

6. Indexstrukturen

- Zugriffsarten und Suchanfragen
- Indexstrukturen für Primärschlüsselzugriff
 - Mehrwegbäume: B-Baum, B*-Baum, Clustered / Non-clustered Index
 - Reduzierung der Baumhöhe (Schlüsselkomprimierung, erweitertes Splitting)
 - Hash-Verfahren (Statische Verfahren, Erweiterbares Hashing)
- Sekundärschlüsselzugriff
 - Invertierung
 - Bitlisten-Indizes
 - Verallgemeinerte Zugriffspfadstruktur (Join-Index)
- Mehrdimensionale Zugriffspfade
 - Verwendung mehrerer 1-dimensionaler Zugriffspfade
 - Grid-File
 - R-Baum
- Textsuche: Invertierte Listen, Signatur-Dateien
- Indexierung von XML-Daten



Zugriffsarten

- Sequentieller Zugriff
 - auf alle Sätze eines Satztyps (Scan)
 - in Sortierreihenfolge eines Attributes
- Direkter Zugriff
 - über den Primärschlüssel
 - über einen Sekundärschlüssel (Nicht-Primärschlüssel)
 - über zusammengesetzte Schlüssel (Attribute)
 - über mehrere unabhängige Schlüssel (Attribute)
- Navigierender Zugriff von einem Satz zu dazugehörigen Sätzen desselben oder eines anderen Satztyps
- Wichtige Anfragearten
 - exakte Anfrage (exact match queries)
 - Präfix-Match-Anfragen
 - Bereichsanfrage (range query)
 - Extremwertanfragen
 - Join-Anfragen
 -



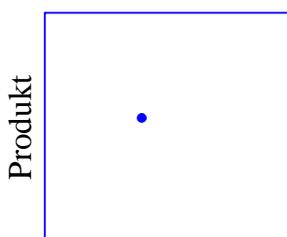
Mehrdimensionale Zugriffe

- besonders wichtig zur effizienten Verwaltung räumlicher Objekte (Punkte, Polygone, Quader, ...)
 - Kartographie, CAD, Computer-Vision und Robotics
- Annahmen:
 - Satztyp/Datei $R = (A_1, \dots, A_n)$
 - Jeder Satz $t = (a_1, a_2, \dots, a_n)$ ist ein Punktobjekt
 - Attribute A_1, \dots, A_k ($k < n$) seien Schlüssel \Rightarrow k -dimensionaler Datenraum D
 - Anfrage Q spezifiziert Bedingungen, die von den Schlüsselwerten der Sätze in der Treffermenge erfüllt sein müssen
- Generelle Anfrageklassen:
 - Intersection Queries: gesuchte Sätze bilden einen Durchschnitt mit der Menge der vorhandenen Sätze (exakte und partielle Anfragen, Bereichsanfragen)
 - Enthaltenseins- oder Überlappungsanfragen (containment queries): gesuchte räumliche Objekte sind ganz oder überlappend in einem Suchfenster enthalten
 - Punktanfrage (point query): Gegeben ist ein Punkt im Datenraum D ; finde alle Objekte, die ihn enthalten.
 - Gebietsanfrage (region query): Gegeben ist Anfragegebiet; finde alle Objekte, die es schneiden.
 - Nächster-Nachbar-Anfragen (best match query, nearest neighbor query): Suche nach "nächstgelegenen" Objekten



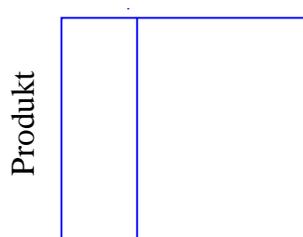
Schnittbildende Suchfragen

1. Exakte Anfrage (exact match query): spezifiziert für jeden Schlüssel einen Wert
 $Q = (A_1 = a_1) \wedge (A_2 = a_2) \wedge \dots \wedge (A_k = a_k)$
 2. Partielle Anfrage (partial match query): spezifiziert $s < k$ Schlüsselwerte
 $Q = (A_{i_1} = a_{i_1}) \wedge (A_{i_2} = a_{i_2}) \wedge \dots \wedge (A_{i_s} = a_{i_s})$ mit $1 < s < k$ und $1 < i_1 < i_2 < \dots < i_s < k$
 3. (Exakte) Bereichsanfrage (range query): spezifiziert einen Bereich $r_i = [l_i < a_i < u_i]$ für jeden Schlüssel A_i
 $Q = (A_1 = r_1) \wedge \dots \wedge (A_k = r_k) \equiv (A_1 > l_1) \wedge (A_1 < u_1) \wedge \dots \wedge (A_k > l_k) \wedge (A_k < u_k)$
 4. Partielle Bereichsanfrage (partial range query): spezifiziert für $s < k$ Schlüssel einen Bereich
 $Q = (A_{i_1} = r_{i_1}) \wedge \dots \wedge (A_{i_s} = r_{i_s})$ mit $1 < s < k$ und $1 < i_1 < \dots < i_s < k$ und $r_{i_j} = [l_{i_j} < a_{i_j} < u_{i_j}]$, $1 < j < s$
- \rightarrow bei den schnittbildenden Anfragen lassen sich alle 4 Fragetypen als allgem. Bereichsanfrage ausdrücken
- genauer Bereich $[l_i = a_i = u_i]$
 - vs. unendlicher Bereich $[-\infty < a_i < \infty]$



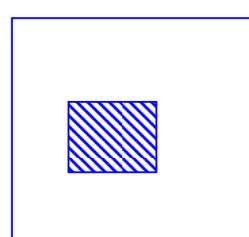
Zeit (Tage)

Punktanfrage
(exact match)



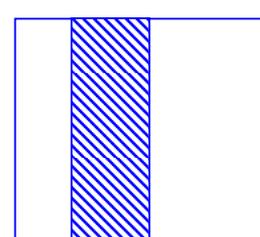
Zeit (Tage)

Partielle Punkt-anfrage
(partial match)



Zeit (Tage)

Bereichsanfrage



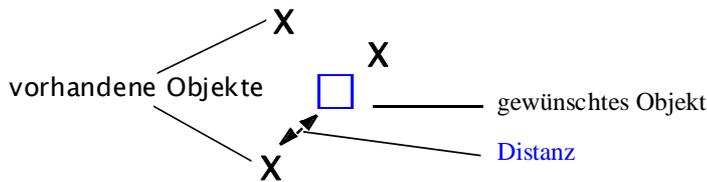
Zeit (Tage)

Partielle Bereichs-anfrage
(partial range)



Best-Match (Nearest Neighbor)-Anfragen

- gewünschtes Objekt nicht vorhanden
=> Frage nach möglichst ähnlichen Objekten



– "best" wird bestimmt über verschiedene Arten von Distanzfunktionen

- Beispiele:
 - räumliche Distanz
 - Objekt erfüllt nur 8 von 10 geforderten Eigenschaften
 - Objekt ist durch Synonyme beschrieben

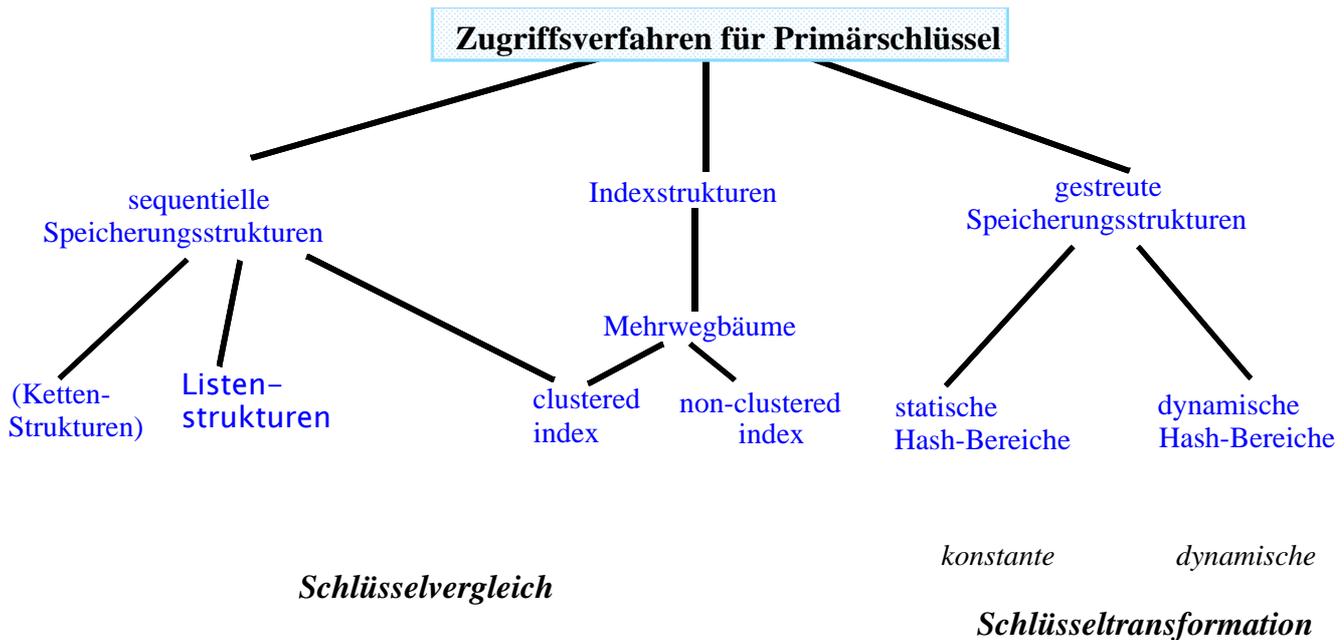
Nearest Neighbor:

D = Distanzfunktion

B = Sammlung von Punkten im k-dim. Raum

Gesucht: nächster Nachbar von p (in B)

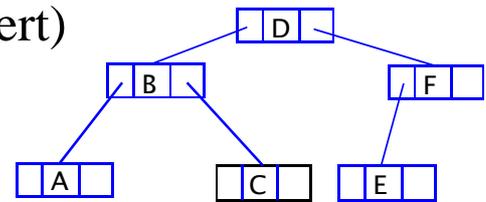
Der nächste Nachbar ist q, wenn $(\forall r \in B) \{r \neq q \Rightarrow [D(r,p) > D(q,p)]\}$



Mehrweg-Bäume

■ Ausgangspunkt: Binäre Bäume (balanciert)

