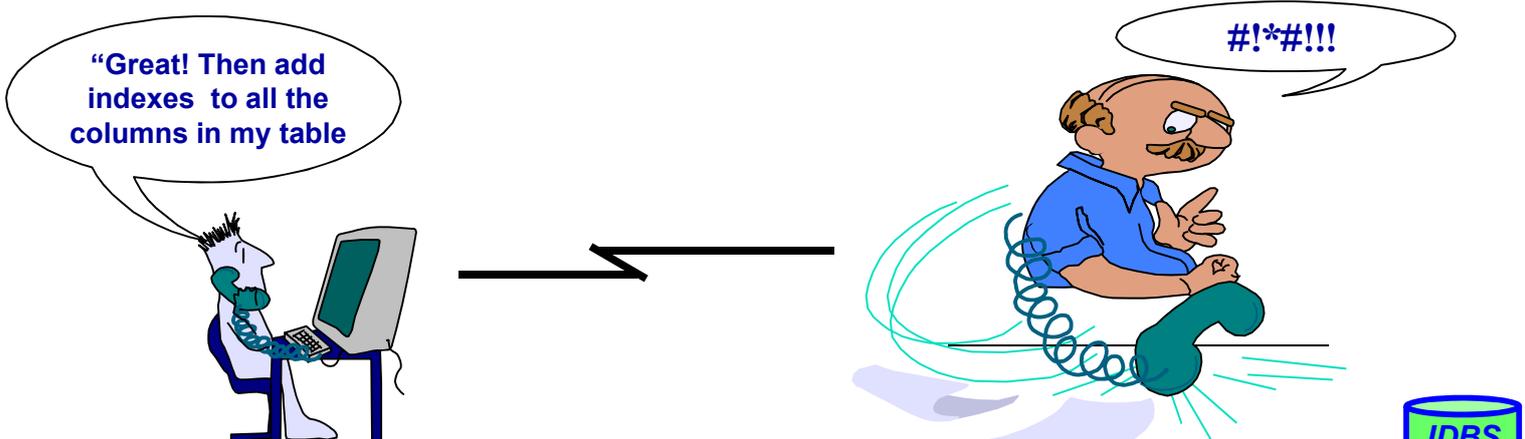
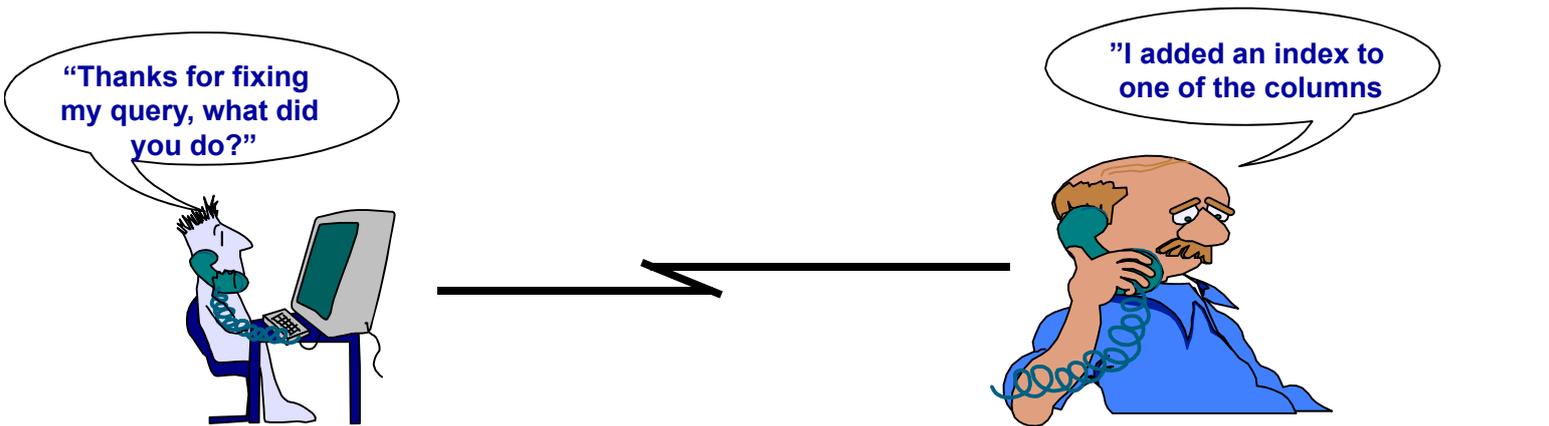


5. Indexstrukturen

- Einführung
 - Zugriffsarten und Suchanfragen
- B*-Baum
 - clustered / non-clustered Index
 - Optimierungen: Schlüsselkomprimierung, erweitertes Splitting
- Hash-Verfahren
 - statische Verfahren
 - erweiterbares Hashing
- Sekundärschlüsselzugriff
 - Invertierung
 - Bitlisten-Indizes
 - verallgemeinerte Indexstruktur (Join-Index)
- mehrdimensionale Indexstrukturen
 - eindimensionale Einbettungen
 - Grid-File
 - R-Baum
- Textsuche/Matching:
invertierte Listen, Signatur-Dateien, Nutzung von Bitvektoren



Zugriffsarten

- sequenzieller Zugriff
 - auf alle Sätze eines Satztyps (Scan)
 - in Sortierreihenfolge eines Attributes
- direkter Zugriff
 - über den Primärschlüssel (z.B. PNR=4711)
 - über einen Sekundärschlüssel / Nicht-Primärschlüssel (z.B. Wohnort=,Leipzig‘)
 - über zusammengesetzte Schlüssel / Attribute
 - über mehrere unabhängige 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

- Suchbedingungen bezüglich mehrerer Attribute / Dimensionen
 - besonders wichtig für räumliche Objekte (Punkte, Polygone, Quader, ...) -> Geo-Anwendungen, etc.
- 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 ($k \leq n$) 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 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

$$Q = (A_{i_1} = a_{i_1}) \wedge (A_{i_2} = a_{i_2}) \wedge \dots \wedge (A_{i_s} = a_{i_s}) \text{ mit } 1 < s < k \text{ und } 1 < i_1 < i_2 < \dots < i_s < k$$

3. **(exakte) Bereichsanfrage (range query)**: spezifiziert einen Bereich $r_i = [l_i < a_i < u_i]$ für jeden Schlüssel A_i

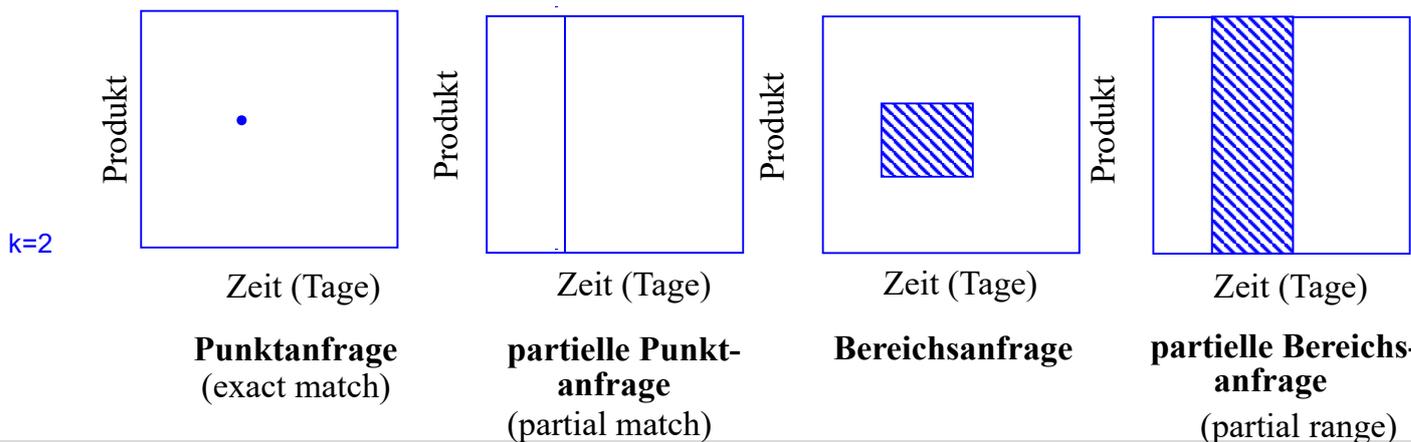
$$Q = (A_1 = r_1) \wedge \dots \wedge (A_k = r_k) \equiv (A_1 > l_1) \wedge (A_1 < u_1) \wedge \dots \wedge (A_k > l_k) \wedge (A_k < u_k)$$

4. **partielle Bereichsanfrage (partial range query)**: spezifiziert für $s < k$ Schlüssel einen Bereich

$$Q = (A_{i_1} = r_{i_1}) \wedge \dots \wedge (A_{i_s} = r_{i_s}) \text{ mit } 1 < s < k \text{ und } 1 < i_1 < \dots < i_s < k \text{ und } r_{i_j} = [l_{i_j} < a_{i_j} < u_{i_j}], 1 < j < s$$

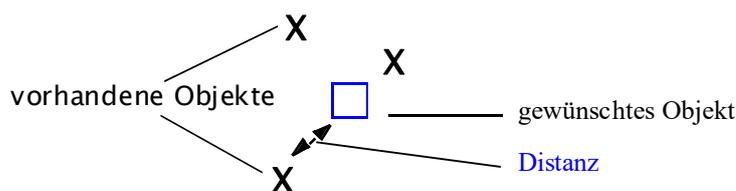
→ bei den schnittbildenden Anfragen lassen sich alle 4 Fragetypen als allgem. Bereichsfrage ausdrücken

- genauer Bereich $[l_i = a_i = u_i]$
- vs. unendlicher Bereich $[-\infty < a_i < \infty]$



Best-Match (Nearest Neighbor)-Anfragen

- gewünschtes Objekt nicht vorhanden
=> Suche 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

