Implementierung von Datenbanksystemen

Kapitel 5: Eindimensionale Zugriffspfade

Kapitel 5: Eindimensionale Zugriffspfade

Eindimensionale Zugriffspfade

- Ziele
  - Entwurfsprinzipien für Zugriffspfade auf die Sätze einer Tabelle, bei denen ein Suchkriterium unterstützt wird
  - Abbildungsmöglichkeiten für hierarchische Zugriffsanforderungen
- Wichtige Kenngrößen:
  - n = Anzahl der Sätze eines Satztyps
  - b = mittlere Anzahl der Sätze pro Seite (Blockungsfaktor)
  - q = Anzahl der Treffer einer Anfrage
  - N_S = Anzahl der Seitenzugriffe
  - N_B = Anzahl der Blattseiten eines B*-Baums
  - h_B = Höhe des B*-Baums

Anforderungen an Zugriffspfade

- Folgende Arten von Zugriffen müssen unterstützt werden:
  - Sequentieller Zugriff auf alle Sätze eines Satztyps (Scan)
    Select * From Pers
  - Sequentieller Zugriff in Sortierreihenfolge eines Attributs
    ... Order by Name
  - Direkter Zugriff über den Primärschlüssel
    ... Where Pnr = 0815
  - Direkter Zugriff über einen Sekundärschlüssel
    ... Where Beruf = 'Programmierer'
  - Direkter Zugriff über zusammengesetzte Schlüssel und komplexe Suchausdrücke (Wertintervalle, ...)
    ... Where Gehalt Between 50K And 100K
  - Navigierender Zugriff von einem Satz zu einer dazugehörigen Satzmenge desselben oder eines anderen Satztyps
    ... Where P.Pnr = A.Pnr

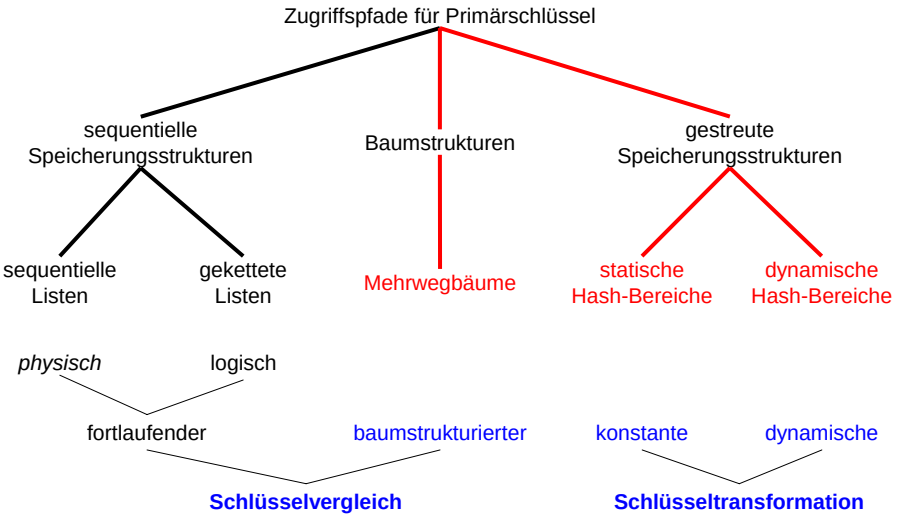
Anforderungen an Zugriffspfade

- Wenn kein geeigneter Zugriffspfad vorhanden ist, sind alle Zugriffsarten durch fortlaufende Suche (Scan) abzuwickeln
- Scan:
  - muss von allen DBMS unterstützt werden!
  - ist ausreichend / effizient bei:
    - kleinen Satztypen (z. B. ≤ 5 Seiten)
    - Anfragen mit großen Treffermengen (z. B. > 3%)
  - DBMS kann Prefetching zur Scan-Optimierung nutzen

Kapitel 5: Eindimensionale Zugriffspfade

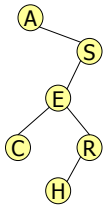
- Ziele
- Anforderungen und Klassifikation
- Zugriffspfade für Primärschlüssel
  - Mehrwegbäume
    - B-Bäume
    - B*-Bäume
  - Hash-Verfahren
    - Statische Verfahren
    - Erweiterbares Hashing
- Zugriff über Sekundärschlüssel
  - Einstiegs- und Verknüpfungsstruktur
  - Einsatz von Zeigerlisten und komprimierten Bitlisten
- Hierarchische Zugriffspfade
- Verallgemeinerte Zugriffspfadstruktur

Klassifikation

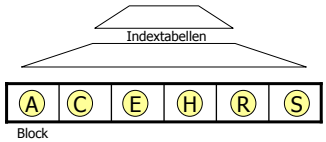


Mehrwegbäume

- Ähnliche Konzepte:
  - Binäre Suchbäume
    - geeignete Regeln der Seitenzuordnung fehlen



  - Vorfahr: ISAM
    - Sätze in physisch sequentiellen Blockstrukturen
    - mehrstufige Indextabellen
    - statische Organisation
    - aufwändiger Änderungsdienst
    - periodische Reorganisation

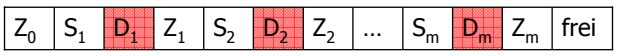


B-Baum, B*-Baum

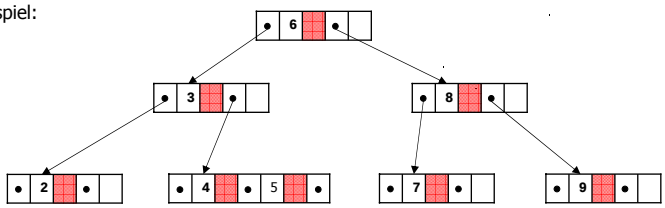
- Merkmale:
  - Knoten = Seite = Transporteinheit zum Externspeicher
  - referenzierte und materialisierte Speicherung der Datensätze
  - dynamische Reorganisation durch Splitten und Mischen von Seiten
  - Balancierte Struktur
    - unabhängig von Schlüsselmenge
    - unabhängig von Einfügereihenfolge
- Funktion
  - direkter Schlüsselzugriff
  - sortiert sequentieller Zugriff
- Realisierung von Index-organisierten Tabellen
  - oft nach Primärschlüssel geordnet
  - Cluster-Bildung durch eingebettete Datensätze

B-Baum

- Def.: 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:



  Z_i = Zeiger Sohnseite
  S_i = Schlüssel
  D_i = Daten des Satzes oder Verweis auf den Satz (materialisiert oder referenziert)
- Beispiel:

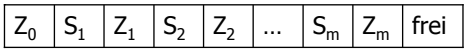


bei 4KB Seiten:

Z=4 B, S=4 B, D=92 B	⇒	100 B pro Eintrag	⇒	ca. 40 Söhne
Z=4 B, S=4 B, D=4 B	⇒	12 B pro Eintrag	⇒	ca. 330 Söhne

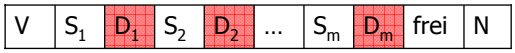
B*-Baum

- Def.: Ein B*-Baum vom Typ (k, 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. Jedes Blatt hat mindestens k* Einträge.
  3. Jeder Zwischenknoten hat höchstens 2k+1 Söhne. Jedes Blatt hat höchstens 2k* Einträge.
- Zwischenknoten:

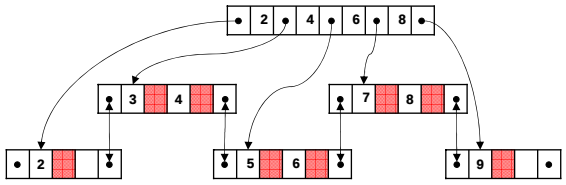


Z_i = Zeiger Sohnseite, S_i = Schlüssel

- Blattknoten:



D_i = Verweis auf Satz (materialisiert oder referenziert)
N = Nachfolger-Zeiger
V = Vorgänger-Zeiger



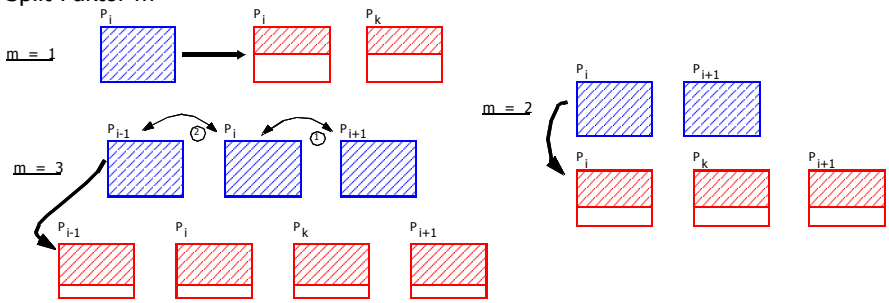
Z=4 B, S=4 B ⇒ 8 B pro Eintrag ⇒ ca. 500 Söhne bei 4 KB Seite

Mehrwegbäume

- Reduktion der Höhe durch Verbesserung der Baumbreite (fan-out)
  - Schlüsselkomprimierung
  - Nutzung von „Wegweisern" in B*-Bäumen
  - Präfix-B-Bäume
- Verbesserung des Belegungsgrades
  - verallgemeinertes Splittingverfahren mit Slit-Faktor m

Splitting bei B*-Bäumen

- Split-Faktor m



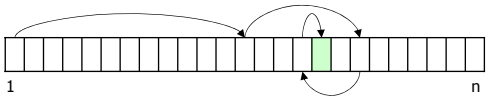
- Verbesserung des Belegungsgrades:

Belegung	m = 1	m = 2	m
worst case:	$\frac{1}{1+1}$	$\frac{2}{2+1}$	$\frac{m}{m+1}$
avg. case:	ln 2 (69 %)		$m \cdot \ln\left[\frac{m+1}{m}\right]$

$m \leq 3$: sonst zu aufwendig!

Suche in der Seite

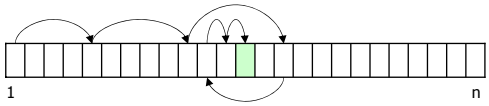
- Interne Struktur sei eine Liste mit n Einträgen
- sequentielle Suche
  - sortierte oder ungeordnete Schlüsselmenge
  - $C_{avg}(n) \approx n/2$
  - nur geringe Verbesserungen auf sortierten Listen (bei erfolgloser Suche)
- Binärsuche
  - wesentlich effizienter (Divide-and-Conquer-Strategie)
  - Voraussetzung: Sortierung und Einträge fester Länge
  - $C_{avg}(n) \approx \log_2(n+1) - 1$ für große n



Suche in der Seite

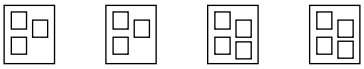
- Sprungsuche
  - Voraussetzung: Sortierung und Einträge fester Länge
  - Prinzip:
  - zunächst wird Liste in Sprüngen von m Einträgen überquert, um Abschnitt zu lokalisieren, der ggf. den gesuchten Schlüssel enthält
  - danach wird der Schlüssel im gefundenen Abschnitt nach irgendeinem Verfahren gesucht
  - $C_{avg}(n) = \frac{1}{2} a \frac{n}{m} + \frac{1}{2} b(m-1)$

    wenn ein Sprung a und ein sequentieller Vergleich b Einheiten kostet



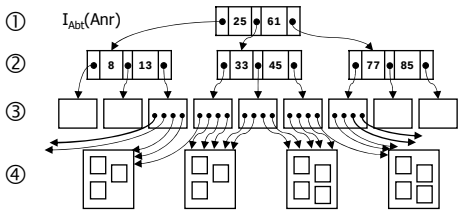
Zugriff zu allen Sätzen eines Satztyps

- Tabellen-Scan

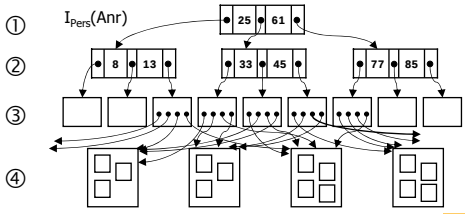


① Wurzelseite
② Zwischenseiten
③ Blattseiten
④ Datenseiten

- Index-Scan mit Cluster-Bildung



- Index-Scan ohne Cluster-Bildung



Kapitel 5: Eindimensionale Zugriffspfade

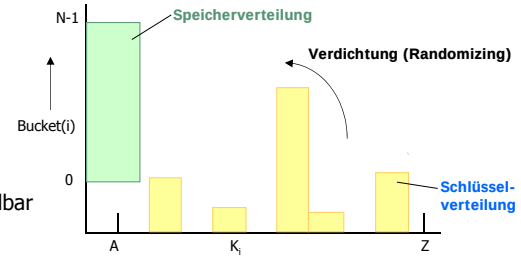
- Ziele
- Anforderungen und Klassifikation
- Zugriffspfade für Primärschlüssel
  - Mehrwegbäume
    - B-Bäume
    - B*-Bäume
  - Hash-Verfahren
    - Statische Verfahren
    - Erweiterbares Hashing
- Zugriff über Sekundärschlüssel
  - Einstiegs- und Verknüpfungsstruktur
  - Einsatz von Zeigerlisten und komprimierten Bitlisten
- Hierarchische Zugriffspfade
- Verallgemeinerte Zugriffspfadstruktur

Hash-basierte Zugriffspfade

- Merkmale:
  - Nutzung der Schlüsseltransformation als Entwurfsprinzip für Zugriffspfade auf die Sätze einer Tabelle bei denen ein Suchkriterium unterstützt wird.
  - Einschränkung auf Schlüsselzugriff, keine Bereichssuche usw.
  - Idealerweise 1 Seitenzugriff
- Wichtige Kenngrößen:
  - n = Anzahl der Sätze eines Satztyps
  - b = Anzahl der Sätze pro Bucket (Kapazität)
  - N = Anzahl der Buckets
  - β = Belegungsfaktor

Gestreute Speicherungsstrukturen (Hash-Verfahren)

- Hash-Funktion
  h: S → {0, 1, ..., N-1} S = Schlüsselraum, N = Größe des statischen Hash-Bereichs in Seiten (Buckets)
- Idealfall: h ist injektiv (keine Kollisionen)
  - nur in Ausnahmefällen möglich ('dichte' Schlüsselmenge)
  - jeder Satz kann mit einem Seitenzugriff gefunden werden
- Statische Hash-Bereiche mit Kollisionsbehandlung
  - vorhandene Schlüsselmenge K (K ⊆ S) soll möglichst gleichmäßig auf die N Buckets verteilt werden
  - Behandlung von Synonymen
    - Aufnahme im selben Bucket, wenn möglich
    - ggf. Anlegen und Verketten von Überlaufseiten
  - typischer Zugriffsfaktor: 1.1 bis 1.4
- Vielzahl von Hash-Funktionen anwendbar z. B. Divisionsrestverfahren, Faltung, Codierungsmethode, ...