- entwickelt für Hauptspeicher
- für DB-Anwendungen unzureichend



■ Weiterentwicklung: B- und B*-Baum

- Seiten (Blöcke) als Baumknoten
- Zusammenfassung mehrerer Sätze/Schlüssel in einem Knoten
- höherer Verzweigungsgrad (Mehrwegbaum), wesentlich niedrigere Baumhöhe

■ Balancierte Struktur

- unabhängig von Schlüsselmenge, unabhängig von Einfügereihenfolge
- dynamische Reorganisation durch Splitten und Mischen von Seiten

■ Speicherung von Datensätzen: direkt im Mehrweg-Baum eingebettet oder separat

- *direkter Index* vergrößert Baumhöhe, erspart jedoch separaten Satzzugriff über Verweis (z.B. TID)
- direkte Speicherung nur für einen Index pro Satztyp möglich (i.a. für Primärschlüssel)
- *indirekter Index* enthält Verweis (TID) pro Datensatz
- Aggregatfunktionen u.ä. können ohne Zugriff auf Datenseiten berechnet werden

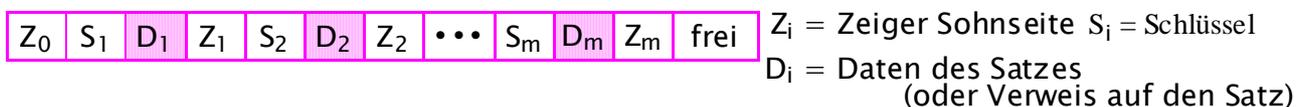


B-Baum

■ Ein B-Baum vom Typ (k, h) ist ein Baum mit folgenden Eigenschaften

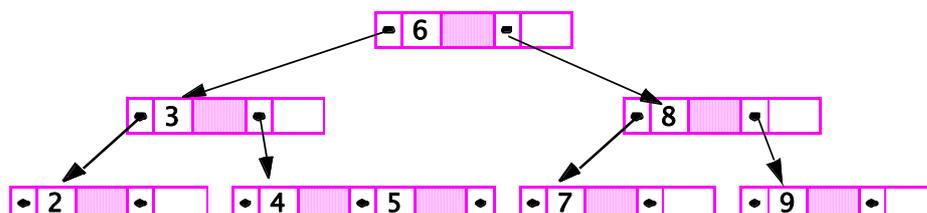
1. Jeder Weg von der Wurzel zum Blatt hat die Länge h
2. Jeder Zwischenknoten hat mindestens k+1 Söhne. Die Wurzel ist ein Blatt oder hat mindestens 2 Söhne
3. Jeder Knoten hat höchstens 2k+1 Söhne

■ Seitenformat:



für alle Schlüssel s im Teilbaum zu Z_i gilt: $S_i < s < S_{i+1}$ (i=1, ..., m-1)
 bzgl. Z_0 gilt: $s < S_1$ und bzgl. Z_m gilt: $S_m < s$

■ Beispiel



■ Verzweigungsgrad (Seitengröße 8 KB, Z = 4 B, S = 4 B):



B*-Baum

- Ein B*-Baum vom Typ (k, k^*, h) ist ein Baum mit folgenden Eigenschaften
 - Jeder Weg von der Wurzel zum Blatt hat die Länge h
 - Jeder Zwischenknoten hat mindestens $k+1$ Söhne. Die Wurzel ist ein Blatt oder hat mindestens 2 Söhne. Jedes Blatt hat mindestens k^* Einträge.
 - Jeder Zwischenknoten hat höchstens $2k+1$ Söhne. Jedes Blatt hat höchstens $2k^*$ Einträge

Seitenformate:

Zwischenknoten:

Z_0	S_1	Z_1	S_2	Z_2	\dots	S_m	Z_m	frei
-------	-------	-------	-------	-------	---------	-------	-------	------

Z_i = Zeiger Sohnseite
 S_i = Schlüssel

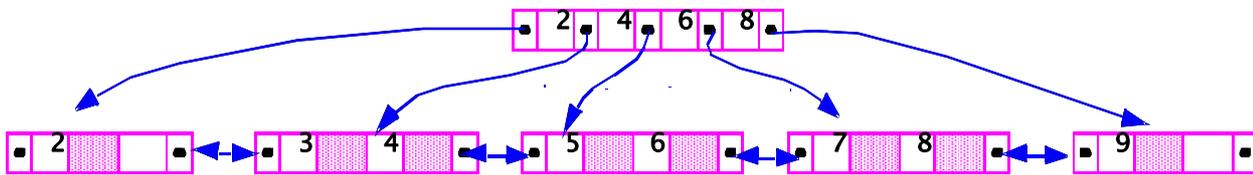
Blattknoten:

V	S_1	D_1	S_2	D_2	\dots	S_m	D_m	frei	N
---	-------	-------	-------	-------	---------	-------	-------	------	---

D_i = Daten bzw. Verweis
 V = Vorgänger-Zeiger
 N = Nachfolger-Zeiger

für alle Schlüssel s im Teilbaum zu Z_i gilt: $S_i < s \leq S_{i+1}$ ($i=1, \dots, m-1$)
 bzgl. Z_0 gilt: $s \leq S_1$ und bzgl. Z_m gilt: $S_m < s$

Beispiel:



B*-Baum (2)

- Verzweigungsgrad (Seitengröße 8 KB, $Z = 4$ B, $S = 4$ B):

- Höhe h eines B*-Baums mit N Datenelementen:

$$1 + \log_{2k+1} \left(\frac{N}{2k^*} \right) \leq h \leq 2 + \log_{k+1} \left(\frac{N}{2k^*} \right) \quad (h \geq 2)$$

- Typischer Wert für die Höhe eines B*-Baums:
 Höhe 3 – 4 bei $10^5 - 10^7$ Datensätzen

Performance-Optimierung

- Niedrige Höhe / hoher Verzweigungsgrad
- möglichst kurze Schlüsselwerte / große Seiten / hohe Seitenbelegung



B*-Baum (3)

■ Unterschiede gegenüber B-Baum

- innere Knoten enthalten nur Verzweigungsinformation
- geringere Baumhöhe möglich
- Schlüsselwerte teilweise redundant gespeichert
- für jeden Satz müssen h Seiten gelesen werden
- Kette der Blattknoten liefert alle Sätze nach Schlüsselwert sortiert
- bessere Clustereigenschaften für B*-Baum

■ Suche im B*-Baum

- beginnend im Wurzelknoten; jeder Knoten wird von links nach rechts durchsucht
- Ist der gesuchte Wert kleiner oder gleich S_i , wird die Suche in der Wurzel des von Z_{i-1} identifizierten Teilbaums fortgesetzt
- Ist der gesuchte Wert größer als S_i , wird der Vergleich mit S_{i+1} fortgesetzt. Ist auch der letzte Schlüsselwert S_m im Knoten noch kleiner als der gesuchte Wert, wird die Suche im Teilbaum von Z_m fortgesetzt
- der derart ermittelte Blattknoten wird von links nach rechts nach dem Schlüsselwert durchsucht; wird er nicht gefunden, war die Suche erfolglos (kein Treffer)

■ unterstützte Funktionen:

- direkter Schlüsselzugriff
- sortiert sequentieller Zugriff
- Unterstützung für exakte Anfragen, Präfix-Match-Anfragen, Bereichsanfragen, Extremwertanfragen ...

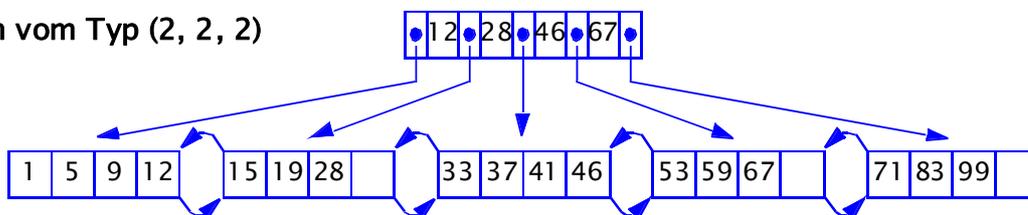


Einfügen im B*-Baum

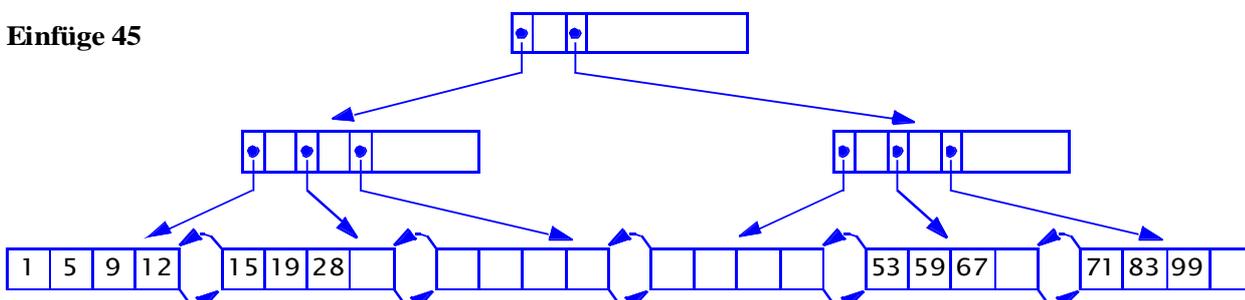
■ Vorgehensweise

- zunächst Abstieg durch den Baum wie bei der Suche
- Einfügen des Satzes stets im Blattknoten
- falls Blattknoten bereits voll, muss ein neuer Knoten erzeugt und ein gleichmäßiges Aufteilen der $(2k^*+1)$ Sätze auf zwei Seiten vorgenommen werden (Splitting)
- Splitting erfordert Anpassung der Verzweigungsinformation in den Vorgängerknoten

B*-Baum vom Typ (2, 2, 2)



