9 Eindimensionale Zugriffspfade

Motivation von Zugriffspfaden

- Index-Scan versus Table-Scan
- Klassifikation von Verfahren

B/B-Baum*

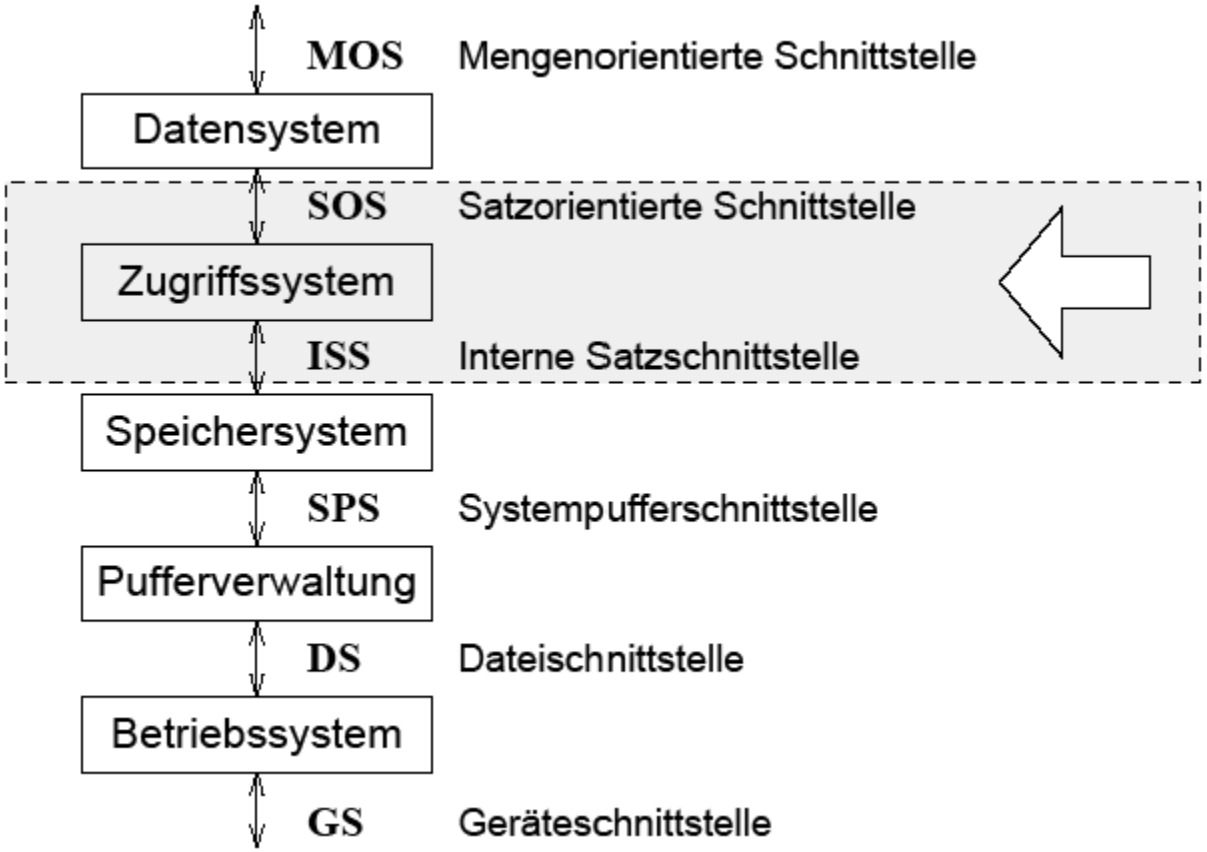
- Struktur und Operationen: Einfügen, Löschen

Bitmap-Indexstrukturen

- Vorteile gegenüber TID-Listen

Hash-Verfahren

- lineares Hashing, virtuelles Hashing, Hashing mit Separatoren, ...

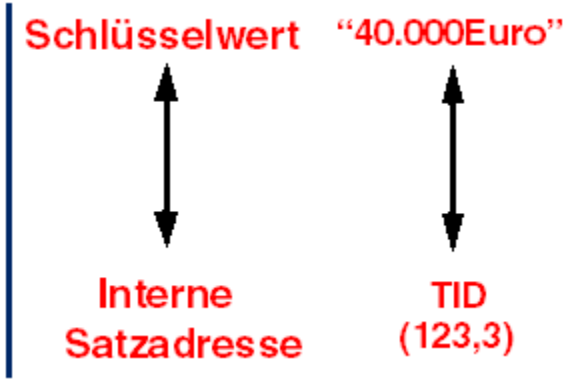

MOS
Mengenorientierte Schnittstelle
Datensystem
SOS
Satzorientierte Schnittstelle
Zugriffssystem
ISS
Interne Satzschnittstelle
Speichersystem
SPS
Systempufferschnittstelle
Pufferverwaltung
DS
Dateischnittstelle
Betriebssystem
GS
Geräteschnittstelle

Arten von Zugriffen

- Sequentieller Zugriff auf alle Sätze eines Satztyps (Scan)
- Sequentieller Zugriff in Sortierreihenfolge eines Attributes
- Direkter Zugriff über den Primärschlüssel (z.B.: Kennzeichen = “DD-EK 2332”)
- Direkter Zugriff über einen Sekundärschlüssel (z.B. Farbe = “silber” and Automarke = “VW”)
- Direkter Zugriff über zusammengesetzte Schlüssel und komplexe Suchausdrücke (Wertintervalle, ...)
- Navigierender Zugriff von einem Satz zu einer dazugehörigen Satzmenge desselben oder eines anderen Satztyps

Anforderungen an Zugriffspfade

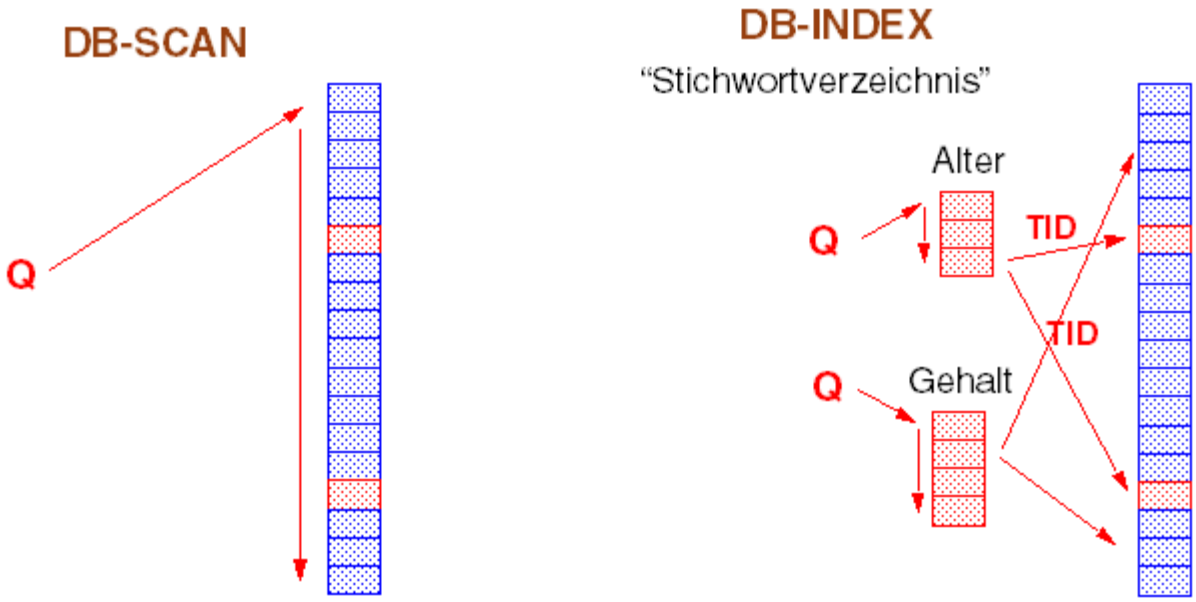
- effizientes (direktes) Auffinden von Datensätzen bzgl. inhaltlichen Kriterien
- Vermeiden von sequentiellem Durchsuchen aller Datensätze
- Erleichterung von Zugriffskontrollen durch vorgegebene Zugriffspfade (constraints)
- Erhaltung topologischer Beziehungen



Idee

- Einführung eines Zwischenschrittes

Pers(PID, NAME, ALTER, GEHALT, ...)



DB-Scan

- alle Blöcke müssen gelesen und alle Sätzen in den eingelesenen Seiten müssen hinsichtlich dem Suchkriterium untersucht werden
- wird von allen DBMS unterstützt
- ist ausreichend / effizient bei:
  - kleinen Satztypen (z. B. < 5 Seiten)
  - Anfragen mit großen Treffermengen (z. B. > 1 %)
- DBMS kann Prefetching zur Scan-Optimierung nutzen

Index

- zwei Klassen von Indexstrukturen
  1. Schlüsselwerte werden transformiert um die betreffenden Seiten/Blöcke zu ermitteln
  2. Schlüsselwerte werden redundant in einer eigenen Struktur gehalten und mit dem Suchkriterium verglichen
- ... wenn kein geeigneter Zugriffspfad vorhanden (oder dessen Nutzung nicht ökonomischer) ist, müssen alle Zugriffsarten durch einen SCAN abgewickelt werden

Bestandteile einer Indexstruktur

- Name des Zugriffspfades
- Typ des Zugriffspfades
  - Primärschlüssel-Index (Garantie der Eindeutigkeit)
  - Sekundärschlüssel-Index (mehrere Tupel für einen Schlüsselwert)
- Liste der betreffenden Attributnamen plus potentiell weitere Attribute
- optional: Sortierung

Schlüsselzugriff/Schlüsseltransformation

- Schlüsselzugriff: Zuordnung von Primär- oder Sekundärschlüsselwerten zu Adressen in Hilfsstruktur wie Indexdatei
  - Beispiel: indexsequentielle Organisation, B-Baum, KdB-Baum, ...
- Schlüsseltransformation: berechnet Tupeladresse durch Formel aus Primär- oder Sekundärschlüsselwerten (statt Indexeinträgen nur Berechnungsvorschrift gespeichert)
  - Beispiel: Hash-Verfahren

Statische Zugriffstruktur

- optimal nur bei bestimmter (fester) Anzahl von verwaltenden Datensätzen
- Beispiel
  - Adresstransformation für Personalausweisnummer p von Personen mit p mod 5
  - 5 Seiten, Seitengröße 1 KB, durchschnittliche Satzlänge 200 Bytes, Gleichverteilung der Personalausweisnummern für 25 Personen optimal, für 10.000 Personen nicht mehr ausreichend
- unterschiedliche Verfahren: Heap, indexsequentiell, indiziert-nichtsequentiell
- oft grundlegende Speichertechnik in RDBS für direkte Organisation
  - Vorteil: keine Hilfsstruktur, keine Adressberechnung

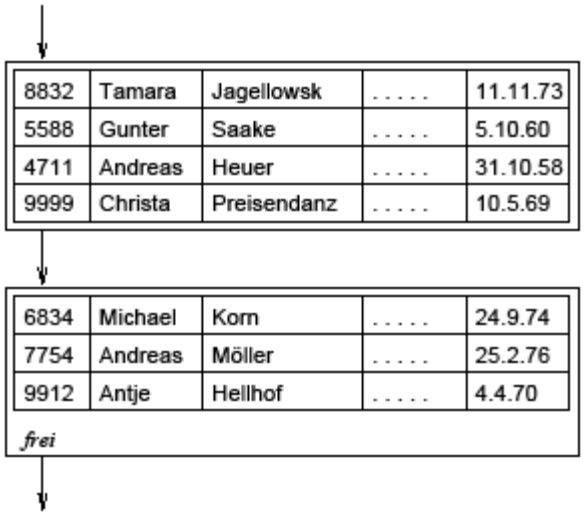
Dynamische Zugriffstruktur

- unabhängig von der Anzahl der Datensätze optimal
  - dynamische Adresstransformationsverfahren:
    --> dynamische Anpassung des Bildbereichs der Transformation
  - dynamische Indexverfahren: dynamische Anpassung der Anzahl der Indexstufen

Physische Dateiorganisation

Heap-Organisation

- völlig unsortierte Speicherung
- physische Reihenfolge der Datensätze entspricht der zeitlichen Reihenfolge der Aufnahme von Datensätzen
- Insert-Operation
  - Zugriff auf letzte Seite der Datei
  - Falls genügend freier Platz -> Satz anhängen
  - Ansonsten nächste freie Seite holen
- Delete-Operation
  - lookup, dann Löschbit setzen
- Lookup-Operation
  - sequenzielles Durchsuchen der Gesamtdatei
  - maximaler Aufwand (Heap-Datei meist zusammen mit Sekundärindex eingesetzt)
- Komplexitätsbetrachtung: Neuaufnahme von Daten O(1), Suchen O(n)



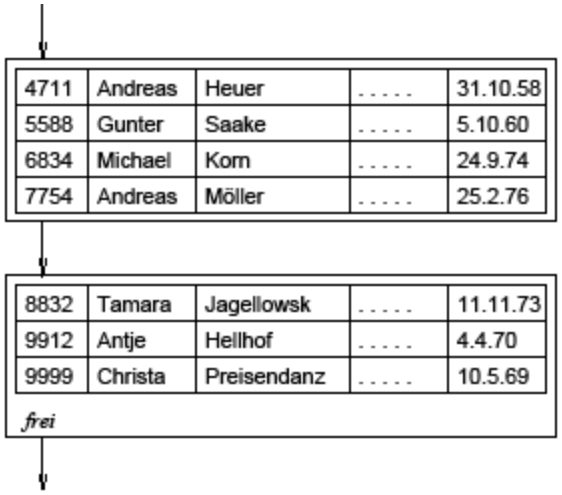
8832	Tamara	Jagellowsk	. . . . .	11.11.73
5588	Gunter	Saake	. . . . .	5.10.60
4711	Andreas	Heuer	. . . . .	31.10.58
9999	Christa	Preisendanz	. . . . .	10.5.69

6834	Michael	Korn	. . . . .	24.9.74
7754	Andreas	Möller	. . . . .	25.2.76
9912	Antje	Hellhof	. . . . .	4.4.70

frei

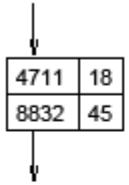
Prinzip

- Sortieres Speichern der Datensätze nach einem **anwendungsseitig** vorgegebenen Schlüsselkriterium
- Insert-Operation
  - Seite suchen und Datensatz einsortieren
  - Füllgrad: beim Anlegen oder sequenziellen Füllen einer Datei jede Seite nur bis zu gewissem Grad (etwa 66%) füllen
- Delete-Operation
  - lookup, dann Löschbit setzen
- normalerweise in Verbindung mit zusätzlichem Index --> indexsequenzielle Dateiorganisation

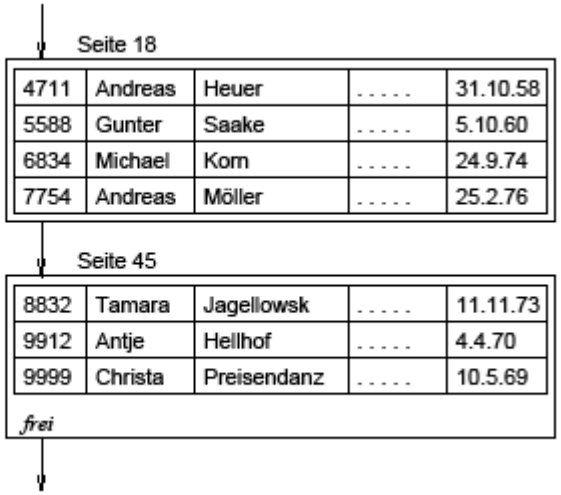


Prinzip

- sequenziell organisierte Hauptdatei
- zusätzliche Indexdatei
  - schnellerer Lookup
  - mehr Platzbedarf (für Index)
  - mehr Zeitbedarf (für Insert und Delete-Operationen)



4711	18
8832	45



Seite 18

4711	Andreas	Heuer	.....	31.10.58
5588	Gunter	Saake	.....	5.10.60
6834	Michael	Korn	.....	24.9.74
7754	Andreas	Möller	.....	25.2.76

Seite 45

8832	Tamara	Jagellowsk	.....	11.11.73
9912	Antje	Hellhof	.....	4.4.70
9999	Christa	Preisendanz	.....	10.5.69

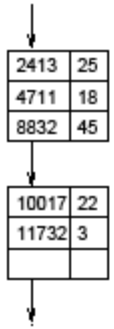
frei

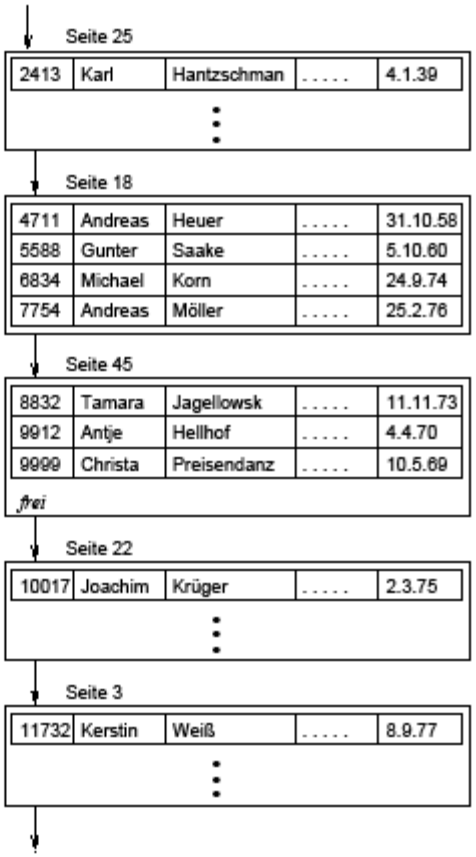
Organisation

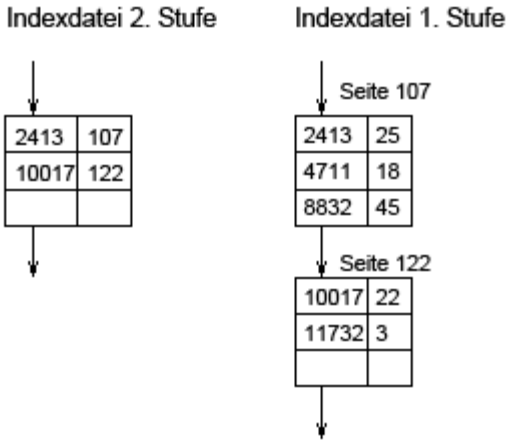
- mindestens zweistufiger Baum
  - Blattebene ist Hauptdatei (Datensätze)
  - jede andere Stufe ist Indexdatei mit Einträgen: (Primärschlüsselwert, Seitennummer)
  - zu jeder Seite in der Hauptdatei genau ein Index-Datensatz in der Indexdatei
- Zwang zu mehrstufigen Indexstrukturen, falls Seitengröße überstiegen wird

