimierung verdichten.

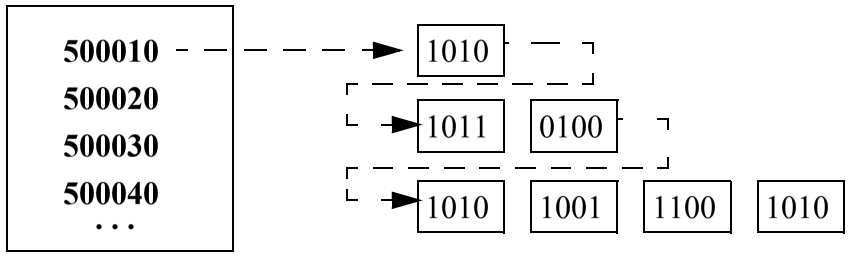
Die Idee, auf das Directory wiederum eine Blockkomprimierung anzuwenden, führt auf die hierarchische Blockkomprimierung. Sie läßt sich rekursiv solange fortsetzen, bis sich die Eliminierung von Nullfolgen nicht mehr lohnt. Ausgehend von der obersten Hierarchiestufe kann die unkomprimierte Bitliste (Indextiefe d) leicht rekonstruiert werden.

Beispiele



Beispiel

- Knotenlänge l = 4 und Indextiefe d = 3
- indexierte Menge S = {2, 3, 9, 12, 13, 14, 38, 40}
- physische Speicherung



Optimale Codes

- **Erweiterte Binärbäume mit minimaler externer Pfadlänge lassen sich zum Entwurf optimaler Codes für n+1 Zeichen heranziehen.**

- **zu codierende Zeichenfolge**

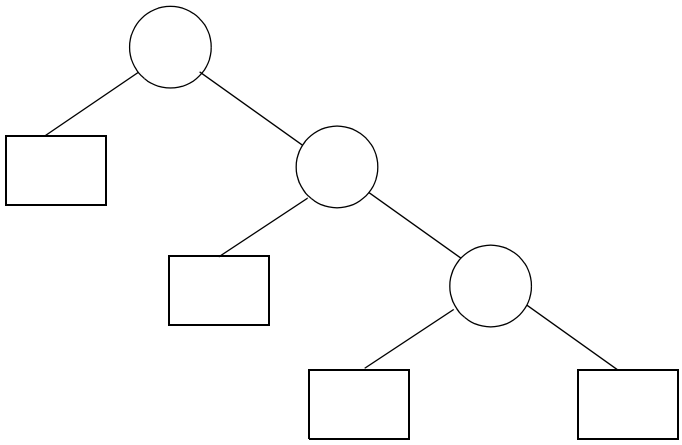
  A A B C A A B B C A D B A B A 15 Zeichen

- **Codierung fester Länge: 2 Bit** **A = 00**
  . . .
  D = 11

  $C_{2Bit} = 15 \cdot 2 = 30$

- **Gibt es bessere Codierungen?**

  Zeichen Häufigkeit Code



➥ kein Zeichen ist Präfix eines anderen

$E_w = C_{Code}$

➥ Die Decodierung läßt sich mit dem gleichen erweiterten Binärbaum, der zur Bestimmung der Codes gefunden wurde, durchführen

- **Arbeitsweise**

  A A B C A . . . = 0 0 1 0 1 1 0 0 . . .
  0|0|10|110|0| . . = A A B C A . . .

Huffman-Algorithmus

- Die minimale Codierung erhält man mit Hilfe von erweiterten Binärbäumen minimaler gewichteter externer Pfadlänge. Die daraus resultierenden Codes bezeichnet man auch als Huffman-Codes.

- Algorithmus zur Konstruktion von
  Binärbäumen mit minimaler gewichteter externer Pfadlänge

  Gegeben:
  Liste von Bäumen, die anfänglich aus n externen Knoten als Wurzeln besteht. In den Wurzeln der Bäume sind die Häufigkeiten q_i eingetragen

  Idee:
  Suche die beiden Bäume mit den geringsten Häufigkeiten und entferne sie aus der Liste. Die beiden gefundenen Bäume werden als linker und rechter Unterbaum mit Hilfe eines neuen Wurzelknotens zu einem neuen Baum zusammengefügt und in die Liste eingetragen.

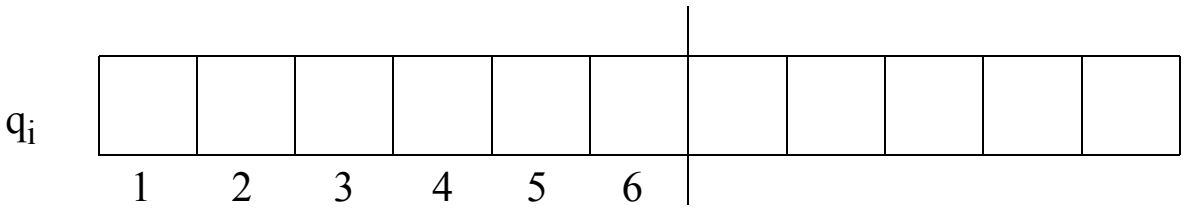
  ➥ n externe Knoten → n-1 Bäume = interne Knoten