Einfüge 45



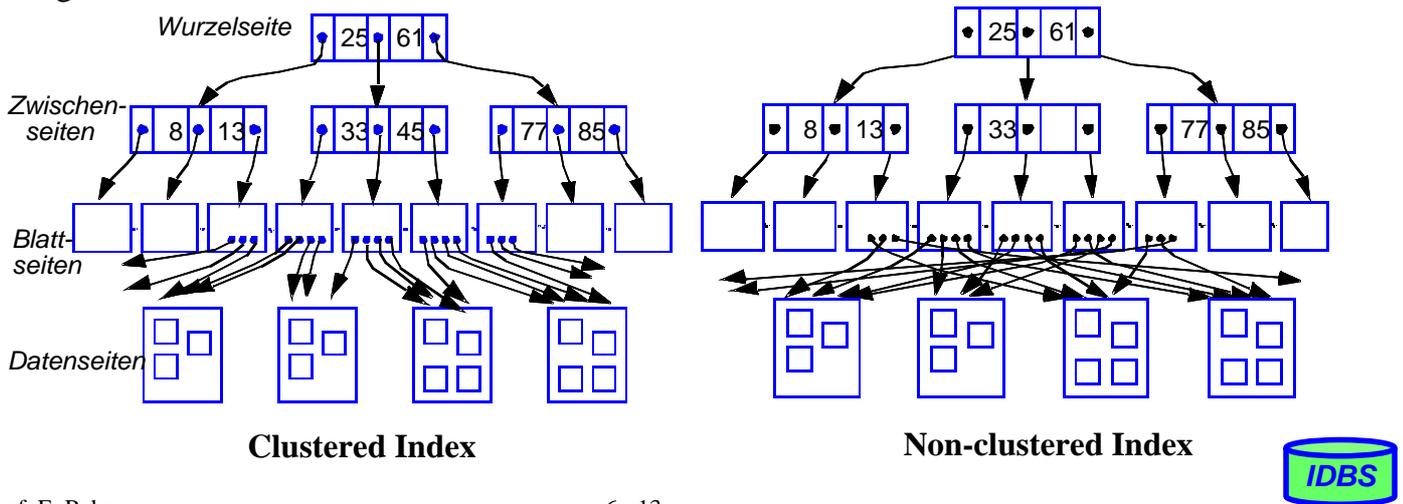
B*-Baum mit/ohne Clusterung der Sätze

■ Index mit Clusterbildung (clustered index)

- Clusterbildung der Sätze in den Datenseiten mit Sortierung nach Indexattribut oder direkte Speicherung der Sätze in Indexblättern
- sehr effiziente Bearbeitung von Präfix-Match- und Bereichsanfragen sowie sortierter Ausgabe
- maximal 1 Clustered Index pro Relation

■ Non-clustered Index

- impliziert indirekten Index
- gut v.a. für selektive Anfragen und Berechnung von Aggregatfunktionen sowie bei relativ großen Sätzen



Reduzierung der Baumhöhe

■ geringere Baumhöhe durch Vergrößerung der Baumbreite (Fan-Out) -> mehr Einträge pro Seite

- größere Seiten
- höherer Füllgrad der Seiten (-> verallgemeinerte Splitting-Verfahren)
- kleinere Einträge, z.B. durch Beschränkung der Daten/-Verweise auf Blätter (B*-Baum) oder/und durch Schlüsselkomprimierung

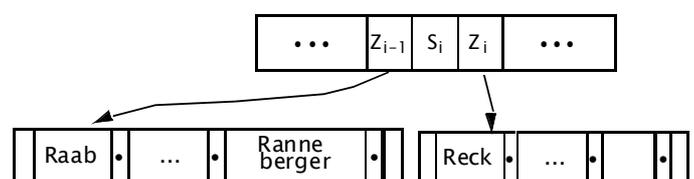
■ Zeichenkomprimierung ermöglicht weit höhere Anzahl von Einträgen pro Seite, v.a. für lange, alphanumerische Schlüssel (z.B. Namen)

■ Präfix-Komprimierung

- mit Vorgängerschlüssel übereinstimmender Schlüsselanfang (Präfix) wird nicht wiederholt
- v.a. wirkungsvoll für Blattseiten in B*-Bäumen
- höherer Aufwand zur Schlüsselrekonstruktion

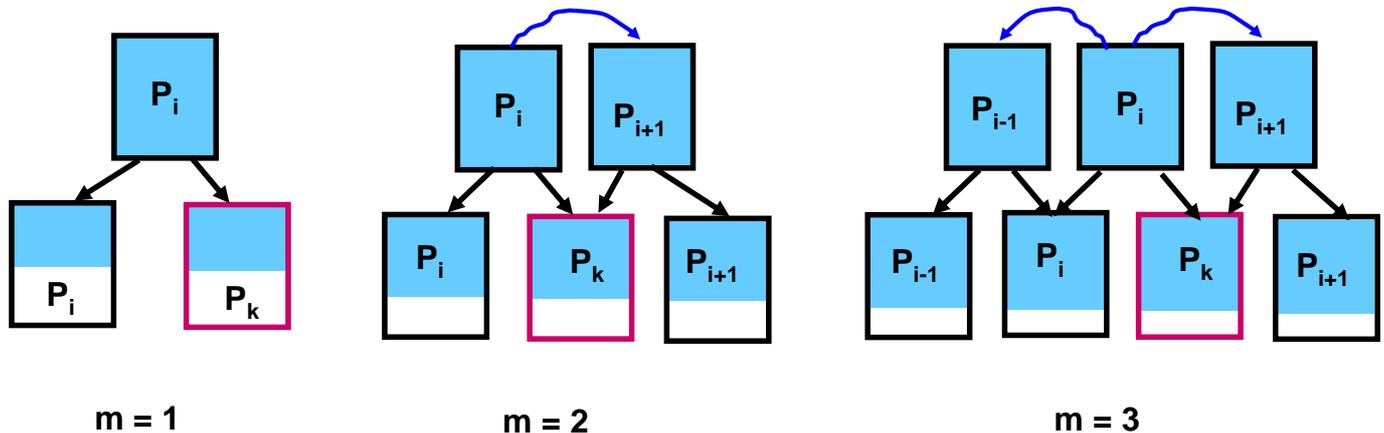
■ Suffix-Komprimierung

- für innere Knoten ist vollständige Wiederholung von Schlüsselwerten für Wegweiserfunktion meist nicht erforderlich
- Weglassen des zur eindeutigen Lokalisierung nicht benötigten Schlüsselendes (Suffix)
- **Präfix-B*-Bäume:** Verwendung minimaler Separatoren (Präfixe) in inneren Knoten



Verallgemeinertes Splitting bei B*-Bäumen

- Splitting erfolgt für m volle Seiten
- Standard $m=1$: Überlauf führt zu zwei halb vollen Seiten:
 $b_{\min}=0,5$ (50%), $b_{\text{avg}}=\ln 2$ (ca. 69%), $b_{\max}=1$ (100%)
- $m>1$ verbessert Speicherbelegung (reduziert Seitenzahl / Baumhöhe)
 $b_{\min}=m/(m+1)$, $b_{\text{avg}}=\ln((m+1)/m)$, $b_{\max}=1$



Hash-Verfahren

- direkte Berechnung der Speicheradresse eines Satzes über Schlüssel (Schlüsseltransformation)
- Hash-Funktion $h: S \rightarrow \{1, 2, \dots, n\}$
 $S = \text{Schlüsselraum}$
 $n = \text{Größe des statischen Hash-Bereiches in Seiten (Buckets)}$
- Vielzahl von Hash-Funktionen nutzbar, z.B. Divisionsrestverfahren
 - alphanumerische Schlüssel können zunächst auf numerischen Wert abgebildet werden, z.B. durch "Faltung"
- Idealfall: jeder Satz wird in zugeordneter Seite gefunden: Zugriffsfaktor 1
 - aber: injektive Hash-Funktion meist nicht möglich \rightarrow Kollisionsbehandlung
- Statische Hash-Bereiche mit Kollisionsbehandlung
 - vorhandene Schlüsselmenge K ($K \subseteq S$) soll möglichst gleichmäßig auf die n Buckets verteilt werden
 - Behandlung von Synonymen: Aufnahme im selben Bucket, wenn möglich; ggf. Anlegen und Verketteten von Überlaufseiten
 - typischer Zugriffsfaktor: 1.1 bis 1.4

Statisches Hash-Verfahren mit Überlaufbereichen: Beispiel

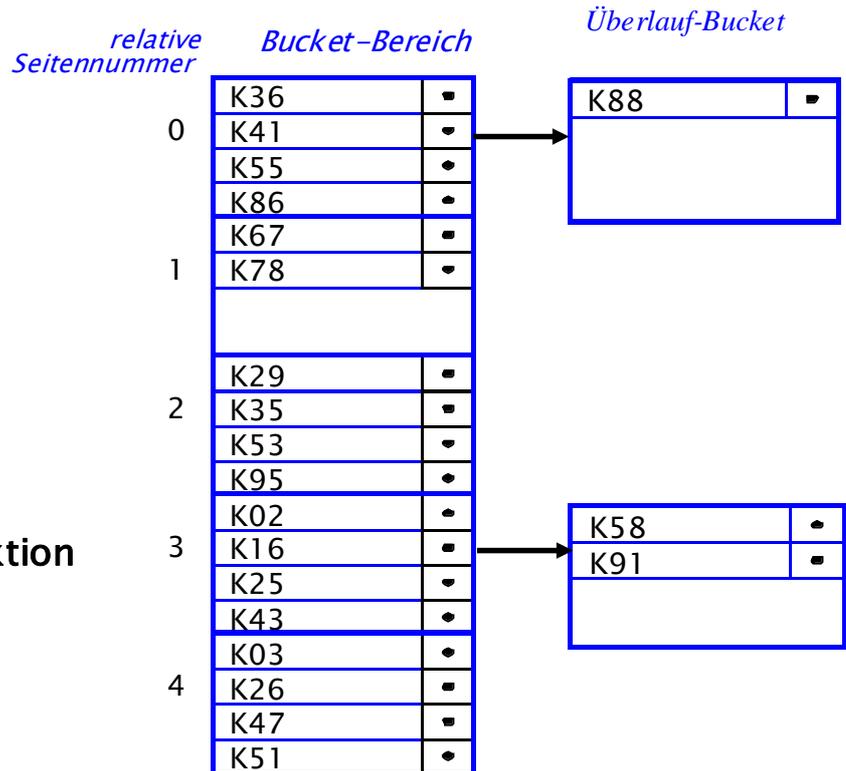
Adreßberechnung
für Schlüssel K02

1. Faltung

$$\begin{array}{r}
 1101\ 0010 \\
 \oplus 1111\ 0000 \\
 \oplus 1111\ 0010 \\
 \hline
 1101\ 0000 = 208_{10}
 \end{array}$$

2. Anwendung der Hash-Funktion

$$208 \bmod 5 = 3$$

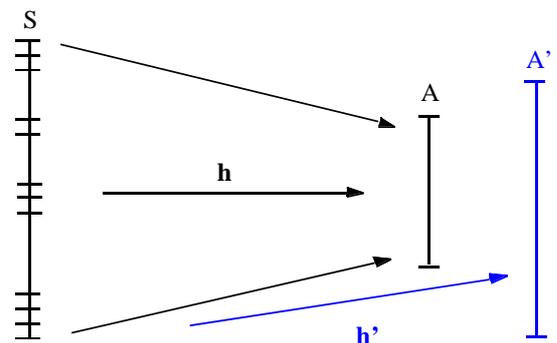


Dynamische Hash-Verfahren

■ Wachstumsproblem bei statischen Verfahren

- Statische Allokation von Speicherbereichen: Speicherausnutzung?
- Bei Erweiterung des Adressraumes: Re-Hashing
- Alle Sätze erhalten eine **neue Adresse**

sehr hoher Reorganisationsaufwand!