Aufbau der Indexdatei

- Indexdatei wiederum indexsequentiell verwalten
- Wurzel darf nur aus einer Seite bestehen







Lookup-Operation

- Gesucht wird Datensatz zum Schlüsselwert w
- Sequenzielles Durchlaufen der Indexdatei und Suche von (v_1,s) mit $v_1 \leq w$
  - (v_1, s) ist letzter Satz der Indexdatei
    --> Datensatz zu w kann höchstens auf dieser Seite gespeichert sein (wenn er existiert)
  - nächster Satz (v_2, s') im Index hat $v_2 > w$
    --> Datensatz zu w, wenn vorhanden, ist in Seite s gespeichert
- (v_1, s) überdeckt Zugriffsattributwert w

Insert-Operation

- Seite mit Lookup-Operation finden
- Falls Platz, Satz sortiert in gefundener Seite speichern
  Index anpassen, falls neuer Satz der erste Satz in der Seite
- Falls kein Platz, neue Seite von Freispeicherverwaltung holen
  Sätze der „zu vollen" Seite gleichmäßig auf alte und neue Seite verteilen; für neue Seite Indexeintrag anlegen (ggf. Anlegen einer Überlaufseite)

Delete-Operation

- Seite mit Lookup-Operation finden
- Satz auf Seite löschen (Löschbit setzen)
  - Falls erster Satz auf Seite --> Index anpassen
  - Falls Seite nach Löschen leer
    --> Index anpassen und Seite an Freispeicherverwaltung zurückgeben

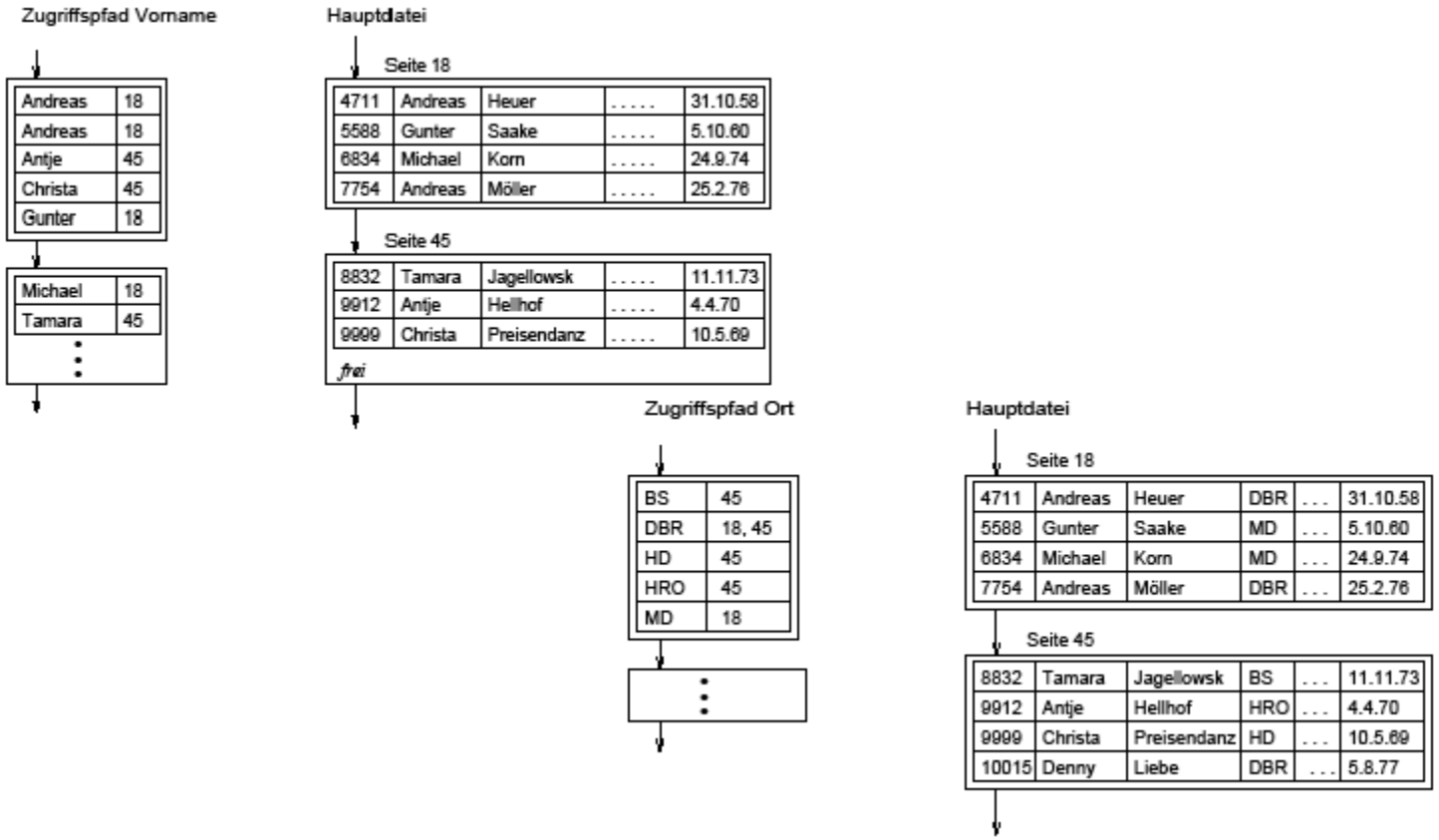
Bewertung

- stark wachsende Dateien: Zahl der linear verketteten Indexseiten wächst; automatische Anpassung der Stufenanzahl nicht vorgesehen
- stark schrumpfende Dateien: nur zögernde Verringerung der Index- und Hauptdatei-Seiten
- unausgeglichene Seiten in der Hauptdatei (unnötig hoher Speicherplatzbedarf, zu lange Zugriffszeit)

Idee für Organisation für einen Index

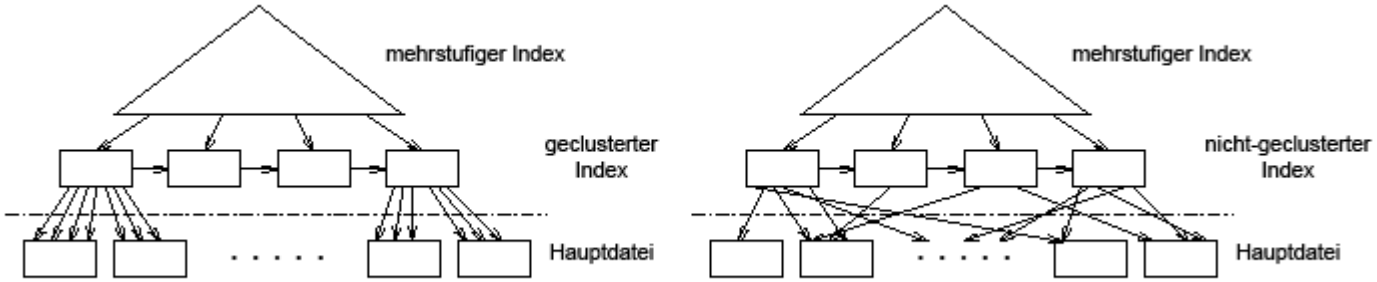
- analog zu einem Stichwortverzeichnis in einen Buch:
  für jeden Schlüsselwert, die Stellen, an denen der Wert auftritt
- Unterstützung von Sekundärschlüsseln
  --> mehrere Zugriffspfade (Sekundärindexe) pro Relation möglich
  - zu jedem Satz der Relation existiert ein Satz (Sekundärschlüsselwert, Seite/TID) im Index
  - Nicht-Eindeutigkeit: mehrere Einträge oder {Seite/TID}
- Mehrstufige Organisation, wobei höhere Indexstufen wieder indexsequentiell organisiert sind -> Baumverfahren mit dynamischer Stufenzahl
- Lookup-Operation
  - Schlüsselwert kann mehrfach auftreten
- Insert-Operation
  - Anpassung des Index-Eintrags erforderlich
- Delete-Operation
  - Eintrag aus dem Index entfernen (ggf. auch die Einträge auf höherer Ebene)

> Beispiel zu Sekundärindex



Klassifikation

- (Primär-)Index: bestimmt Dateiorganisationsform
  - unsortierte Speicherung von Tupeln: Heap-Organisation
  - sortierte Speicherung von internen Tupeln: sequentielle Organisation
  - gestreute Speicherung von internen Tupeln: Hash-Organisation
  - Speicherung in mehrdimensionalen Räumen: mehrdimensionale Dateiorganisationsformen
  - Normalfall: Primärschlüssel über Primärindex/geclusterter Index
- Sekundärindex
  - redundante Zugriffsmöglichkeit, zusätzlicher Zugriffspfad



Ziel

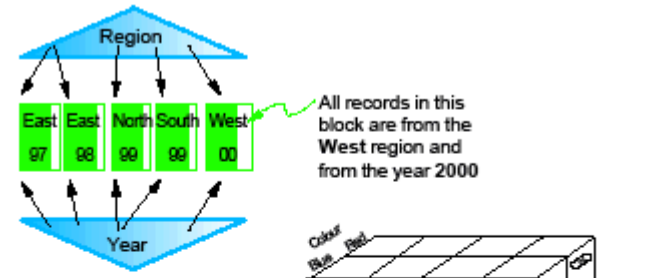
- Erhaltung der topologischen Struktur und Abbildung auf physisches Medium
- offensichtlich: nur ein geclusterter Index pro Relation (Primärindex)

Cluster-Verhältnis (cluster ratio)

- Grad des Clusterings in Prozent
- Cluster-Verhältnis nimmt ab,
  falls freier Platz pro Seite erschöpft ist

Multidimensionales Clustering

- über mehrere Richtungen
- erfordert multidimensionale
  Indexstrukturen !



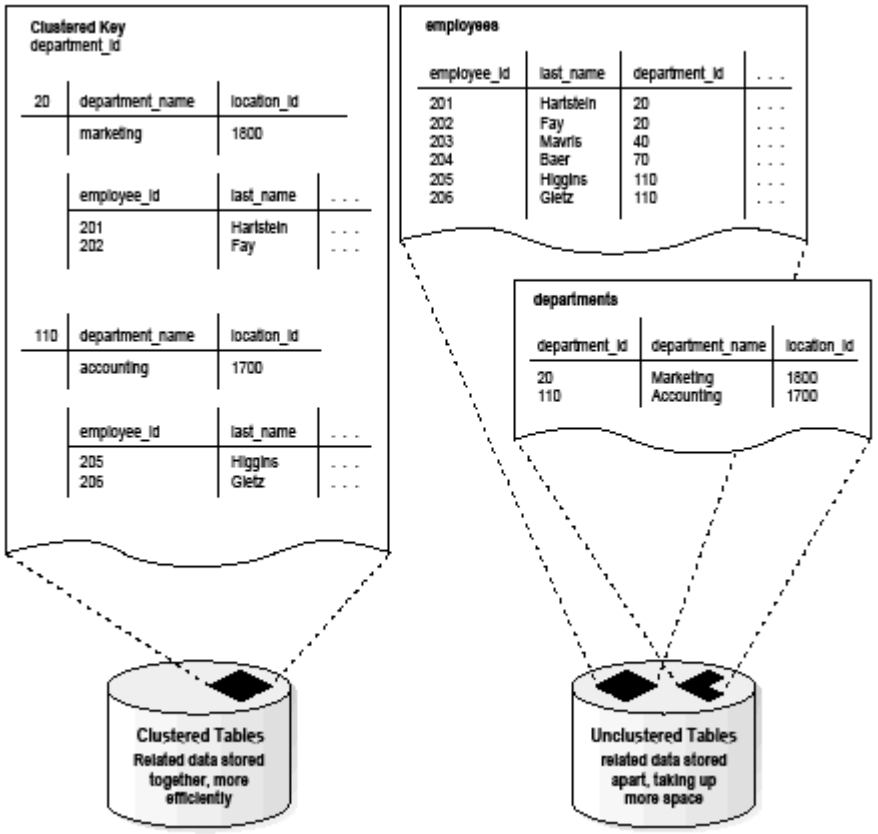

```
CREATE TABLE MDC1 (
   Date DATE,
   Province CHAR(2),
   Color VARCHAR(10),
   YearAndMonth generated as INTEGER(Date)/100, ... )
DIMENSIONS ( YearAndMonth, Province, Colour )
```

Cluster

- Menge von Relationen, bei denen die Einträge nach einem gemeinsamen Attribut organisiert werden
- Ballung basierend auf Fremdschlüsselattributen, d.h. Datensätze, die einen Attributwert gemeinsam haben, werden möglichst auf der gleichen Seite abgelegt.

Vorteil

- logisch zusammengehörige Tupel sind physisch an einem Block gespeichert



... am Beispiel von Oracle

```
create cluster AuftragCluster
   (Auftragsnr number(3))
   pctused 80 pctfree 5;
create table T_Auftrag(
   Auftragsnr number(3) primary key,
   ...)
   cluster AuftragCluster (Auftragsnr);
create table T_Auftragspositionen(
   Position number(3),
   Auftragsnr number(3) references T_Auftrag,
   …
   constraint AuftragPosKey
   primary key (Position, Auftragsnr))
   cluster AuftragCluster (Auftragsnr);
```

Indexierte Cluster

- entspricht normalem Index für den Cluster-Schlüssel

```
create index AuftragClusterIndex
   on cluster AuftragCluster
```

Datenbank Indexstrukturen

Formulierung in SQL

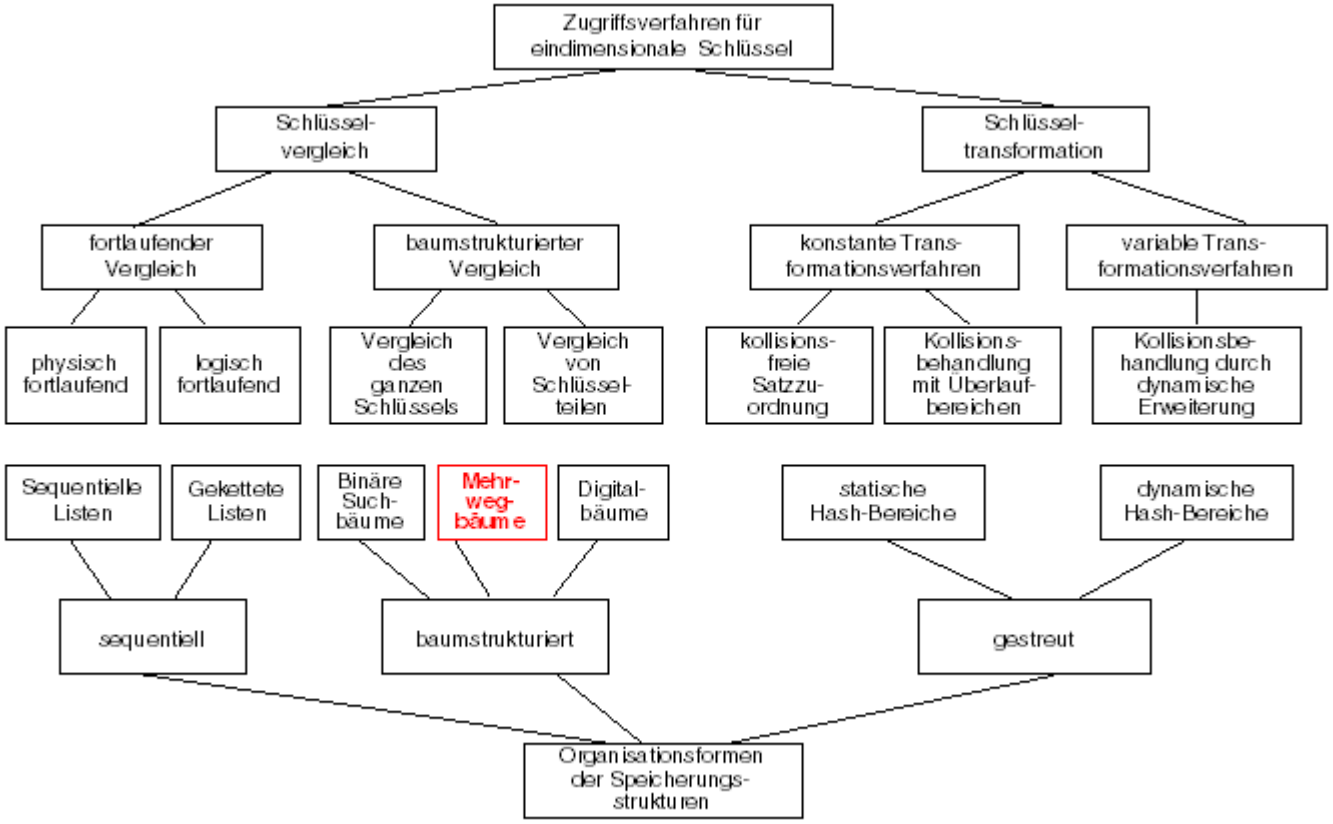
- CREATE UNIQUE INDEX pnr_idx ON pers (pnr) ALLOW REVERSE SCANS
  - ermöglicht bidirektionale Index-Scans (Standard)
- CREATE UNIQUE INDEX pnr_idx ON pers (pnr) INCLUDE (pname)
  - zusätzliche Spalten zur Vermeidung des Zugriffs auf Relation
- CREATE INDEX pgehalt_idx ON pers (gehalt)
- CREATE INDEX pgehalt_idx ON pers (gehalt) DISALLOW REVERSE SCANS COLLECT DETAILED
- CREATE INDEX alt_geh_idx ON pers (alter, gehalt)
  wichtig: unterschiedlich zu
  CREATE INDEX geh_alt_idx ON pers (gehalt, alter)
  (siehe Kapitel “Multidimensionale Indexstrukturen)