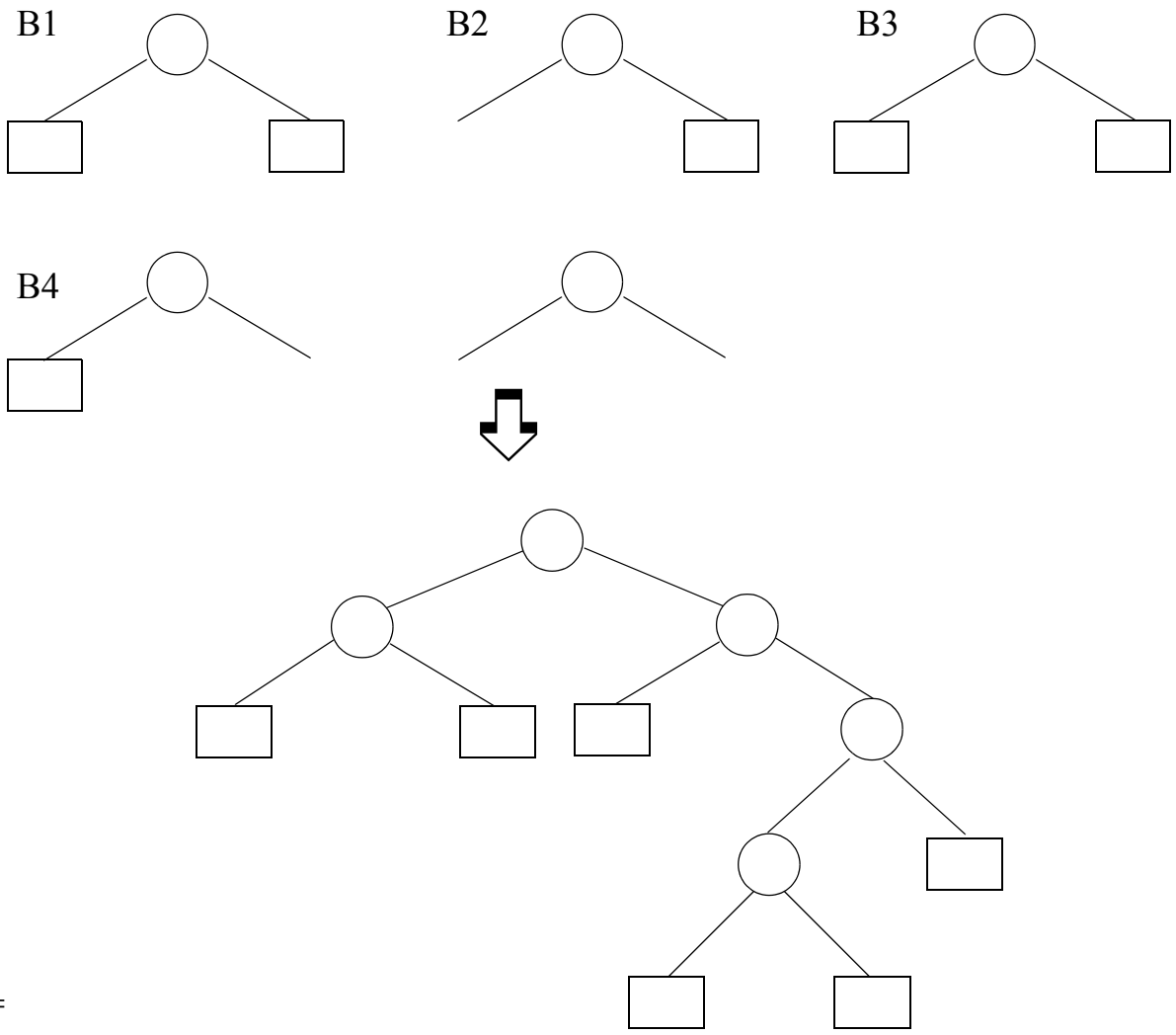
Algorithmus: Huffman(BaumListe liste, int n)

```
for (i = 1; i < n; i += 1)
{  p1 = "kleinstes Element aus liste"
   "Entferne p1 aus liste"
   p2 = "kleinstes Element aus liste"
   “Entferne p2 aus liste"
   “Erzeuge Knoten p"
   “Hänge p1 und p2 als Unterbäume an p an"
   "Bestimme das Gewicht von p als Summe der Gewichte p1 und p2”
   "Füge p in liste ein"
}
```

Huffman-Algorithmus (2)

- **Durchlauf-Beispiel**



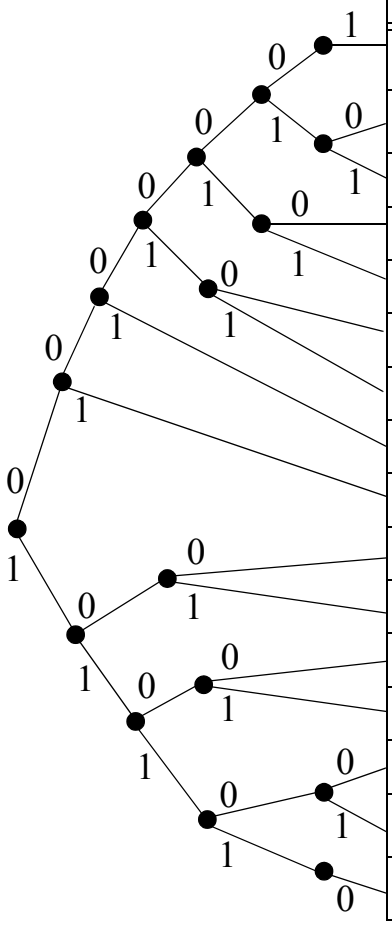


$E_w =$

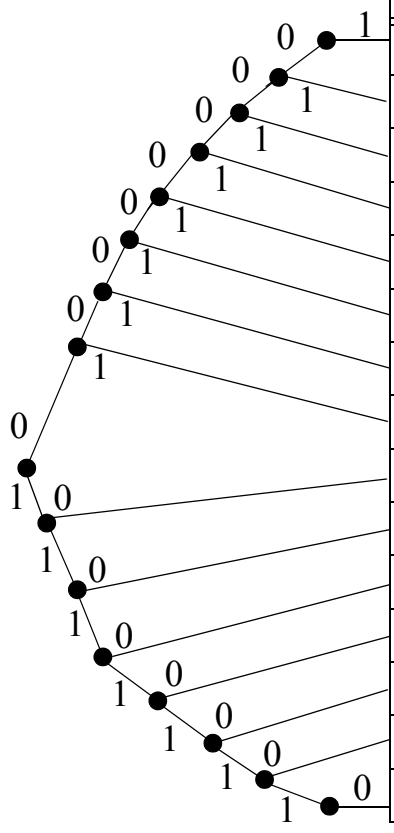
- **Kosten:**

$$C = C_0 \sum_{i=1}^{n-1} (n+i) = C_0(n-1)\left(n+\frac{n}{2}\right) = O(n^2)$$

Zuordnung von Huffman-Codes – Beispiel



Bitstring	L_i	O_i value range
0000001	48	[-2.8x10^{14}, -4.3x10^{9}]
0000010	32	[-4.3x10^{9}, -69977]
0000011	16	[-69976, -4441]
000010	12	[-4440, -345]
000011	8	[-344, -89]
00010	6	[-88, -25]
00011	4	[-24, -9]
001	3	[-8, -1]
01	3	[0, 7]
100	4	[8, 23]
101	6	[24, 87]
1100	8	[88, 343]
1101	12	[344, 4439]
11100	16	[4440, 69975]
11101	32	[69976, 4.3x10^{9}]
11110	48	[3.3x10^{9}, 2.8x10^{14}]

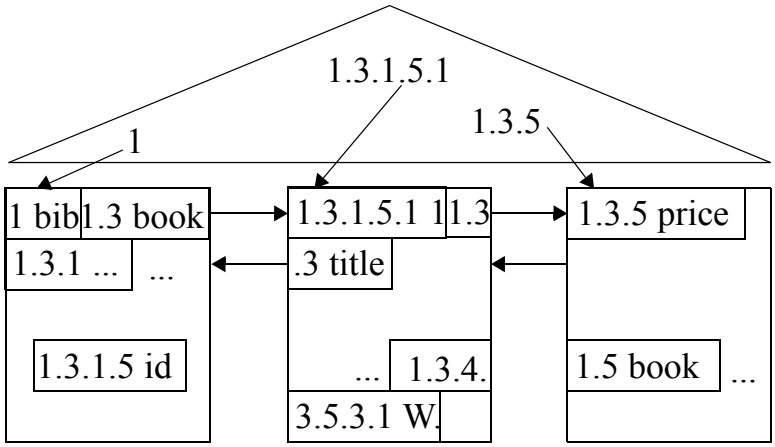


Bitstring	L_i	O_i value range
000000001	20	[-1118485, -69910]
00000001	16	[-69909, -4374]]
0000001	12	[-4373, -278]
000001	8	[-277, -22]
00001	4	[-21, -6]
0001	2	[-5, -2]
001	1	[-1, 0]
01	0	[1, 1]
10	1	[2, 3]
110	2	[4, 7]
1110	4	[8, 23]
11110	8	[24, 279]
111110	12	[280, 4375]
1111110	16	[4376, 69911]
11111110	20	[69912, 1118487]