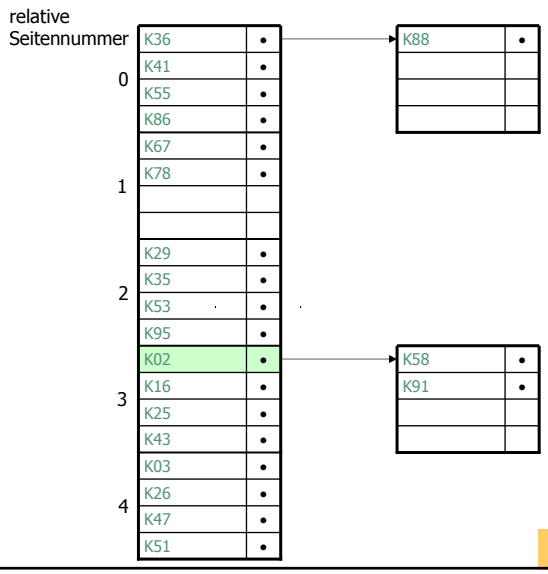


Statisches Hash-Verfahren mit Überlaufbereichen

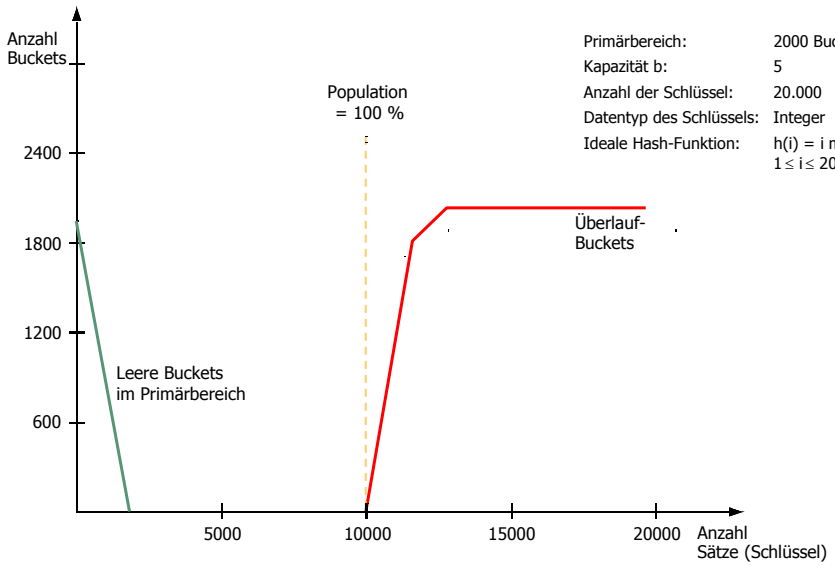
- Adressberechnung für Schlüssel K02:

  1101 0010
  ⊕ 1111 0000
  ⊕ 1111 0010
  1101 0000 = 208_{10}

  208 mod 5 = 3

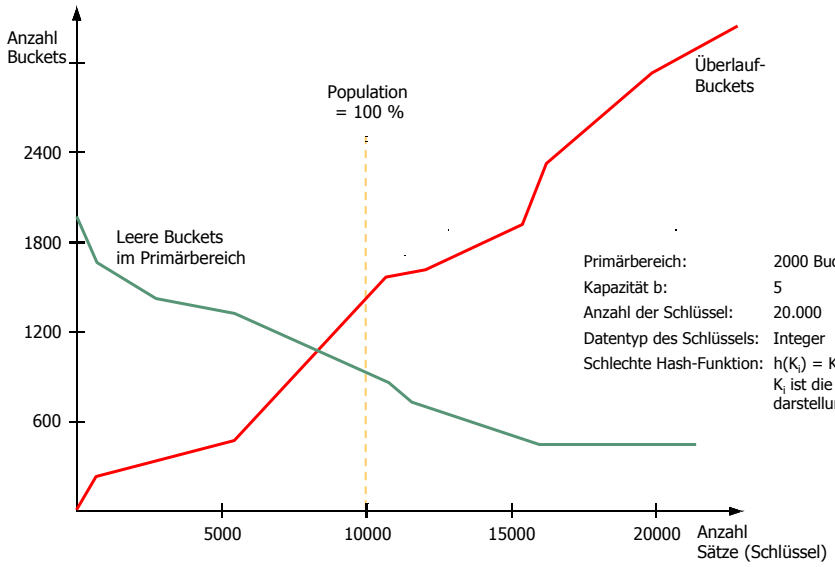


Belegung von Hash-Bereichen – Messung



Primärbereich:	2000 Buckets
Kapazität b:	5
Anzahl der Schlüssel:	20.000
Datentyp des Schlüssels:	Integer
Ideale Hash-Funktion:	$h(i) = i \bmod N$ $1 \le i \le 20.000$

Belegung von Hash-Bereichen – Messung



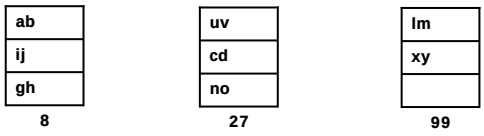
Primärbereich:	2000 Buckets
Kapazität b:	5
Anzahl der Schlüssel:	20.000
Datentyp des Schlüssels:	Integer
Schlechte Hash-Funktion:	$h(K_i) = K_i \bmod N$ K_i ist die Zeichen-darstellung von i

Externes Hashing ohne Überlaufbereiche

- Ziel:
  - Jeder Satz kann mit genau einem E/A-Zugriff gefunden werden
  - ➢ Gekettete Überlaufbereiche können nicht benutzt werden
- Statisches Hashing:
  - n Sätze, N Buckets mit Kapazität b
  - Belegungsfaktor $\beta = \frac{n}{N \cdot b}$
- Überlaufbehandlung
  - Open Adressing (ohne Kette oder Zeiger)
  - Bekannteste Schemata: Lineares Sondieren und Double Hashing
  - Sondierungsfolge für einen Satz mit Schlüssel k:
    - H(k) = (h_1(k), h_2(k), …, h_n(k))
    - bestimmt Überprüfungsreihenfolge der Buckets (Seiten) beim Einfügen und Suchen
    - wird durch k festgelegt und ist eine Permutation der Menge der Bucketadressen {0, 1, …, n-1}

Externes Hashing ohne Überlaufbereiche

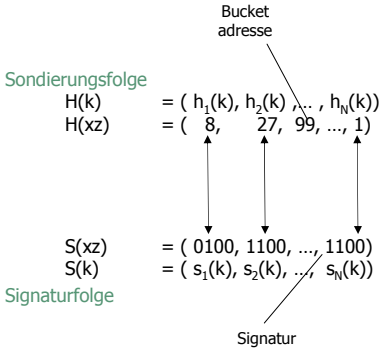
- Erster Versuch:
  - Aufsuchen und Einfügen von k = xy
  - Sondierungsfolge sei H(xy) = (8, 27, 99)



  - viele E/A-Zugriffe
  - Wie geschieht das Einfügen?