■ Entwurfsziele für bessere Hash-Verfahren

- dynamische Struktur, die Wachstum und Schrumpfung des Hash-Bereichs erlaubt
- keine Überlauftechniken
- Zugriffsfaktor ≤ 2 für die direkte Suche

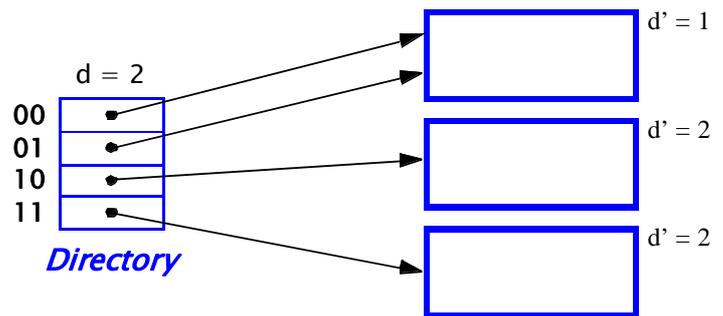
■ Viele konkurrierende Ansätze

- Extendible Hashing (Fagin et al., 1978)
- Virtual Hashing und Linear Hashing (Litwin, 1978, 1980)
- Dynamic Hashing (Larson, 1978)



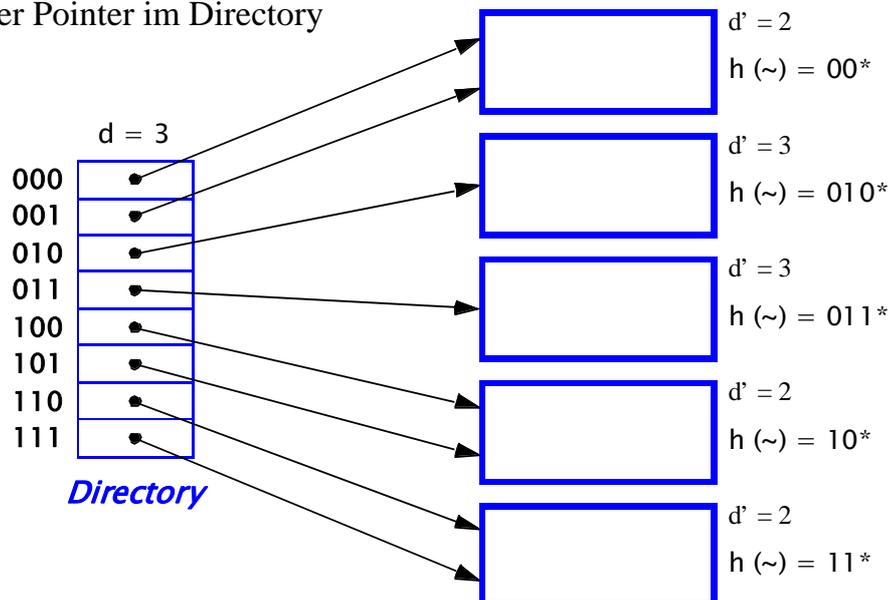
Erweiterbares Hash-Verfahren (Extendible Hashing)

- dynamisches Wachsen und Schrumpfen des Hash-Bereiches
 - Buckets werden erst bei Bedarf bereitgestellt
 - hohe Speicherplatzbelegung möglich
- keine Überlauf-Bereiche, jedoch Zugriff über Directory
 - max. 2 Seitenzugriffe
 - Hash-Funktion generiert Pseudoschlüssel zu einem Satz
 - d Bits des Pseudoschlüssels werden zur Adressierung verwendet ($d =$ globale Tiefe)
 - Directory enthält 2^d Einträge; Eintrag verweist auf Bucket, in dem alle zugehörigen Sätze gespeichert sind
 - in einem Bucket werden nur Sätze gespeichert, deren Pseudoschlüssel in den ersten d' Bits übereinstimmen ($d' =$ lokale Tiefe)
 - $d = \text{MAX}(d')$



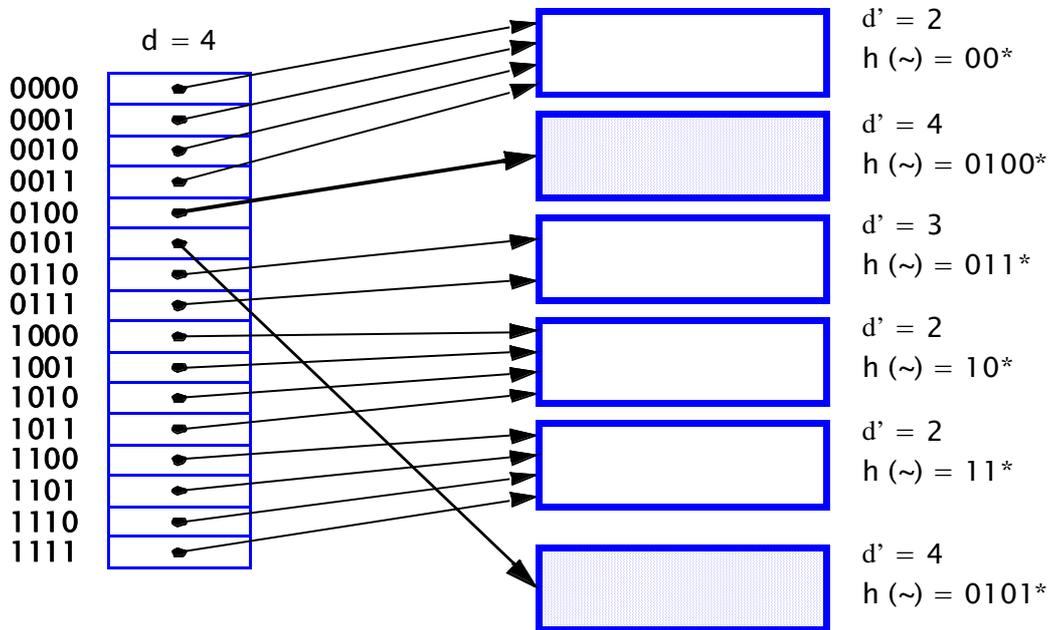
Erweiterbares Hashing: Splitting von Buckets

- Fall 1: Überlauf für Bucket mit lokaler Tiefe kleiner der globalen Tiefe d
 - lokale Neuverteilung der Daten
 - Erhöhung der lokalen Tiefe
 - lokale Korrektur der Pointer im Directory



Erweiterbares Hashing: Splitting von Buckets (2)

- Fall 2: Überlauf eines Buckets, dessen lokale Tiefe gleich der globalen Tiefe ist
 - lokale Neuverteilung der Daten (Erhöhung der lokalen Tiefe)
 - Verdopplung des Directories (Erhöhung der globalen Tiefe)
 - globale Korrektur/Neuverteilung der Pointer im Directory



Grobvergleich der Zugriffskosten (#Seiten)

Beispielangaben für $N = 10^6$

Speicherungs-/Indexstruktur	Direkter Zugriff	Sortiert-sequentielle Verarbeitung
gekettete Liste	$O(N) \approx 5 \cdot 10^5$	$O(N) \approx 10^6$
sequentielle Liste/Clusterung	$O(N) \approx 10^4$	$O(N) \approx c \cdot 10^4$
Mehrwegbäume (B*-Bäume)	$O(\log_k N) \approx 3 - 5$	$O(N) \approx 10^6$
Clustered index	$O(\log_k N) \approx 3 - 5$	$O(N) \approx c \cdot 10^4$
Statisches Hashing mit Überlaufbereich	$O(1) (\approx 1.1 - 1.4)$	$O(N \log_2 N) (*)$
Erweiterbares Hashing	$O(1) = 2$	$O(N \log_2 N) (*)$

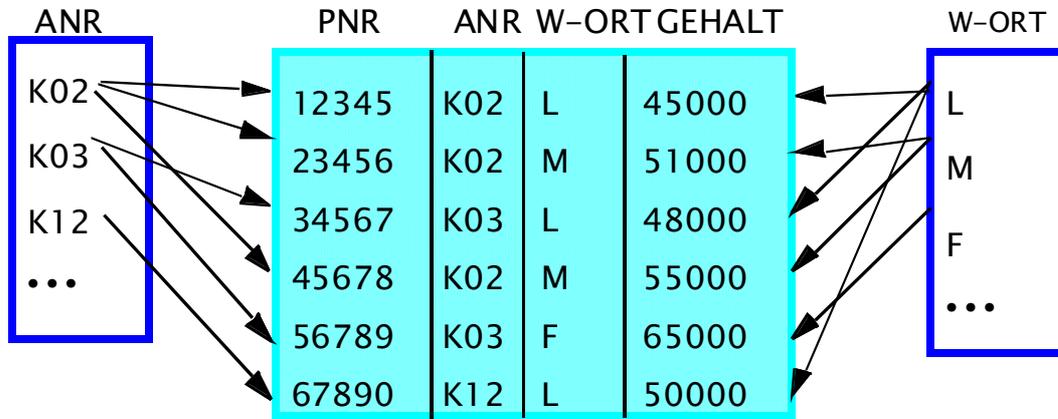
c vergleichsweise kleine Konstante

(*) Physisch sequentielles Lesen, Sortieren und sequentielles Verarbeiten der gesamten Sätze