Idee

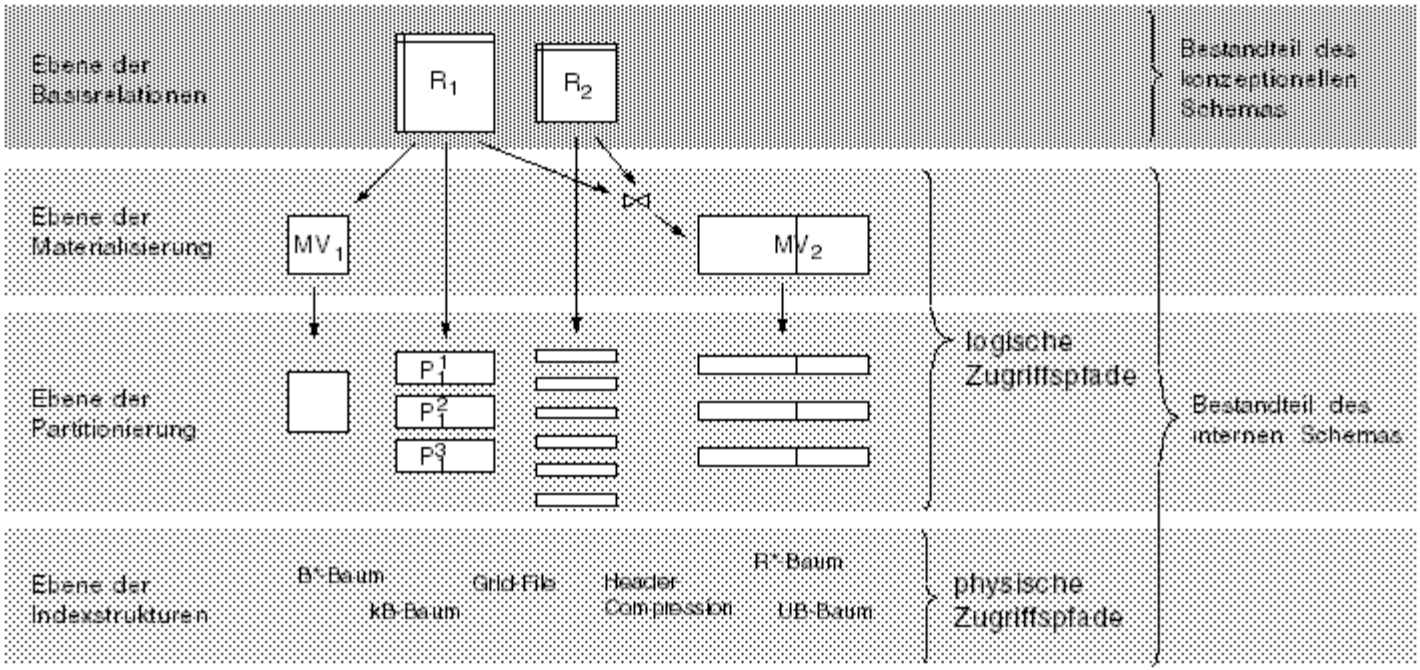
- Definition von Indexstrukturen auf Funktionen (z.B. Oracle)
- Benutzung von Anfragen, die exakt auf die gleiche Funktion bzw. auf “äquivalenten” algebraischen Ausdruck zurückgreifen
- Einschränkung
  Funktion ist als DETERMINISTIC gekennzeichnet

Beispiel

- CREATE INDEX idx ON table_x (a + b * (c - 1), a, b);
  wird benutzt, um folgende Anfrage zu unterstützen
  SELECT a FROM table_1 WHERE a + b * (c - 1) < 100;
- CREATE INDEX uppercase_idx ON employees (UPPER(first_name));

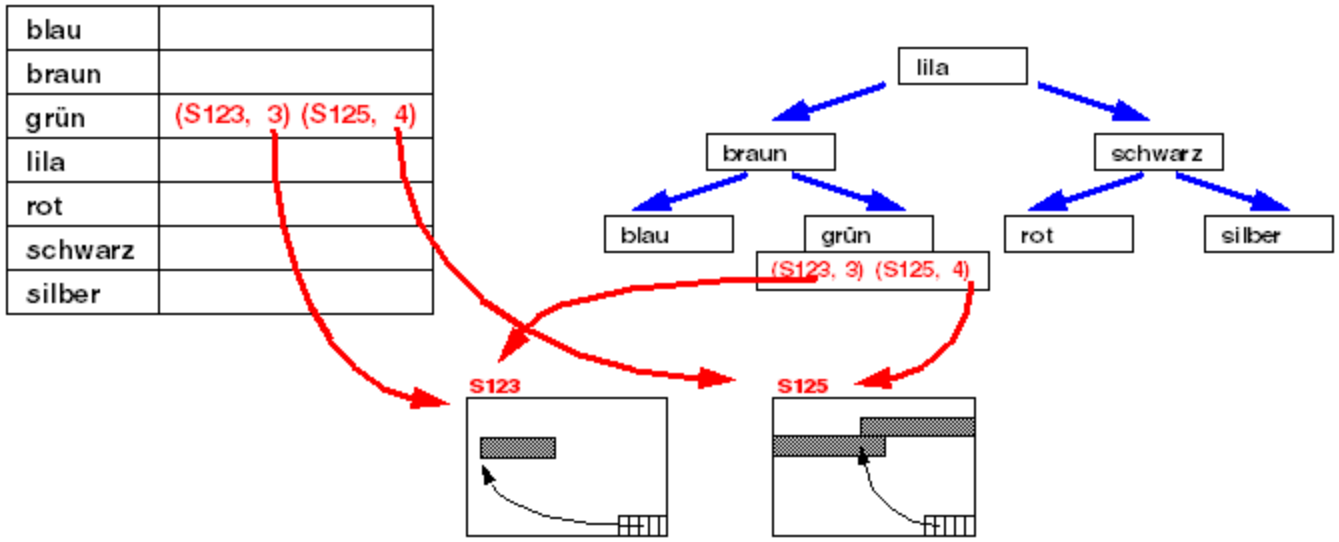

Zugriffsverfahren für eindimensionale Schlüssel
Schlüssel-vergleich
Schlüssel-transformation
fortlaufender Vergleich
baumstrukturierter Vergleich
konstante Trans-formationsverfahren
variable Trans-formationsverfahren
physisch fortlaufend
logisch fortlaufend
Vergleich des ganzen Schlüssels
Vergleich von Schlüssel-teilen
kollisions-freie Satzzu-ordnung
Kollisions-behandlung mit Überlauf-bereichen
Kollisionsbe-handlung durch dynamische Erweiterung
Sequentielle Listen
Gekettete Listen
Binäre Such-bäume
Mehr-weg-bäume
Digital-bäume
statische Hash-Bereiche
dynamische Hash-Bereiche
sequentiell
baumstrukturiert
gestreut
Organisationsformen der Speicherungs-strukturen

Auszug aus 3-Schema-Schichtenarchitektur



Verwaltung der Indexeinträge

- Variante 1: Liste von Einträge
- Variante 2: Organisation als Binärbaum / Binärer Suchbaum
  - Baumstruktur mit einem linken und rechten Kind
  - ausgeglicherender balancierter Suchbaum



Mehrwegbaum

- Baumstruktur mit mehreren Kindern
- Idee:
  Die maximale Größe eines Knotens entspricht exakt der Speicherkapazität einer Seite

B-Baum

- Variante eines Mehrwegbaumes zur Abbildung von Schlüsselwerten auf interne Satzadressen
- entworfen für den Einsatz in Datenbanksystemen (Bayer, McCreight, 1972)

Funktion

- dynamische Reorganisation durch Splitten und Mischen von Seiten
- direkter Schlüsselzugriff
- sortierter sequentieller Zugriff (insbes. B*-Baum)

Definition

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

- Jeder Pfad von der Wurzel zum Blatt hat die gleiche Länge h
- Jeder Knoten (außer Wurzel und Blätter) hat mindestens k + 1 Nachfolger. Die Wurzel ist ein Blatt oder hat mindestens 2 Nachfolger
- Jeder Knoten hat höchstens 2k + 1 Nachfolger

Seitenformat

- (K_i, D_i, P_i) = Eintrag, K_i = Schlüssel
- D_i = Daten des Satzes oder Verweis auf den Satz (materialisiert oder referenziert)
- P_i = Zeiger zu einer Nachfolgerseite

Bedeutung der Zeiger K_i (i = 0, 1, ... p)

- P_0 weist auf einen Teilbaum mit Schlüsseln kleiner als K_1
- P_i (i = 1, 2, ..., l - 1) weist auf einen Teilbaum, dessen Schlüssel zwischen K_i und K_{i+1} liegen
- P_p weist auf einen Teilbaum mit Schlüsseln größer als K_p
- In den Blattknoten sind die Zeiger nicht definiert

Parameter k (Ordnung des Baumes)

- errechnet sich aus der Seitengröße
- $k = \left\lceil \frac{n}{2} \right\rceil$, d.h. (2*k) ist die maximale Anzahl von Einträgen pro Seite

Parameter h (Höhe des Baumes)

- ergibt sich aus der Anzahl der gespeicherten Datenelemente und der Einfügereihenfolge

Maximale Höhe h_{max}

- B-Baum der Ordnung k mit n Schlüsseln
- Level 2 hat ≥ 2 Knoten
- Level 3 hat ≥ 2(k+1) Knoten
- Level 4 hat ≥ $2(k+1)^2$ Knoten
- . . . .
- Level h+1 hat n+1 ≥ $2(k+1)^{h-1}$ (äußere) Knoten

→

$$h \leq 1 + \log_{k+1}\left(\frac{n+1}{2}\right)$$

- und somit:

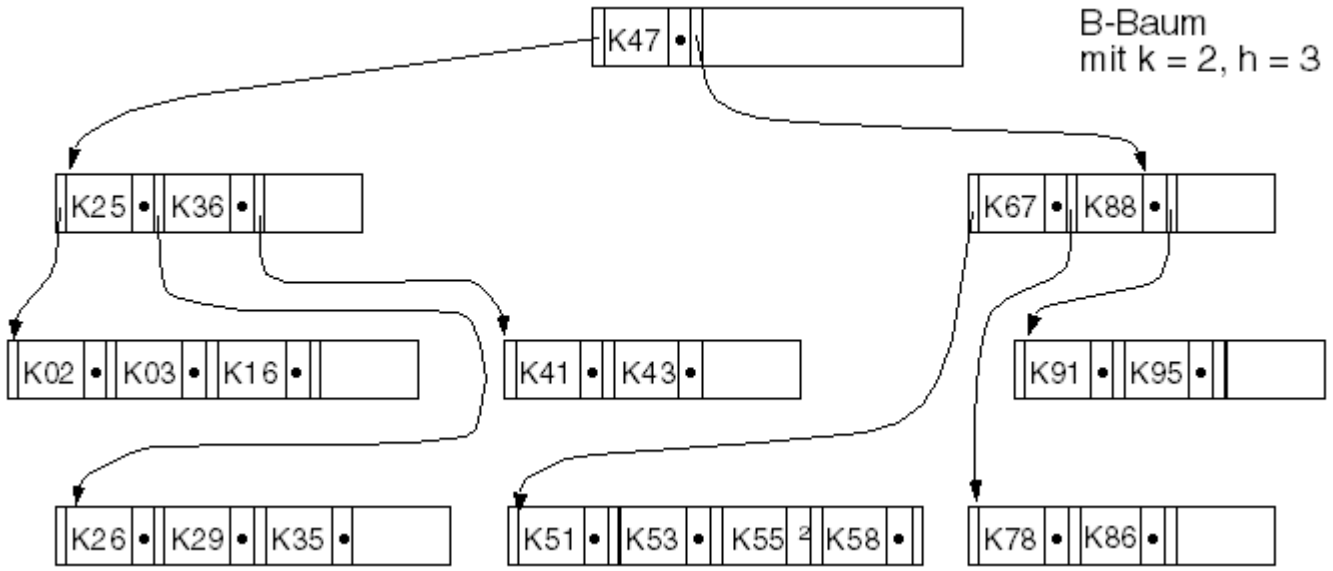
$$\lceil \log_{2k+1}(n+1) \rceil \leq h \leq \left\lfloor \log_{k+1}\left(\frac{n+1}{2}\right) \right\rfloor + 1$$

Beobachtung

- Jeder Knoten (außer der Wurzel) ist mindestens mit der Hälfte der möglichen Schlüssel gefüllt.

  --> **Speicherplatzausnutzung ≥ 50 %**!

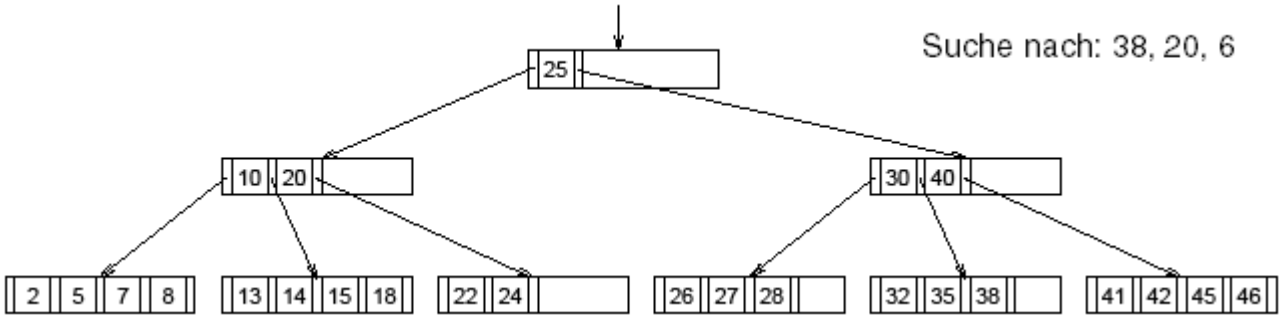
B-Baumstruktur als Zugriffspfad für den Primärschlüssel ANR



Operationen

- Suchen eines Datensatzes mit vorgegebenem Schlüsselwert
- Einfügen und Löschen eines Datensatzes

- Beginnend mit dem Wurzelknoten, wird ein Knoten jeweils von links nach rechts durchsucht
  - 1) Stimmt K_i mit dem gesuchten Schlüsselwert überein, ist der Satz gefunden. (Weitere Sätze mit gleichem Schlüsselwert befinden sich ggf. in dem Teilbaum, auf den P_{i-1} zeigt.)
  - 2) Ist K_i größer als der gesuchte Wert, wird die Suche in der Wurzel des von P_{i-1} identifizierten Teilbaums fortgesetzt.
  - 3) Ist K_i kleiner als der gesuchte Wert, wird der Vergleich mit K_{i+1} wiederholt.
  - 4) Ist auch K_{2k} noch kleiner als der gesuchte Wert, wird die Suche im Teilbaum von P_{2k} fortgesetzt.
- Ist der weitere Abstieg in einen Teilbaum (2. oder 4.) nicht möglich (Blattknoten):
  - Suche abbrechen, kein Satz mit gewünschtem Schlüsselwert vorhanden.



Regel

- Eingefügt wird nur in Blattknoten!

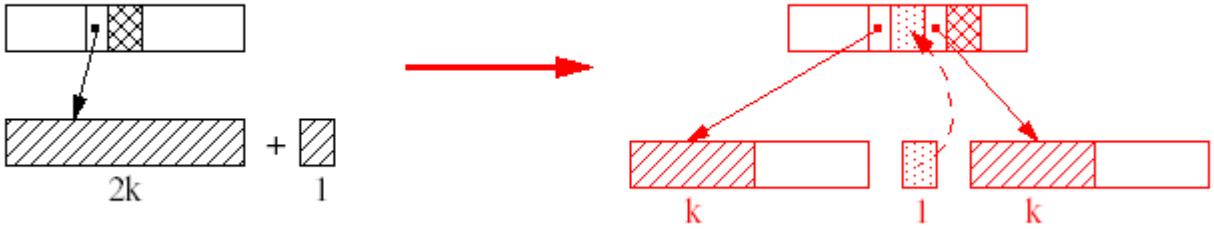
Vorgehen

- zunächst Abstieg durch den Baum wie bei Suche:
  - $S \leq K_i$: folge P_{i-1}
  - $S > K_i$: prüfe K_{i+1}
  - $S > K_{2k}$: folge P_{2k}
- im so gefundenen Blattknoten:
  - Satz entsprechend der Sortierreihenfolge einfügen
  - Sonderfall: Blattknoten ist schon voll (enthält 2k Sätze)

  => Splitt des Blattknotens

Vorgehen beim Splitt

- einen neuen Blattknoten erzeugen
- die 2k+1 Sätze (in Sortierordnung!) halbe-halbe zwischen altem und neuem Blattknoten aufteilen
  - die ersten k Sätze in die erste (die linke) Seite
  - die letzten k Sätze in die zweite (die rechte) Seite
- den mittleren (k+1-ten) Satz als neuen “Diskriminator” (als Verzweigungs-information bei der Suche) in den eine Stufe höheren Knoten einfügen, der auf den Blattknoten verweist



zwei mögliche Situationen nach einem Splitt

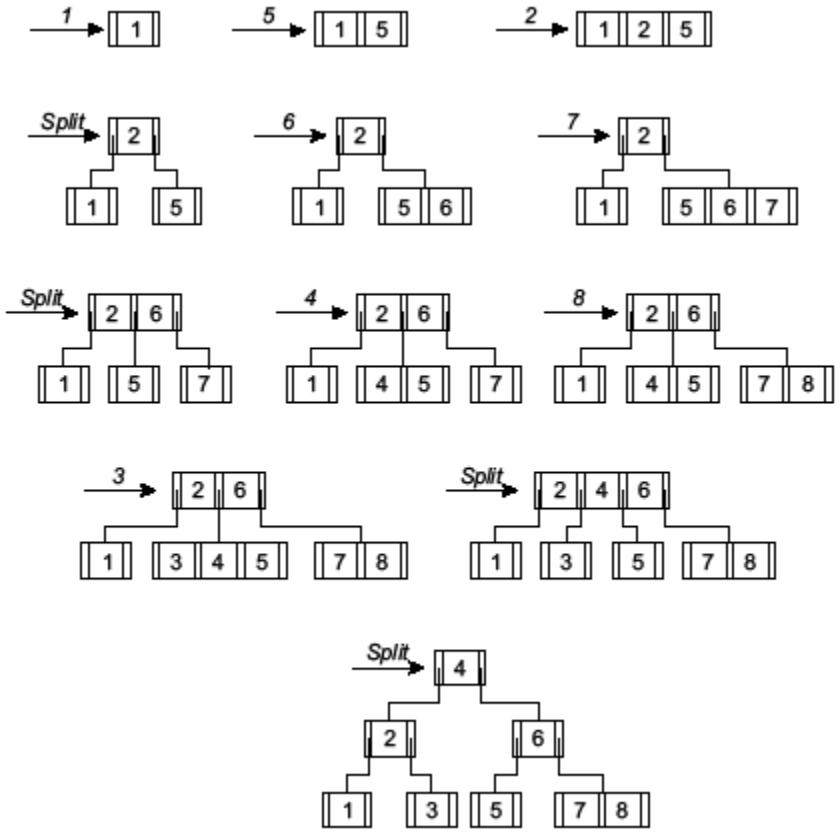
- der übergeordnete Knoten ist voll
  => Splitt auf dieser Ebene wiederholen
- ausreichend Platz
  => FERTIG

Weiterer Sonderfall

- Splitt des Wurzelknotens
  => Erzeugung von zwei neuen Knoten
  => Neue Wurzel mit zwei Nachfolgeknoten
- Höhe des Baums wächst um 1
  (Man sagt bildlich: Der Baum “reißt von unten nach oben auf”.)