Externes Hashing mit Separatoren

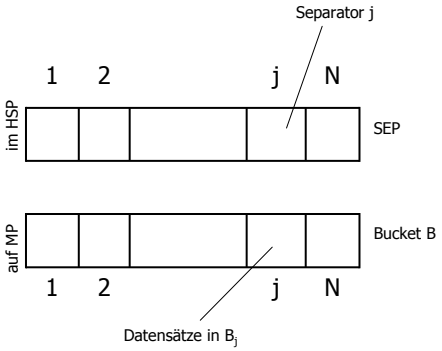
- Zugriffspfad für Primärschlüssel
- Einsatz von Signaturen
  - Jede Signatur $s_i(k)$ ist ein t-Bit Integer
  - Für jeden Satz mit Schlüssel k wird eine Signaturfolge benötigt: $S(k) = (s_1(k), s_2(k), \ldots, s_N(k))$
- Die Signaturfolge wird eindeutig durch k bestimmt
- Die Berechnung von S(k) kann durch einen Pseudozufallszahlengenerator mit k als Saat erfolgen (Gleichverteilung der t Bits wichtig)
- Nutzung der Signaturfolge zusammen mit der Sondierungsfolge
  - Bei Sondierung $h_i(k)$ wird $s_i(k)$ benutzt, i = 1,2,…,N
- Für jede Sondierung wird eine neue Signatur berechnet!



Quelle: [LK84]

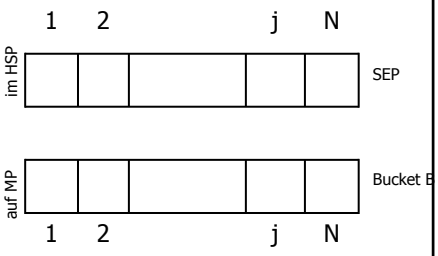
Externes Hashing mit Separatoren

- Einsatz von Separatoren
  - Ein Separator besteht aus t Bits
  - Separator j, j = 0, 1, 2, ..., N-1, gehört zu Bucket j
  - Eine Separatortabelle SEP enthält die N Separatoren und wird im Hauptspeicher gehalten.
  - Separatoren entscheiden darüber, in welchem Bucket seiner Sondierungsfolge ein Satz tatsächlich gespeichert wird.
- Speicherungsmodell



Nutzung der Separatoren

- In welcher Situation sind die Separatoren wichtig?
  - Bucket B_j wird r-mal (r > b) sondiert, d.h. r Sätze sollten nach ihrer Sondierungsfolge in B_j gespeichert werden.
  - Kapazität b des Buckets reicht nicht aus.
  - Mindestens (r – b) Sätze müssen abgewiesen werden, d.h. sie müssen das nächste Bucket in ihrer Sondierungsfolge aufsuchen.



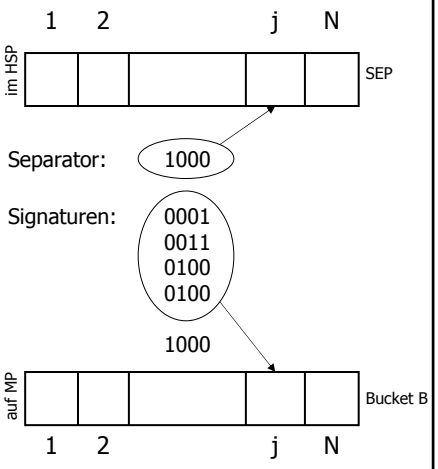
- Beispiel: r=5, t=4

Signaturen: 0001, 0011, 0100, 0100, 1000 → für Bucket B_j

Nutzung der Separatoren

- Für die Entscheidung, welche Sätze im Bucket gespeichert werden, sind die r Sätze nach ihren momentanen Signaturen zu sortieren.
- Sätze mit niedrigen Signaturen werden in B_j gespeichert.
- Sätze mit hohen Signaturen müssen weitersuchen.
- Eine Signatur, die die Gruppe der niedrigen eindeutig von der der höheren Signaturen trennt, wird als Separator j für B_j in SEP aufgenommen. Separator j enthält den niedrigsten Signaturwert der Sätze, die weitersuchen müssen.
- ➢ Ein Separator partitioniert also die r Sätze von B_j.

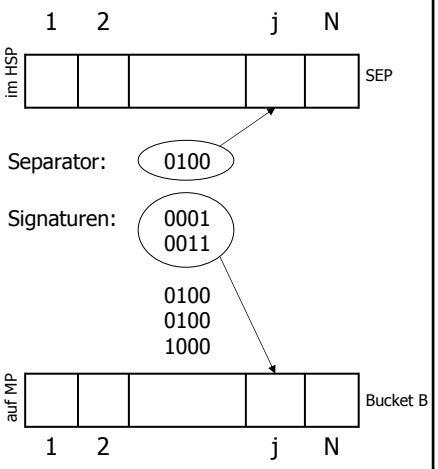
- Beispiel: b=4, Aufteilung 4:1



Nutzung der Separatoren

- Wenn die ideale Partitionierung (b, r – b) nicht gewählt werden kann, wird eine der folgenden versucht:
  (b-1,r-b+1), (b-2,r-b+2), …, (0,r)
- ➢ Ein Bucket mit Überlaufsätzen kann weniger als b Sätze gespeichert haben.

- Beispiel: b=3, Aufteilung 2:3



Aufsuchen und Einfügen

- Aufsuchen
  - In der Signaturfolge S(k) werden die $s_i(k)$ mit $SEP[h_i(k)]$, i=1,2,…,n im Hauptspeicher verglichen.
  - Sobald ein $SEP[h_i(k)] > s_i(k)$ gefunden wird, ist die richtige Bucketadresse lokalisiert.
  - Das Bucket wird eingelesen und durchsucht.
  - Wenn der Satz nicht gefunden wird, existiert er nicht.
  - ➢ Es ist genau ein E/A-Zugriff erforderlich

- Einfügen
  - Kann Verschieben von Sätzen und Ändern von Separatoren erfordern.
  - Wenn für einen Satz $h_i(k) = j$ und $s_i(k) < SEP[j]$ gilt, muss er in Bucket B_j eingefügt werden.
  - Falls B_j schon voll ist, müssen ein oder mehrere Sätze verschoben und SEP[j] entsprechend aktualisiert werden.
  - Alle verschobenen Sätze müssen dann in Buckets ihrer Sondierungs-folgen wieder eingefügt werden.
  - Dieser Prozess kann kaskadieren
  - β nahe bei 1 ist unsinnig, da die Einfügekosten explodieren; Empfehlung: $\beta < 0.8$

Beispiel: Startsituation

- Initialisierung der Separatoren mit 2^t-1
  - Separator eines Buckets, das noch nicht übergelaufen ist, soll größer als alle Signaturen sein
  - ➢ einfachere Algorithmen
  - daher Bereich der Signaturen: 0, 1, …, 2^t-2