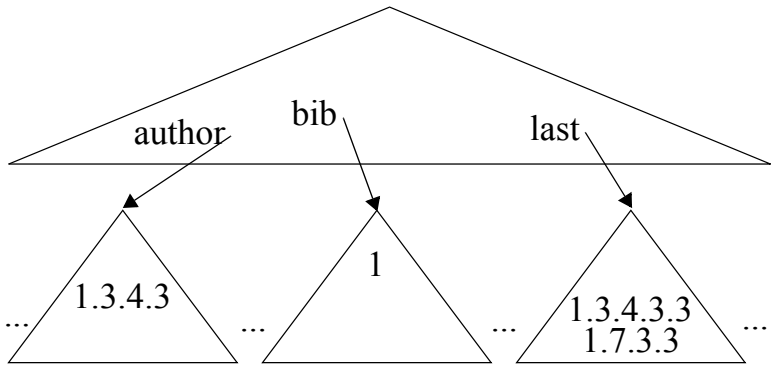
Implementation of DeweyIDs

- **Use of DeweyIDs**
  - Document storage using B*-trees
  - DeweyIDs have to be compressed effectively and used as **variable-length keys and pointers**

a) Storage structure



b) Element index



  - The storage structure consists of a ***document index*** and a ***document container*** and the element index of a ***name directory*** and a ***node-reference index***
  - DeweyIDs
    - are effectively encoded and
    - further compressed by **prefix compression**
  - The nodes are stored in document order (left-most depth-first) in the doubly chained document container which preserves the **sort sequence** of the DeweyIDs which should make prefix compression very effective.

Encoding of DeweyIDs

- **Fixed length field**

TL	L_0	O_0	L_1	O_1	. . .	L_k	O_k

TL = totel length

A division value O needs a **variable-length representation** which could be achieved in the simplest case by attaching a fixed-length field L_f representing the actual length of O. However, what is an adequate value l_f for L_f? Most division values are expected to be rather small (<100), but some of them could reach $>16*10^6$. While for the former example value 7 bits would be sufficient, the latter would require >24 bits. Furthermore, whatever reasonable value for l_f is chosen, it is **not space optimal** and additionally introduces an implementation restriction.

- **Fixed- and variable-length length fields**

TL	L_{f0}	L_{v0}	O_0	L_{f1}	. . .	L_{vk}	O_k

The length indicator itself can be made of variable length. A straightforward approach is to spend a fixed-length field L_{fi} of length l_f to describe the actual length of L_{vi}. A length l_f of L_{fi} allows—given length 0 does not occur—the representation of a length

2^{l_f} of L_v limiting the length of divisions O to $L_O = 2^{2^{l_f}}$. and their values to 2^{L_O}.

While $l_f = 2$ is not big enough for the general case, $l_f = 3$ definitely is for all practical applications. However, such a scheme carries the penalty for the frequent divisions with small values.

Encoding of DeweyIDs (2)

- **3-based decimal representation**

  Interpret (variable-length) 2-bit codes as numbers with base 3
  0: 00, 1: 01, 2: 10, ".": 11

  DeweyID 1.11.7: 0111010010111001

  Very good space efficiency, no adaptation to value distributions

  Is 2-based or 7-based decimal representation useful?

- **Huffman codes**

  Each division value is encoded by a template C_i | O_i where the dots are implicitly represented. C_i is a Huffman code which is indexing a translation table. Thus, we can freely assign length values L_i to the value ranges of O_i and, therefore, adjust them to value distributions

  Example for Huffman codes and length assignment

code	L_i	value range of O_i
01	3	1-8
100	4	9-24
101	6	25-88
1100	8	89-344
1101	12	345-4440
11100	16	4441-69976
11101	20	69977-1118552
11110	24	1118553-17895768
11111	31	17895769-2147483647

  DeweyID representation:

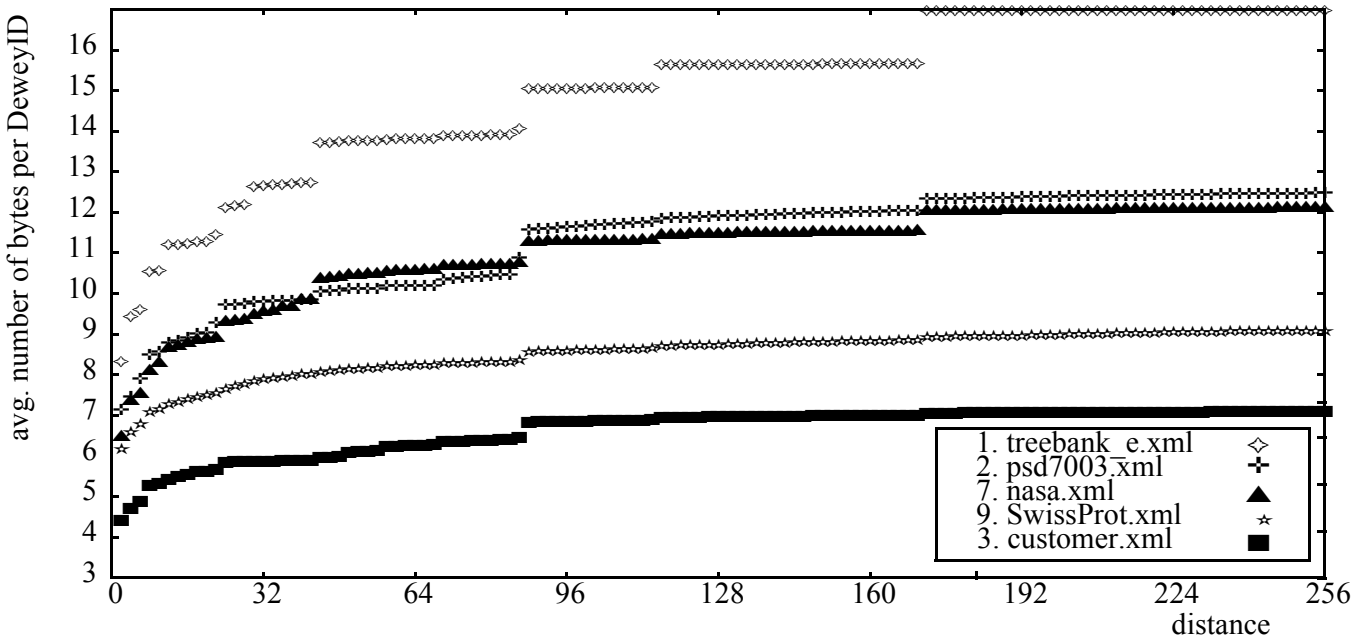
TL	C_0	O_0	C_1	O_1	. . .	C_k	O_k

Encoding of DeweyIDs (3)