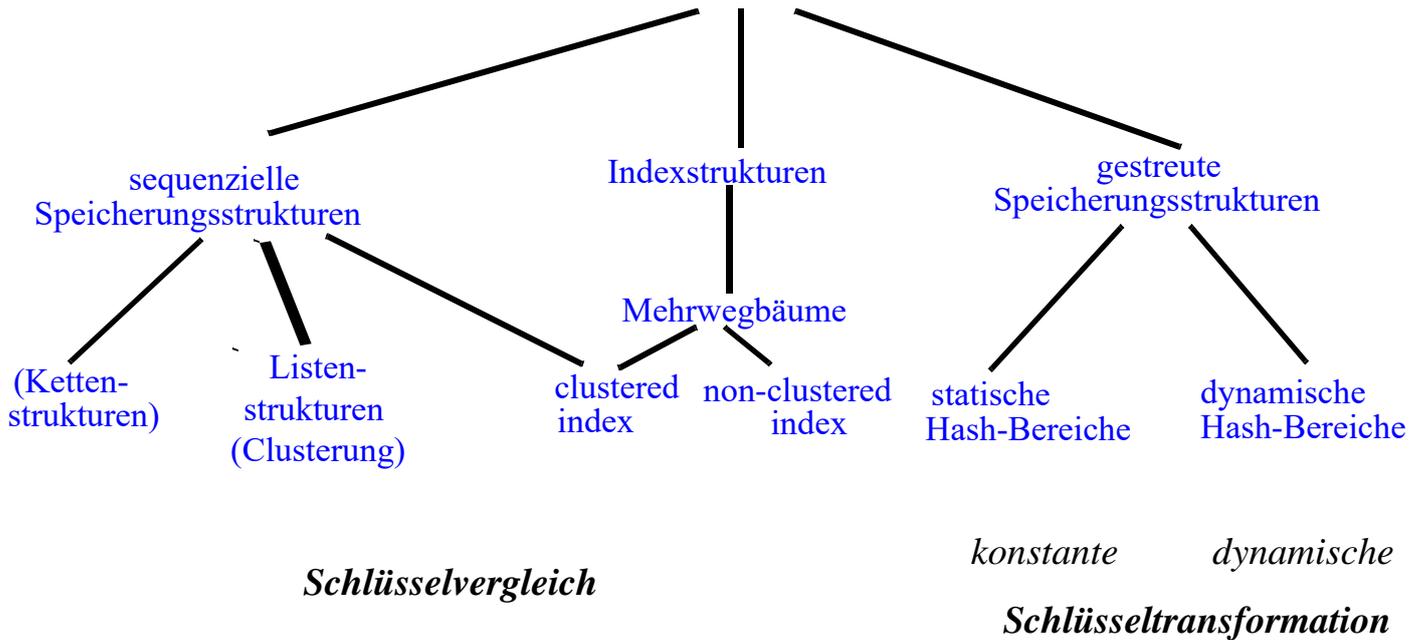
B = Sammlung von Punkten im k -dim. Raum

Gesucht: nächster Nachbar von p (in B)

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



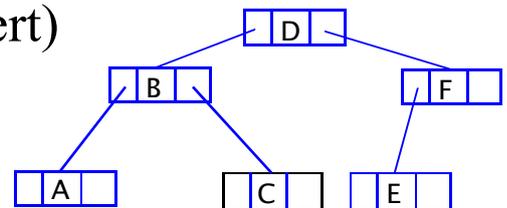
Zugriffsverfahren für Primärschlüssel



Mehrweg-Bäume

■ Ausgangspunkt: binäre Bäume (balanciert)

- entwickelt für Hauptspeicher
- für DB-Anwendungen unzureichend



■ Weiterentwicklung: B- und B*-Baum

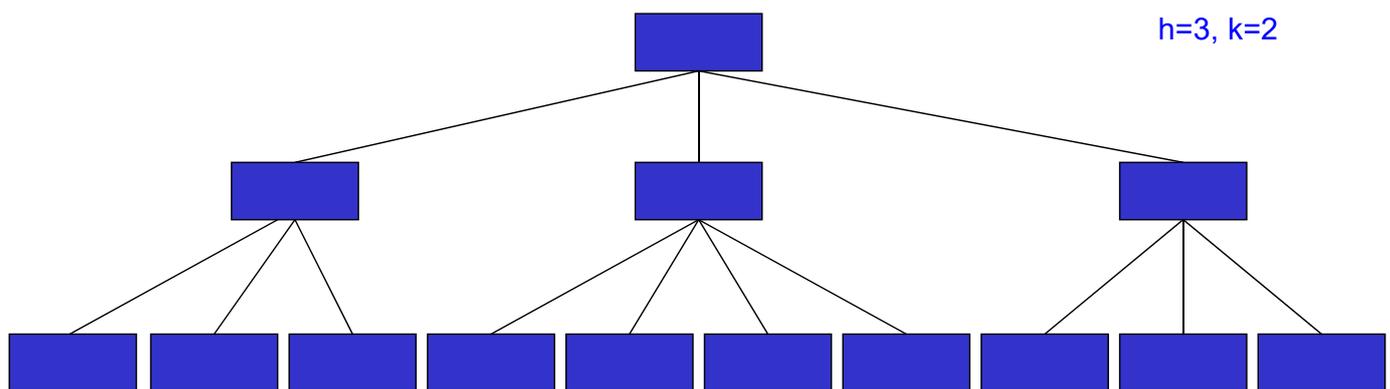
- Seiten (Blöcke) als Baumknoten
- Zusammenfassung mehrerer Sätze/Schlüssel in einem Knoten
- höherer Verzweigungsgrad (Mehrwegbaum), wesentlich niedrigere Baumhöhe
- balancierte Struktur: unabhängig von Schlüsselmenge und Einfügereihenfolge
- dynamische Reorganisation durch Splitten und Mischen von Seiten

Mehrweg-Bäume (2)

- Speicherung von Datensätzen: direkt/eingebettet im Mehrweg-Baum oder separat
- **direkter Index**: eingebettete Datensätze
 - kann Baumhöhe erhöhen, erspart jedoch separaten Satzzugriff über Verweis.
 - direkte Speicherung nur für einen Index pro Satztyp (i.a. für Primärschlüssel) !
- **indirekter Index**: enthält Verweis (TID) pro Datensatz
 - mehr Einträge pro Knoten
 - einige Abfragen (z.B. Aggregatfunktionen) können direkt auf Index beantwortet werden



B*-Baum



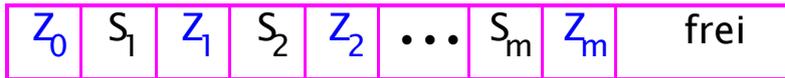
B*-Baum vom Typ (k, k^*, h) ist ein Baum mit folgenden Eigenschaften

- jeder Weg von der Wurzel zum Blatt hat die Länge h
- jeder Zwischenknoten hat mindestens $k+1$ Kinder. Die Wurzel ist ein Blatt oder hat mindestens 2 Kinder.
- jeder Zwischenknoten hat höchstens $2k+1$ Kinder.
- jedes Blatt hat mindestens k^* und höchstens $2k^*$ Einträge



B*-Baum: Seitenformate

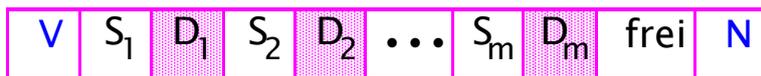
- Zwischenknoten: k bis $2k$ Einträge, $k+1$ bis $2k+1$ Verweise auf Kinder



S_i = Schlüssel
 Z_i = Zeiger auf Kindseite

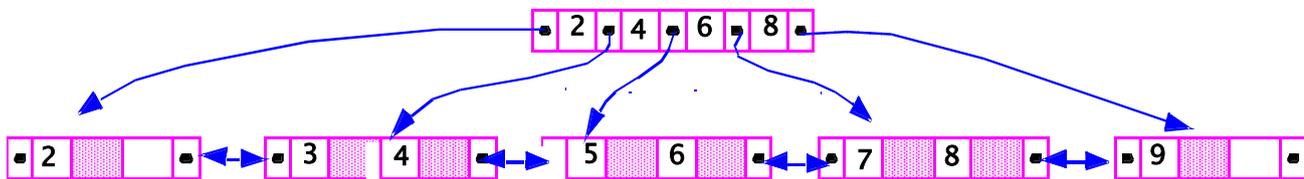
alle Schlüssel in einem Knoten sind sortiert: $S_i \leq S_{i+1}$ ($i=1, \dots, m-1$)
 für alle Schlüssel s im Teilbaum zu Z_i gilt: $S_i < s \leq S_{i+1}$ ($i=1, \dots, m-1$)
 bzgl. Z_0 gilt: $s \leq S_1$
 und bzgl. Z_m gilt: $S_m < s$

- Blattknoten: k^* bis $2k^*$ Einträge, Verkettung mit Nachbarseiten



S_i = Schlüssel
 D_i = Datensatz bzw. Verweis (TID)
 V = Zeiger Vorgängerseite
 N = Zeiger Nachfolgerseite

- Beispiel ($h=2, k=2, k^*=1$):



B*-Baum: Verzweigungsgrad, Höhe

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

Eintragsgröße Zwischenknoten: $4B+4B=8$ B $\rightarrow 2k=8000/8=1000 \rightarrow k=500$

indirekter Index: Größe D (=TID): 4 B \rightarrow Eintragsgröße 8B $\rightarrow 2k^*=8000/8=1000, k^*=500$

direkter Index: Größe $D+S$ sei 100 B $\rightarrow 2k^*=8000/100=80 \rightarrow k^*=40$

- Anzahl von Sätzen (N) für unterschiedliche Baumhöhen

$$h=1: N \leq 2k^* = 1000$$

$$h=2: N \leq 1001 * 1000 \approx 1 \text{ Million Sätze}$$

$$h=3: N \leq 1001^2 * 1000 \approx 10^9 \text{ Sätze}$$

- 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:

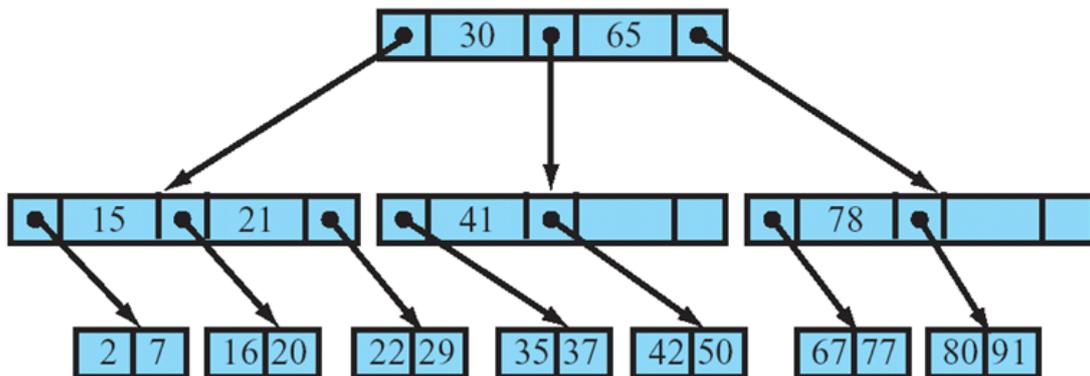
Höhe 2 – 4 bei $10^5 - 10^8$ Datensätzen (für 8 KB-Seiten)



B*-Baum (3)

Suche im B*-Baum (Lookup)

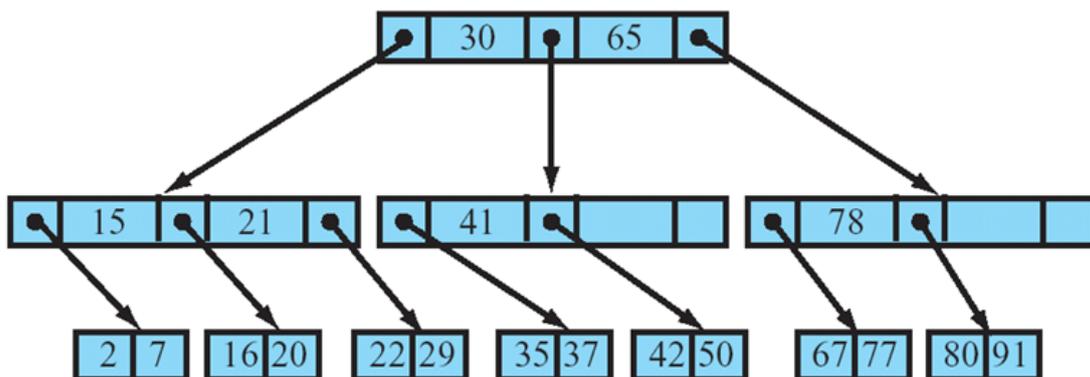
- beginnend im Wurzelknoten; jeder Knoten wird von links nach rechts durchsucht
- ist der gesuchte Wert kleiner oder gleich S_i , wird die Suche in der Wurzel des von Z_{i-1} identifizierten Teilbaums fortgesetzt
- ist der gesuchte Wert größer als S_i , wird der Vergleich mit S_{i+1} fortgesetzt. Ist 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)



B*-Baum: Anfragebearbeitung

unterstützte Zugriffe/Suchanfragen:

- direkter Schlüsselzugriff sowie sortiert sequenzieller Zugriff
- exakte Anfragen und Bereichsanfragen
- Präfix-Match-Anfragen, Extremwertanfragen ...



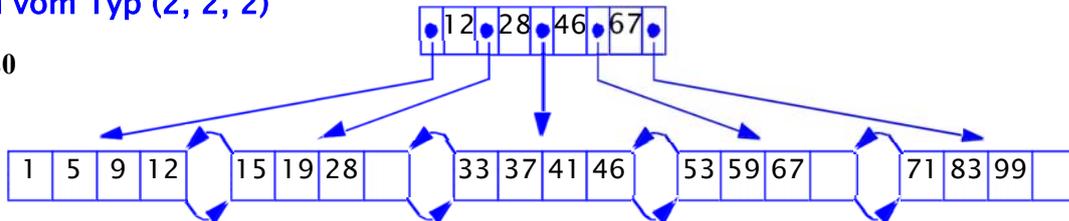
Einfügen im B*-Baum

Vorgehensweise

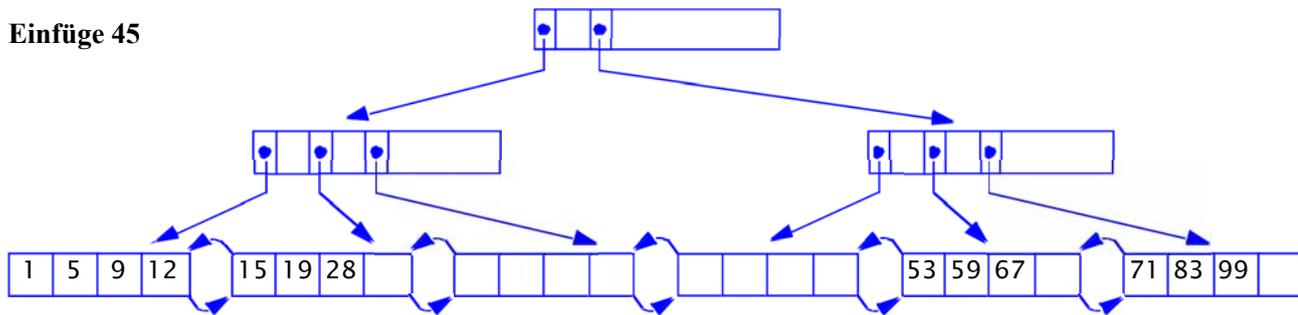
- zunächst Abstieg durch den Baum wie bei der Suche
- Einfügen des Satzes stets im Blattknoten
- falls Blattknoten bereits voll, muss ein neuer Knoten erzeugt und ein gleichmäßiges Aufteilen der $(2k^*+1)$ Sätze auf zwei Seiten vorgenommen werden (Splitting)
- Splitting erfordert Anpassung der Verzweigungsinformation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

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

Einfüge 20



Einfüge 45



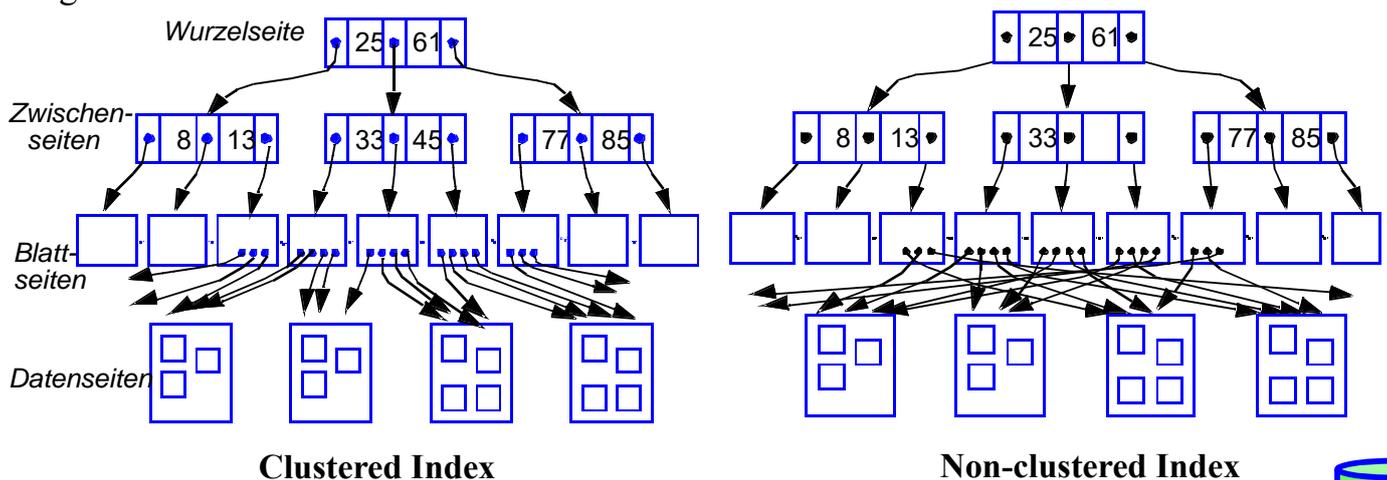
B*-Baum mit/ohne Clusterung der Sätze

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



B*-Baum: Optimierungen

■ Merkmale B*-Baum

- innere Knoten enthalten nur Verzweigungsinformation (kleine Einträge günstig für geringe Baumhöhe)
- für jeden Satz müssen h Seiten gelesen werden
- Kette der Blattknoten liefert alle Sätze nach Schlüsselwert sortiert (->Clustermöglichkeit)

■ Optimierungsziel: wenige Seiten und niedrige Höhe

- möglichst große Seiten
- möglichst hoher Verzweigungsgrad (Fan-Out) -> mehr Einträge pro Seite
- möglichst kurze Schlüsselwerte (-> Schlüsselkomprimierung)
- möglichst hohe Seitenbelegung (-> verallgemeinerte Splitting-Verfahren)