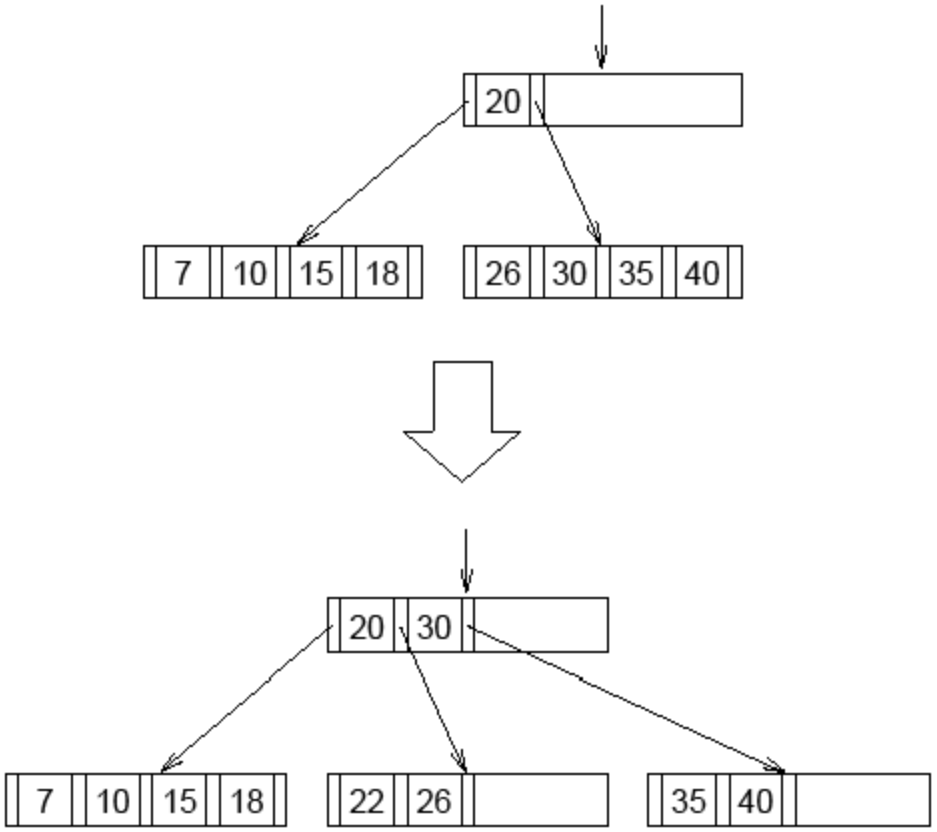
Dynamische Reorganisation

- kein Entladen und Laden erforderlich
- Baum immer balanciert

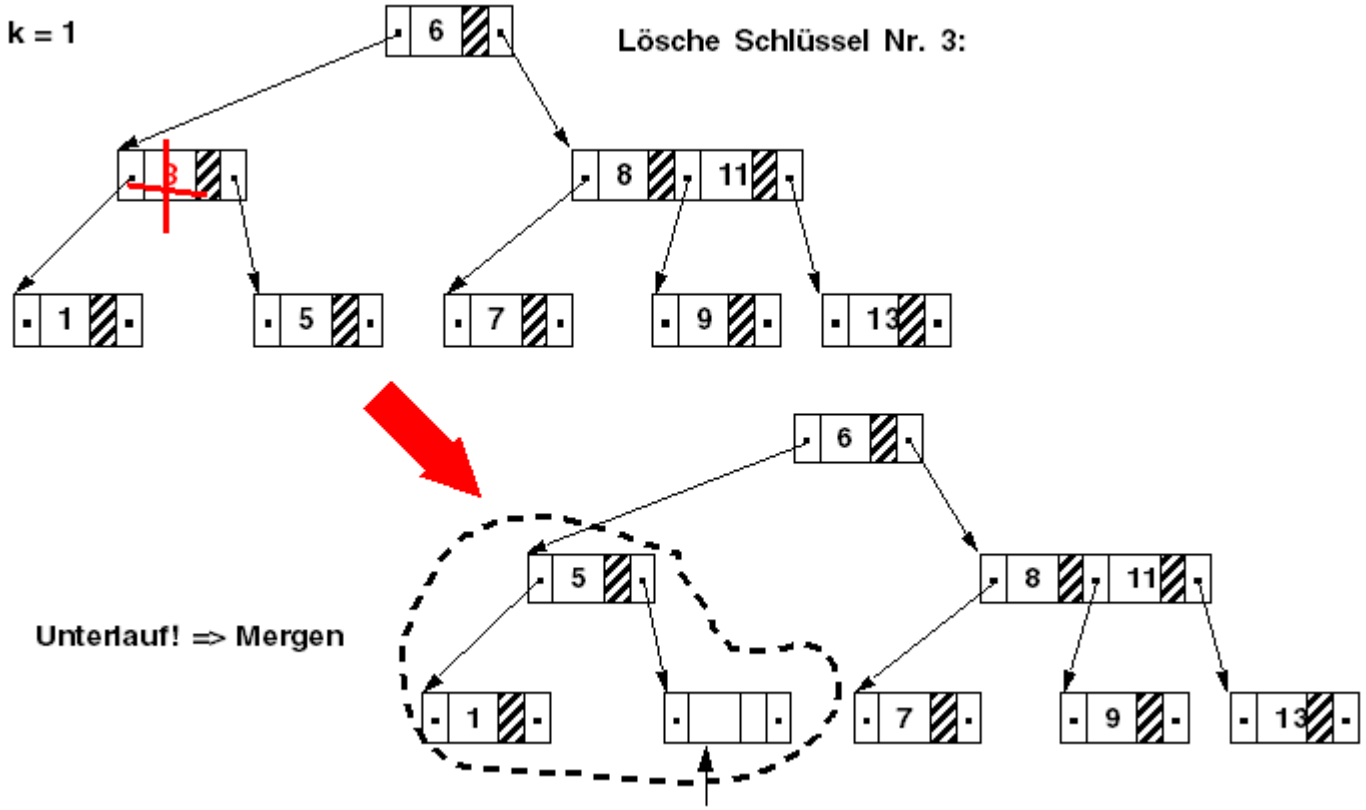

1
1
5
1 5
2
1 2 5
Split
2
1 5
6
2
1 5 6
7
2
1 5 6 7
Split
2 6
1 5 7
4
2 6
1 4 5 7
8
2 6
1 4 5 7 8
3
2 6
1 3 4 5 7 8
Split
2 4 6
1 3 5 7 8
Split
4
2 6
1 3 5 7 8

Problem

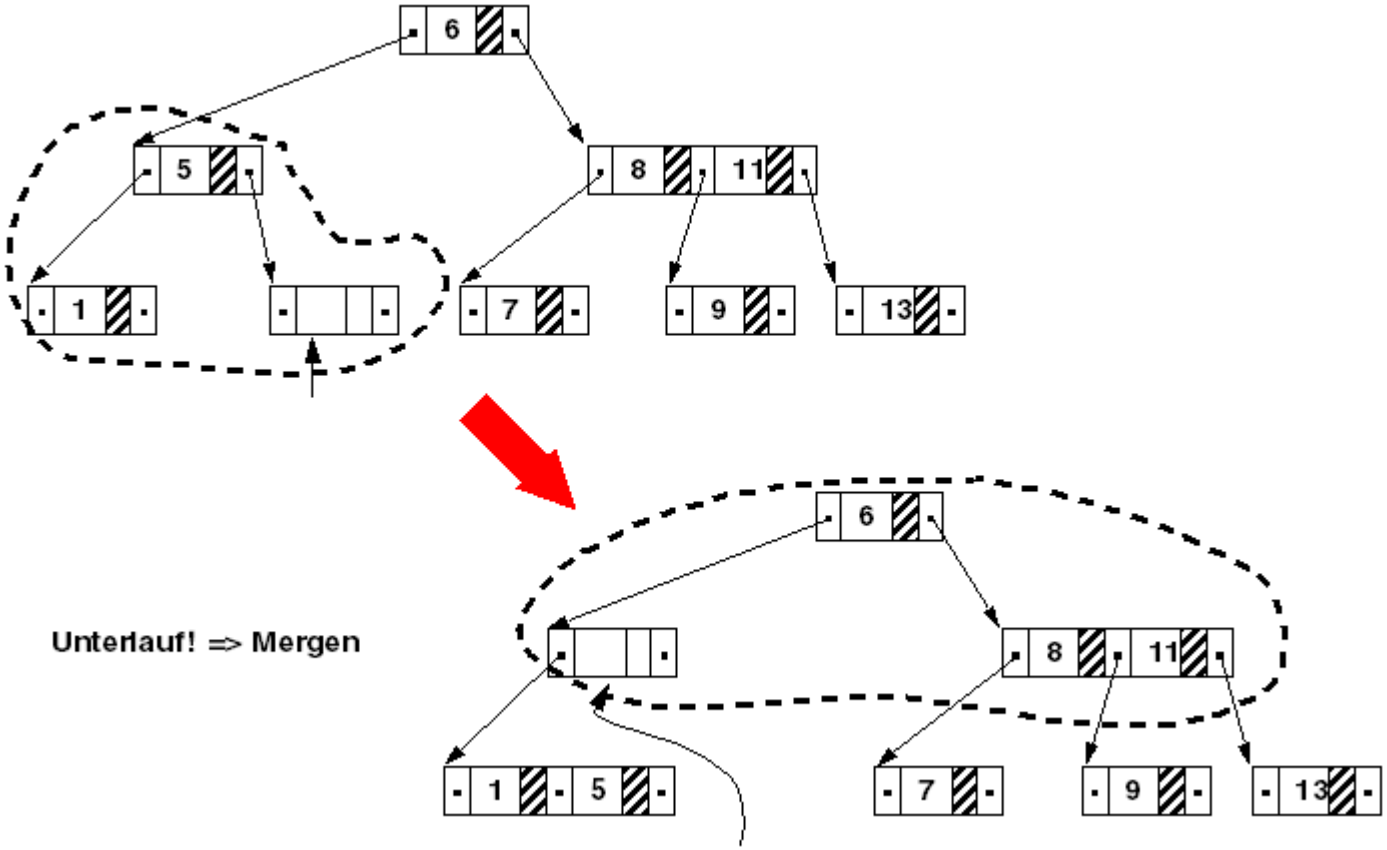
- Einfügen kann Überlauf erzeugen
- Löschen kann Unterlauf und Überlauf erzeugen
- Beispiel: Einfügen und Löschen von Schlüssel Nr. 22

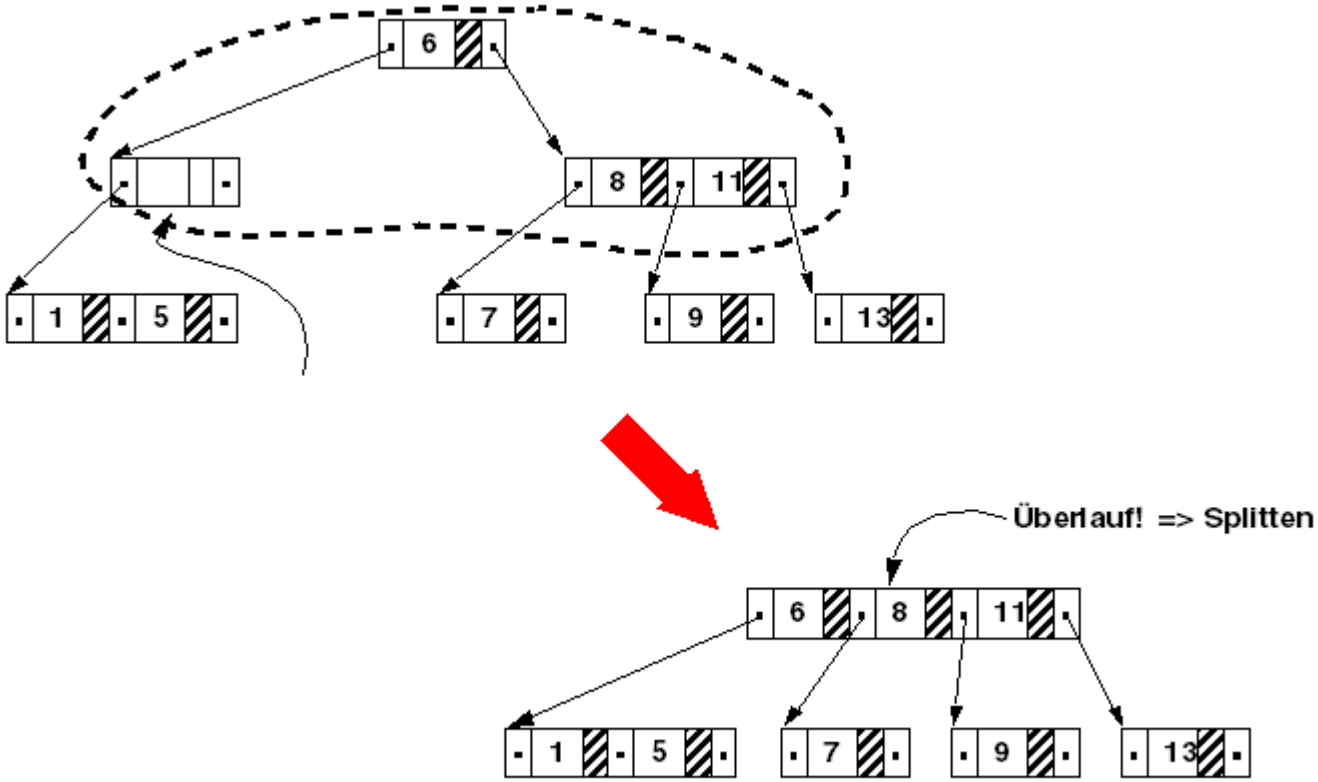


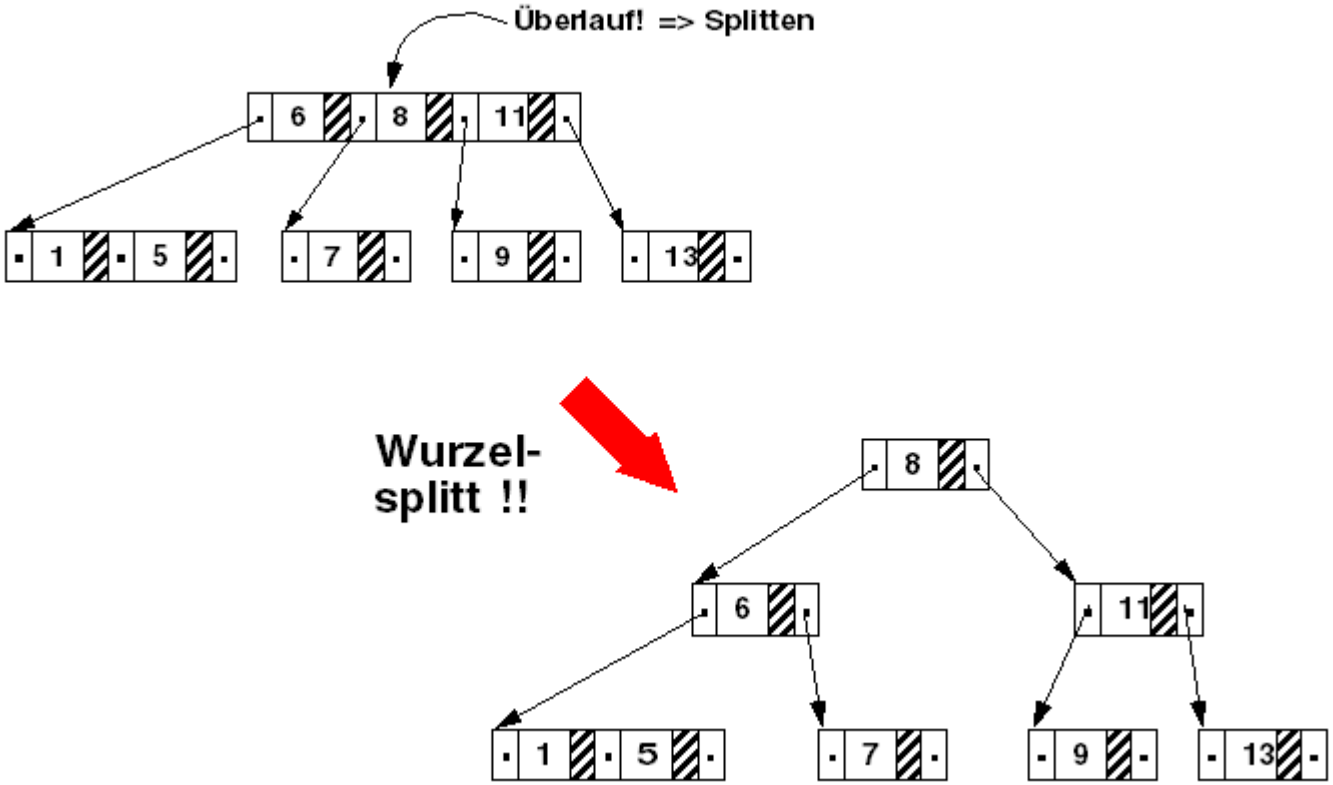
... erstmal am Beispiel !!



> Löschen im B-Baum (2)




6
8
11
1
5
7
9
13
Überlauf! => Splitten
6
8
11
1
5
7
9
13


Überlauf! => Splitten
6
8
11
1
5
7
9
13
Wurzel-
splitt !!
8
6
11
1
5
7
9
13

Beispiel - es gibt verschiedene Algorithmen

- Suche den Knoten, in dem der zu löschende Schlüssel S liegt
- Falls Schlüssel S in Blattknoten, dann lösche Schlüssel in Blattknoten und behandle evtl. entstehenden Unterlauf
- Falls Schlüssel S in einem inneren Knoten, dann untersuche linken und rechten Nachfolgerknoten zu dem zu löschenden Schlüssel S:
- untersuche, welcher Nachfolgerknoten von S mehr Elemente hat, der linke oder der rechte. Falls beide gleich viele Elemente haben, dann entscheide für einen.
- Ersetze zu löschenden Schlüssel S durch direkten Vorgänger S’ aus linken Nachfolgeknoten bzw. durch direkten Nachfolger S’’ aus rechten Nachfolgeknoten.
- Lösche S’ bzw. S’’ aus dem entsprechenden Nachfolgeknoten (rekursiv)

Anmerkungen

- ein entgültiger Unterlauf entsteht bei obigen Algorithmus erst auf Blattebene!
- Unterlaufbehandlung wird durch einen Merge des Unterlaufknotens mit seinem Nachbarknoten und dem darüberliegenden Diskriminator durchgeführt
- Wurde einmal mit dem Mergen auf Blattebene begonnen, so setzt sich dieses Mergen nach oben hin fort
- Das Mergen auf Blattebene wird solange weitergeführt, bis kein Unterlauf mehr existiert, oder die Wurzel erreicht ist
- Wird die Wurzel erreicht, kann der Baum in der Höhe um eins schrumpfen. Beim Mergen kann es auch wieder zu einem Überlauf kommen. In diesem Fall muss wieder gesplittet werden.