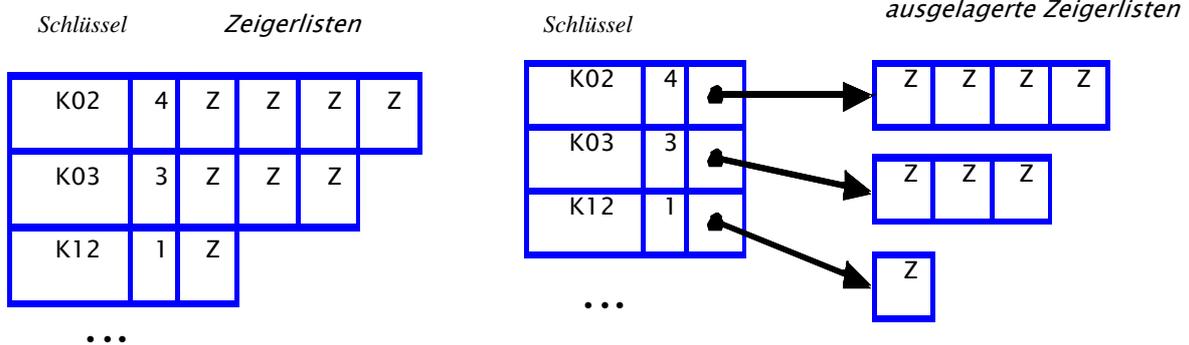
Zugriffspfade für Sekundärschlüssel

- Suche nach Tupeln mit einem vorgegebenen Wert in einem nichteindeutigen Attribut (*Sekundärschlüssel*)
- i.a. mehr als 1 Treffer (Satzmengen)



Invertierungstechniken

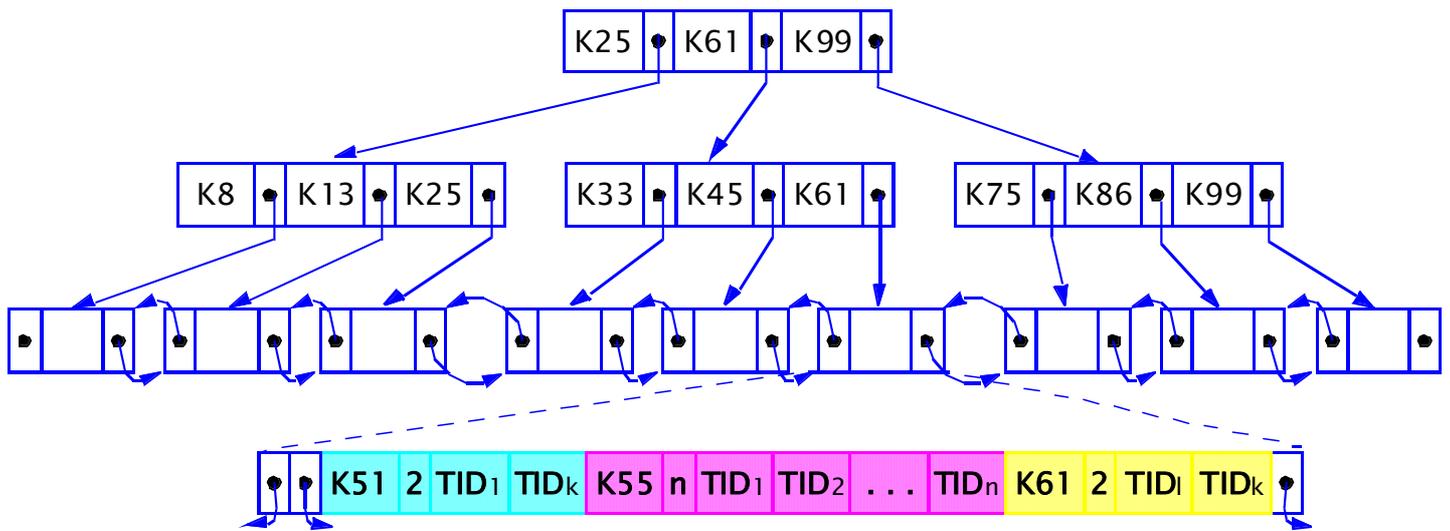
- Invertierung: Indexstruktur mit Zeigerlisten bzw. Bitvektoren pro Attributwert
- effiziente Berechnung mengenalgebraischer Operationen



- Speicherplatzbedarf für Zeigerlisten (TID-Listen) vs. Bitlisten
 - N Sätze; k unterschiedliche Attributwerte



B*-Baum-Nutzung für Sekundärschlüssel



Bitlisten-Indizes

- herkömmliche Indexstrukturen ungeeignet für Suchbedingungen geringer Selektivität
 - z.B. für Attribute (Dimensionen) mit nur wenigen Werten (Geschlecht, Farbe, Jahr ...)
 - pro Attributwert sehr lange Verweislisten (TID-Listen) auf zugehörige Sätze
 - nahezu alle Datenseiten zu lesen
- Standard-Bitlisten-Index (**Bitmap Index**):
 - Index für Attribut A umfasst eine Bitliste (Bitmap, Bitvektor) B_{A_j} für jeden der k Attributwerte $A_1 \dots A_k$
 - Bitliste umfasst 1 Bit pro Satz (bei N Sätzen Länge von N Bits)
 - Bitwert 1 (0) an Stelle i von B_{A_j} gibt an, dass Satz i Attributwert A_j aufweist (nicht aufweist)

Kunde

KID	Geschlecht	Lieblingsfarbe
122	W	Rot
123	M	Rot
124	W	Weiß
125	W	Blau
...

Blau 00011000100011000011000000001001000
 Rot 110000000001001000001000000110000001
 Weiß 001000000110000001000001110000000110
 Grün 000001110000000110000110001000110000 ...



Bitlisten Indizes (2)

■ Vorteile

- geringer Speicherplatzbedarf bei kleinen Wertebereichen
- effiziente AND-, OR-, NOT-Verknüpfung zur Auswertung mehrdimensionaler Suchausdrücke
- effiziente Unterstützung von Data-Warehouse-Anfragen (Joins)

■ Beispielanfrage

Select ... WHERE A1 = C1 AND A2=C2 AND A3=C3 AND A4=C4
10 Millionen Sätze; pro Teilbedingung Selektivität 5%



Kodierte Bitlisten-Indizes

■ Speicherplatzersparnis durch kodierte Bit-Listen (encoded bitmap indexing)

- Standardverfahren: pro Satz ist nur in einer der k Bitlisten das Bit gesetzt
- jede der k Wertemöglichkeiten kann durch $\log_2 k$ Bits kodiert werden
=> nur noch $\log_2 k$ Bitlisten

	Kodierung	F ₁	F ₀	2 Bitvektoren
Blau	0001100010001 ...	Blau		
Rot	1100000000010 ...	Rot		F ₁
Weiß	0010000001100 ...	Weiß		F ₀
Grün	0000011100000 ...	Grün		

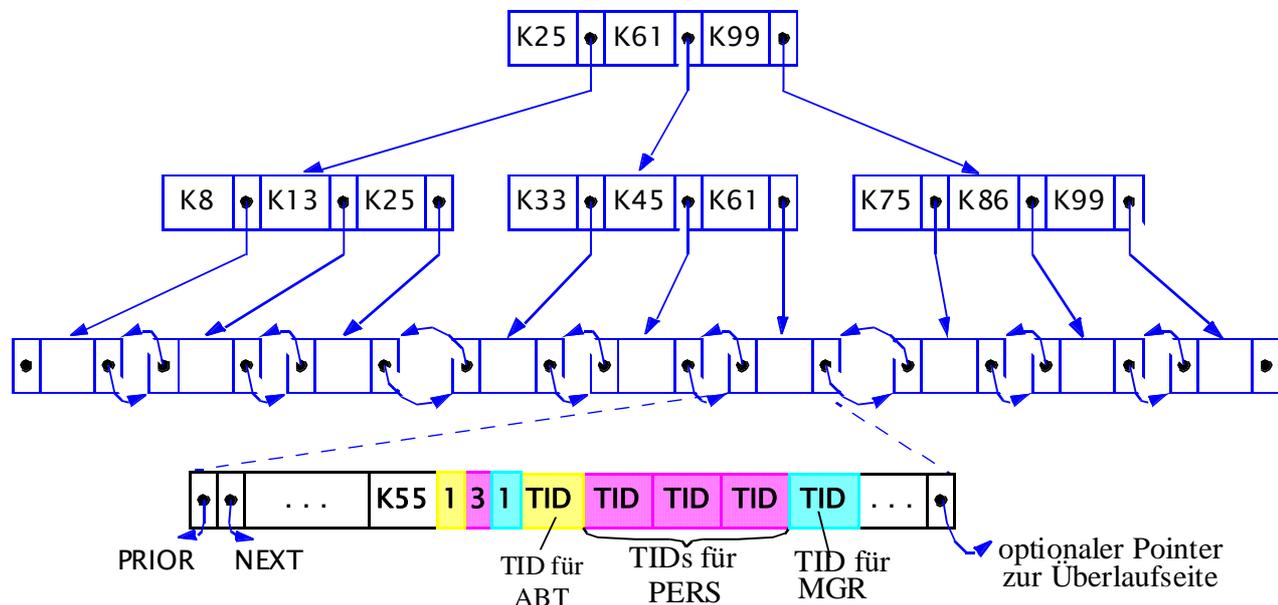
■ Auswertung von Suchausdrücken

- höherer Aufwand bei nur 1 Bedingung ($\log_2 k$ Bitlisten statt 1 abuarbeiten)
- bei mehreren Bedingungen wird auch Auswertungsaufwand meist reduziert



Verallgemeinerte Zugriffspfadstruktur (Join-Index)

- B*-Baum kann als gemeinsamer Index für Primär- und Fremdschlüssel verwendet werden
- Unterstützung von Joins bzw. hierarchischen Zugriffen zwischen Relationen
- Beispiel: ANR-Index für ABT und PERS und MGR



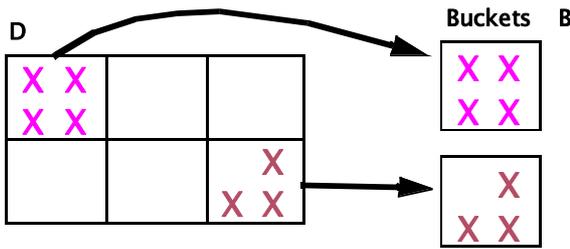
Verallgemeinerte Zugriffspfadstruktur: Bewertung