Präfix/Suffix-Komprimierung

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

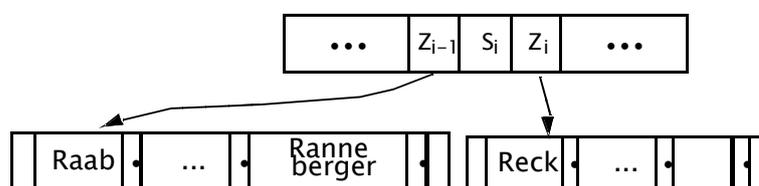
■ Präfix-Komprimierung

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

Schmidt
Schmied
Schmitt

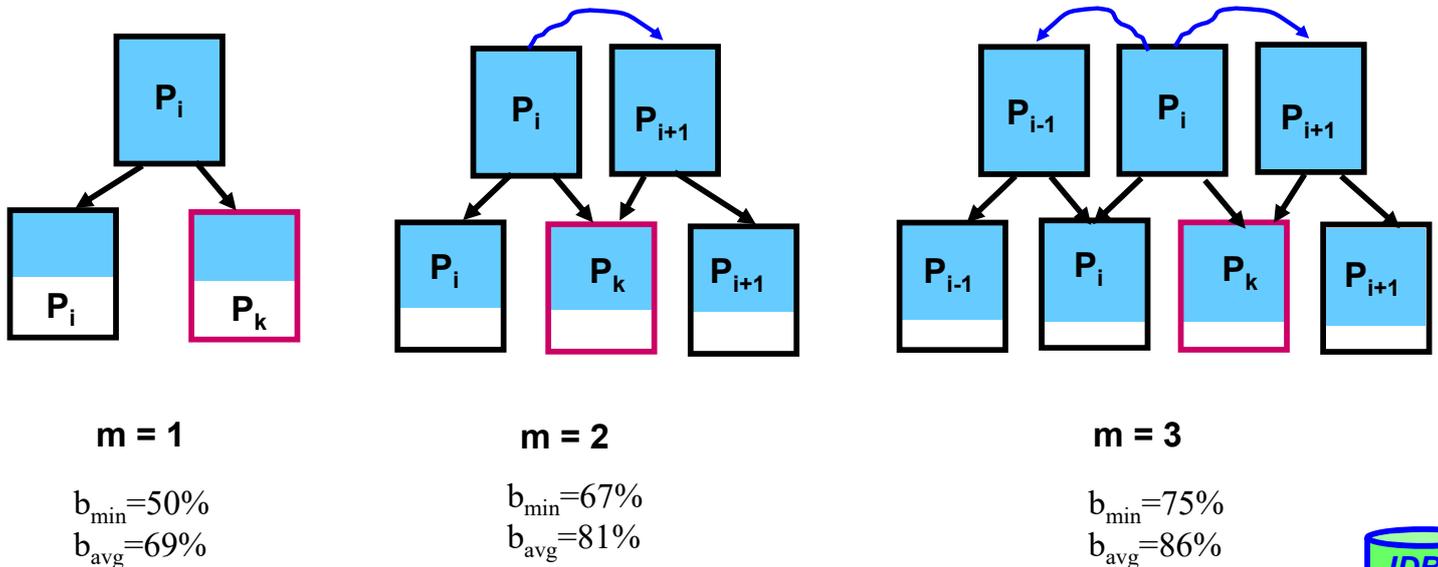
■ Suffix-Komprimierung

- 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)
 $b_{\min}=m/(m+1)$, $b_{\text{avg}}=m \ln((m+1)/m)$, $b_{\max}=1$



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 -> 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 Verkettung von Überlaufseiten
 - typischer Zugriffsfaktor: 1.1 bis 1.4

Statisches Hash-Verfahren mit Überlaufbereichen: Beispiel

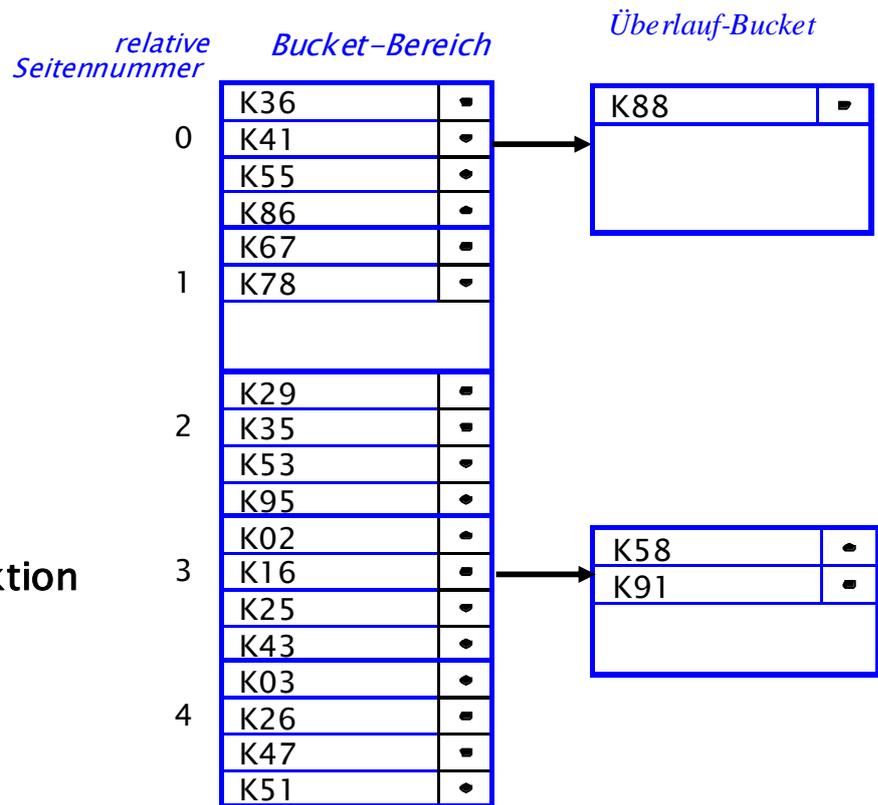
Adreßberechnung
für Schlüssel K02

1. Faltung

$$\begin{array}{r}
 1101\ 0010 \quad K \\
 \oplus 1111\ 0000 \quad 0 \\
 \hline
 \oplus 1111\ 0010 \quad 2 \\
 \hline
 1101\ 0000 = 208_{10}
 \end{array}$$

2. Anwendung der Hash-Funktion

$$208 \bmod 5 = 3$$



n=5, B=4

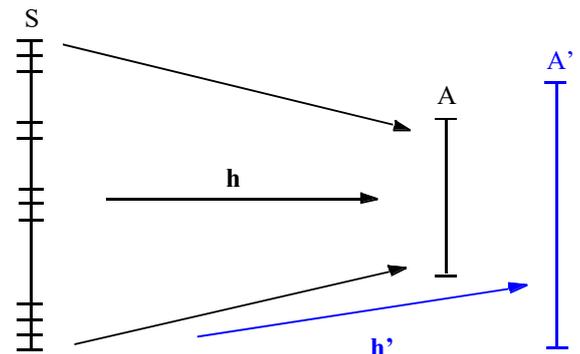


Dynamische Hash-Verfahren

■ Wachstumsproblem bei statischen Verfahren

- statische Allokation von Speicherbereichen führt oft zu schlechter 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

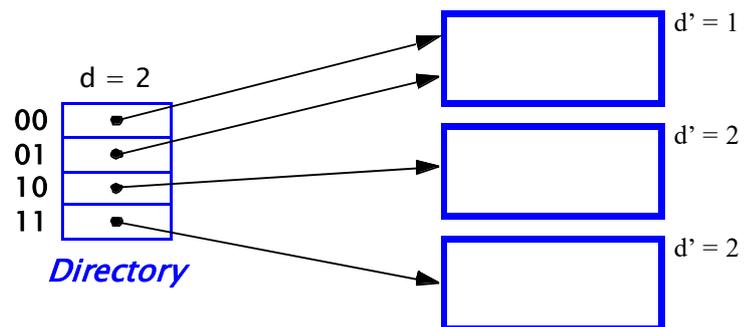
■ viele konkurrierende Ansätze

- Extendible Hashing
- Virtual Hashing und Linear Hashing
- Dynamic Hashing



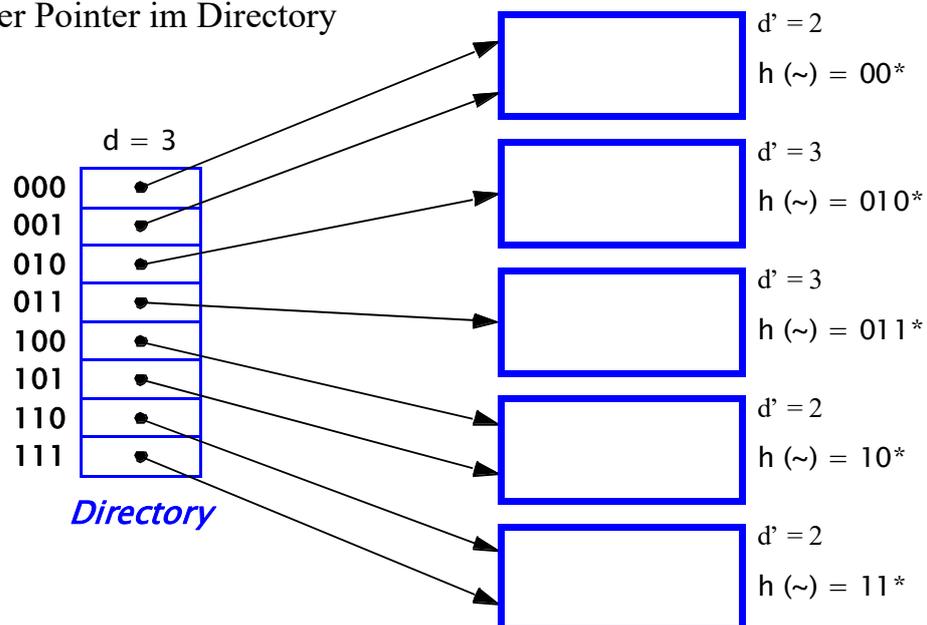
Erweiterbares Hash-Verfahren (Extendible Hashing)

- 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: Directory + Bucket
 - 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)
 - $d = \text{MAX}(d')$



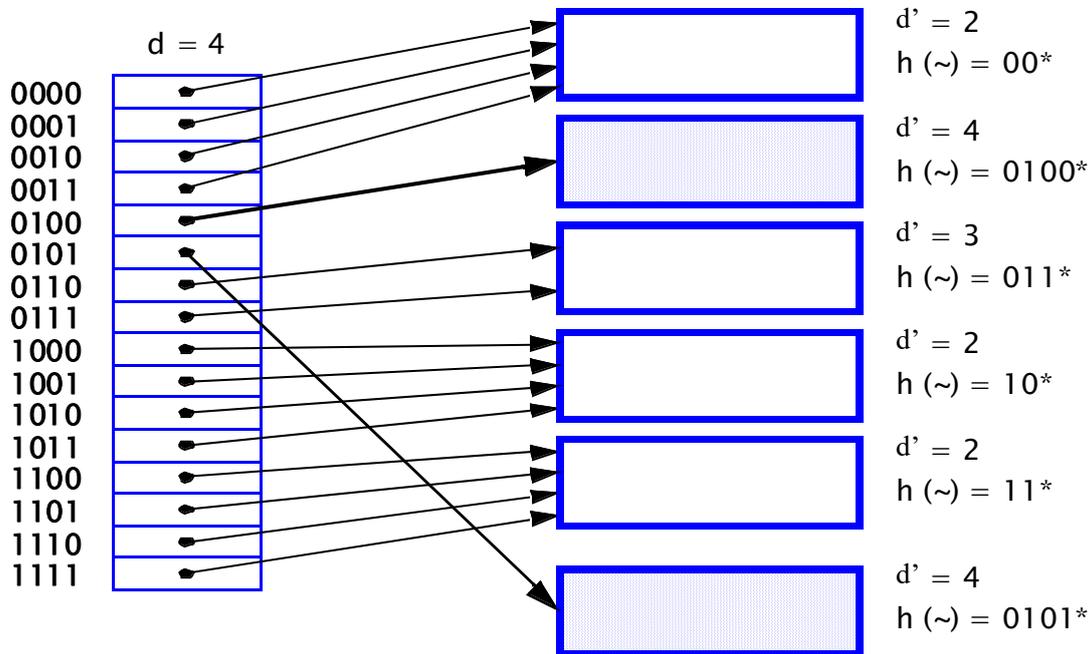
Erweiterbares Hashing: Splitting von Buckets