1111

Key	Sign.
ab	0100

8

1111

Key	Sign.
ef	0010
cd	0101

18

1111

Key	Sign.
uv	0101

27

1111

Key	Sign.
lm	0100
xy	0110

99

- Einfügen von k = gh mit $h_1(gh) = 18$, $s_1(gh) = 1110$

1111

Key	Sign.
ab	0100

8

1111

Key	Sign.
ef	0010
cd	0101
gh	1110

18

1111

Key	Sign.
uv	0101

27

1111

Key	Sign.
lm	0100
xy	0110

99

Beispiel: Bucketüberlauf

1111

Key	Sign.
ab	0100

8

1111

Key	Sign.
ef	0010
cd	0101
gh	1110

18

1111

Key	Sign.
uv	0101

27

1111

Key	Sign.
lm	0100
xy	0110

99

- Einfügen von k = ij mit $h_1(ij) = 18$, $s_1(ij) = 0101$
- k = gh muss weiter sondieren, z.B. $h_2(gh) = 99$, $s_2(gh) = 1010$

1111

Key	Sign.
ab	0100

8

1110

Key	Sign.
ef	0010
cd	0101
ij	0101

18

1111

Key	Sign.
uv	0101

27

1111

Key	Sign.
lm	0100
xy	0110
gh	1010

99

Beispiel: Bucketüberlauf

- Situation nach weiteren Einfügungen und Löschungen

1000

Key	Sign.
ab	0100

8

1110

Key	Sign.
ef	0010
cd	0101
ij	0101

18

1111

Key	Sign.
uv	0101
mn	1001

27

1000

Key	Sign.
lm	0010
xy	0110

99

- Einfügen von k = qr mit H(qr) = (8,18,…) und S(qr) = (1011,0011,…)
- Sondierungs- und Signaturfolgen von cd und ij seien:
  H(cd) = (18,27,…) und S(cd) = (0101,1011,…)
  H(ij) = (18,99,8,…) und S(ij) = (0101,1110,0100,…)

1000

Key	Sign.
ab	0100
ij	0110

8

0101

Key	Sign.
ef	0010
qr	0011

18

1111

Key	Sign.
uv	0101
mn	1001
cd	1011

27

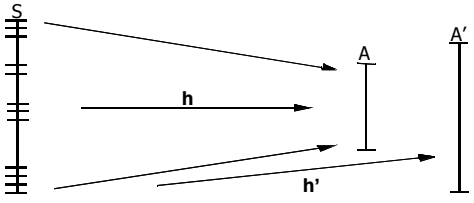
1000

Key	Sign.
lm	0010
xy	0110

99

Dynamische Hash-Verfahren

- Wachstumsproblem bei statischen Verfahren
  - Statische Allokation von Speicherbereichen: Speicherausnutzung?
  - Bei Erweiterung des Adressraums: Re-Hashing

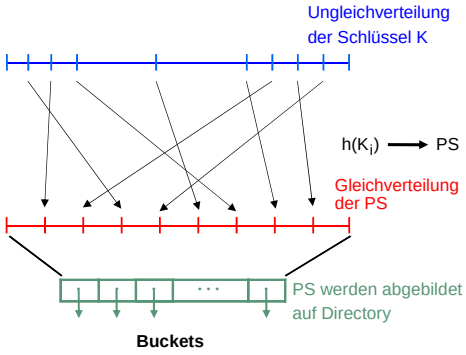


  - Alle Sätze erhalten eine neue Adresse
    - Kosten
    - Verfügbarkeit
    - Adressierbarkeit

- Entwurfsziele:
  - Dynamische Struktur erlaubt Wachstum und Schrumpfung des Hash-Bereichs (Datei)
  - 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)
  - Lösungsvorschläge mit und ohne Index (Hilfsdaten)

Erweiterbares Hashing

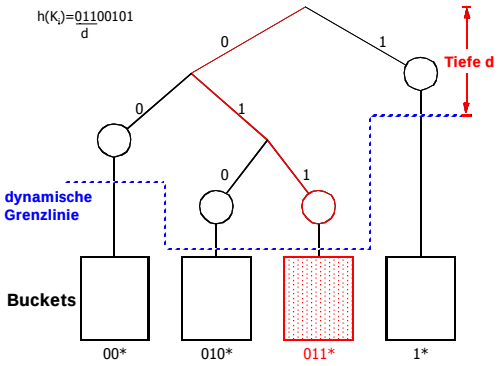
- **Lösungsidee: Verknüpft**
  - Split- und Mischtechniken von Seiten der B-Bäume zur Konstruktion eines dynamischen Hash-Bereichs
  - Adressierungstechnik der Digitalbäume zum Aufsuchen eines Speicherplatzes



- **Prinzipielle Vorgehensweise**
  - Die einzelnen Bits eines Schlüssels steuern den Weg durch den zur Adressierung benutzten Digitalbaum.
  - $K_i = (b_0, b_1, b_2, \ldots)$. Es ist prinzipiell möglich, die Bitfolge von K_i direkt für die Adressierung heranzuziehen. Bei Ungleichverteilung der Schlüssel ist dann ein unausgewogener Digitalbaum zu erwarten.
  - Da Digitalbäume keinen Mechanismus zur Balancierung ihrer Höhe besitzen, muss die Ausgewogenheit "von außen" aufgezwungen werden.
  - $h(K_i) = (b_0, b_1, b_2, \ldots)$. Die Verwendung von $h(K_i)$ als sog. Pseudoschlüssel (PS) soll bessere Gleichverteilung gewährleisten.
  - Gleichverteilung der Pseudoschlüssel (PS) impliziert minimale Höhe des Digitalbaumes

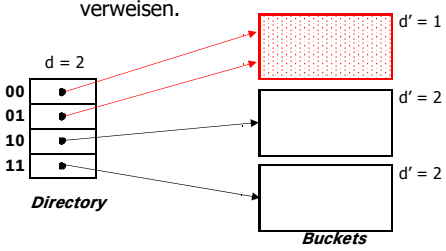
Erweiterbares Hashing

- Prinzipielle Abbildung der Pseudoschlüssel
  - Zur Adressierung eines Buckets sind d Bits erforderlich, wobei sich dafür i.A. eine dynamische Grenzlinie variierender Tiefe ergibt.
  - Die Digitalbaum-Adressierung bricht ab, sobald ein Bucket den ganzen Teilbaum aufnehmen kann.
- ausgeglichener Digitalbaum garantiert minimales d_{max}
- Dynamisches Wachsen und Schrumpfen des Hash-Bereiches
  - Buckets werden erst bei Bedarf bereitgestellt
  - Knoten unterschiedlicher Tiefe verweisen auf ein Bucket
- hohe Speicherplatzbelegung möglich



Erweiterbares Hashing

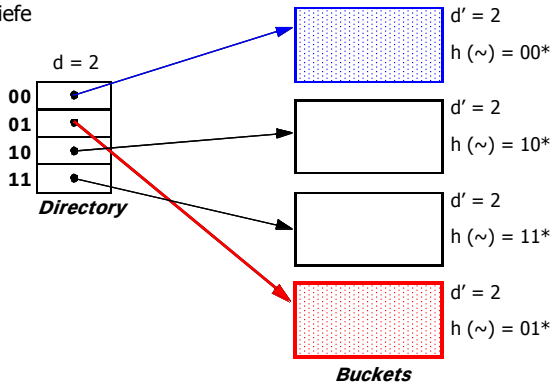
- keine Überlaufbereiche, jedoch Zugriff über Directory (Index)
  - Binärer Digitalbaum der Höhe d wird durch einen (2^d)-Digitalbaum der Höhe 1 implementiert (Trie der Höhe 1 mit 2^d Einträgen)
  - d wird festgelegt durch den längsten Pfad im binären Digitalbaum
  - In einem Bucket werden nur Sätze gespeichert, deren PS in den ersten d' Bits übereinstimmen (d' = lokale Tiefe)
  - d = MAX (d'): d Bits des PS werden zur Adressierung verwendet (d = globale Tiefe)
  - Directory enthält 2^d Einträge
- Speicherungsstruktur:
  - Der Trie lässt sich als Directory oder Adressverzeichnis auffassen.
  - Die d Bits von $h(K_i)$ zeigen im Directory auf einen Eintrag mit der Adresse des Buckets, das den Schlüssel K_i enthält.
  - Wenn d' < d, können (benachbarte) Einträge auf dasselbe Bucket verweisen.