- Prinzip
 - kombinierte Realisierung von Primär-, Sekundär- und hierarchischen Zugriffspfaden
 - gemeinsame Verwendung von B*-Baum für mehrere Satztypen, für die Beziehungen (1:1, 1:n, n:1, n:m) durch Wertegleichheit in demselben Attribut (z.B. ANR) repräsentiert sind
 - innere Baumknoten bleiben unverändert; Blätter enthalten Verweise für primäre und sekundäre Zugriffspfade
- Erhöhung der Anzahl der Blattseiten
 - mehr Seitenzugriffe beim sequentiellen Lesen einer Relation in Sortierordnung
 - aber: Höhe des Baumes bleibt meist erhalten: ähnliches Leistungsverhalten beim Aufsuchen von Daten und beim Änderungsdienst
- Vorteile
 - einheitliche Struktur für alle Zugriffspfadtypen: Vereinfachung der Implementierung
 - Schlüssel werden nur einmal gespeichert: Speicherplatzersparnis
 - Unterstützung der Join-Operation sowie bestimmter statistischer Anfragen
 - einfache Überprüfung der referentiellen Integrität sowie weiterer Integritätsbedingungen (z.B. Kardinalitätsrestriktionen)

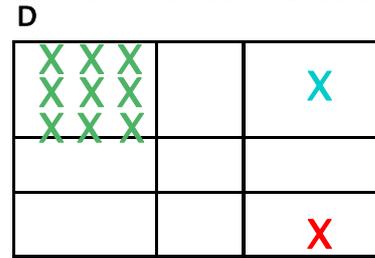


Grundprobleme räumlicher (mehrdim.) Zugriffe

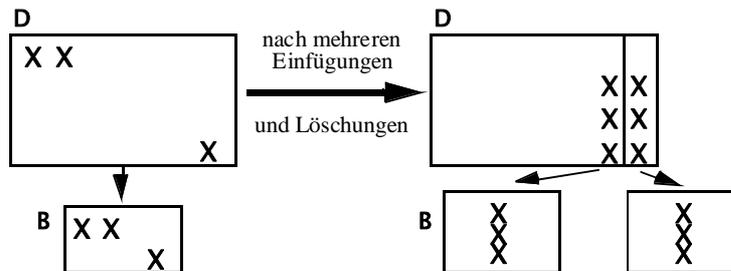
1. Erhaltung der topologischen Struktur



2. Stark variierende Dichte der Objekte



3. Dynamische Reorganisation



4. Objektdarstellung: Punktoobjekte sowie Objekte mit Ausdehnung

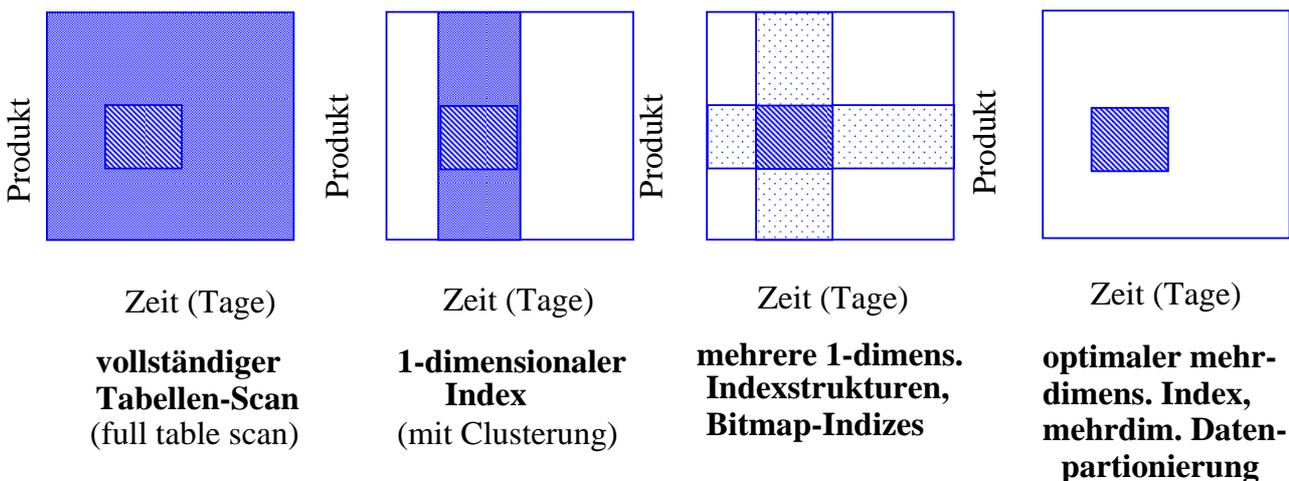
5. Balancierte Zugriffsstruktur

- Beliebige Belegungen und Einfüge-/Löschreihenfolgen
- Garantie eines gleichförmigen Zugriffs: 2 oder 3 Externspeicherzugriffe



Indexunterstützung für mehrdimens. Anfragen

■ Eingrenzung des Datenraumes



■ Verwendung eindimensionaler Indexstrukturen

- Mischen mehrerer Trefferlisten für Anfragen wie $(\text{Key1} = \text{K1i}) \left\{ \begin{array}{c} \text{OR} \\ - \\ \text{AND} \end{array} \right\} (\text{Key2} = \text{K2j})$
- Clustering höchstens bezüglich eines Attributs (eine Dimension)
- Topologie wird nicht erhalten
- Index auf konkatenierten Attributen unterstützt nur sehr eingeschränkt Mehrattributsuche



Grid-File (Gitterdatei)

■ Organisation des Datenraumes D

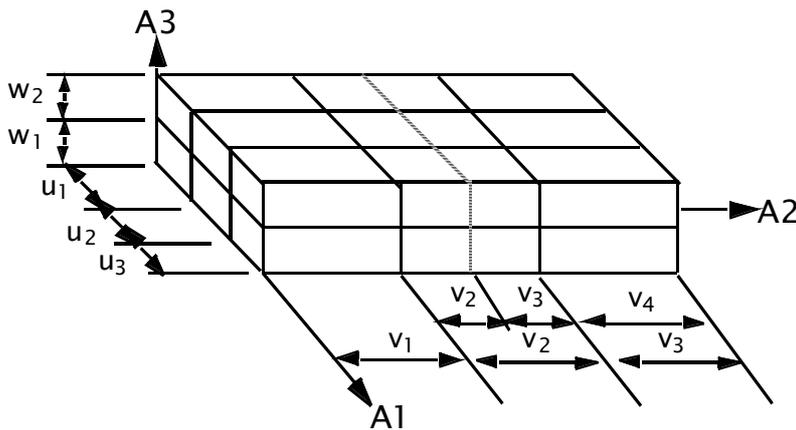
- D wird durch ein **orthogonales Raster** (grid) partitioniert, so dass k-dimensionale Zellen (Grid-Blöcke) entstehen
- die in den Zellen enthaltenen Objekte werden Buckets zugeordnet
- es muss eine eindeutige Abbildung der Zellen zu den Buckets gefunden werden

■ dynamische Anpassung über Dimensionsverfeinerung

- ein Abschnitt in der ausgewählten Dimension wird durch einen vollständigen Schnitt durch D verfeinert

■ Ziele: Erhaltung der Topologie

- effiziente Unterstützung aller Fragetypen
- vernünftige Speicherplatzbelegung



3-dim. Datenraum $D = A_1 \times A_2 \times A_3$

Zellpartition $P = U \times V \times W$

Abschnitte der Partition $U = (u_1, u_2, \dots, u_n)$

$V = (v_1, v_2, \dots, v_m)$

$W = (w_1, w_2, \dots, w_n)$

Veranschaulichung eines Split-Vorganges im Intervall v_2

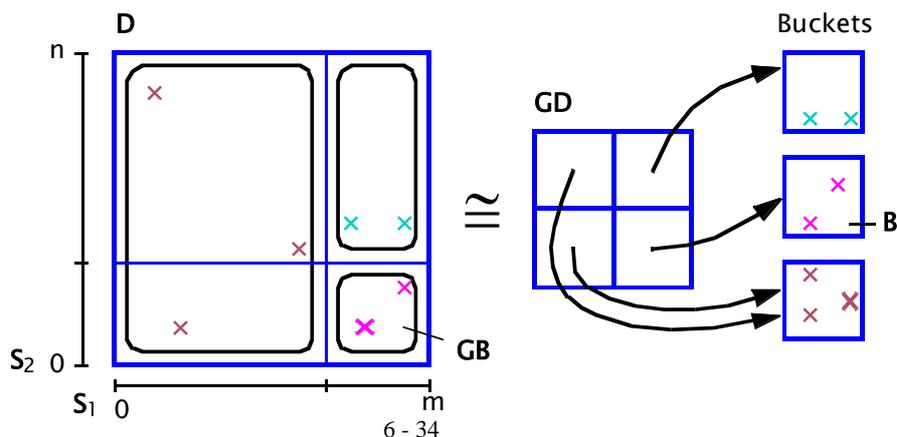
Grid-File (2)

■ Komponenten

- dynamische Aufteilung von D in Gridblöcke (GB)
- k **Skalierungsvektoren** (Scales) definieren Grid auf k-dimensionalen Datenraum D
- **Grid Directory** GD: dynamische k-dim. Matrix zur Abbildung von D auf die Menge der Buckets
- Bucket: Speicherung der Objekte eines oder mehrerer Gridblöcke (Bucketbereich)

■ Eigenschaften

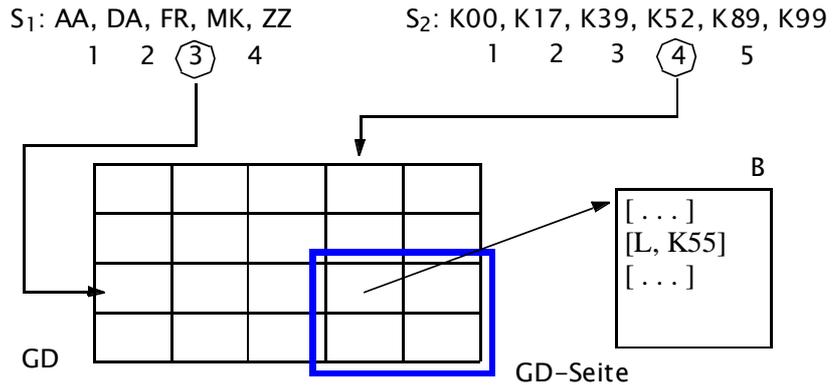
- 1:1-Beziehung zwischen Gridblock GB und Element von GD
- Element von GD = Pointer zu Bucket B
- n:1-Beziehung zwischen GB und B