- 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 $N = 10^6$ Sätze

Speicherungs-/Indexstruktur	direkter Zugriff	sortiert-sequenzielle Verarbeitung
gekettete Liste	$O(N) \approx 5 \cdot 10^5$	$O(N) \approx 10^6$
sequenzielle Liste/Clusterung	$O(N) \approx 10^4$	$O(N) \approx c \cdot 10^4$
Mehrwegbäume (B*-Bäume)	$O(\log_k N) \approx 2 - 4$	$O(N) \approx 10^6$
Clustered index	$O(\log_k N) \approx 2 - 4$	$O(N) \approx c \cdot 10^4$
statisches Hashing mit Überlaufbereich	$O(1) (\approx 1.1 - 1.4)$	$O(N \log_2 N) (*)$
erweiterbares Hashing	$O(1) = 2$	$O(N \log_2 N) (*)$

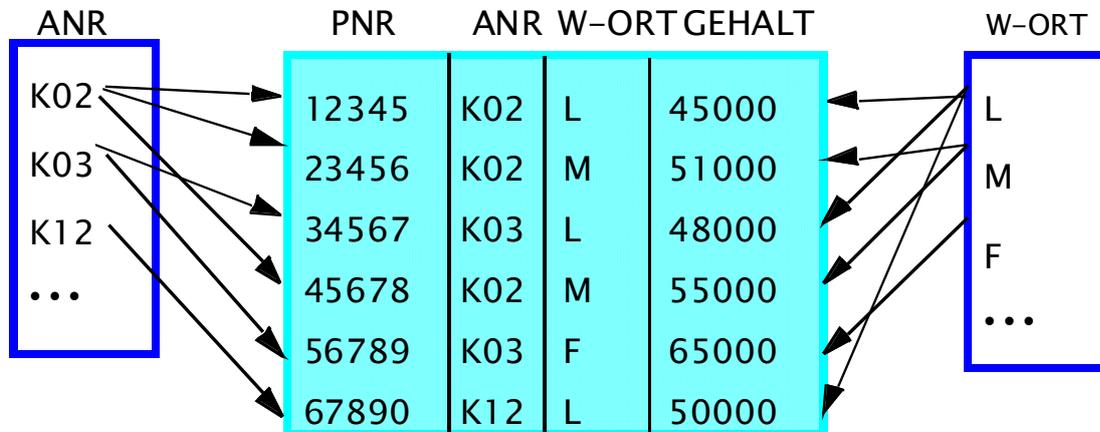
c vergleichsweise kleine Konstante

(*) physisch sequenzielles Lesen, Sortieren und sequenzielles Verarbeiten der gesamten Sätze



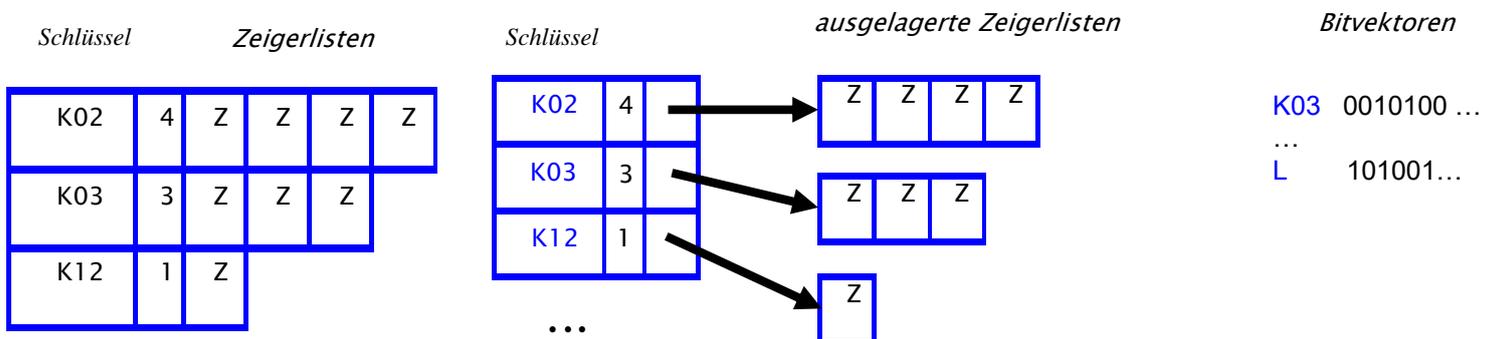
Zugriffspfade für Sekundärschlüssel

- Sekundärschlüssel: nicht-identifizierendes Attribut; i.a. keine Eindeutigkeit
- Suchanfragen liefern i.a. mehr als 1 Treffer (Satzmengen)



Invertierungstechniken

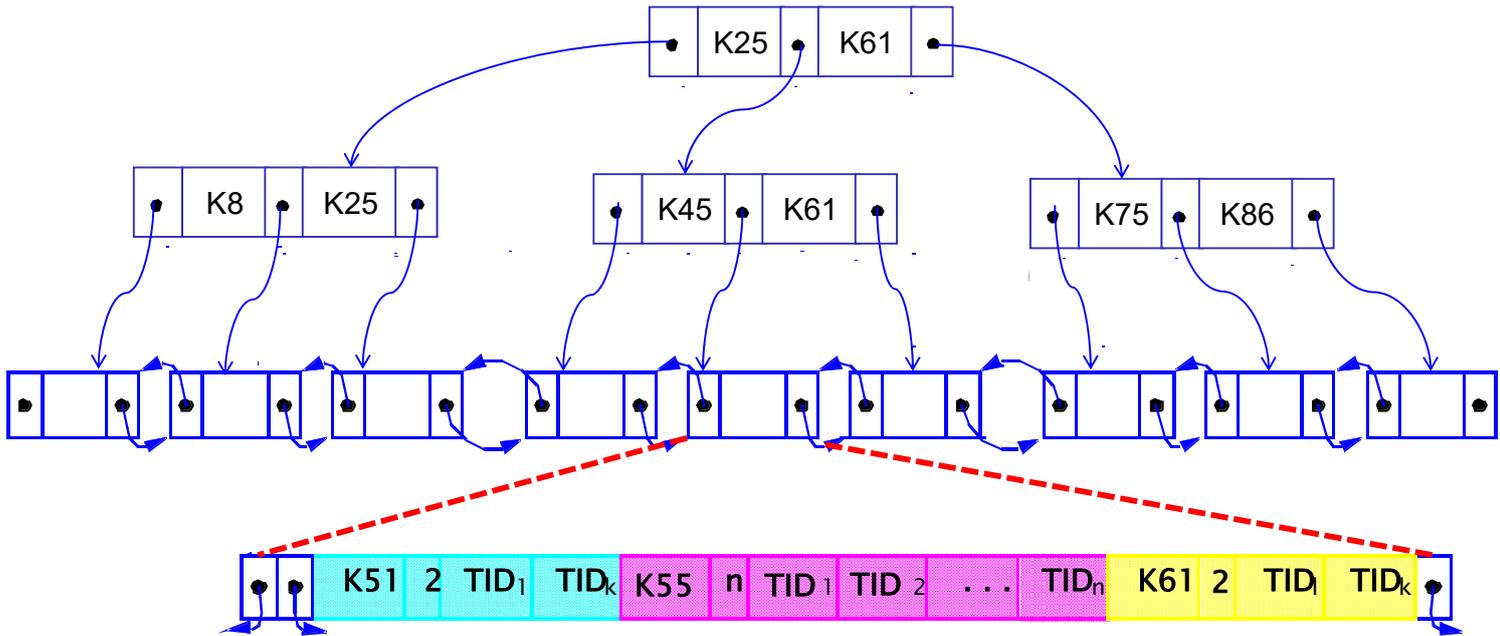
- Invertierung: Indexstruktur mit Zeigerlisten bzw. Bitvektoren (N Bit pro Wert) pro Attributwert
- effiziente Berechnung mengenalgebraischer Operationen (Durchschnitt/AND-Bedingung, Vereinigung/OR-Bedingung; Komplement / NOT)



- Anfrage: ANR=„K03“ AND W-ORT=„L“
 $\{Z3, Z5, Z7\}$ $\{Z1, Z3, Z6\}$

- Speicherplatzbedarf für Zeigerlisten (TID-Listen) vs. Bitlisten
 - N Sätze; k unterschiedliche Attributwerte, TID-Länge t (4 bzw. 8 B)
 - Bitlistenumfang: $k * N \text{ Bit} = k * N / 8 \text{ Bytes}$
 - TID-Umfang $N * t$. Bitlisten kompakter falls $k < 8 * t$ (also < 32 bzw. 64 Attributwerte)