➢ Kosten der direkten Suche: max. 2 Seitenzugriffe

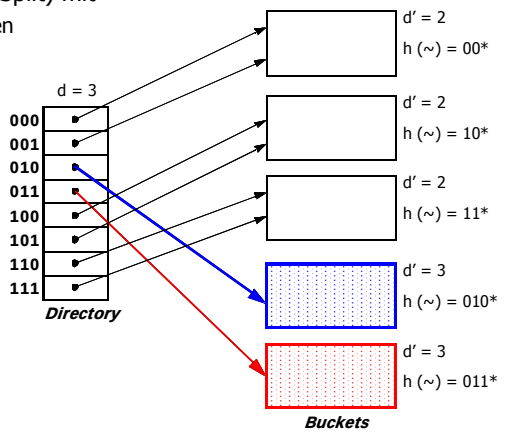
Splitting von Buckets

- Fall 1:
  - Überlauf eines Buckets, dessen lokale Tiefe kleiner als die globale Tiefe d ist
  - Anlegen eines neuen Buckets (Split) mit
    - lokaler Neuverteilung der Daten
    - Erhöhung der lokalen Tiefe
    - lokaler Korrektur der Verweise im Directory



Splitting von Buckets

- Fall 2:
  - Überlauf eines Buckets, dessen lokale Tiefe gleich der globalen Tiefe ist
  - Anlegen eines neuen Buckets (Split) mit
    - lokaler Neuverteilung der Daten (Erhöhung der lokalen Tiefe)
    - Verdopplung des Directories (Erhöhung der globalen Tiefe)
    - globaler Korrektur/ Neuverteilung der Verweise im Directory



Vergleich der wichtigsten Zugriffsverfahren

Zugriffsverfahren	Speicherungsstruktur	Direkter Zugriff	Sequentielle Verarbeitung	Änderungsdienst (Ändern ohne Aufsuchen)
Fortlaufender Schlüsselvergleich	Sequentielle Liste Gekettete Liste	$O(n)=10^4$ $O(n) \approx 5 \cdot 10^5$	$O(n) \approx 2 \cdot 10^4$ $O(n) \approx 10^6$	$O(1) \leq 2$ $O(1) \leq 3$
Baumstrukturierter Schlüsselvergleich	Balancierte Binärbäume Mehrwegbäume	$O(\log_2 n) \approx 20$ $O(\log_k n) \approx 3 – 4$	$O(n) \approx 10^6$ $O(n) \approx 10^6$ *	$O(1) = 2$ $O(1) = 2$
Konstante Schlüsseltrans-formationsverfahren	Externes Hashing mit separatem Überlaufbereich Externes Hashing mit Separatoren	$O(1) \approx 1.1 – 1.4$ $O(1) = 1$	$O(n \log_2 n)$** $O(n \log_2 n)$ **	$O(1) \approx 1.1$ $O(1) = 1$ (+D)
Variable Schlüsseltrans-formationsverfahren	Erweiterbares Hashing Lineares Hashing	$O(1) = 2$ $O(1) > 1$	$O(n \log_2 n)$** $O(n \log_2 n)$**	$O(1) \approx 1.1$ (+R) $O(1) < 2$

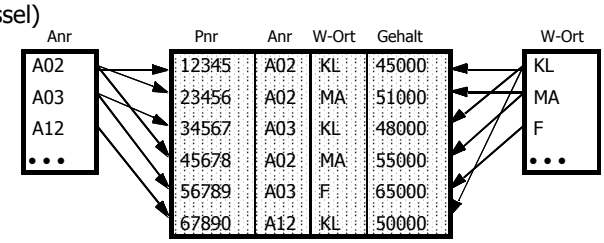
Beispielangaben für $n = 10^6$ (D: Dominoeffekt, R: Reorganisationskosten)
* Bei Clusterbildung bis zu Faktor 50 weniger
** Physisch sequentielles Lesen, Sortieren und sequentielles Verarbeiten der gesamten Sätze

Kapitel 5: Eindimensionale Zugriffspfade

- Ziele
- Anforderungen und Klassifikation
- Zugriffspfade für Primärschlüssel
  - Mehrwegbäume
    - B-Bäume
    - B*-Bäume
  - Hash-Verfahren
    - Statische Verfahren
    - Erweiterbares Hashing
- Zugriff über Sekundärschlüssel
  - Einstiegs- und Verknüpfungsstruktur
  - Einsatz von Zeigerlisten und komprimierten Bitlisten
- Hierarchische Zugriffspfade
- Verallgemeinerte Zugriffspfadstruktur

Zugriffspfade für Sekundärschlüssel

- Suche nach Sätzen mit einem vorgegebenen Wert eines nicht-identifizierenden Attributs (Sekundärschlüssel)
- Ergebnis ist eine Satzmenge



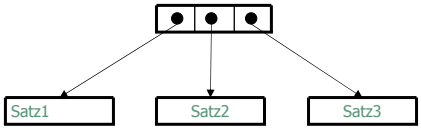
- Realisierung: Einstiegsstruktur + Verknüpfungsstruktur
  - Primärschlüssel-Zugriffspfade als Einstiegsstruktur auf Satzmengen anwendbar
  - Prinzipiell lassen sich alle Verknüpfungsstrukturen für Satzmengen heranziehen
  - vor allem Verwendung von B*-Bäumen und spezielle Invertierungstechniken
- Standardlösung bei der Invertierung sind sequentielle Verweislisten (oft OID-Listen oder TID-Listen genannt)
  - effiziente Durchführung von Mengenoperationen
  - kosteneffektive Wartung

Verknüpfungsstrukturen für Satzmengen

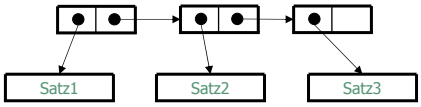
- Materialisierte Speicherung
  - Physische Nachbarschaft der Sätze (Cluster-Bildung, Listen)

  - Verkettung der Sätze

- Ausgelagerte Hilfsdateien
  - Physische Nachbarschaft der Zeiger (Invertierung)

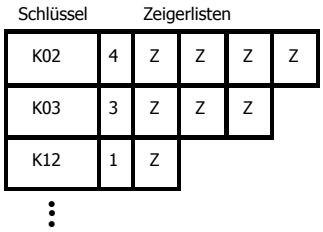


  - Verkettung der Zeiger

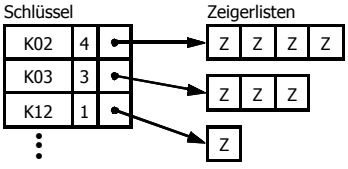


Zugriffspfade für Sekundärschlüssel

- Häufige Realisierung: Invertierung
  - Trennung der Zugriffspfaddaten von den Datensätzen (referenzierte Speicherung)
  - Verweis Z realisiert als TID, DBK/PPP, ...
- Zwei mögliche Darstellungsmethoden:
  1. Gemeinsame Verwaltung der Suchstruktur und der Zeigerlisten
     - relativ kurze Zeigerlisten erforderlich!