- **Characteristics of XML documents considered**

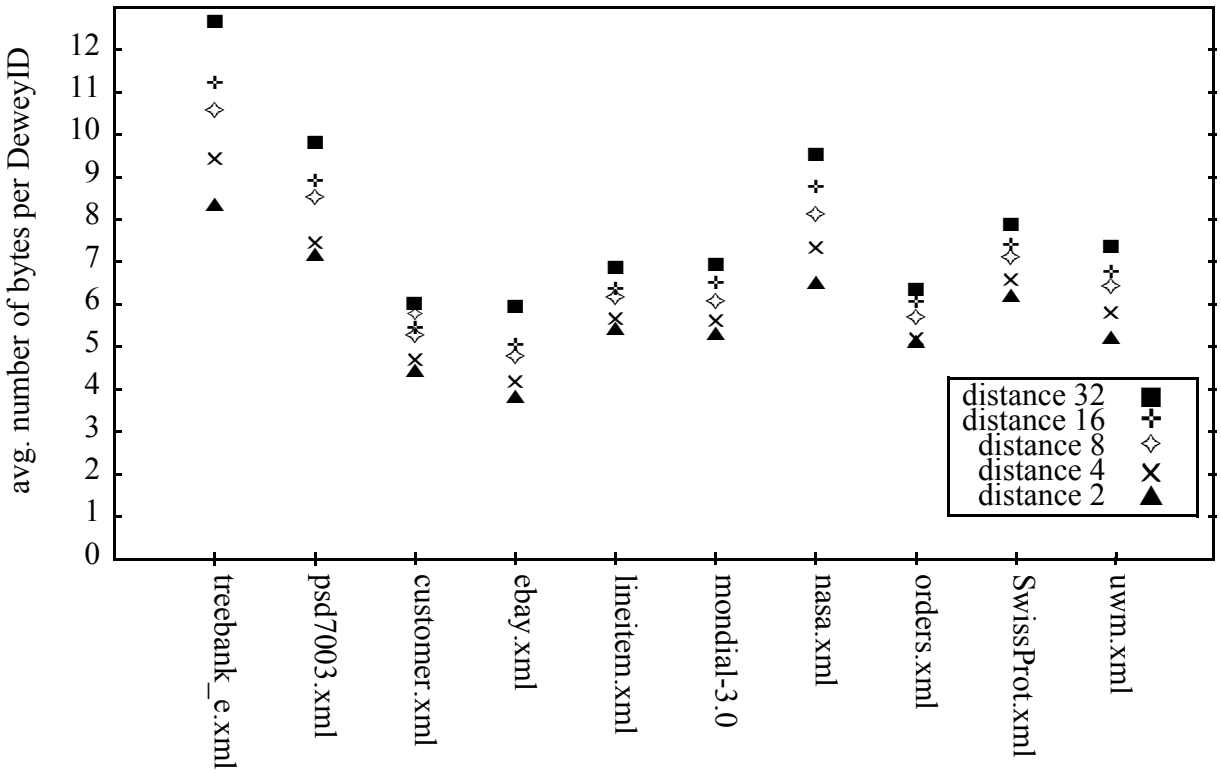
file name	description	size (bytes)	number of XML nodes	attributes	max. depth	∅–depth	max. fanout	∅–fan-out of elems
1) treebank_e.xml	Encoded DB of English records of Wall Street Journal	86082517	2437666	1	38	8.97	56385	2.33
2) psd7003.xml	DB of protein sequences	716853016	21305818	1290647	9	6.2	262527	3.99
3) customer.xml	Customers from TPC-H bench-mark	515660	13501	1	5	3.92	1501	8.99
4) ebay.xml	Ebay auction data	35562	156	0	7	4.76	12	5.0
5) lineitem.xml	Line items from TPC-H bench-mark	32295475	1022976	1	5	3.96	60176	17.0
6) mondial-3.0.xml	Geographical DB of diverse sources	1784825	22423	47423	8	5.25	955	4.43
7) nasa.xml	Astronomical data	25050288	476646	56317	10	6.62	2435	2.79
8) orders.xml	Orders from TPC-H Benchmark	5378845	150001	1	5	3.93	15001	10.0
9) SwissProt.xml	DB of protein sequences	114820211	2977031	2189859	7	4.9	50000	6.75
10) uwm.xml	Courses of a University Web-site	2337522	66729	6	7	4.83	2112	4.21

- **Avg. sizes of DeweyIDs grouped by the documents avg. depth**



Encoding of DeweyIDs (4)

- **Influence of the distance parameter**



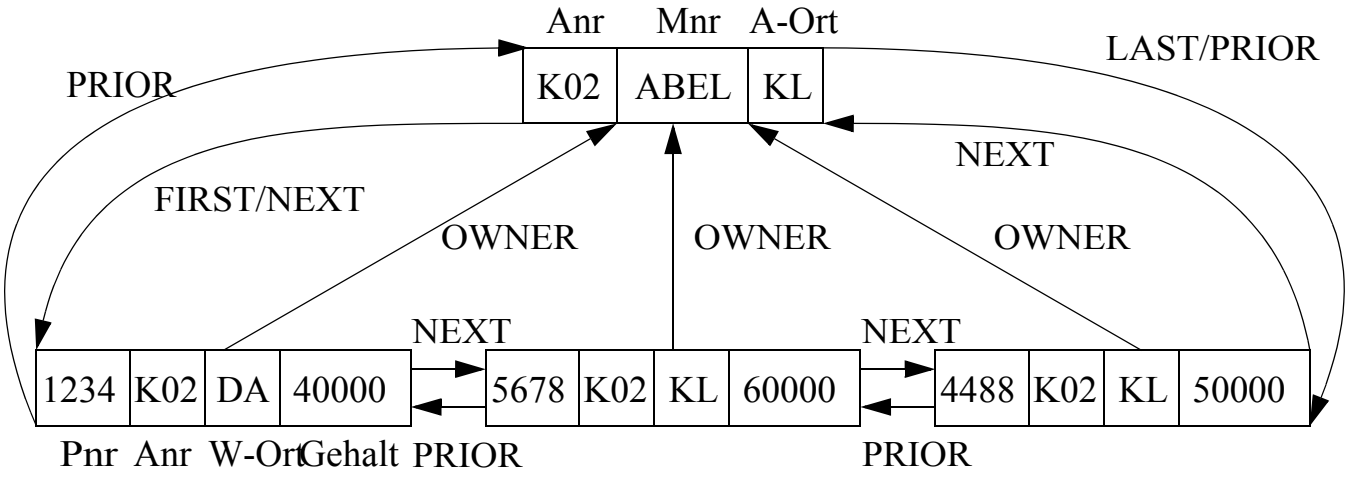
- **Provoking DeweyID reorganizations: worst-case node insertions**