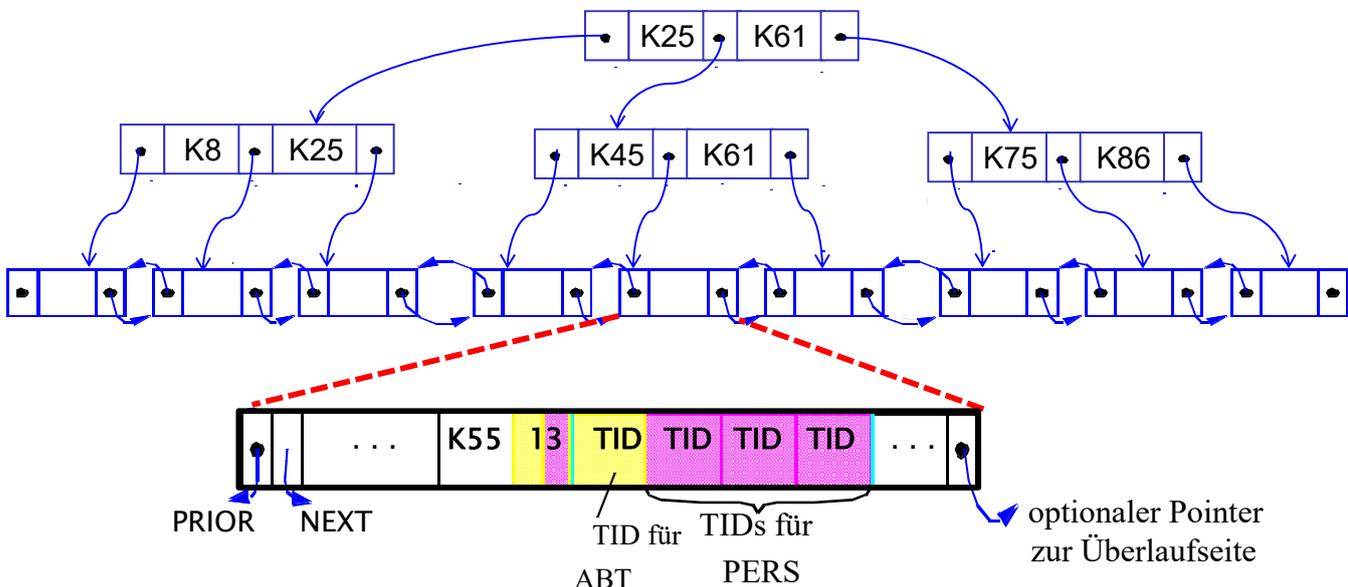
B*-Baum-Nutzung für Sekundärschlüssel



ANR-Index für Mitarbeiter-Tabelle

Verallgemeinerte Indexstruktur (Join-Index)

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



Verallgemeinerte Indexstruktur: Bewertung

■ Prinzip

- kombinierte Realisierung für Primär- und Sekundärschlüssel sowie mehrere Relationen mit demselben Attribut (z.B. ANR)
- innere Baumknoten bleiben unverändert; Blätter enthalten mehr Verweise

■ Erhöhung der Anzahl der Blattseiten

- mehr Seitenzugriffe beim sequenziellen Lesen einer Relation in Sortierordnung
- aber: Höhe des Baumes bleibt meist erhalten: ähnliches Leistungsverhalten beim Aufsuchen von Daten und beim Änderungsdienst

■ Vorteile

- weniger Indexstrukturen, 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)



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)

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

Kunde

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)

■ Ausführungszeitvergleich für Beispielanfrage

SELECT ... WHERE A1 = C1 AND A2=C2 AND A3=C3 AND A4=C4
10 Millionen Sätze; 200 B pro Satz -> 2 GB
pro Teilbedingung Selektivität 5%

Verarbeitungsdauer ohne Index (Scan): 100 s bei 20 MB/s

Bitlisten-Umfang: 1,25 MB pro Bitliste -> 5 MB für 4 Bitlisten -> 0,25 s Einlesedauer
Anzahl Treffer im Suchergebnis: $10^7 * (5 * 10^{-2})^4 = 5^4/10 \approx 63$
Einlesen Ergebnissätze: $63 * 10 \text{ ms} = 0,63 \text{ s}$
Gesamtzeitbedarf: $0,25+0,63 = 0,88 \text{ s}$



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
=> statt k nur noch $\log_2 k$ Bitlisten, z.B. 20 statt 1 Million = 2^{20} Bitlisten

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

■ Auswertung von Suchausdrücken

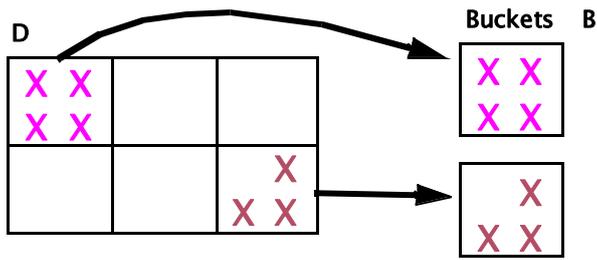
- höherer Aufwand bei nur 1 Bedingung ($\log_2 k$ Bitlisten statt 1 abuarbeiten)

Farbe = „Blau“:

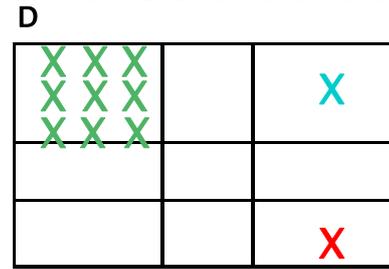


Grundprobleme räumlicher (mehrdim.) Zugriffe

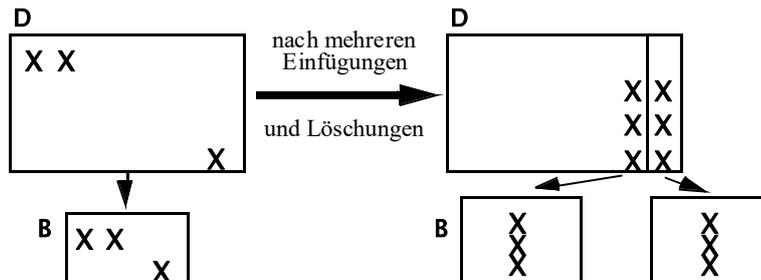
1. Erhaltung der topologischen Struktur



2. Stark variierende Dichte der Objekte



3. Dynamische Reorganisation



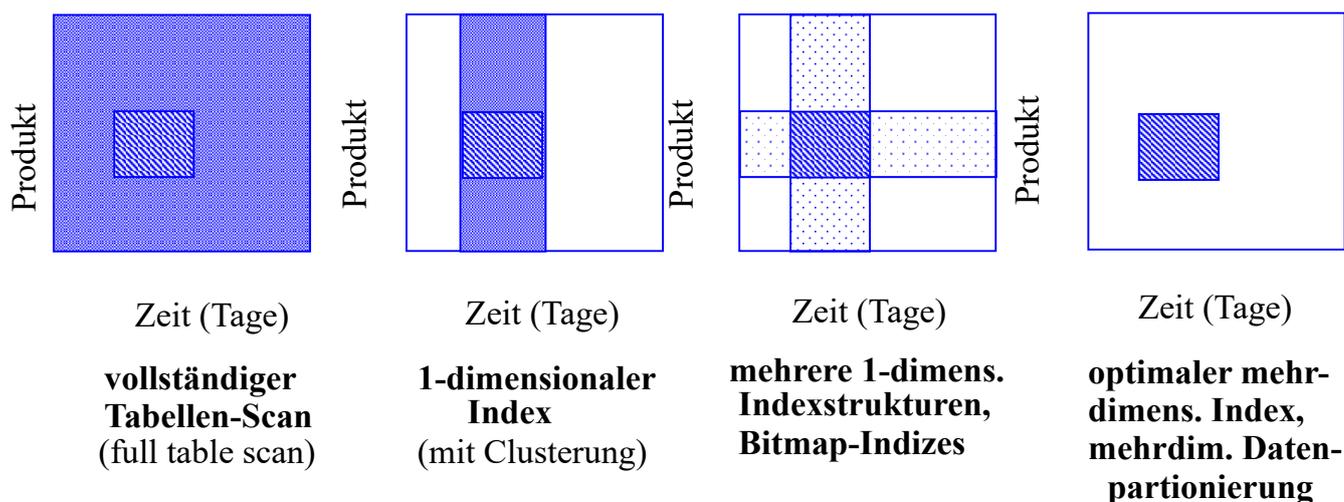
4. Objektdarstellung: Punktobjekte sowie Objekte mit Ausdehnung

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

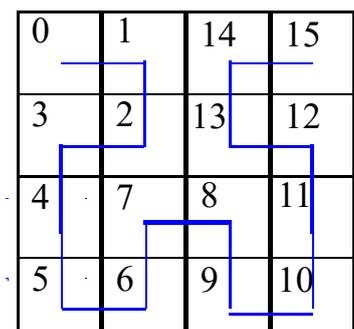
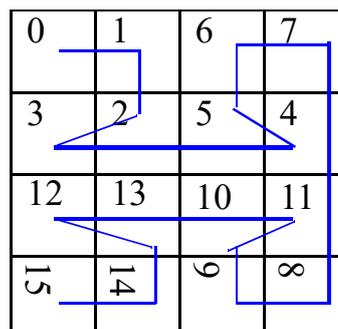
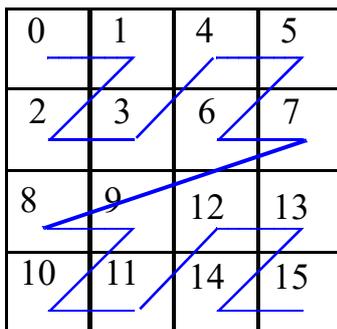
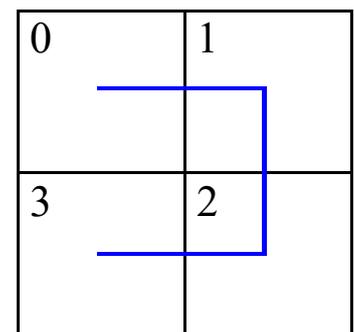
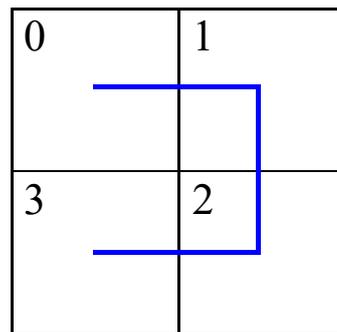
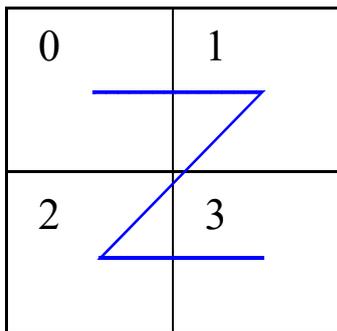
- Mischen mehrerer Trefferlisten für Anfragen wie $(\text{Key1} = \text{K1i}) \begin{cases} \text{OR} \\ - \\ \text{AND} \end{cases} (\text{Key2} = \text{K2j})$
- Clusterung höchstens bezüglich eines Attributs (eine Dimension)
- Topologie wird nicht erhalten
- Index auf konkatenierten Attributen unterstützt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