  2. 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

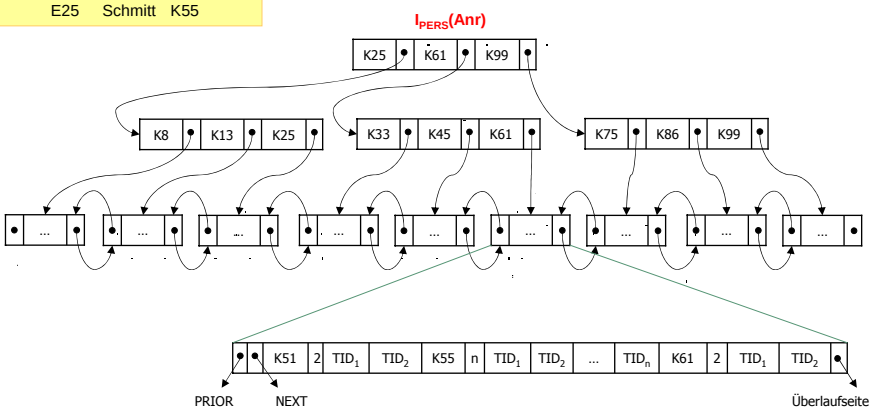


Zugriffspfade für Sekundärschlüssel

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!)
- mit Vorwärts- und Rückwärtsverkettung



Invertierung mit Bitlisten

- Adressierung von Datensätzen oder Dokumenten (Information Retrieval)
  - über Zuordnungstabelle ZT
  - direkt bei fester Länge und fortlaufender Speicherung (b Sätze pro Seite)
- Markierungen der Bitliste entsprechen Einträgen von ZT oder berechenbaren Adressen
- Bitmatrix für Attribut A
  - Attribut A habe j Attributwerte a_1, ..., a_j
- Speicherung als vertikale Bitlisten erlaubt Indexierung von mehrwertigen Attributen (Bsp: Warenkorb mit Produkten)

	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		

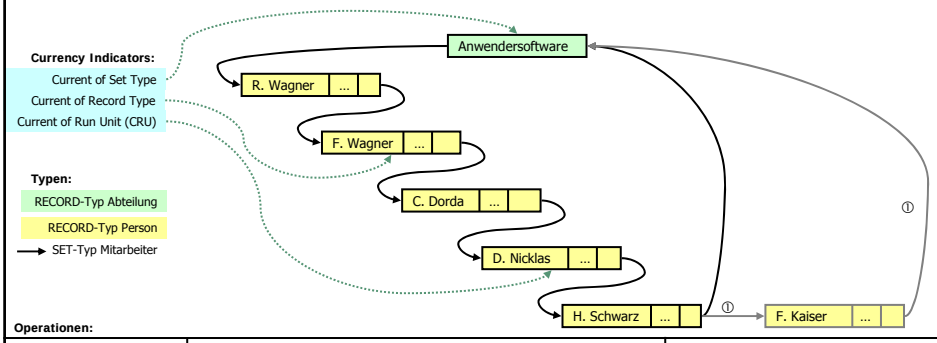
Invertierung mit Bitlisten

- 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 dünn besetzten Matrizen
    - von Multimedia-Objeklten (Image, Audio, Video, ...)
    - von Objekten in Geo-DB, ...
- Viele Komprimierungstechniken verfügbar
  - Bitfolgen-Komprimierung (run length compression)
  - Mehr-Modus-Komprimierung
  - Blockkomprimierung

Kapitel 5: Eindimensionale Zugriffspfade

- Ziele
- Anforderungen und Klassifikation
- Zugriffspfade für Primärschlüssel
  - Mehrwegbäume
    - B-Bäume
    - B*-Bäume
  - Hash-Verfahren
    - Statische Verfahren
    - Erweiterbares Hashing
- Zugriff über Sekundärschlüssel
  - Einstiegs- und Verknüpfungsstruktur
  - Einsatz von Zeigerlisten und komprimierten Bitlisten
- Hierarchische Zugriffspfade
- Verallgemeinerte Zugriffspfadstruktur

Exkurs: Netzwerk-Datenmodell



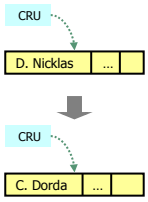
Operationen:

STORE <F. Kaiser, ...>

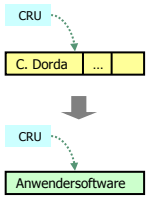
ERASE Person

MODIFY ...

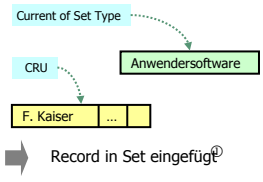
FIND Person USING <C. Dorda>



FIND OWNER WITHIN Mitarbeiter



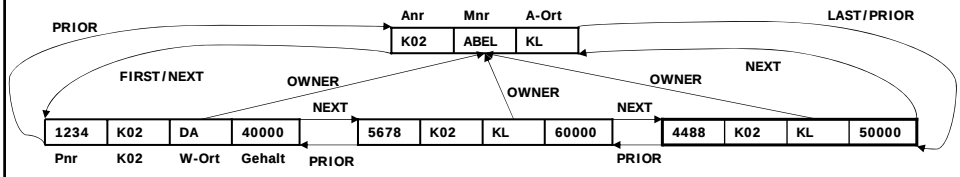
CONNECT Person TO Mitarbeiter

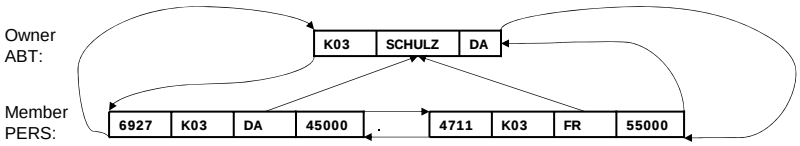


DISCONNECT Person FROM Mitarbeiter

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

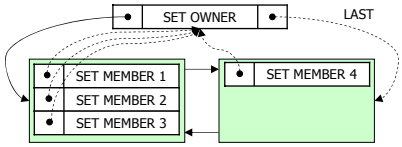




- drei Implementierungen für unterschiedliche Leistungsanforderungen

Implementierung Hierarchischer Zugriffspfade

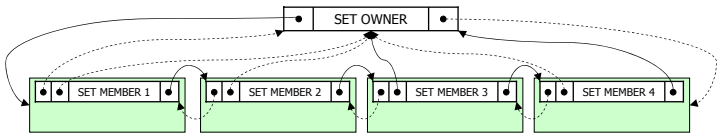
- Sequentielle Liste auf Seitenbasis



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

Implementierung Hierarchischer Zugriffspfade

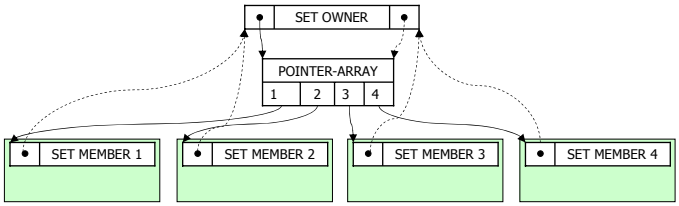
- Gekettete Liste



- Bewertung:
  - Vorteile bei Mitgliedschaft des Member-Satztyps in mehreren Sets
  - billiger Wechsel auf andere Set-Ausprägungen
  - sequentieller Zugriff (geringfügig) schneller als bei Pointer-Array
  - nur gut in kleinen Set-Ausprägungen