Grid File - Suchfragen

■ Punktfrage (exact match)

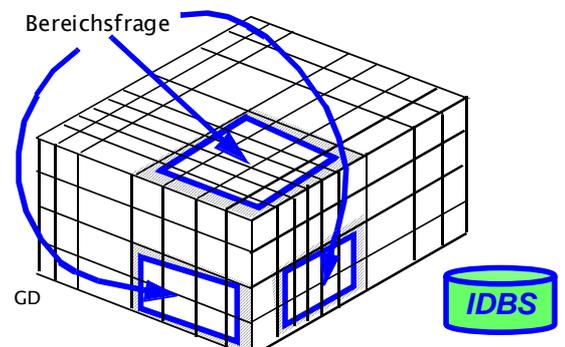
```
SELECT *
FROM PERS
WHERE ORT = 'L'
AND ANR = 'K55'
```



- 2 Plattenzugriffe: unabhängig von Werteverteilungen, #Operationen und #Sätze

■ Bereichsfrage

- Bestimmung der Skalierungswerte in jeder Dimension
- Berechnung der qualifizierten GD-Einträge
- Zugriff auf die GD-Seite(n) und Holen der referenzierten Buckets



R-Baum

■ Zugriffspfad für ausgedehnte räumliche Objekte

- Objektapproximation durch schachtelförmige Umhüllung
- Hauptoperationen: Punkt- sowie Gebietsanfragen

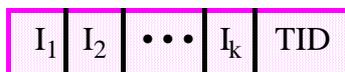
■ Ansatz: Speicherung und Suche von achsenparallelen Rechtecken

- Objekte werden durch Datenrechtecke repräsentiert und müssen durch kartesische Koordinaten beschrieben werden
- Repräsentation im R-Baum erfolgt durch minimale begrenzende (k-dimensionale) Rechtecke/Regionen
- Suchanfragen beziehen sich ebenfalls auf Rechtecke/Regionen

■ R-Baum ist höhenbalancierter Mehrwegebaum

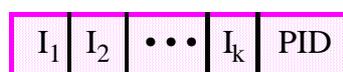
- jeder Knoten entspricht einer Seite
- pro Knoten maximal M, wenigstens m ($\geq M/2$) Einträge

Blattknoteneintrag:



kleinstes umschreibendes Rechteck für TID

Zwischenknoteneintrag:



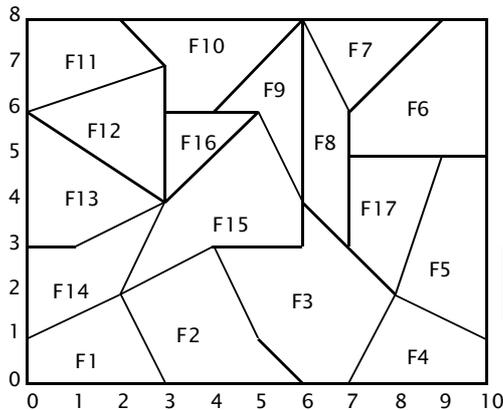
Intervalle beschreiben kleinstes umschreibendes Rechteck für alle in PID enthaltenen Objekte

I_j = geschlossenes Intervall bzgl. Dimension j

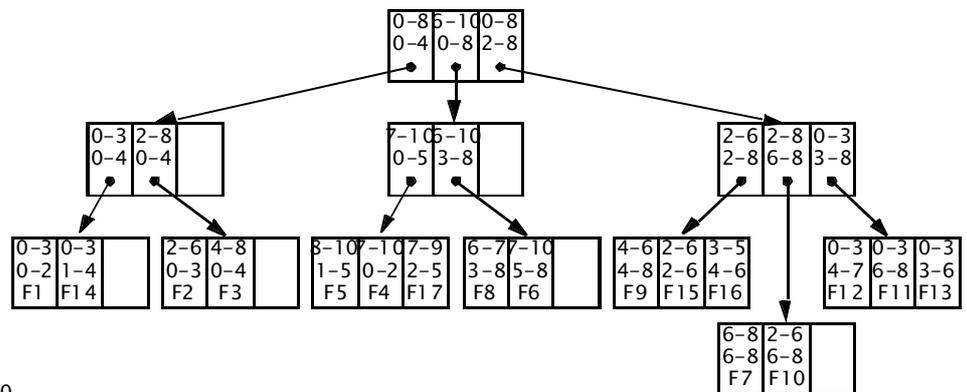
TID: Verweis auf Objekt PID: Verweis auf Sohn

Abbildung beim R-Baum

Flächenobjekt



Zugehöriger R-Baum



■ Eigenschaften

- starke Überlappung der umschreibenden Rechtecke auf allen Baumebenen möglich
- bei Suche nach überlappenden Rechtecken sind ggf. mehrere Teilbäume zu durchlaufen



Textsuche in Dokumentkollektionen

■ DB mit Kollektion von Dokumenten / Texten

```
CREATE TABLE Dokument
( DokID    int primary key,
  URL      varchar (60),
  Autor    varchar (60),
  Titel    varchar (60),
  Jahr     nt,
  Abstract CLOB,
  Volltext CLOB ... )
```

```
SELECT URL
FROM Dokument
WHERE CONTAINS (Volltext,
"(multimedial OR objektorientiert OR
intelligent) AND
(Datenbank OR Informationssystem)" )
AND Jahr > 2005
```

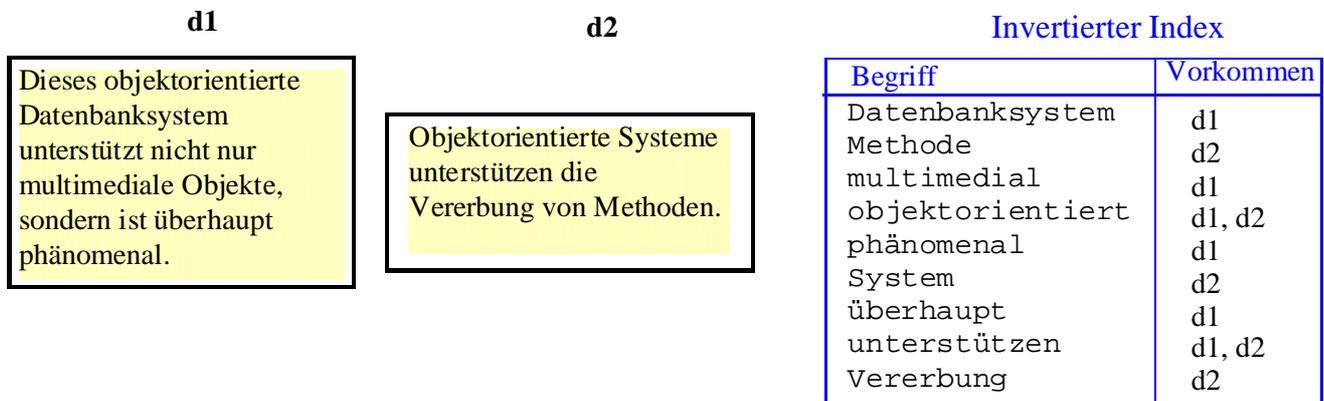
■ Ziel: Unterstützung von Information-Retrieval-Anfragen nach Textinhalten / Deskriptoren

- Suche nach bestimmten Wörtern/Schlüsselbegriffen/Deskriptoren (nicht nach beliebigen Zeichenketten)
- ggf. Abfangen sogenannter „Stop-Wörter“ (der, die, das, ist, er ...)
- Boole'sche Anfragen zur Verknüpfung mehrerer Suchbedingungen
- "unscharfe" Anfragen; Berücksichtigung von Wort-Stammformen
- Ranking von Ergebnissen (Berücksichtigung von Begriffshäufigkeiten, Nachbarschaftvorkommen in einem Satz / Paragraphen ...)



Invertierte Listen

- Invertierung: Verzeichnis (Index) aller Vorkommen von Schlüsselbegriffen
 - lexikographisch sortierte Liste der vorkommenden Schlüsselbegriffe
 - pro Eintrag (Begriff) Liste der Dokumente (Verweise/Zeiger), die Begriff enthalten
 - eventuell zusätzliche Information pro Dokument wie Häufigkeit des Auftretens oder Position der Vorkommen
- Beispiel

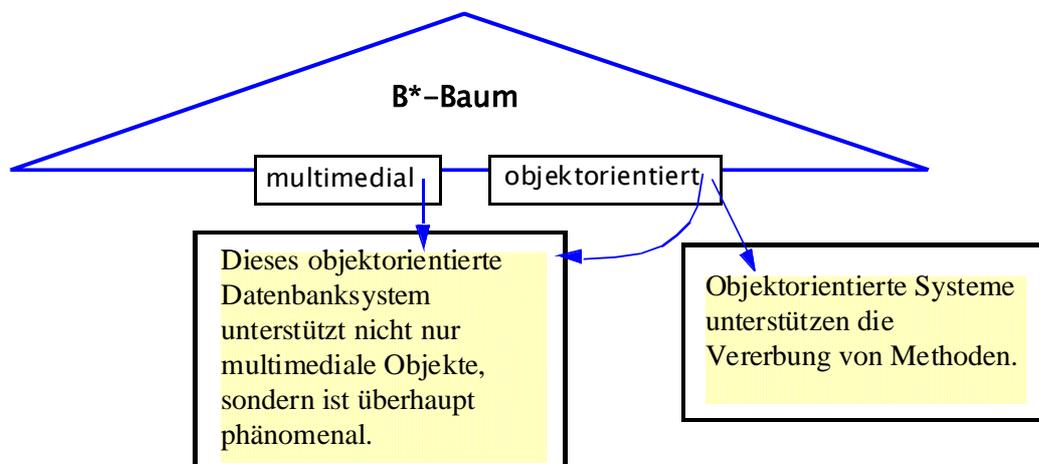


- Zugriffskosten werden durch Datenstruktur zur Verwaltung der invertierten Liste bestimmt (B*-Baum, Hash-Verfahren ...)