Eindimensionale Einbettung

- Transformation mehrdimensionaler Punktobjekte für eindimensionale Repräsentation, z.B. mit B*-Bäumen
- möglichst Wahrung der topologischen Struktur (Unterstützung mehrdimensionaler Bereichs- und Nachbarschaftsanfragen)
- Ansatz
 - Partitionierung des Datenraums D zunächst durch gleichförmiges Raster
 - eindeutige Nummer pro Zelle legt Position in der totalen Ordnung fest
 - Reihenfolge bestimmt eindimensionale Einbettung: *space filling curve*
- Zuordnung aller mehrdimensionalen Punktobjekte einer Zelle zu einem Bucket (Seite)
- jede Zelle kann bei Bedarf separat (und rekursiv) unter Nutzung desselben Grundmusters weiter verfeinert werden



Eindimensionale Einbettungen



a) z-Ordnung

b) Gray-Code

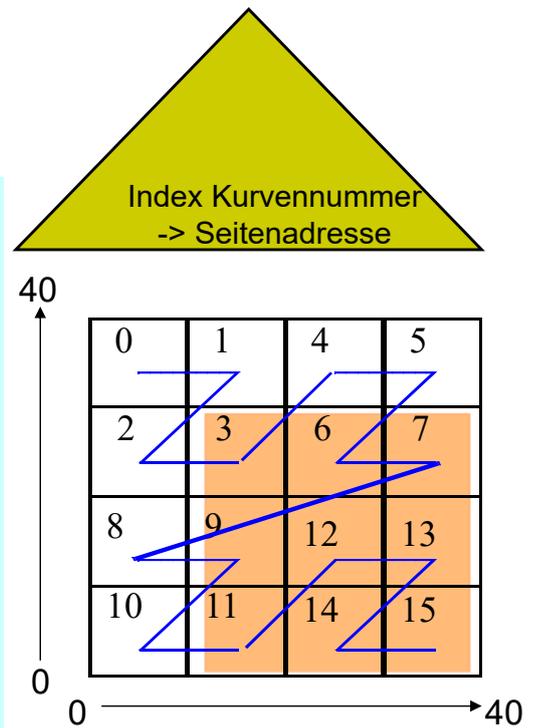
c) Hilbert's Kurve



Beispiel Z-Kurve

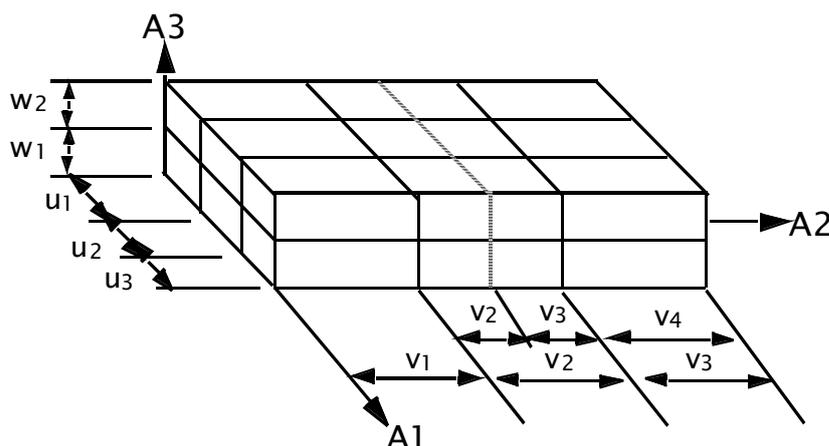
Range-Query bzgl. dem Suchintervall $[12, 38] \times [1, 29]$

- zu lesende Kurvennummern: 3, 6, 7, 9, 11, 12, 13, 14 und 15
- mögliche Suchschritte:
 1. Einstieg in Seite zu Kurvennummer 3
 2. Lesen von Nr. 6 und 7
 3. Lesen von Nr. 9
 4. dann lineares Lesen der Seiten zu 11 bis 15 über Listenverkettung



Grid-File (Gitterdatei)

- Organisation des Datenraumes D
 - D wird durch **orthogonales Raster** (grid) in k -dimensionale Zellen (**Grid-Blöcke**) partitioniert
 - die in den Zellen enthaltenen Objekte werden **Buckets** zugeordnet
 - es muss eine eindeutige Abbildung von Zellen zu Buckets gefunden werden
- dynamische Anpassung über Dimensionsverfeinerung
 - ein Abschnitt in der ausgewählten Dimension wird durch einen vollständigen Schnitt durch D verfeinert
 - **Split-Beispiel:** $3 \times 3 \times 2$ Bereiche (18 Grid-Blöcke) \rightarrow $3 \times 4 \times 2$ Bereiche (24 Grid-Blöcke)
- Ziele: Erhaltung der Topologie
 - effiziente Unterstützung aller Fragetypen
 - vernünftige Speicherplatzbelegung



3-dim. Datenraum $D = A1 \times A2 \times A3$

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

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

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

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

Veranschaulichung eines Split-Vorganges im Intervall v_2



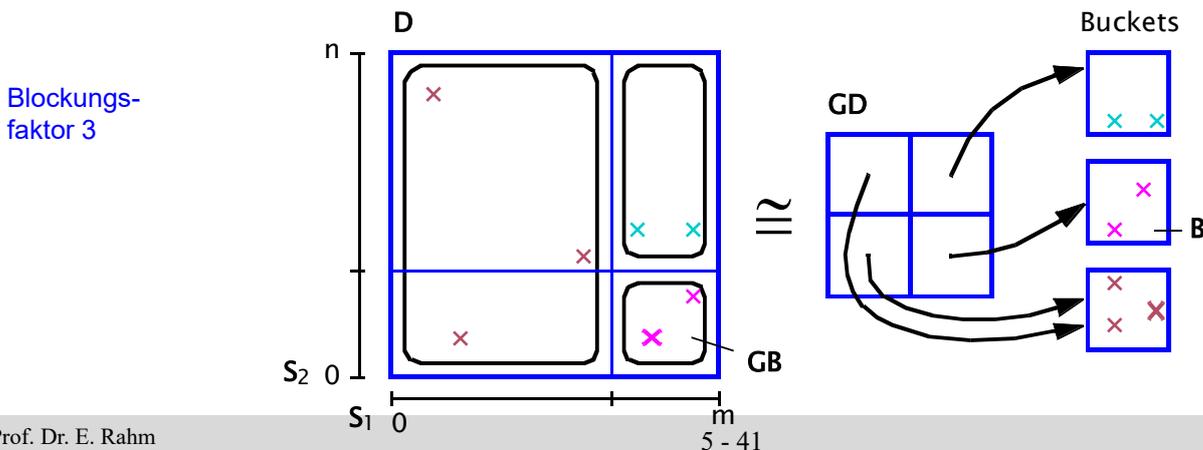
Grid-File (2)

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 Menge der Buckets
- Bucket: Speicherung der Objekte eines oder mehrerer Gridblöcke

Eigenschaften

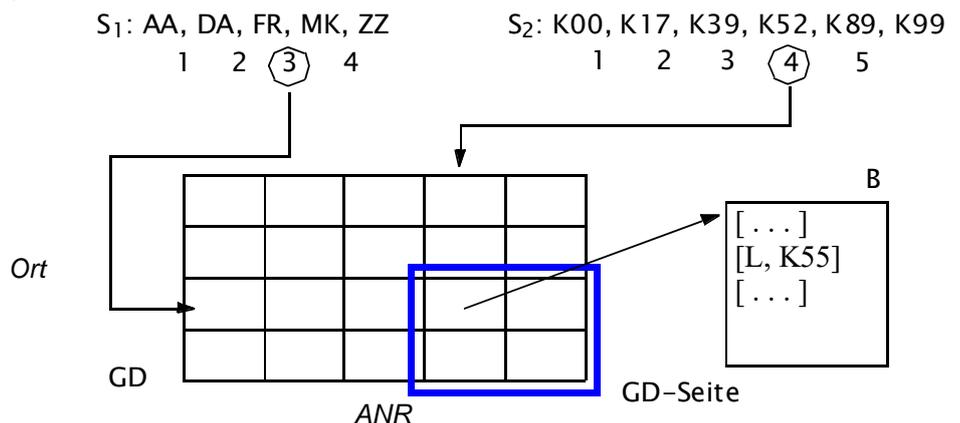
- 1:1-Beziehung zwischen Gridblock GB und Element von GD
- Element von GD = Pointer zu Bucket B
- n:1-Beziehung zwischen GB und B



Grid File - Suchfragen

Punktfrage (exact match)

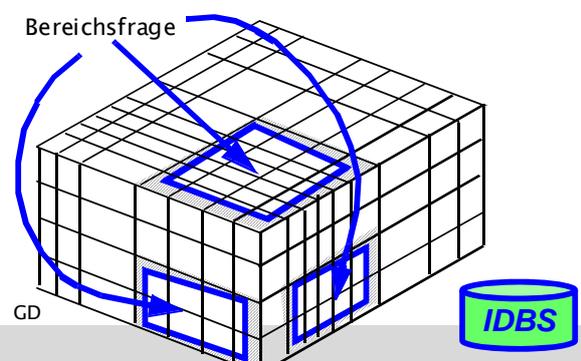
```
SELECT *
FROM PERS
WHERE ORT = 'L'
AND ANR = 'K55'
```



- 2 Plattenzugriffe: Grid-Directory + Bucket
unabhängig von Werteverteilungen, #Operationen und #Sätze

Bereichsfrage

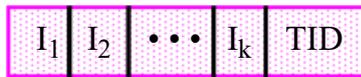
- Bestimmung der Skalierungswerte in jeder Dimension
- Berechnung der qualifizierten GD-Einträge
- Zugriff auf die GD-Seite(n) und Holen der referenzierten Buckets



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 **MBR**: minimale **b**egrenzende (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