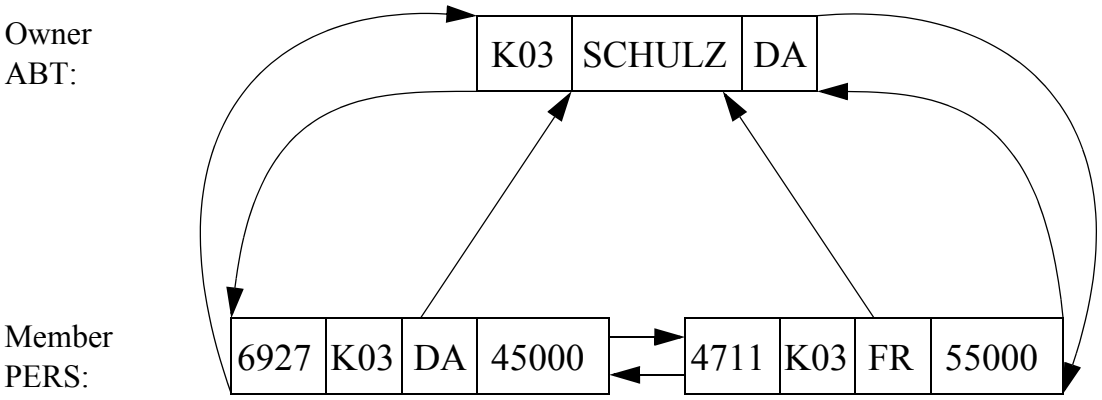
Hierarchische Zugriffspfade

- **Realisierung funktionaler Beziehungen zwischen zwei Satztypen**

  - Owner --> Member: Set-Typen nach dem Netzwerkmodell
  - Jede Ausprägung einer Owner-Satzart wird mit 0..n Ausprägungen der Member-Satzart verknüpft

Logische Sicht:
Darstellung der Navigationsmöglichkeiten

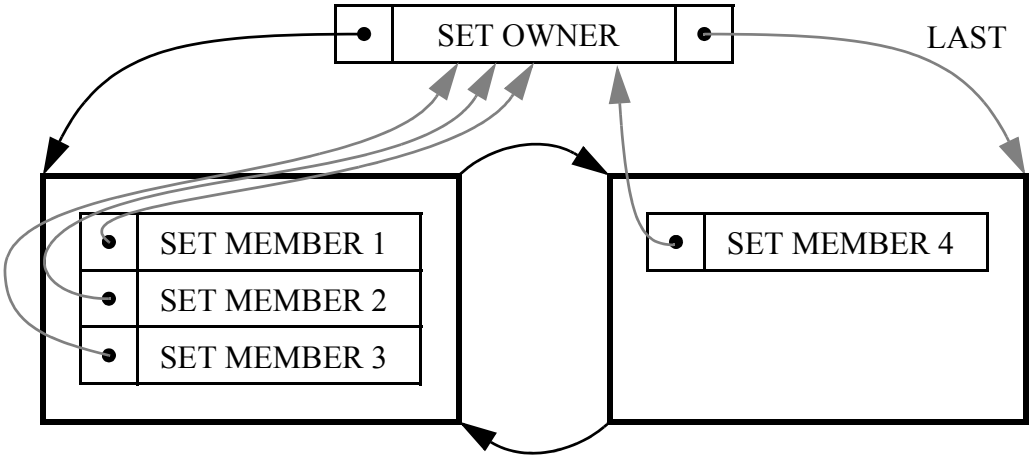




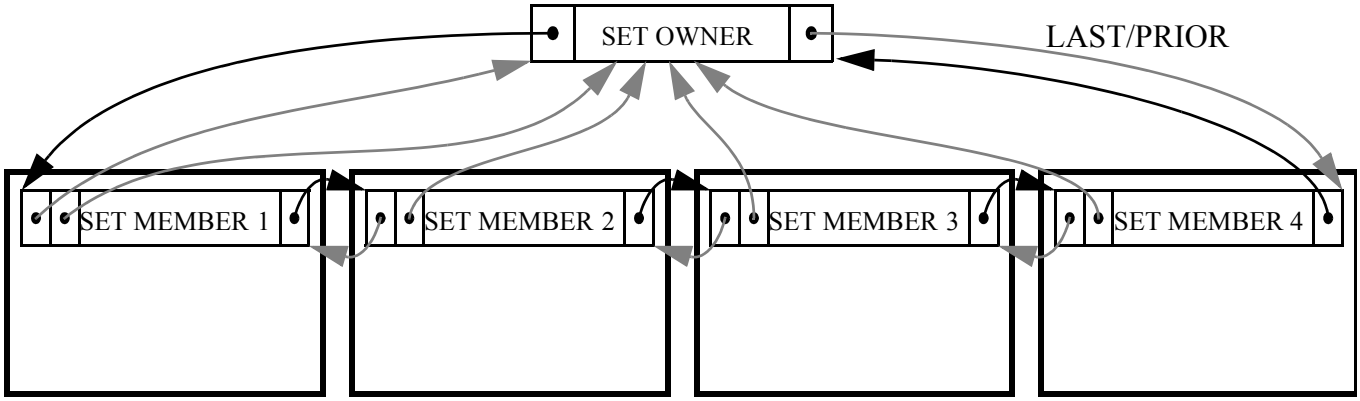
➥ drei Implementierungen für unterschiedliche Leistungsanforderungen

Hierarchische Zugriffspfade – Implementierung

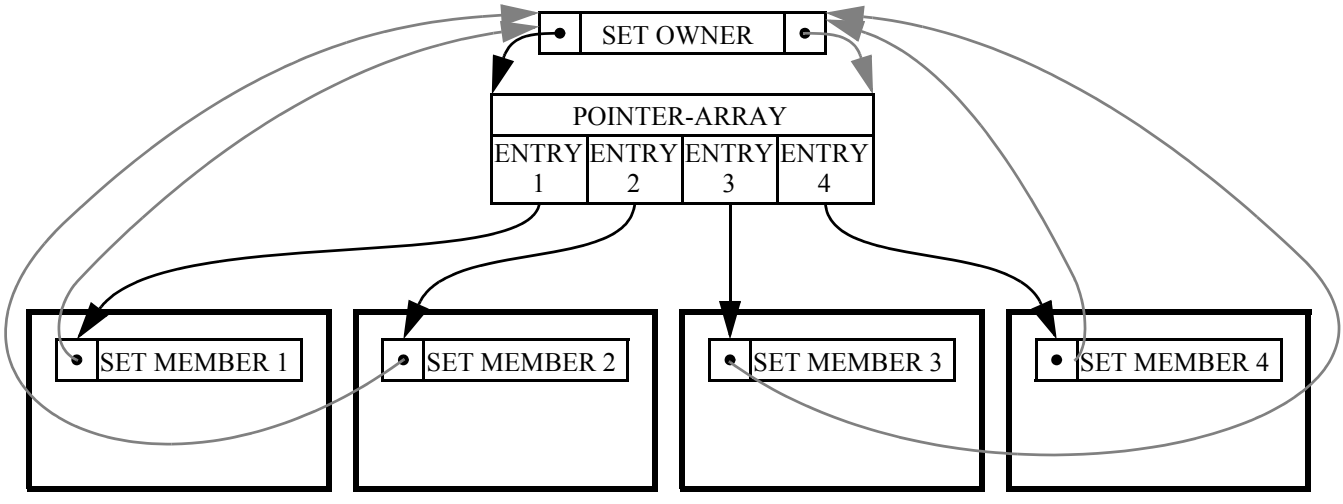
- **Sequentielle Liste auf Seitenbasis**