Implementierung Hierarchischer Zugriffspfade

- Pointer-Array-Struktur



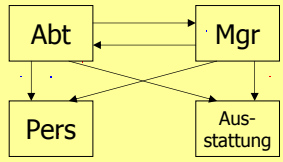
- Bewertung:
  - stabiles Verhalten
  - Verhalten unabhängig vom Set-Wachstum und Set-Reihenfolge
  - 'Standard'-Verfahren bei unscharfen Informationen über Set-Größe und Zugriffshäufigkeit

Kapitel 5: Eindimensionale Zugriffspfade

- Ziele
- Anforderungen und Klassifikation
- Zugriffspfade für Primärschlüssel
  - Mehrwegbäume
    - B-Bäume
    - B*-Bäume
  - Hash-Verfahren
    - Statische Verfahren
    - Erweiterbares Hashing
- Zugriff über Sekundärschlüssel
  - Einstiegs- und Verknüpfungsstruktur
  - Einsatz von Zeigerlisten und komprimierten Bitlisten
- Hierarchische Zugriffspfade
- Verallgemeinerte Zugriffspfadstruktur

Verallgemeinerte Zugriffspfadstruktur

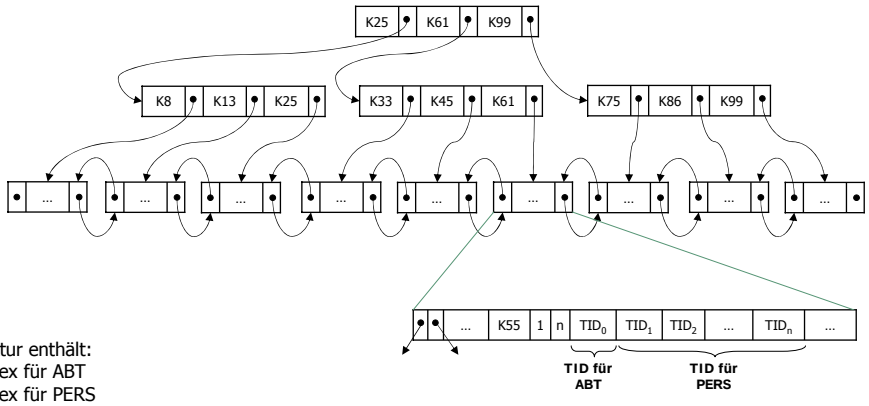
- Idee:
  - gemeinsame Verwendung einer Indexstruktur (B*-Baum) für mehrere Satztypen, für die Beziehungen (1:1, 1:n, n:m) über demselben Wertebereich (z. B. für Anr) definiert und durch Gleichheit von Attributwerten repräsentiert sind.



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

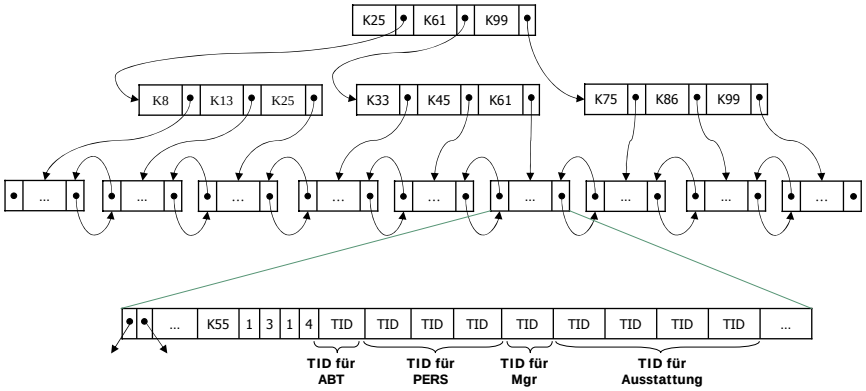
- Nutzung der Indexstruktur für
  - Primärschlüsselzugriff z.B. als I_{Abt}(Anr)
  - Sekundärschlüsselzugriff z.B. als I_{Pers}(Anr)
  - hierarchischen Zugriff z.B. von Abt(Anr) nach Pers(Anr) oder in umgekehrter Richtung
  - Verbundoperationen (Join) z. B. von Abt.Anr = Pers.Anr
- Kombinierte Realisierung 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
- Index für PERS
- Link für ABT-PERS mit direktem Zugriff von:
  - OWNER zu jedem MEMBER,
  - jedem MEMBER zu jedem anderen MEMBER
  - jedem MEMBER zum OWNER

B*-Baum als verallgemeinerte Zugriffspfadstruktur



Zugriffspfadstruktur umfasst:
- 4 Indexstrukturen
- 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

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

Zugriffspfade in kommerziellen Datenbanksystemen

DB2	B*-Baum (clustered, non-clustered), partitionierte Tabellen, …
Informix	B-Baum, statisches Hashing, ISAM, HEAP, …
Oracle	B*-Baum (mit Präfix-/Suffix-Komprimierung), (Join-) Cluster-Bildung, …
Sybase	B*-Baum (clustered, non-clustered), …
RDB (DEC)	B*-Baum (clustered, non-clustered), Hashing, Join-Cluster-Bildung, …
NonStop SQL (Tandem)	B*-Baum (clustered, non-clustered) mit Präfix-Komprimierung, …
UDS (Siemens)	B*-Baum, statisches Hashing, Cluster-Bildung (LIST), Invertierung (Pointer-Array), Kettung (CHAIN)

Zusammenfassung

- Cluster-Bildung optimiert (sortiert) sequentielle Zugriffe
- Standard-Zugriffspfadstruktur: B*-Baum (the ubiquitous B*-tree)
  - materialisierte und referenzierte Speicherung der Datensätze
  - Index-organisierte Tabelle mit Cluster-Bildung
- Schnellerer Schlüsselzugriff erfordert Hash-Verfahren
  - (nur) direkter Zugriff (= 1 Seitenzugriff)
  - Erweiterbares Hashing unterstützt stark wachsende Datenbestände ($\lesssim$ 2 Seitenzugriffe)
- Zugriffspfade für Sekundärschlüssel
  - Einstiegsstruktur: B*-Baum u.a.
  - Verknüpfungsstruktur: Zeigerlisten, Bitlisten
  - Bitlisten sind bei geringer Kardinalität des Wertebereichs hoch effizient
  - Viele Komprimierungsverfahren verfügbar
  - Unterstützung mengentheoretischer Operationen
- 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
  - auch Join-Index genannt
  - Unterstützung von Primärschlüssel-, Sekundärschlüssel- und hierarchischen Zugriffen

Literatur zu diesem Kapitel

[Fag79] Fagin, R., et. al: Extendible hashing - a fast access method for dynamic files. In: ACM Trans. Database Syst. 4:3. 1979.

[LK84] Larson, P.-A. and Kajla, A.: File organization: implementation of a method guaranteeing retrieval in one access. In: Comm. of the ACM 27,7 (1984).

[Gra07] Graefe, G.: Algorithms for merged indexes. In: Proc. BTW 2007.