I_j = geschlossenes Intervall bzgl. Dimension j

TID: Verweis auf Objekt

Zwischenknoteneintrag:



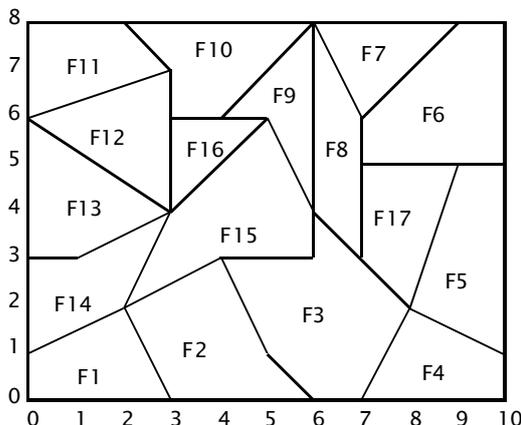
Intervalle beschreiben kleinstes umschreibendes Rechteck für alle in PID enthaltenen Objekte

PID: Verweis auf Nachfolgerseite (nächste Baumebene)

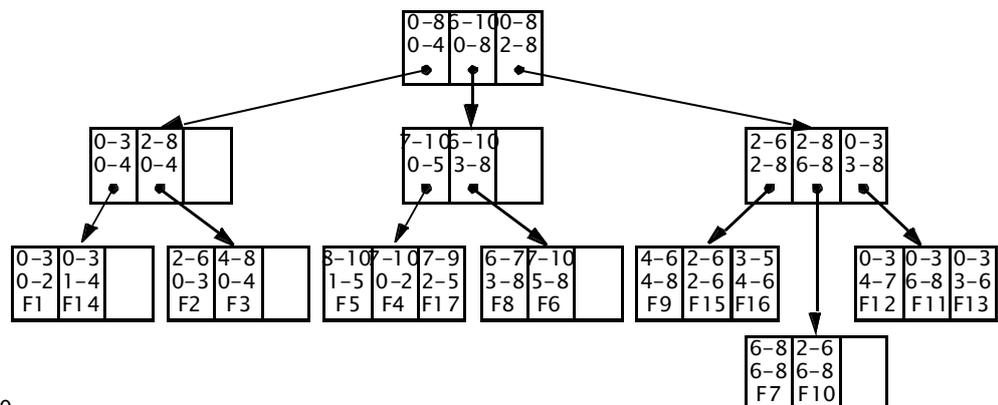


Abbildung beim R-Baum

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 Dokumentkolektionen

■ 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

- 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

d1	d2	Invertierter Index	
		Begriff	Vorkommen
Dieses objektorientierte Datenbanksystem unterstützt nicht nur multimediale Objekte, sondern ist überhaupt phänomenal.	Objektorientierte Systeme unterstützen die Vererbung von Methoden.	Datenbanksystem	d1
		Methode	d2
		multimedial	d1
		objektorientiert	d1, d2
		phänomenal	d1
		System	d2
		überhaupt	d1
		unterstützen	d1, d2
		Vererbung	d2

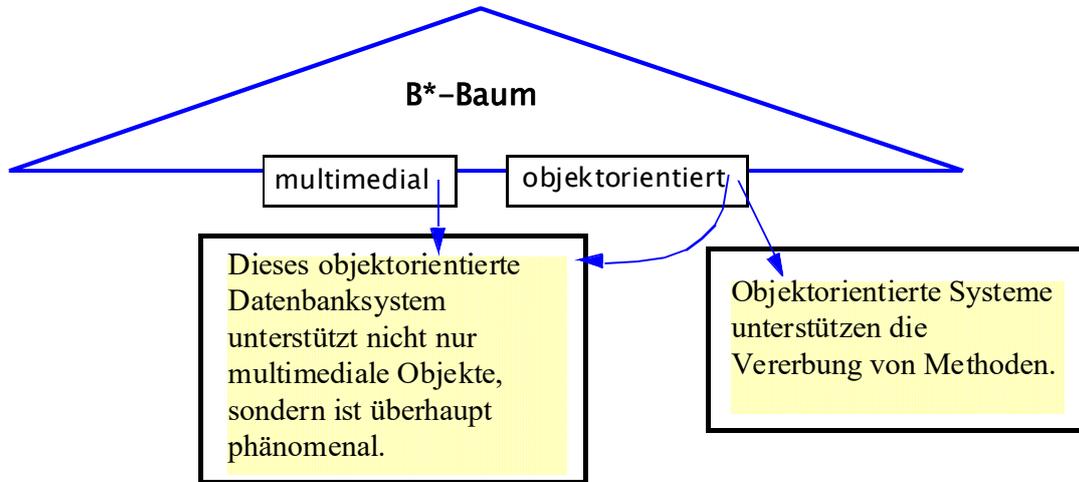
■ Zugriffskosten werden durch Datenstruktur zur Verwaltung der invertierten Liste bestimmt (B*-Baum, Hash-Verfahren ...)



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 die selbe 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. ...

D1

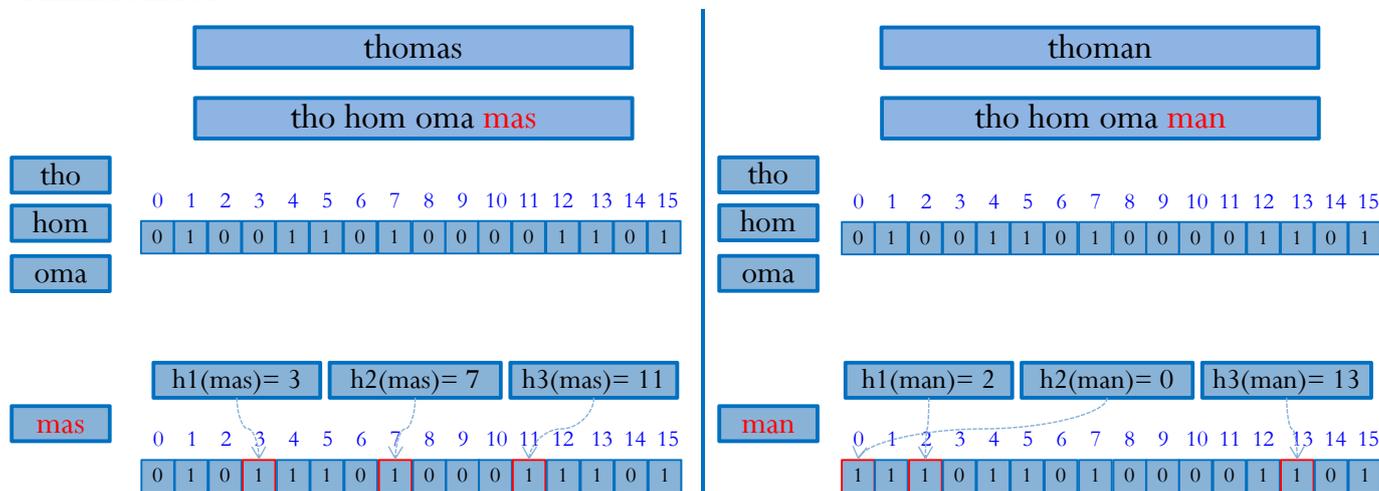
Dieses objektorientierte Datenbanksystem unterstützt nicht nur multimediale Objekte, sondern ist überhaupt phänomenal.

D2



Matching mit Bitvektoren

- verwandte Aufgabe: Matching mit Bitvektoren
 - bestimme ähnliche Dokumente
 - bestimme Matching für mit Bitvektoren verschlüsselte Daten, z.B. Personenangaben (privacy-preserving record linkage)
- Match-Ähnlichkeit über Grad der Überlappung der Bitvektoren, z.B. Jaccard-Ähnlichkeit



$$\text{Sim}_{\text{Jaccard}}(\mathbf{r1}, \mathbf{r2}) = (\mathbf{r1} \wedge \mathbf{r2}) / (\mathbf{r1} \vee \mathbf{r2})$$

$$\text{Sim}_{\text{Jaccard}}(\mathbf{r1}, \mathbf{r2}) = 7/11=0,64$$



Matching mit Bitvektoren (2)

- Performance-Problem bei sehr großen Datenmengen / vielen Bitvektoren
 - unoptimiert: quadratischer Aufwand
- Beschleunigung durch Eingrenzung des Suchraumes und/oder Parallelisierung
- Eingrenzung des Suchraumes
 - Blocking
 - Indexierung/Filterung
- Blocking
 - Partitionierung der Sätze (z.B. Dokumente aus bestimmtem Gebiet oder Publikationszeitraum)
 - Vergleich nur zwischen Sätzen ähnlicher Partitionen
- Indexierung/Filterung für minimale Ähnlichkeit
 - Längensfilter
 - Test auf minimale Überlappung mit Präfix
 - spezielle Indexstrukturen, z.B. Signaturbäume, Multibit Trees
 - Nutzung der Dreiecksungleichung für metrische Ähnlichkeitsmaße



Bitvektoren: Längenfilter

- nur Bitvektoren (kodierte Sätze/Dokumente) ähnlicher Länge können ähnlich sein
 - Länge bzw. Kardinalität: Anzahl gesetzter Bits
- für minimale Jaccard-Ähnlichkeit t gilt:

$$\text{Sim}_{\text{Jaccard}}(x, y) \geq t \Rightarrow |x| \geq |y| * t$$

- Beispiel für $t=0.8$

ID	Bit vector	card.
B	1 0 1 0 0 0 0 0 1 1 0 0 0	4
C	0 0 0 1 1 1 1 1 1 1 0 0 0	7
A	0 1 0 1 1 1 1 1 1 1 0 0 0	8
D	0 1 0 1 1 1 1 1 1 1 0 0 1	9

length filter
 $7 * 0.8 = 5.6 > 4$

- Satz B mit Länge 4 kann kein Match mit C und allen Sätzen größerer Länge eingehen, z.B., A



Zusammenfassung

- Standard-Indexstruktur in DBS: B*-Baum ("the ubiquitous B*-tree")
 - direkter vs. indirekter; clustered vs. non-clustered Index
 - direkter und sortiert sequenzieller 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
 - 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
 - Suchraumeingrenzung, z.B. für bestimmte Ähnlichkeitsfunktionen (Längenfilter)