- **Gekettete Liste**



- **Pointer-Array-Struktur**



: optionaler Zeiger

Hierarchische Zugriffspfade – Bewertung der Implementierungstechniken

- **Pointer-Array**
  - stabiles Verhalten
  - Verhalten unabhängig vom Set-Wachstum und Set-Reihenfolge
  - 'Standard'-Verfahren bei unscharfen Informationen über Set-Größe und Zugriffshäufigkeit

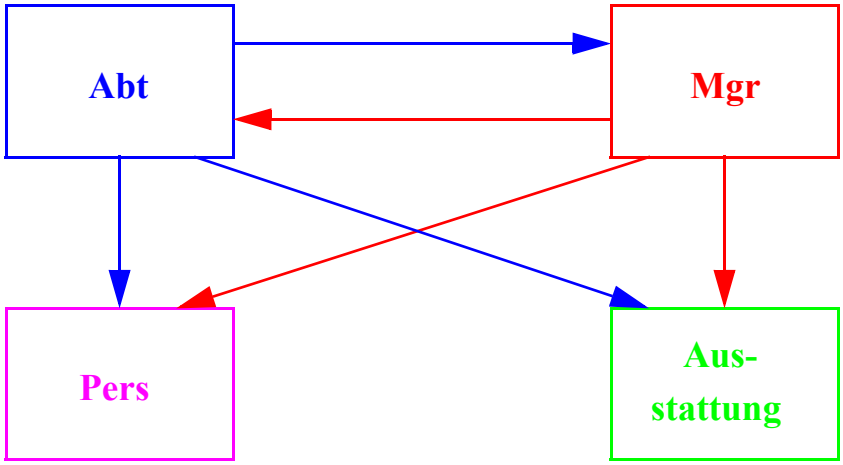
- **Sequentielle Liste**
  - auf einen Set-Typ pro Member-Satztyp beschränkt (Cluster-Bildung)
  - schnelles Aufsuchen / Einfügen in Set-Reihenfolge
  - Ändern teurer als bei Pointer-Array

- **Gekettete Liste**
  - Vorteile bei Mitgliedschaft des Member-Satztyps in mehreren Sets
  - billiger Wechsel auf andere Set-Ausprägungen
  - sequentieller Zugriff schneller als bei Pointer-Array
  - nur gut in kleinen Set-Ausprägungen

Verallgemeinerte Zugriffspfadstruktur

- **Idee:**

  gemeinsame Verwendung einer Indexstruktur (B*-Baum)
  für mehrere Satztypen, für die Beziehungen (1:1, 1:n, n:m)
  über demselben Wertebereich (z. B. für Anr) definiert und
  durch Gleichheit von Attributwerten repräsentiert sind

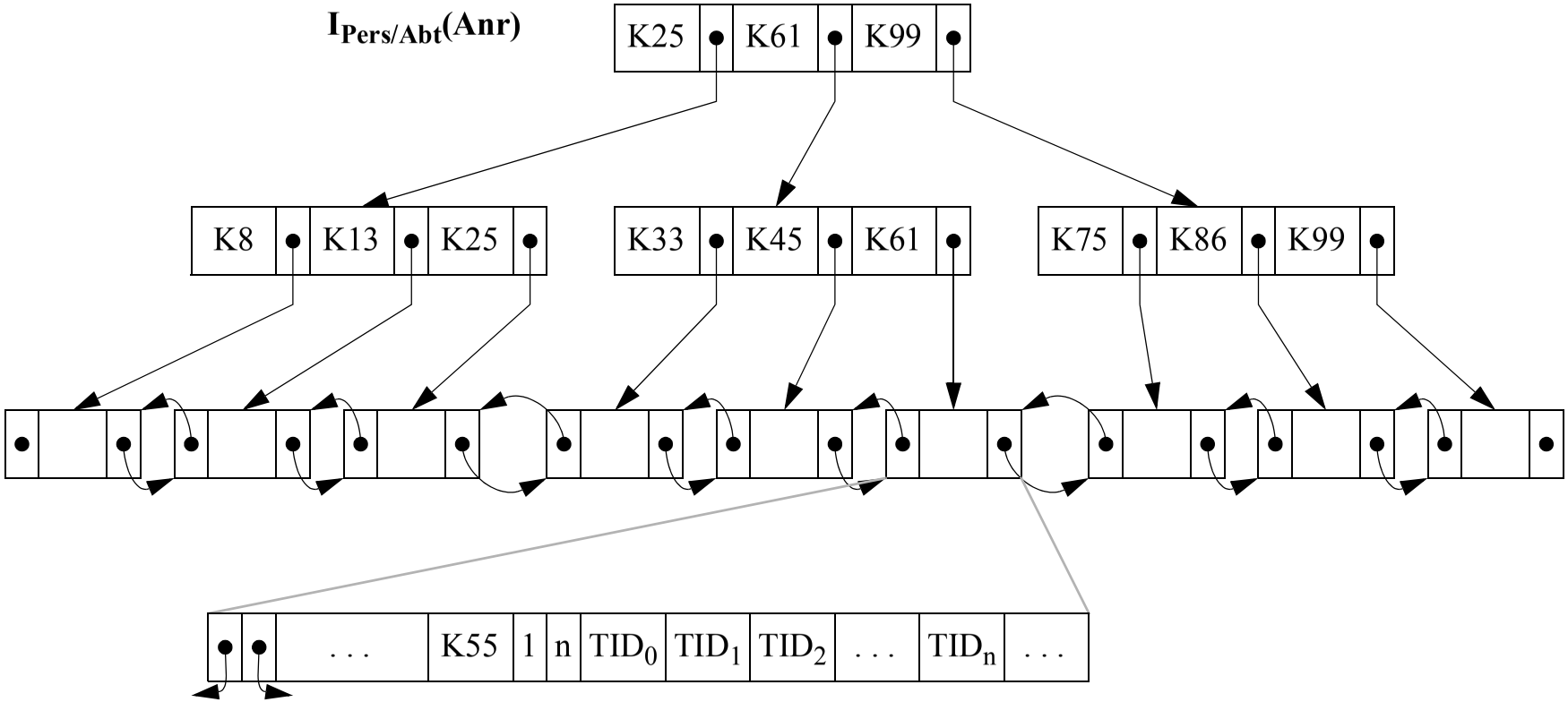


Alle Tabellen besitzen ein Attribut (z. B. Anr), das auf dem Wertebereich Abtnr definiert ist

- **Nutzung der Indexstruktur** für
  - Primärschlüsselzugriff z. B. als I_{Abt}(Anr)
  - Sekundärschlüsselzugriff z. B. als I_{Pers}(Anr)
  - hierarchischen Zugriff z. B. von Abt(Anr) nach Pers(Anr)
    oder in umgekehrter Richtung
  - Verbundoperationen (Join) z. B. von Abt.Anr = Pers.Anr