Aufwandsabschätzung

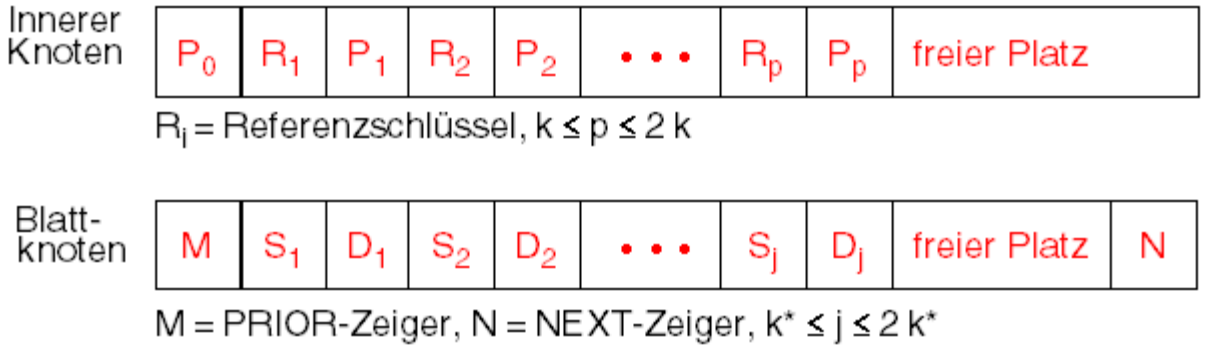
- Einfügen, Suchen und Löschen: O(logk(n)) Operationen
- entspricht Höhe eines Baumes
- Ziel: geringere Höhe -> größere Breite

Konkretes Beispiel

- Seiten der Größe 4 KB, Zugriffsattributwert 32 Bytes, 8-Byte-Zeiger
  --> zwischen 50 und 100 Indexeinträge pro Seite; Ordnung dieses B-Baumes 50
  1.000.000 Datensätze: log50(1.000.000) = 4 Seitenzugriffe im schlechtesten Fall
  Wurzelseite jedes B-Baumes normalerweise im Puffer: drei Seitenzugriffe

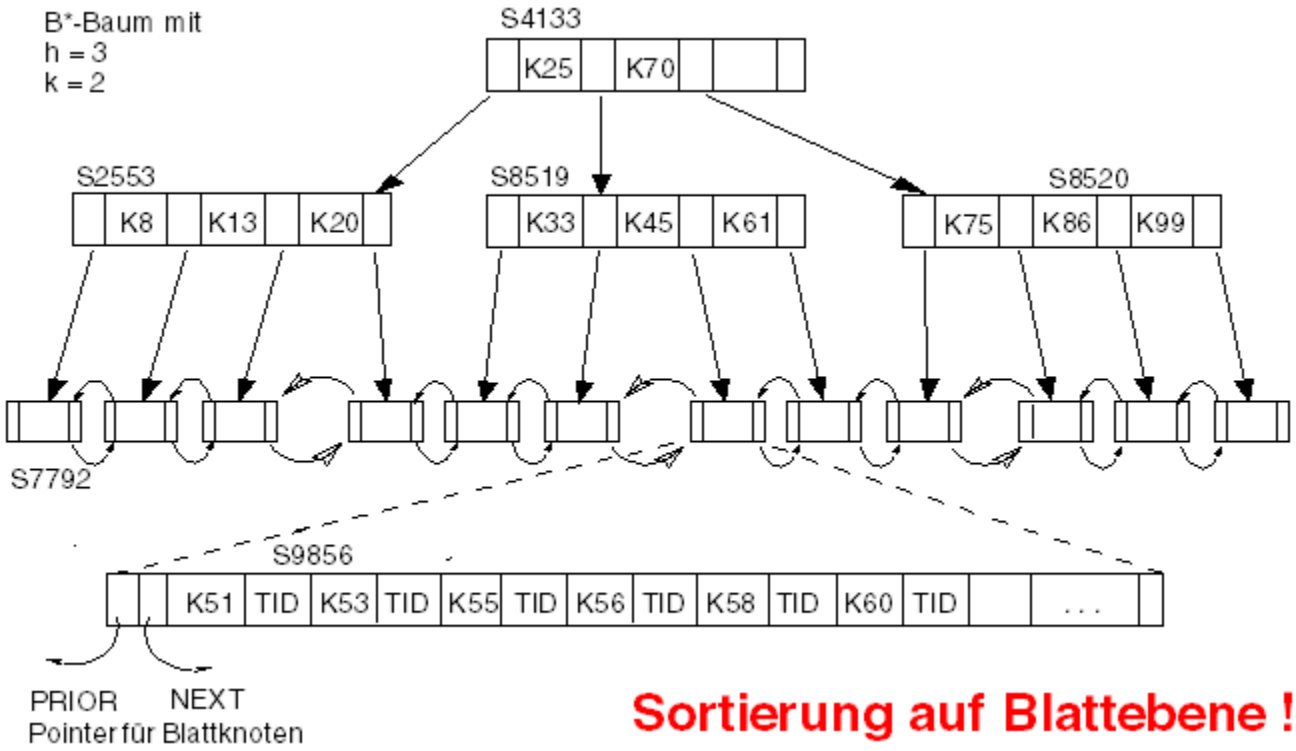
Eigenschaften und Unterschiede zum B-Baum

- Alle Sätze (bzw. Schlüsselwerte mit TID's) werden in den Blattknoten abgelegt.
- Innere Knoten enthalten nur Verzweigungsinformation (also u.U. auch Schlüsselwerte, die in keinem Satz vorkommen), aber keine Daten.
- Aufbau von B*-Baum-Knoten:



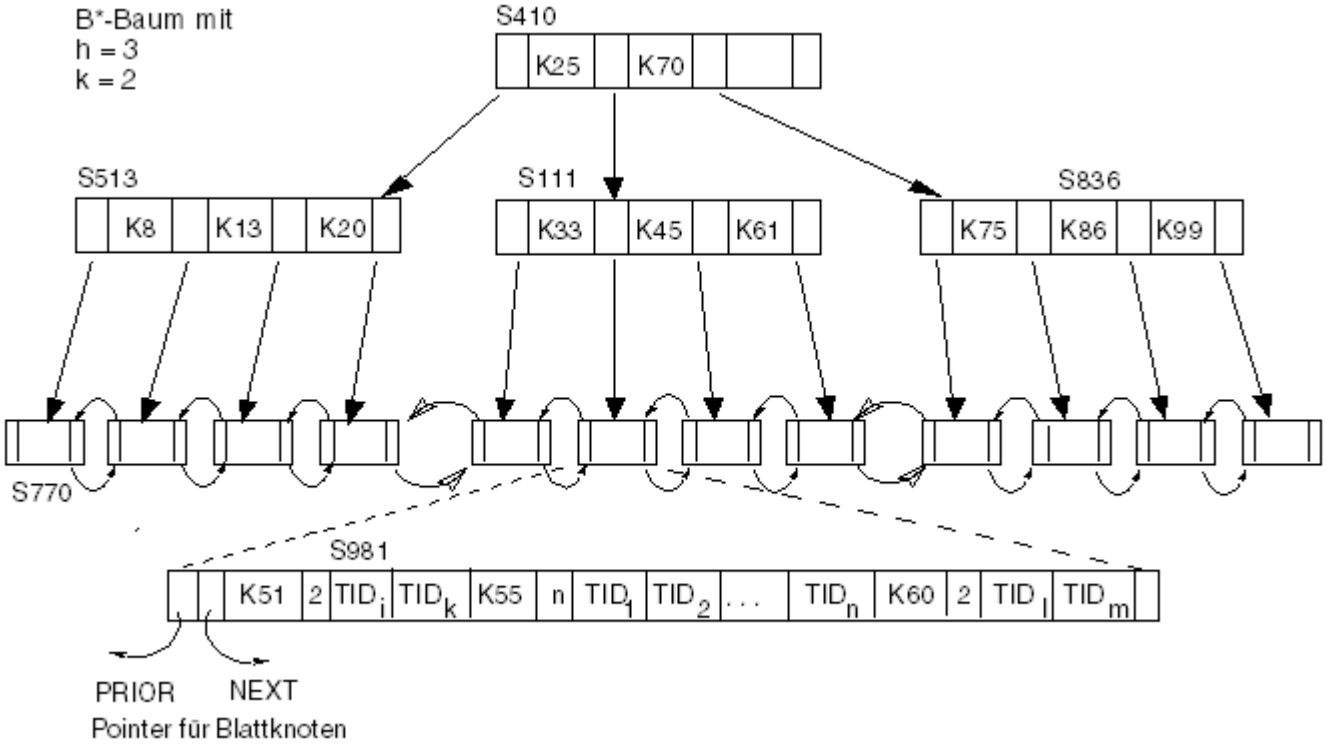
Beispiel

- ANR ist Primärschlüssel in der Relation ABT(ANR, ORT, MNR)



Beispiel

- ANR ist Sekundärschlüssel in der Relation PERS(PNR, NAME, ALTER, ANR)

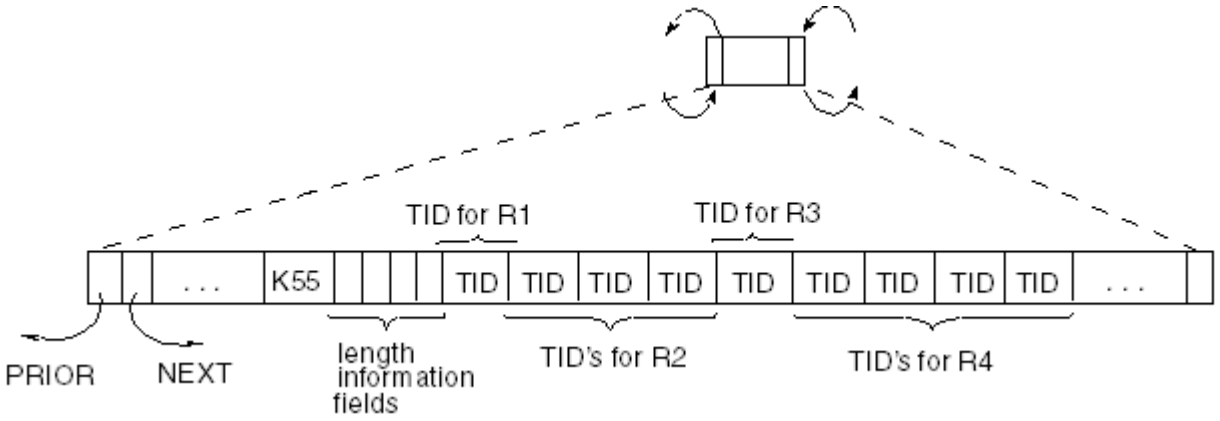


Idee

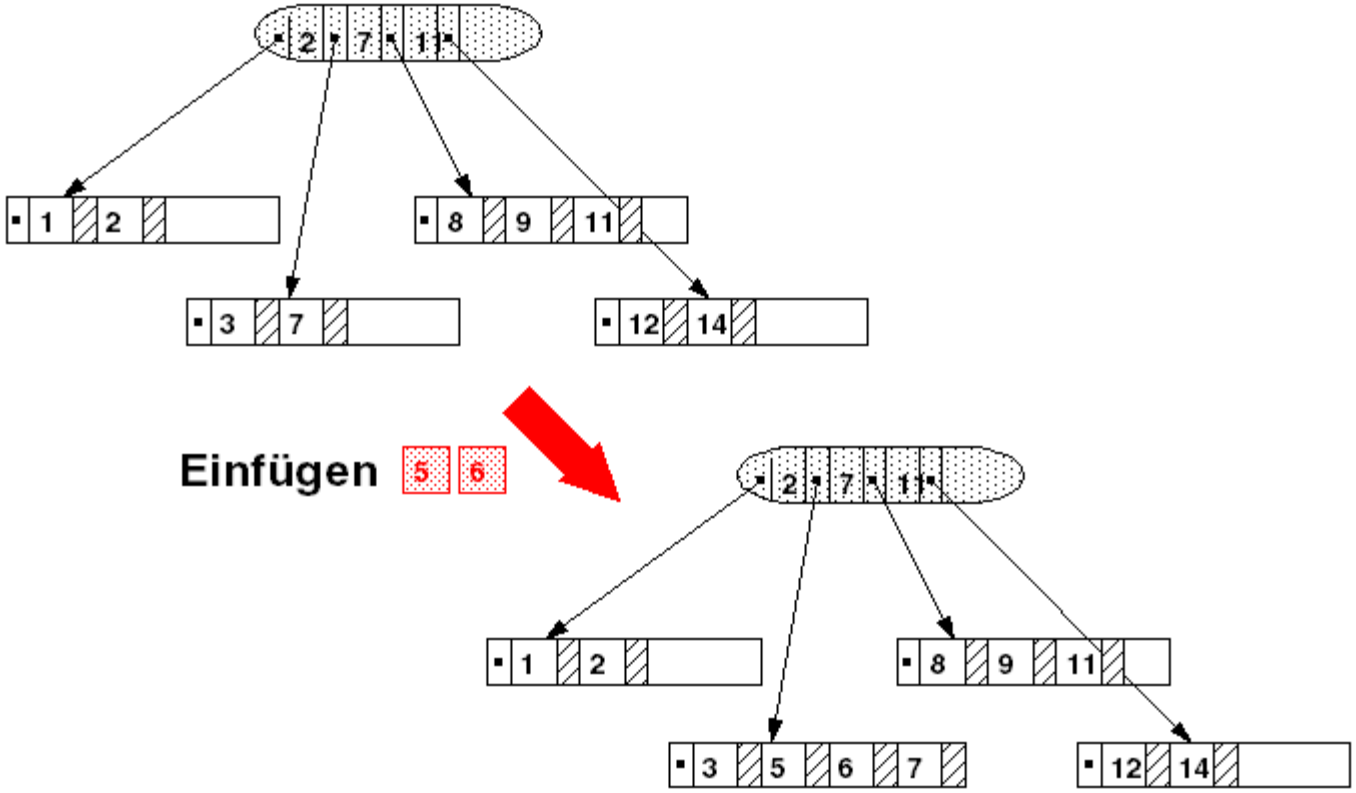
- Indexstruktur für Attribute mit gleichem Wertebereich über mehrere Relationen
- Trennung von Primär- und Sekundärschlüssel-Zugriffspfad wird irrelevant !

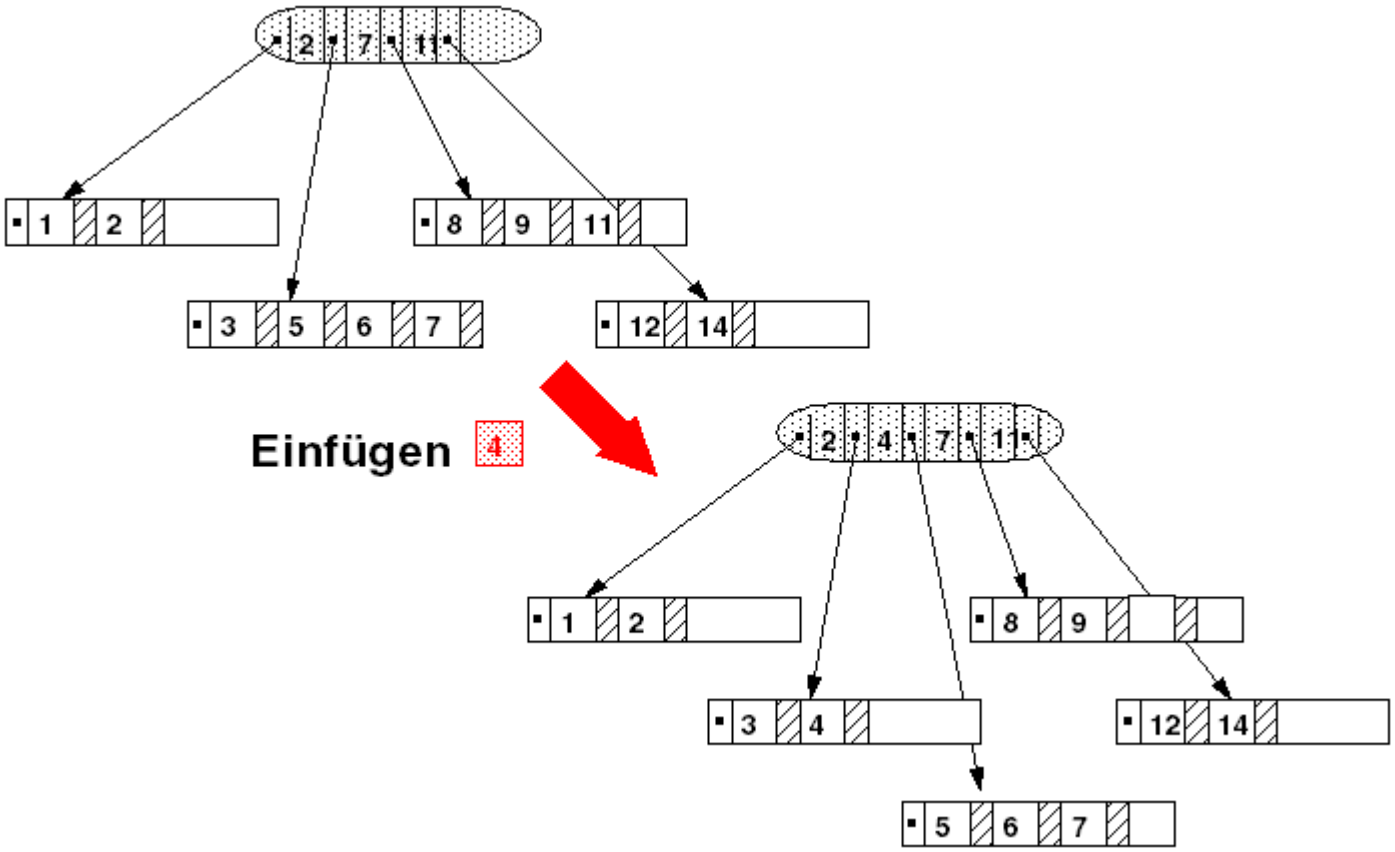
Beispiel: Index über DNO für

R1 = DEP(DNO, ...)
R2 = EMP(ENO, DNO, ...)
R3 = MGR(MNO, DNO, JCODE, ...)
R2 = EQUIP(TNO, DNO, TYPE, ...)