Invertierte Listen (2)

- effiziente Realisierung über (indirekten) B*-Baum
 - variabel lange Verweis/Zeigerlisten pro Schlüssel auf Blattebene
 - eventuell zusätzliche Information pro Dokument wie Häufigkeit des Auftretens oder Position des ersten Vorkommens



- Boole'sche Operationen: Verknüpfung von Zeigerlisten
 - Beispiel: Suche nach Dokumenten mit „multimedial“ UND „objektorientiert“



Signatur-Dateien

- Alternative zu invertierten Listen: Einsatz von *Signaturen*
 - zu jedem Dokument wird Bitvektor fester Länge (*Signatur*) geführt
 - Begriffe werden über Signaturgenerierungsfunktion (Hash-Funktion) s auf Bitvektor abgebildet
 - OR-Verknüpfung der Bitvektoren aller im Dokument vorkommenden Begriffe ergibt *Dokument-Signatur*
- Suchbegriffe werden über dieselbe Signaturgenerierungsfunktion s auf eine *Anfragesignatur* abgebildet
 - Kombination mehrerer Suchbegriffe durch OR, AND, NOT-Verknüpfung der Bitvektoren
 - wegen Nichtinjektivität der Signaturgenerierungsfunktion muss bei ermittelten Dokumenten geprüft werden, ob tatsächlich ein Treffer vorliegt
- Eigenschaften
 - geringer Platzbedarf für Dokumentensignaturen
 - Zugriffskosten aufgrund Nachbearbeitungsaufwand bei „False Matches“ meist höher als bei invertierten Listen

Signaturgenerierungsfunktion:

objektorientiert -> Bit 0
 multimedial -> Bit 2
 Datenbanksystem -> Bit 4
 Vererbung -> Bit 2

Anfrage: Dokumente mit Begriffen "objektorientiert" und "multimedial"

Anfragesignatur:

Signaturen der Dokumente

1 0 1 0 0 0
1 0 1 0 1 0
0 0 1 0 1 1
...

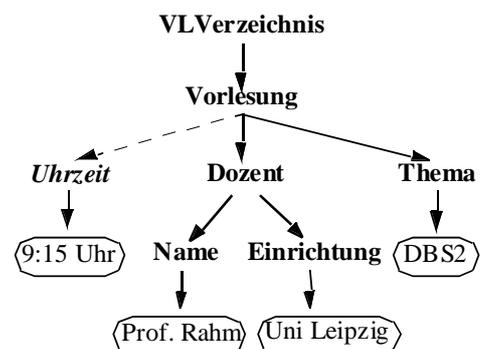
Objektorientierte Systeme unterstützen die Vererbung von Methoden. ...

Dieses objektorientierte Datenbanksystem unterstützt nicht nur multimediale Objekte, sondern ist überhaupt phänomenal.



Indexierung von XML-Daten

- Beispiel-XML-Dokument repräsentiert als DOM-Baum
 - Element-Knoten
 - Attribut-Knoten
 - Text-Knoten
- Indexierung abhängig von Datenstruktur zur Speicherung von XML-Dokumenten
- hier: generisches Datenmodell auf Relationen



Elemente

DocID	ID	ElementName	VaterKnoten	Pos	Wert
1	1	VLVerzeichnis	Null	1	Null
1	2	Vorlesung	1	1	Null
1	3	Dozent	2	1	Null
1	4	Name	3	1	Prof. Rahm
1	5	Einrichtung	3	2	Uni Leipzig
1	6	Thema	2	2	DBS2

Attribute

DocID	ElementID	AttributName	Wert
1	2	Uhrzeit	9:15 Uhr



Indexierung von XML-Daten (2)

■ Elementindex (Namensindex)

Aufsuchen aller Knoten eines bestimmten Elementtyps/
Attributnamens

- Index auf ElementName (B*-Baum)
Bsp.: //Dozent

■ Strukturindex (Pfadnavigation)

- Index auf VaterKnoten (B*-Baum);
beschleunigt Child-Achse
Bsp.: /VLVerzeichnis/*

■ Wertindex

Aufsuchen von Elemente/
Attributen über ihren Inhalt/Wert

- Index auf Wert (B*-Baum)
Bsp.: //*[@Uhrzeit='9:15 Uhr']
- bei Vorliegen von Typinformationen der Werte auch mehrere typspezifische Indizes möglich
z.B. für //Person[@Alter < 20]

■ Textindex

- Elemente können als Wert längere Texte enthalten
- Wertindex nicht für Suche nach Teilausdrücken nutzbar
- Textindex als invertierte Liste über den Elementinhalt
Bsp.: //Vorlesung/Thema[contains(text(), 'Datenbanken')]

Elemente

DocID	ID	ElementName	VaterKnoten	Pos	Wert
1	1	VLVerzeichnis	Null	1	Null
1	2	Vorlesung	1	1	Null
1	3	Dozent	2	1	Null
1	4	Name	3	1	Prof. Rahm
1	5	Einrichtung	3	2	Uni Leipzig
1	6	Thema	2	2	DBS2

Attribute

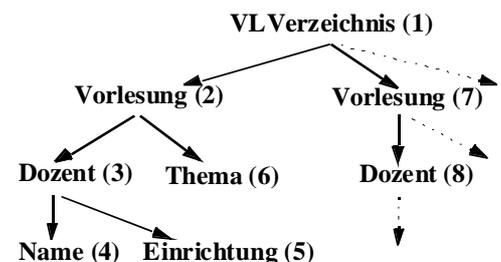
DocID	ElementID	AttributName	Wert
1	2	Uhrzeit	9:15 Uhr



Indexierung von XML-Daten (3)

■ Pfadindex

- Problem: XPath-Ausdrücke müssen Schritt für Schritt abgearbeitet werden (z.B. /VLVerzeichnis/Vorlesung/Dozent)
- Pfadindex liefert alle Knoten, die über einen bestimmten Pfad erreichbar sind



Pfad	Knoten
/VLVerzeichnis	{1}
/VLVerzeichnis/Vorlesung	{2,7,...}
/VLVerzeichnis/Vorlesung/Dozent	{3,8,...}
/VLVerzeichnis/Vorlesung/Dozent/Name	{4,...}
/VLVerzeichnis/Vorlesung/Dozent/Einrichtung	{5,...}
/VLVerzeichnis/Vorlesung/Thema	{6,...}

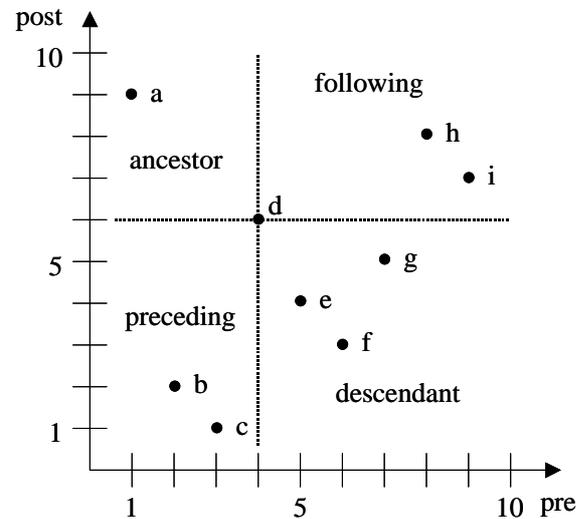
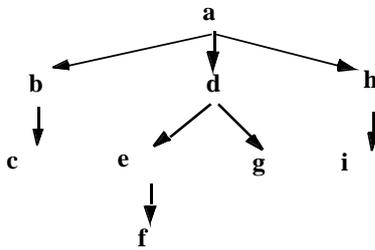
- XPath-Anfrage wird als Zeichenkette komplett oder als Teilausdruck mit Pfadindex verglichen
- auch Wildcards können in der Anfrage vorkommen
Bsp.: /VLVerzeichnis//Name
/VLVerzeichnis/*/Dozent
- Anwendbarkeit eingeschränkt, wenn Prädikate enthalten sind
Bsp.: /VLVerzeichnis/Vorlesung[@Uhrzeit='9:15 Uhr']/Dozent/Name



Indexierung von XML-Daten (4)

■ Erweiterte Knotennummerierung

- **Problem:** bisheriger Ansatz erfordert Baum-Traversierung zur Auswertung der XPath-Achsen *ancestor* (Vorfahre), *descendant* (Nachkomme), *preceding* (Vorangehende) und *following* (Folgende)
- **Lösung:** spezielle Knotennummerierungen erlauben Auswertung der Achsen über Bereichsanfragen
- Bsp.: Knotennummerierung mit pre- und postorder Werten



- $descendant(d) = \{ x \mid x.pre > d.pre \text{ AND } x.post < d.post \}$
- Indexvarianten:
 - je ein B*-Baum für pre- und postorder Wert
 - R-Tree
- Einschränkung: Nummerierungsschema eignet sich nur für statische XML-Dokumente (Neunummerierung bei Einfügen/Löschen von Knoten)

* Grust, T.: Accelerating XPath Location Steps. In Proc. of ACM SIGMOD Conf., 2002



Zusammenfassung

- Standard-Indexstruktur in DBS: B*-Baum ("the ubiquitous B*-tree")
 - direkter vs. indirekter; clustered vs. non-clustered Index
 - direkter und sortiert sequentieller Zugriff für Primär- und Sekundärschlüssel
 - verallgemeinerte Zugriffspfadstruktur, Join-Index
 - Verbesserung der Baumbreite: Schlüsselkomprimierung, verallgemeinertes Splitting
- schnellerer Schlüsselzugriff erfordert Hash-Verfahren
 - (nur) direkter Zugriff (< 1.4 Seitenzugriffe)
 - dynamische Hash-Verfahren unterstützen stark wachsende Datenbestände
 - Beispiel-Implementierung: Erweiterbares Hashing (2 Seitenzugriffe)
- Bitlisten-Indexierung u.a. für Attribute geringer Kardinalität
- Mehrdimensionale Anfrageunterstützung
 - Intersection / Containment / Nearest Neighbor queries
 - Erhaltung der topologischen Struktur des Datenraumes
 - eindimensionale Indexstrukturen begrenzen Datenraum unzureichend
 - mehrdimensionale Beispielansätze: Grid-File und R-Baum
- Textsuche (Schlüsselwortsuche in großen Dokumentkollektionen)
 - Verwendung von invertierten Listen (B*-Bäumen) oder Signaturen
- XML: verschiedene Indexvarianten für Elemente, Pfade, Werte