- **Kombinierte Realisierung** von Primärschlüssel-, Sekundärschlüssel- und hierarchischen Zugriffspfaden mit einem erweiterten B*-Baum
  - Innere Baumknoten bleiben unverändert
  - Blätter enthalten Verweise für primäre und sekundäre Zugriffspfade

B*-Baum als kombinierte Zugriffspfadstruktur

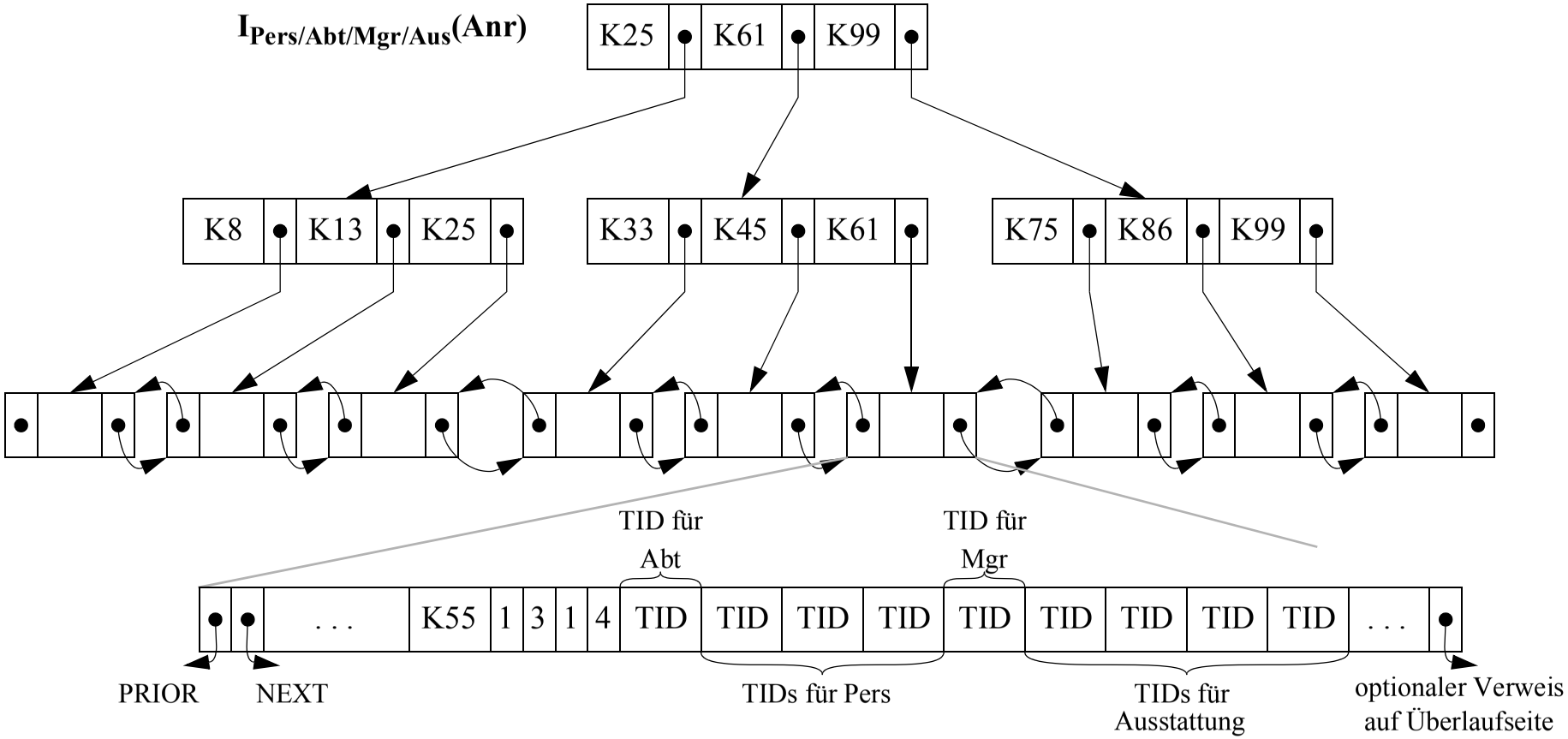


Struktur enthält Index für Abt, Pers und

Link für Abt-Pers mit direktem Zugriff von

1. OWNER zu jedem MEMBER,
2. jedem MEMBER zu jedem anderen MEMBER,
3. jedem MEMBER zum OWNER

B*-Baum als verallgemeinerte Zugriffspfadstruktur



Zugriffspfadstruktur umfaßt

- 4 Index-Strukturen
- 6 Link-Strukturen

Verallgemeinerte Zugriffspfadstruktur – Bewertung

- **Schlüssel werden nur einmal gespeichert**

  ➥ Speicherplatzersparnis

- **Einheitliche Struktur für alle Zugriffspfadtypen**

  ➥ Vereinfachung der Implementierung

- **Unterstützung der Join-Operation sowie bestimmter statistischer Anfragen**

- **Einfache Überprüfung der referentiellen Integrität sowie weiterer Integritätsbedingungen**

  (z. B. Kardinalitätsrestriktionen)

- **Erhöhung der Anzahl der Blattseiten**

  ➥ mehr Seitenzugriffe beim sequentiellen Lesen aller Sätze eines Satztyps in Sortierordnung

- **Höhe des Baumes bleibt meist erhalten**

  ➥ ähnliches Leistungsverhalten beim Aufsuchen von Daten und beim Änderungsdienst

Verbund- und Pfadindexe

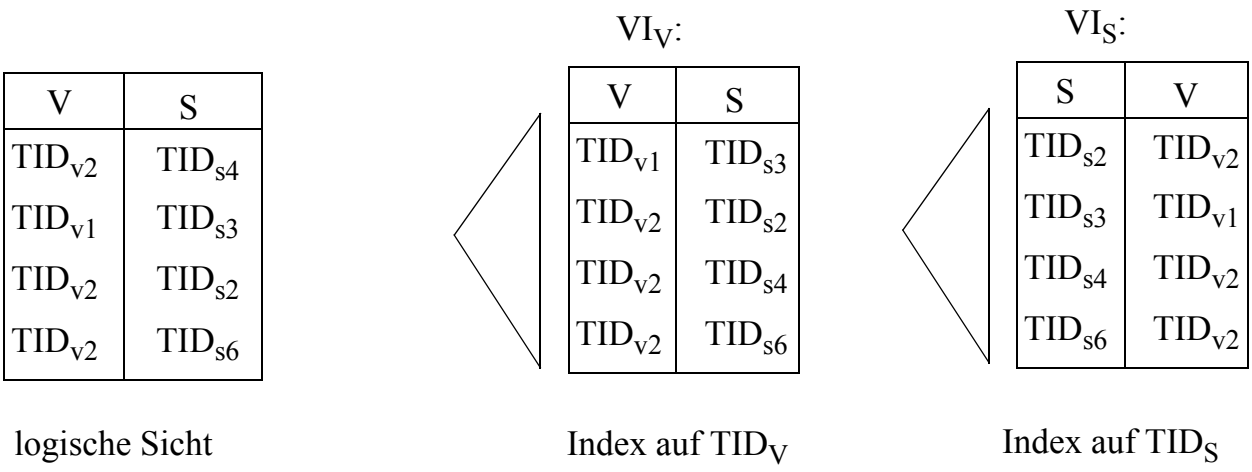
- **Verbundindex**

  - Der Verbundindex VI zwischen zwei Tabellen V und S (nicht notwendigerweise verschieden) mit den Verbundattributen A und B ist folgendermaßen definiert:

    $VI = \{(v.TID, s.TID) \mid f(v.A, s.B) \text{ ist TRUE}, v \in V, s \in S\}$

  - f bezeichnet eine Boolesche Funktion, die das Verbundprädikat, das sehr komplex sein kann, definiert. Insbesondere lassen sich dadurch Θ-Verbunde ($\Theta \in \{=, \neq, <, \leq, >, \geq\}$) spezifizieren.