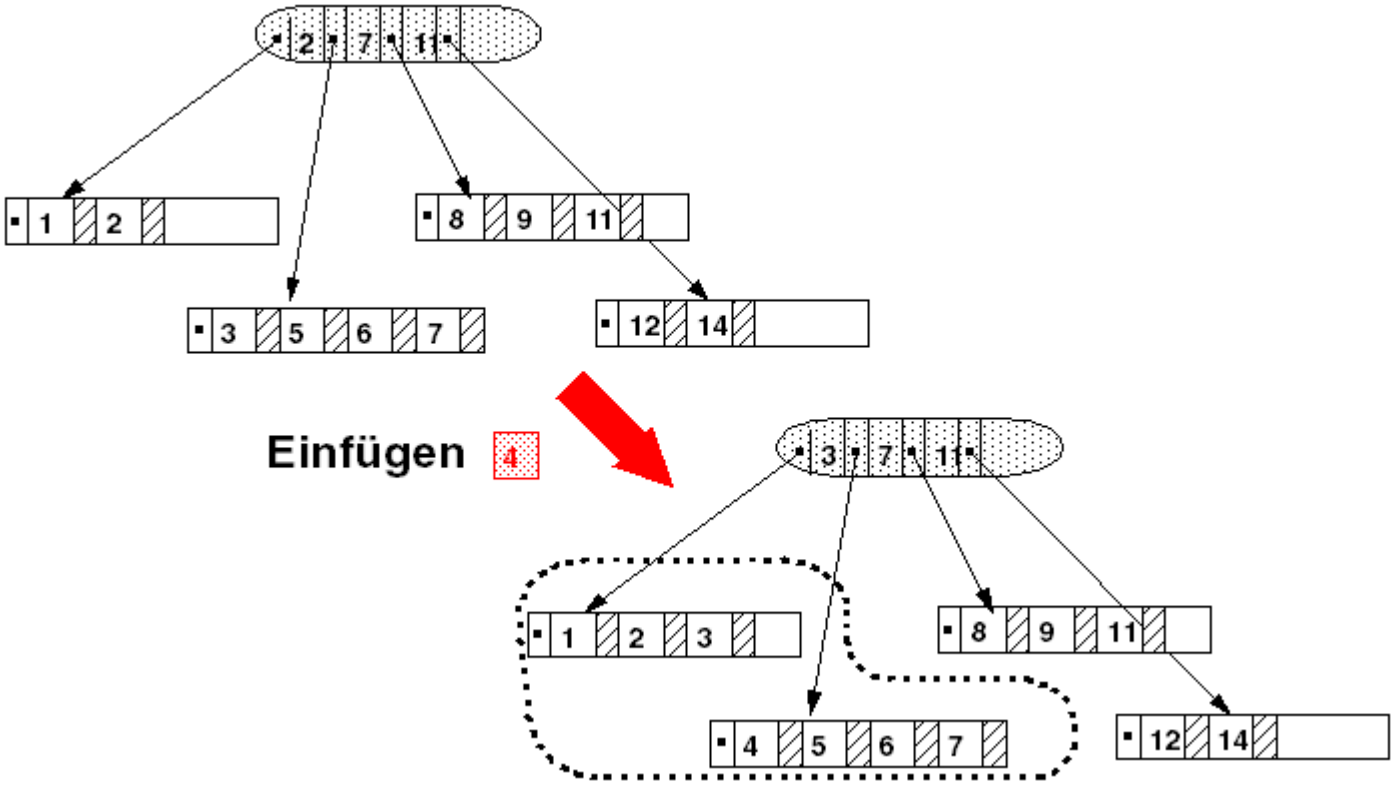


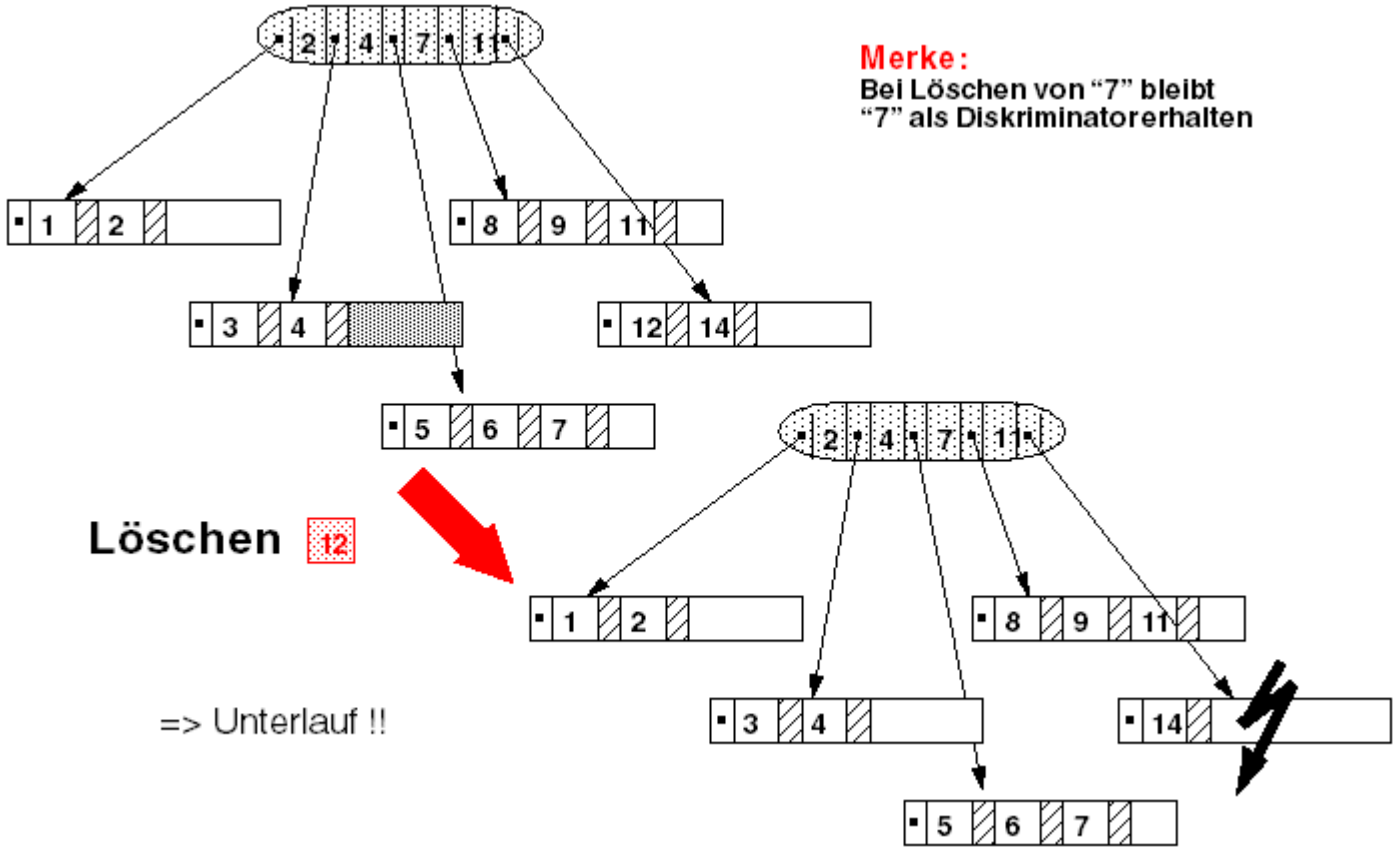
... am Beispiel

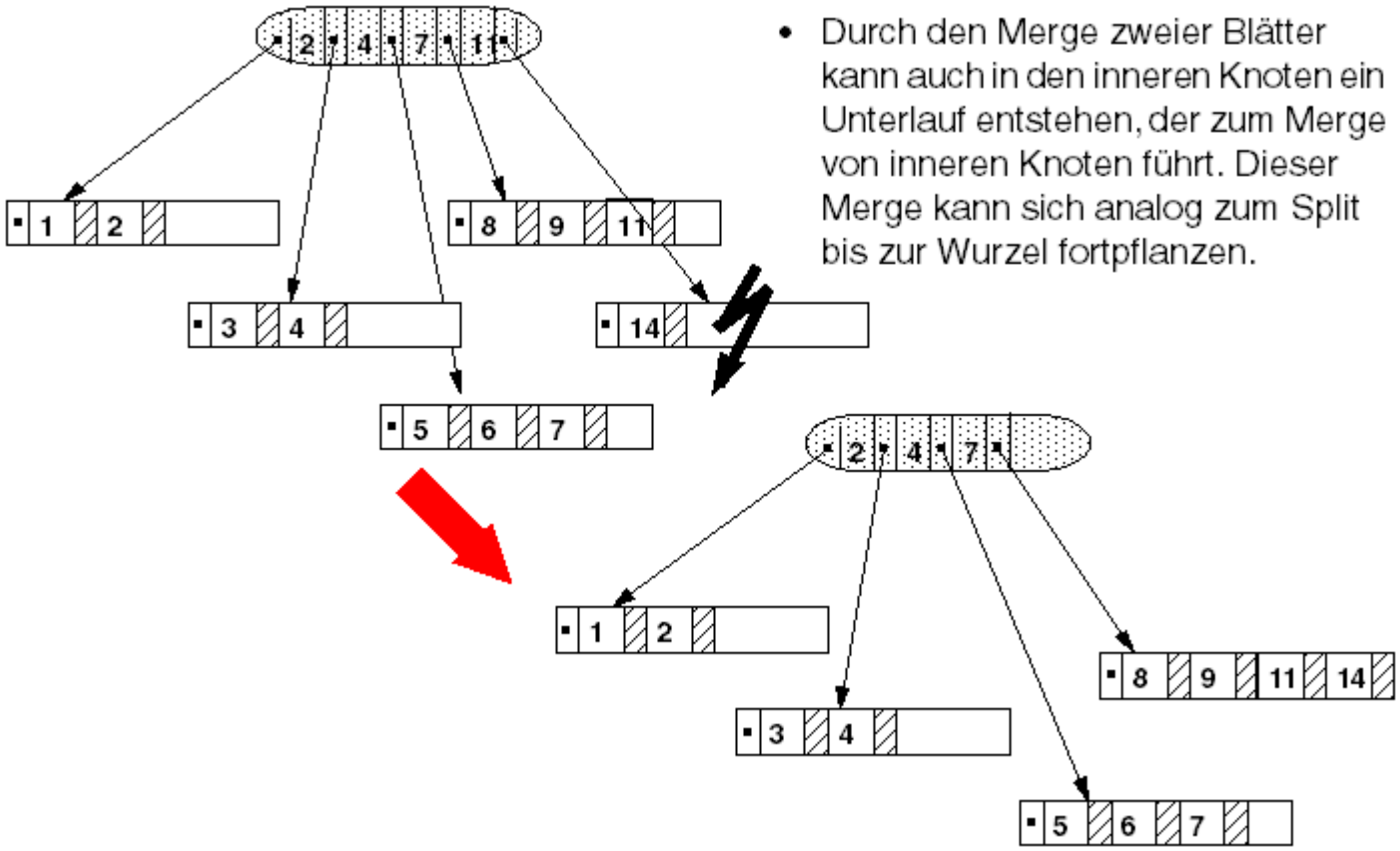



2 7 11
1 2
8 9 11
3 5 6 7
12 14
Einfügen
4
2 4 7 11
1 2
8 9
3 4
12 14
5 6 7

Statt Splitt bei Überlauf, Neuverteilung der Einträge unter Berücksichtigung eines oder mehrerer benachbarter Knoten




2 4 7 11
1 2
8 9 11
3 4
12 14
5 6 7
Merke:
Bei Löschen von "7" bleibt
"7" als Diskriminatorerhalten
Löschen 12
2 4 7 11
1 2
8 9 11
3 4
14
5 6 7
=> Unterlauf !!



- Durch den Merge zweier Blätter kann auch in den inneren Knoten ein Unterlauf entstehen, der zum Merge von inneren Knoten führt. Dieser Merge kann sich analog zum Split bis zur Wurzel fortpflanzen.

1. *Suche den zu löschenden Eintrag im Baum*
2. *Entsteht durch das Löschen ein Unterlauf? (#Einträge < k?)*

NEIN

- Entferne den Satz aus dem Blatt
  (Eine Aktualisierung des Diskriminators im Vaterknoten ist nicht erforderlich!)

JA

- Mische das Blatt mit einem Nachbarknoten:
- Ist die Summe der Einträge in beiden Knoten größer als 2k?

  NEIN
  - Fasse beide Blätter zu einem Blatt zusammen
  - falls dabei ein Unterlauf im Vaterknoten entsteht: mische die inneren Knoten analog

  JA
  - Teile die Sätze neu auf beide Knoten auf, so daß ein Knoten jeweils die Hälfte der Sätze aufnimmt
  - Der Diskriminator im Vaterknoten ist entsprechend zu aktualisieren

Anmerkung

- Vielzahl von Varianten bzgl. Aufteilung/Neuverteilung nach UDI-Operationen

B-Baum

- keine Redundanz
- Lesen aller Sätze sortiert nach Schlüsselwert nur mit Verwaltung eines Stacks der max. Tiefe = Baumhöhe h
- bei Einbettung der Datensätze geringe Verzweigungszahl (“Grad” oder “fan-out”), daher größere Höhe
- einige wenige Sätze (die in der Wurzel) werden mit *einem*

B*-Baum

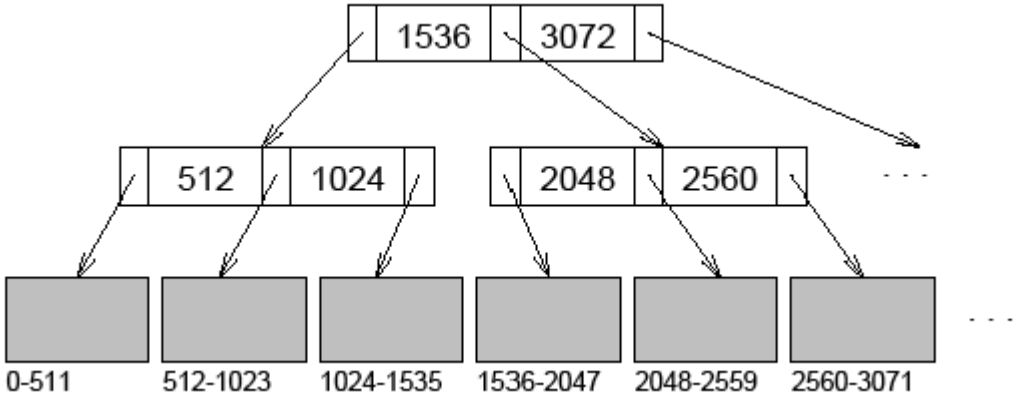
- Schlüsselwerte teilweise redundant gespeichert
- Kette der Blattknoten liefert alle Sätze nach Schlüsselwert sortiert
- hohe Verzweigung in der inneren Knoten, daher geringere Höhe

Idee

- Speicherung der Positionen oder Offset-Angaben statt Zugriffsattributwerte in einem B*-Baum
- BLOB-B*-Baum: Positions-B*-Baum

Anmerkung

- Indexstruktur wird auch benutzt für die Verwaltung von Objekten beispielsweise in objektorientierten Datenbanksystemen



Indexierung von Zeichenketten

- B-Bäume: Betrachtung als atomare Werte
- Lösungsansatz: Digital- oder Präfixbäume aus dem Umfeld des “Information Retrieval”

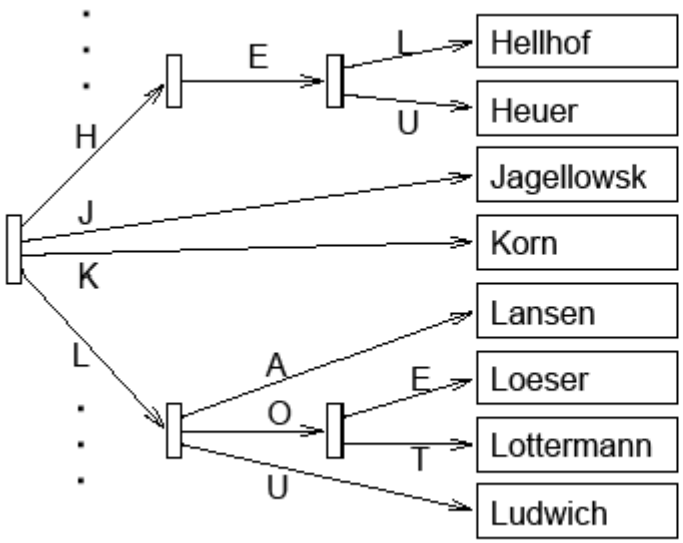
Digitalbäume

- (feste) Indexierung der Buchstaben des zugrundeliegenden Alphabets
- keine Garantie der Balancierung
- Beispiele: Tries, Patricia-Bäume

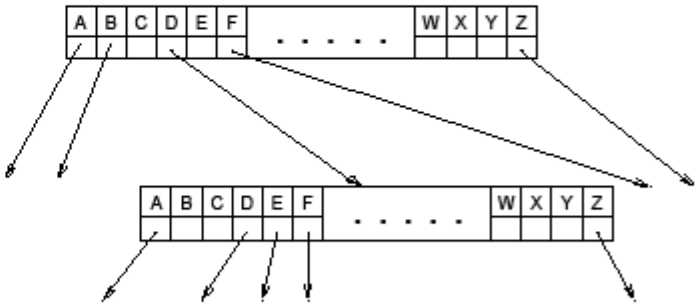
Präfix-Bäume

- Indexierung über Präfixe der Menge von Zeichenketten

Beispiel



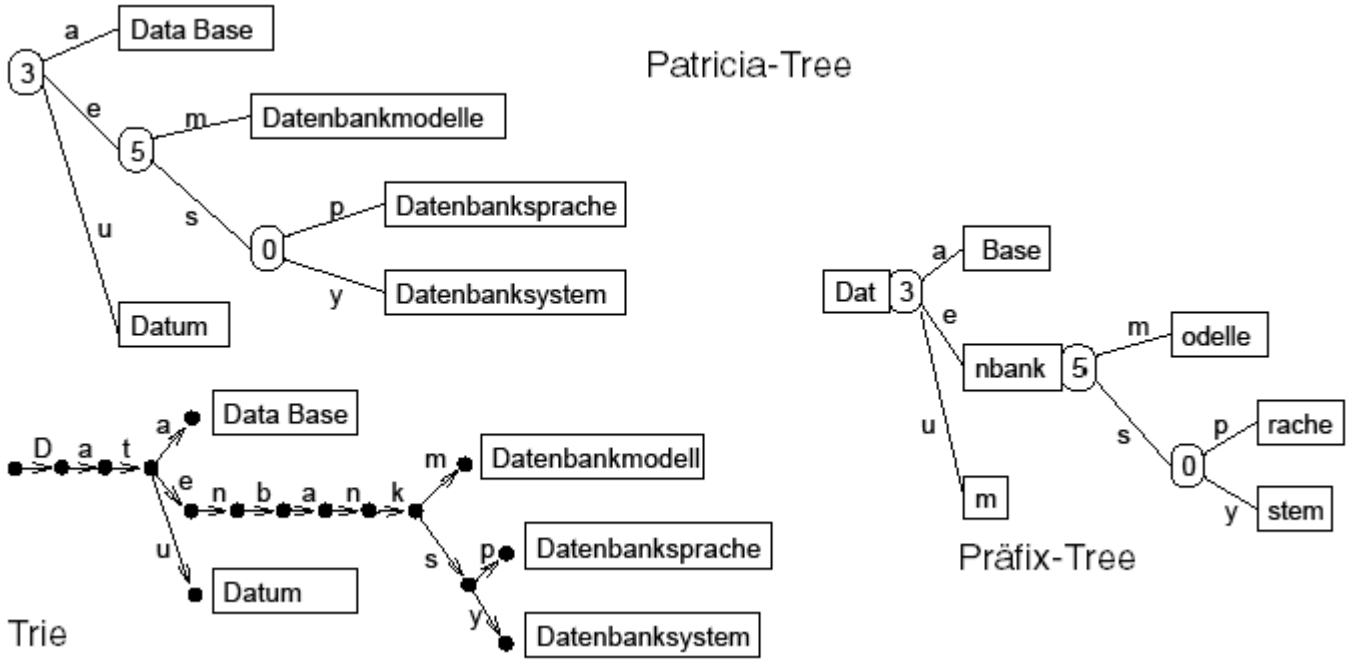
Knoten eines Trie



- Probleme verursachen (fast) leere Knoten / sehr unausgeglichene Bäume
- lange gemeinsame Teilworte
- nicht vorhandene Buchstaben und Buchstabenkombinationen

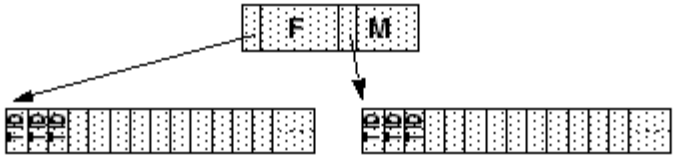
Akronym

- Lösung: **P**ractical **A**lgorithm **T**o **R**etrieve **I**nformation **C**oded **I**n **A**lphanumeric
- Überspringen von Teilworten (zusätzliche Speicherung in Präfix-Bäumen)



Problem

- Beispiel: B-Baum auf Geschlecht bei Kundentabelle mit 100.000 Tupeln resultiert in zwei Listen mit jeweils ca. 500.000 Tupeln



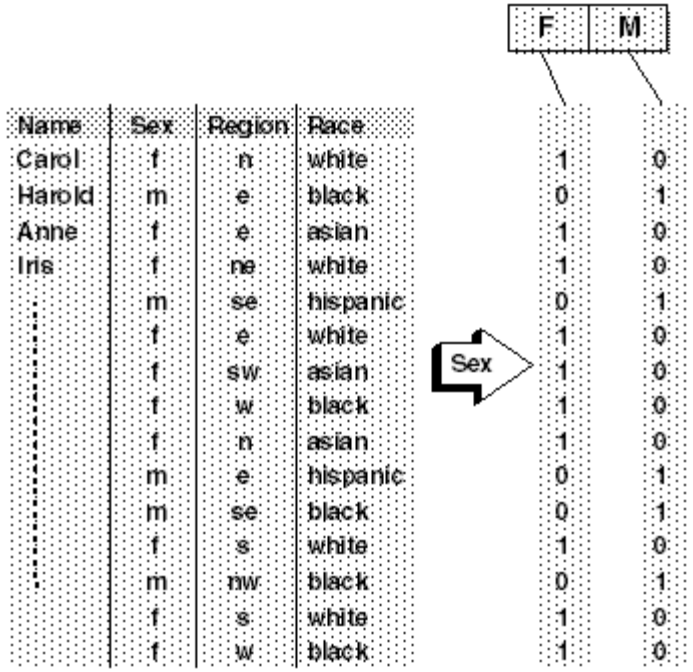
- Anfrage nach allen “weiblichen” Kunden erfordert 500.000 einzelne Zugriffe
  È Table-Scan ist um Längen schneller

Folgerung

- B-Bäume (und auch Hashing) sind sinnvoll für Prädikate mit hoher “Selektivität” (geringer Anteil von zu erwartenden Tupeln gegenüber allen Tupeln einer Relation)
- Daumenregel
- Grenztrefferrate liegt bei ca. 5%.
- Höhere Trefferraten lohnen bereits den Aufwand für einen Indexzugriff nicht mehr

Idee

- bereits in die Jahre gekommen ... (eingesetzt seit 60er Jahren in Model 204 von Computer Corporation of America)
- Lege für jede Attributausprägung eine Bitmap/Bitliste an
- Jedem Tupel der Tabelle ist ein Bit in der Bitmap zugeordnet
- Bitwert 1 heißt der Attributwert wird angenommen, 0 heißt Attributwert wird nicht angenommen
- Notwendig: Fortlaufende Nummerierung der Tupel (TIDs)



Indexgröße: (Anzahl der Tupel) x (Anzahl der Ausprägungen) Bits

- Beispiel: Geschlecht mit 2 Ausprägungen in Relation mit 10k Tupeln, 4 Byte TID
  Bitmap: 2 * 10k Bits = 20k Bits = 2.5k Bytes
- Beispiel: Relation ORDERS mit O_ORDERSTATUS: 150.000 Tupel
- RID-Liste: 600KByte mit je 4Byte pro RID
- Bitliste: 150.000/8 = 18750 Byte je Attribut: 56,25KByte

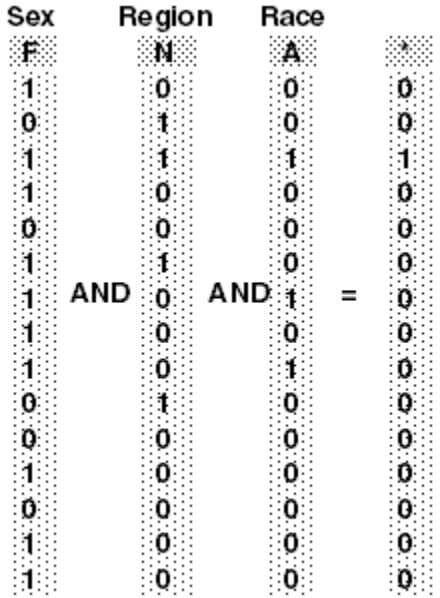
Eigenschaften von Bitmap-Indexstrukturen

- wachsen mit der Anzahl der Ausprägungen
- sind besonders interessant bei Wertigkeiten bis ca. 500
- sind bei kleinen Wertigkeiten (z.B. Geschlecht) nur sinnvoll, wenn entsprechendes Attribut oft in Konjunktionen mit anderen indizierten Attributen auftritt (z.B. Geschlecht und Wohnort)
- Indexgröße nicht so problematisch, da gerade bei höherwertigen Attributen die Bitmaps sehr dünn besetzt sind und Kompressionsverfahren (z.B. RLE) sehr gut einsetzbar sind

Hauptvorteil von Bitmap-Indexen

- einfache und effiziente logische Verknüpfbarkeit
- Beispiel: Bitmaps B1 und B2 in Konjunktion

- for (i=0; i<B1.length; i++)
  B = B1[i] & B2[i];



Beispiel
"Asiatische Frauen der Region 'Nord'"

- Selektivität: 1/2 * 1/8 * 1/4 = 1/64
- Annahme: 10k Tupel mit je 200 Bytes Länge
  (ca. 10 Tupel pro Seite bei 2kB Seiten)
- Table-Scan: 1000 Seiten
- Bitmap-Zugriff: 10k/64 » 150 Seiten (worst case)

Beispiel

```
SELECT SUM(L_QUANTITY) AS SUM_QUAN
  FROM TPCD.LINEITEM, TPCD.ORDERS
  WHERE L_ORDERKEY = O_ORDERKEY
  AND O_ORDERSTATUS = 'F'
  AND O_ORDERPRIORITY = '1-URGENT'
  AND (O_ORDERPRIORITY IN ('4-NOT SPECIFIED', '5-LOW')
  OR O_CLERK = 'CLERK#466');
```

RID-Listen: Sortierung lokaler RID-Listen im Hauptspeicher

Bitlisten: Verknüpfung durch Anwendung logischer Operatoren auf B[ ]

```
For i = 1 To Length(TPCD.ORDERS)
  B[i] := B('F')[i] AND B('1-URGENT')[i]
  AND (B('4-NOT SPECIFIED')[i] OR B('5-LOW')[i] OR B('CLERK#466')[i])
  End For
```

Beachte

- hohe Selektivität nach der konjunktiven Verknüpfung
  im Beispiel: 17/150.000 = 1.1‰

Variante 1: Kette disjunktiver Verknüpfungen

- Beispiel: BETWEEN 2 AND 7

```
For i = 1 To Length(...)
B[i] := B(2)[i] OR B(3)[i] OR B(4)[i] OR B(5)[i] OR B(6)[i] OR B(7)[i]
End For
```

Variante 2: Bereichsbasierte Kodierung (›range-based encoding scheme‹)

- Prinzip: k-te Bitliste wird auf 1 gesetzt, falls
- normal kodierte Bitliste weist eine 1 auf
- vorangegangene Bitliste weist eine 1 auf
- Beispiel: Bereichskodierung von Attribut A

A	$\overline{B}(1)$	$\overline{B}(2)$	$\overline{B}(3)$	$\overline{B}(4)$	$\overline{B}(5)$	$\overline{B}(6)$	$\overline{B}(7)$	$\overline{B}(8)$	$\overline{B}(9)$	$\overline{B}(10)$	$\overline{B}(11)$	$\overline{B}(12)$
1	1	1	1	1	1	1	1	1	1	1	1	1
3	0	0	1	1	1	1	1	1	1	1	1	1
5	0	0	0	0	1	1	1	1	1	1	1	1
8	0	0	0	0	0	0	0	1	1	1	1	1
11	0	0	0	0	0	0	0	0	0	0	1	1
...	...	...	...	...	...	...	...	...	...	...	...	...

Vorteile/Nachteile

- Adressierung von Bereichen mit einer AND- und einer NOT-Operation
- Beispiel: Berechnung des Bereichs [2,7]:

```
For i = 1 To Length(...)
B[i] := NOT(B(1)[i]) AND B(7)[i]
End For
```

- Doppelter Aufwand für Punktanfragen, z.B. Position 5

```
For i = 1 To Length(...)
B[i] := NOT(B(4)[i]) AND B(5)[i]
End For
```

Komprimierung von Bitlisten

- Problem: Dünnbesetztheit von Bitlisten bei Attributen mit hoher Kardinalität
- Naiver Ansatz: Klassische Komprimierungstechniken (z.B. Längenkodierung)
- Besserer Ansatz: Repräsentation der numerischen Schlüsselwerte in einem anderen Zahlensystem

Grundidee am Beispiel a=13

- im regulären 10er-System: $(1,3)_{<10,10>} = 1 \bullet (10^0 \bullet 10^1) + 3 \bullet (10^0 \bullet 10^0) = 13$
- im binären Zahlensystem: $(1,1,0,1)_{<2,2,2,2>} = 1 \bullet (2^0 \bullet 2^1 \bullet 2^1 \bullet 2^1) + 1 \bullet (2^0 \bullet 2^0 \bullet 2^1 \bullet 2^1) + 0 \bullet (2^0 \bullet 2^0 \bullet 2^0 \bullet 2^1) + 1 \bullet (2^0 \bullet 2^0 \bullet 2^0 \bullet 2^0)$
- im Zahlensystem zur Basis <16>: $(13)_{<16>} = 13 \bullet (16^0)$
- im Zahlensystem zur Basis <2,4,3>: $(1,0,1)_{<2,4,3>} = 1 \bullet (2^0 \bullet 4^1 \bullet 3^1) + 0 \bullet (2^0 \bullet 4^0 \bullet 3^1) + 1 \bullet (2^0 \bullet 4^0 \bullet 3^0)$

Nutzung zur Komprimierung von Bitlisten

- Menge von Bitlisten für jede Position im Zahlensystem
- Kombination mit Bereichskodierung möglich

	< 16 >															
A	$B^x(0)$	$B^z(1)$	$B^x(2)$	$B^x(3)$	$B^x(4)$	$B^x(5)$	$B^x(6)$	$B^x(7)$	$B^x(8)$	$B^x(9)$	$B^x(10)$	$B^x(11)$	$B^x(12)$	$B^x(13)$	$B^x(14)$	$B^x(15)$
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
11	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...

	< 2		2		2		2 >	
A	$B^{\alpha}(0)$	$B^{\alpha}(1)$	$B^z(0)$	$B^z(1)$	$B^y(0)$	$B^y(1)$	$B^x(0)$	$B^x(1)$
...	...	...	...	...	...	...	...	...
11	0	1	1	0	0	1	0	1
12	0	1	0	1	0	0	1	0
13	0	1	0	1	0	0	0	1
14	0	1	0	1	0	1	1	0
15	0	1	0	1	0	1	0	1
...	...	...	...	...	...	...	...	...

	< 2		4				3 >		
A	$B^z(0)$	$B^z(1)$	$B^y(0)$	$B^y(1)$	$B^y(2)$	$B^y(3)$	$B^x(0)$	$B^x(1)$	$B^x(2)$
...	...	...	...	...	...	...	...	...	...
11	1	0	0	0	0	1	0	0	1
12	0	1	1	0	0	0	1	0	0
13	0	1	1	0	0	0	0	1	0
14	0	1	1	0	0	0	0	0	1
15	0	1	0	1	0	0	1	0	0
...	...	...	...	...	...	...	...	...	...

Rekonstruktion komprimierter Bitlisten

- Rückgriff auf je eine Bitliste aus der Bitlistenmenge einer Position im Zahlensystem
- Beispiel: $a=13=(1,0,1)_{<2,4,3>}$ erfordert Rückgriff auf $B^x(1)$, $B^y(0)$, $B^z(1)$
- $B(13) := B^z(1)$ AND $B^y(0)$ AND $B^x(1)$

Merke

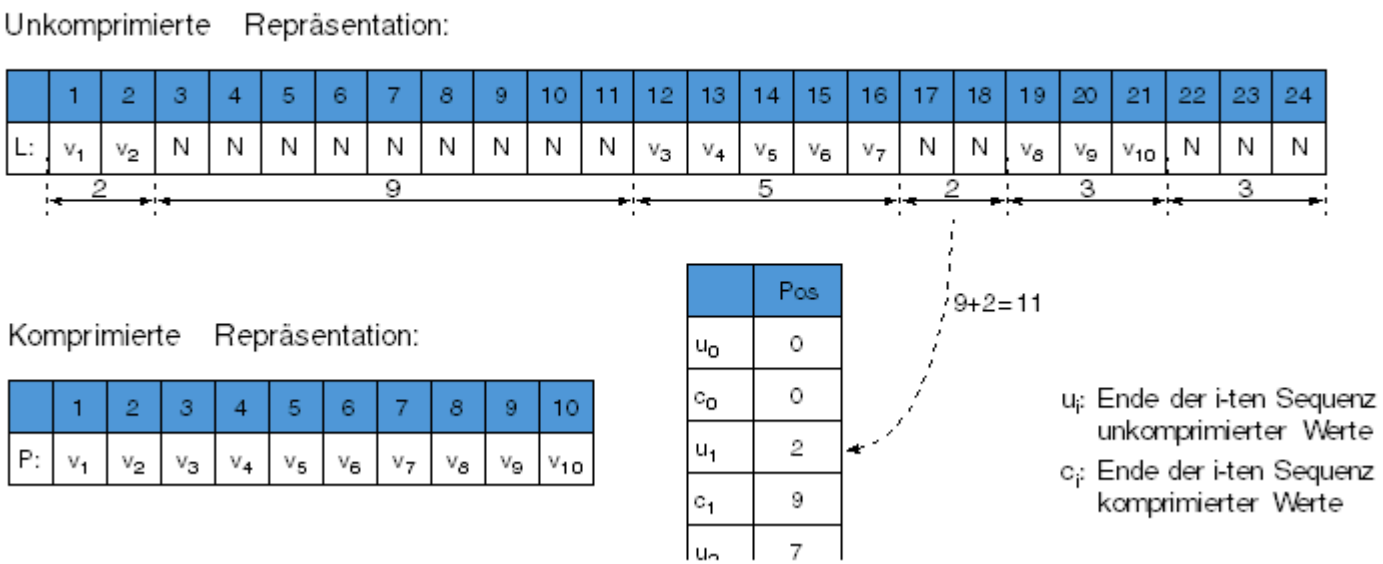
- normale Bitlistenrepräsentation
- im Zahlensystem <N> mit N als Kardinalität des Attributs
- eine Menge von N unterschiedlichen Bitlisten
- maximale Komprimierung
- Binärrepräsentation <2,...,2>
- minimale Anzahl von ld(N) Bitlisten

Prinzip

- Datenbank enthält Vielzahl von NULL-Werten
- Rekonstruktion ohne vollständige Dekomprimierung (Header-Verfahren)

Idee

- ›Header‹-Tabelle zeichnet die kumulierten Teilsequenzen von NULL und tatsächlichen Werten auf (Paare von u_i- und c_i-Werten)
- Direkter Zugriff mit binärer Suche nach ges. Position k auf der Header-Tabelle
- $u_i + c_i < k \leq c_i + u_{i+1}$
  Der unkomprimierte Datensatz befindet sich an der Stelle $k\text{-}c_i$ in der physischen Repräsentation
- $c_{i-1} + u_i < k \leq u_i + c_i$
  Der gesuchte Wert ist eine Konstante (bzw. NULL-Wert) und weist keine physische Repräsentation auf



Fall 1: Wert an Position k=14: i=1 -> Wert ist an Stelle 14-9=5 physisch abgelegt

Fall 2: Wert an Position k=18: i=2 -> NULL-Wert an dieser Position

Hashing

Idee

- direkte Berechnung der Satzadresse über Schlüssel (Schlüsseltransformation)

Hash-Funktion

- h: S {1, 2, ..., n} S = Schlüsselraum
  n = Größe des statischen Hash-Bereiches in Seiten (Buckets)

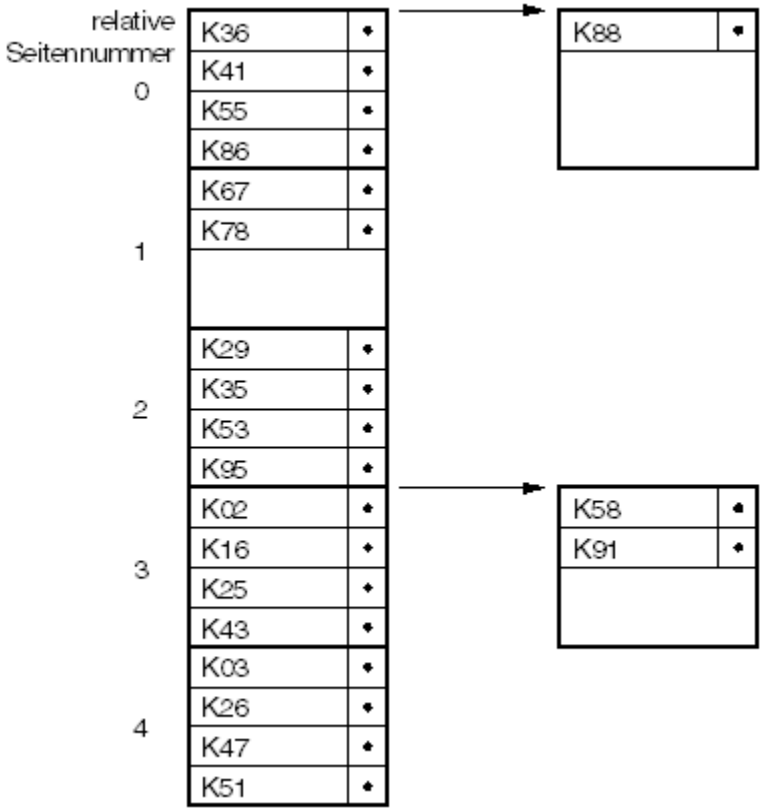
Idealfall: h ist injektiv (keine Kollisionen)

- nur in Ausnahmefällen möglich (’dichte’ Schlüsselmenge)
- jeder Satz kann mit einem einzigen Seitenzugriff referenziert werden

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 Verketten von Überlaufseiten
- typischer Zugriffsfaktor: 1.1 bis 1.4
- Vielzahl von Hash-Funktionen anwendbar
  z. B. Divisionsrestverfahren (Primzahl bestimmt Modul), Faltung, Codierungsmethode, ...