  - Anwendung von **Selektionsprädikaten und Parallelisierung** beim Verbund



- **Mehrverbundindex**

  - **Verallgemeinerung der Idee**, Verbundoperationen über einen vorab berechneten Verbundindex effizient abzuwickeln

  - Index für einen Zwei-Weg-Verbund wird dazu benutzt, die Verbundpartner in einer dritten Relation T zu bestimmen und die Indextabelle um eine Spalte für die TID_{ti} zu erweitern.

  - Falls zwei Indextabellen für VS und ST bereits vorhanden sind, lassen sich diese unmittelbar zu einer erweiterten Indextabelle VST verknüpfen

  - Soll der VST-Verbund nur Attribute von V und T enthalten, kann ein VT-Index angelegt werden. Zur Verbundberechnung ist die S-Spalte unverzichtbar

Verbund- und Pfadindexe (2)

- **Mehrverbundindex (Bsp.)**

  Indextabellen für den Verbund: logische Sicht

V	S
TID_{v1}	TID_{s3}
TID_{v2}	TID_{s4}
TID_{v2}	TID_{s2}

S	T
TID_{s2}	TID_{t1}
TID_{s3}	TID_{t2}
TID_{s3}	TID_{t3}
TID_{s4}	TID_{t4}
TID_{s4}	TID_{t5}

V	S	T
TID_{v1}	TID_{s3}	TID_{t2}
TID_{v1}	TID_{s3}	TID_{t3}
TID_{v2}	TID_{s4}	TID_{t4}
TID_{v2}	TID_{s4}	TID_{t5}
TID_{v2}	TID_{s2}	TID_{t1}

- **Beispiel**

  Gegeben seien Tabellen ABT, PERS, PROJ und PM (PNR, JNR), die eine (n:m)-Beziehung zwischen PERS (PNR, ANR, ...) und PROJ (JNR, ..., ORT) verkörpert.

```
Q2: SELECT A.ANR, A.ANAME
    FROM   ABT A, PERS P, PM M, PROJ J
    WHERE A.ANR = P.ANR
          AND   P.PNR = M.PNR
          AND   M.JNR = J.JNR
          AND   J.ORT = :X
```

  - Verlängerung auf n Tabellen möglich

- **Pfadindex**

  - Integration eines Index Ort in den Mehrverbundindex APMJ
  - erlaubt Auswertung spezieller **Anfragen auf dem Index**

ABT	PERS	PM	PROJ	ORT
TID_{a1}	TID_{p1}	TID_{m1}	TID_{j1}	Berlin
TID_{a1}	TID_{p2}	TID_{m3}	TID_{j1}	Berlin
TID_{a1}	TID_{p2}	TID_{m4}	TID_{j2}	Köln
TID_{a2}	TID_{p3}	TID_{m5}	TID_{j3}	Bonn
. . .	. . .	. . .	. . .	. . .

  - Annahme mehrwertiger Referenzattribute In ORDBMS
  - Analoger Pfadausdruck zu Q2: ABT.Beschäftigt-Pers.Mitarbeiter-an.ORT = :X

Physischer DB-Entwurf

- **Physischer DB-Entwurf**
  - Sorgfältige Wahl der wichtigen Attribute (Spalten), nach deren Werte Cluster-Bildung vorgenommen wird
  - Oft werden die Primärschlüsselattribute oder OWNER-MEMBER-Beziehungen für die Cluster-Bildung ausgewählt
  - Geeignete Wahl des Belegungsgrads beim Laden von Tabellen mit Cluster-Eigenschaft
  - **Welche Attribute sollen indexiert werden?**

- **Indexierungsheuristik**

  Indexstrukturen werden angelegt
  - auf allen Primär- und Fremdschlüsselattributen, was auch mit verallgemeinerten Zugriffspfadstrukturen erreicht werden kann
  - auf Attributen vom Typ DATE
  - auf Attributen, die in (häufigen) Anfragen in Gleichheits- oder IN-Prädikaten vorkommen

  ➥ Primär- und Fremdschlüsselindexierung erlaubt Navigation durch mehrere Tabellen ausschließlich mit Hilfe der Indexstrukturen

- **Alternative Indexierungsheuristik**
  - Indexstrukturen werden auf Primärschlüssel- und (möglicherweise) auf Fremdschlüsselattributen angelegt
  - Zusätzliche Indexstrukturen werden nur angelegt, wenn für eine aktuelle Anfrage der neue Index zehnmal weniger Sätze liefert als irgendein existierender Index

  ➥ **Beide Heuristiken liefern fast die gleichen Indexstrukturen für die meisten Datenbanken und Arbeitslasten**

Zusammenfassung

- **Zugriffspfade für Sekundärschlüssel**
  - Einstiegsstruktur: B*-Baum u.a.
  - Verknüpfungsstruktur: Zeigerlisten, Bitlisten
  - Viele Komprimierungsverfahren verfügbar

  ➥ **Unterstützung mengentheoretischer Operationen**

- **Bitlisten- und DeweyID-Komprimierung**
  - Unterstützung variabel langer Schlüssel und Einträge erforderlich
  - Bitlisten sind bei geringer Kardinalität des Wertebereichs hoch effizient
  - Huffman-Codes erlauben eine flexible Anpassung an Wertverteilungen

- **Hierarchische Zugriffspfade**
  - Unterstützung von Verbund-Operationen (Relationenmodell)
  - effiziente Abwicklung von Set-Operationen (Netzwerk-Modell)
  - Verknüpfungsstruktur: Ketten, Zeigerlisten, Listen
  - vielfältige Abstimmungsmöglichkeiten auf spezielle Arbeitslasten

- **Verallgemeinerte Zugriffspfadstruktur**
  - Unterstützung von Primärschlüssel-, Sekundärschlüssel- und hierarchischen Zugriffen
  - auch als spezieller Verbund-Index einsetzbar

- **Verbund- und Pfadindexe**
  - explizite Konstruktion von Verbundergebnissen und ihre Indexierung
  - Pfadindexe erlauben nur die Optimierung spezieller Anfragen