Schlüsselberechnung für K02



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

208 mod 5 = 3

Ziel

- Jeder Satz kann mit genau einem E/A-Zugriff gefunden werden
  → Gekettete Überlaufbereiche können nicht benutzt werden

Statisches Hashing

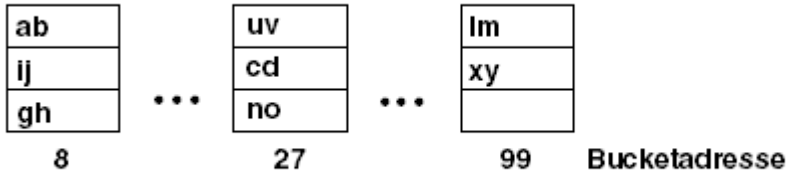
- N Sätze, n Buckets mit Kapazität b
- Belegungsfaktor $\beta = \frac{N}{n \cdot b}$

Überlaufbehandlung

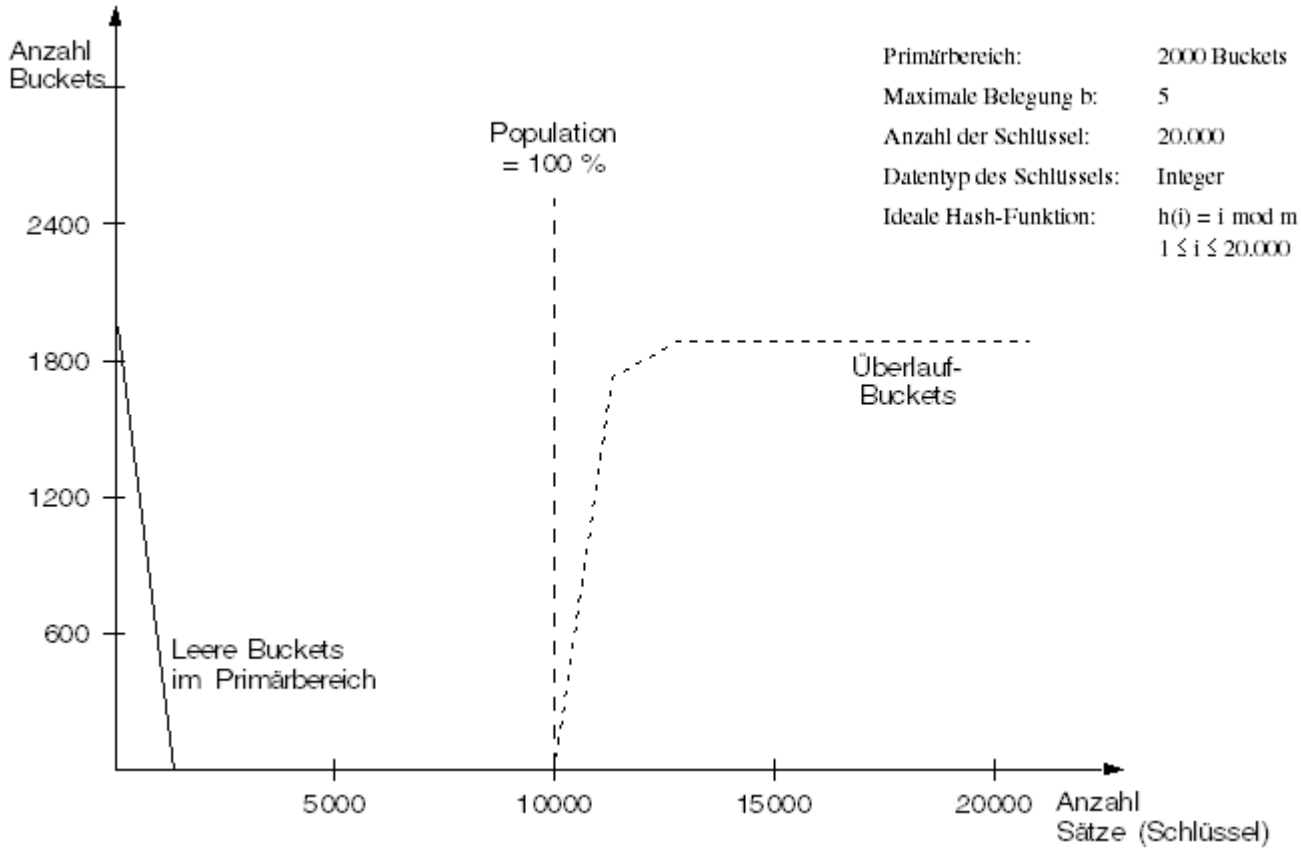
- Open Addressing (ohne Kette oder Zeiger)
- Bekannteste Schemata: Lineares Sondieren und Double Hashing
- Sondierungsfolge für einen Satz mit Schlüssel k:
  - $H(k) = (h_1(k), h_2(k), ..., h_n(k))$
  - bestimmt Überprüfungsreihenfolge der Buckets (Seiten) beim Einfügen und Suchen
  - wird durch k festgelegt und ist eine Permutation der Menge der Bucketadressen {0, 1, ..., n-1}

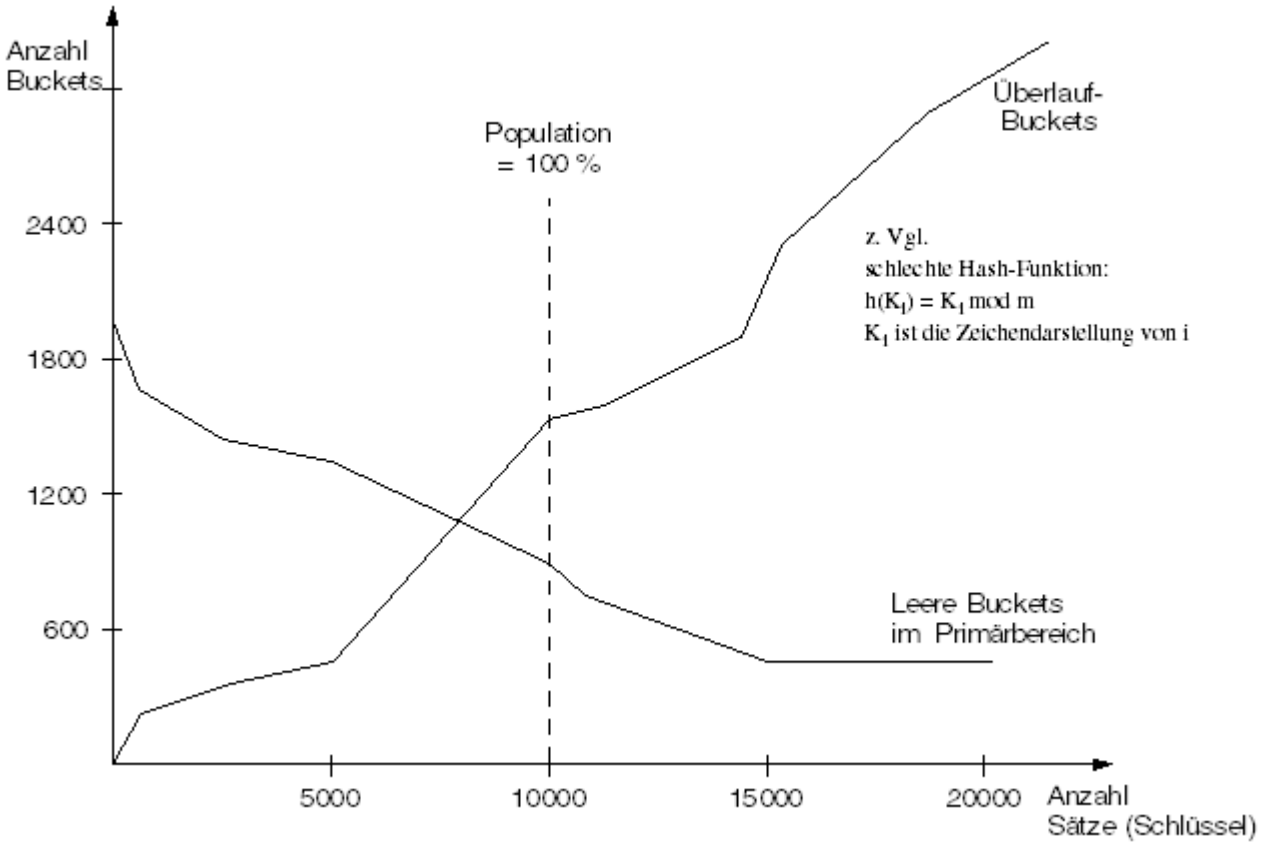
Erster Versuch

- Aufsuchen oder Einfügen von k = xy



- Sondierungsfolge sei H(xy) = (8, 27, 99, ...)
- Viele E/A-Zugriffe
- Wie funktioniert das Suchen und Löschen?


Anzahl Buckets
2400
1800
1200
600
Population = 100 %
Überlauf-Buckets
Leere Buckets im Primärbereich
5000
10000
15000
20000
Anzahl Sätze (Schlüssel)
Primärbereich: 2000 Buckets
Maximale Belegung b: 5
Anzahl der Schlüssel: 20.000
Datentyp des Schlüssels: Integer
Ideale Hash-Funktion: h(i) = i mod m
1 ≤ i ≤ 20.000


Anzahl Buckets
2400
1800
1200
600
Population = 100 %
Überlauf-Buckets
z. Vgl.
schlechte Hash-Funktion:
h(K_i) = K_i mod m
K_i ist die Zeichendarstellung von i
Leere Buckets im Primärbereich
5000
10000
15000
20000
Anzahl Sätze (Schlüssel)

Einsatz von Signaturen

- Jede Signatur $s_i(k)$ ist ein t-Bit Integer
- Für jeden Satz mit Schlüssel k wird eine Signaturfolge benötigt:
  $S(k) = (s_1(k), s_2(k), ..., s_n(k))$
- Die Signaturfolge wird eindeutig durch den Schlüsselwert 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!

Einsatz von Separatoren

- Ein Separator besteht aus t Bits (entspricht also einer Signatur)
- Separator j, j = 0, 1, 2, ..., n-1, gehört zu Bucket j
- Eine Separatortabelle SEP enthält n Separatoren und wird im Hauptspeicher gehalten.

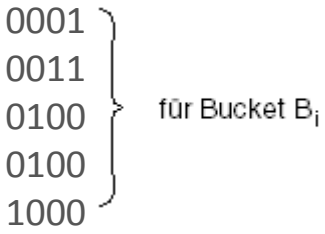
Nutzung der Separatoren

- Wenn Bucket B_i r-mal (r > b) sondiert wurde, müssen mindestens (r - b) Sätze abgewiesen werden; sie müssen das nächste Bucket in ihrer Sondierungsfolge aufsuchen.
- 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_i gespeichert, die mit hohen Signaturen müssen weitersuchen.
- Eine Signatur, die die Gruppe der niedrigen Signaturen eindeutig von der Gruppe 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. Wenn die ideale Partitionierung (b, rb) nicht gewählt werden kann, wird eine der folgenden Partitionierungen versucht: (b-1, r-b+1), (b-2, r-b+2), ..., (o, r)

→ Ein Bucket mit Überlaufsätzen kann weniger als b Sätze gespeichert haben.

Beispiel

- Parameter: r = 5, t= 4
  - Signaturen



  - b = 4: Separator = 1000, Aufteilung (4, 1)
  - → SEP [j] = 1000

  - b = 3: Separator = 0100, Aufteilung (2, 3)
  - → SEP [j] = 0100

Initialisierung der Separatoren mit 2^t-1

- Separator eines Buckets, der noch nicht übergelaufen ist, muss höher als alle tatsächlich auftretenden Signaturen sein -> 2^t-1
- Bereich der Signaturen: 0, 1, ..., 2^t-2

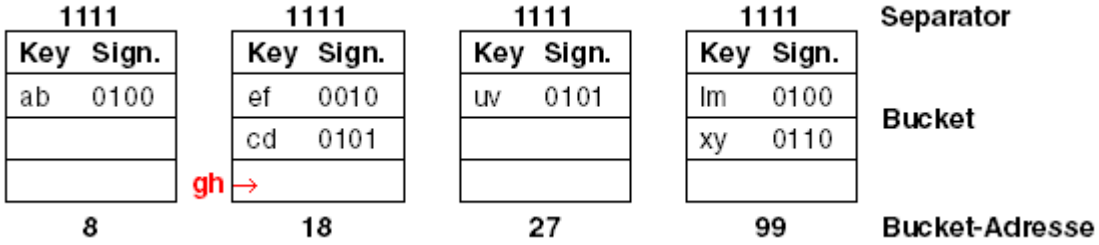
Aufsuchen

- In der Sondierungsfolge 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 $h_i(k)$ lokalisiert.
- Das Bucket wird eingelesen und durchsucht. Wenn der Satz nicht gefunden wird, existiert er nicht.
  → genau ein E/A-Zugriff erforderlich

Einfügen

- Kann Verschieben von Sätzen und Ändern von Separatoren erfordern.
- Ein Satz mit $s_i(k)$<SEP[j] mit j=$h_i(k)$ muss 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 Sondierungsfolgen wieder eingefügt werden
  → Dieser Prozess kann kaskadieren
  → b nahe bei 1 ist unsinnig, da die Einfügekosten explodieren; Empfehlung: b<0.8

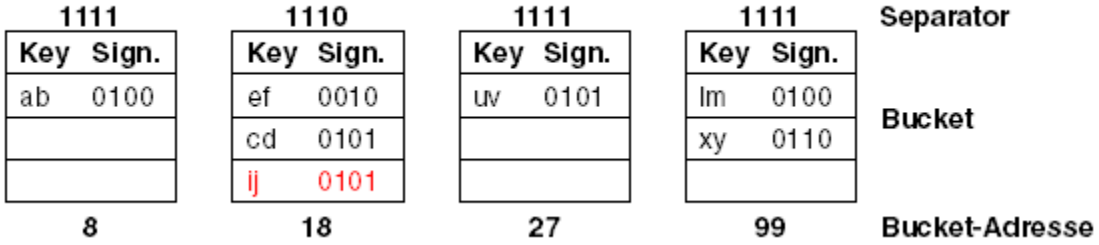
Beispiel 1: Startsituation



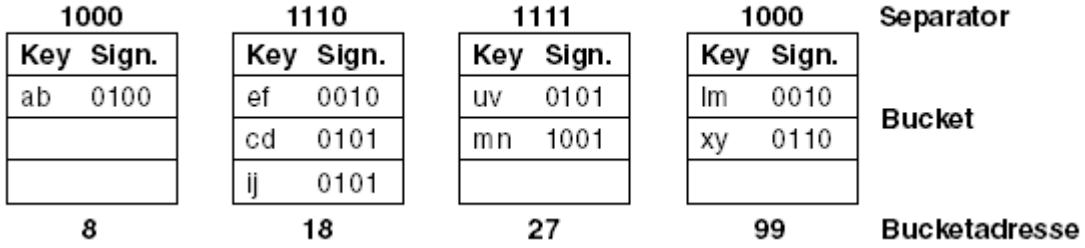
- Einfügen von k = gh mit $h_1(gh)=18$, $s_1(gh) = 1110$
- Einfügen von k = ij mit $h_1(ij) = 18$, $s_1(ij) = 0101$

Erster Bucketüberlauf

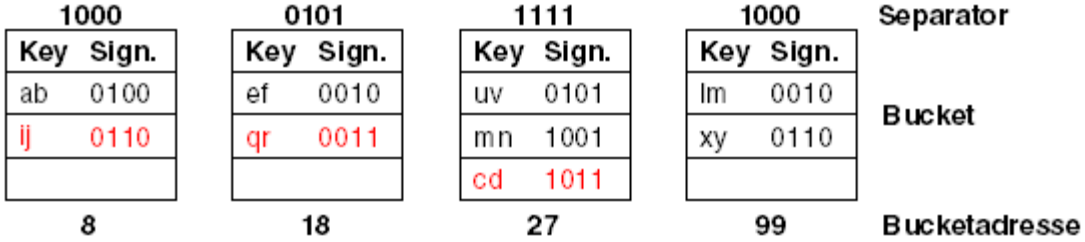
- k = gh muss weiter sondieren: z.B.: $h_2(gh) = 99$, $s_2(gh) = 1010$



Beispiel 2: Situation nach weiteren Einfügungen und Löschungen



- Einfügung von H(qr) = (8, 18, ...) und S(qr) = (1011, 0011, ...)



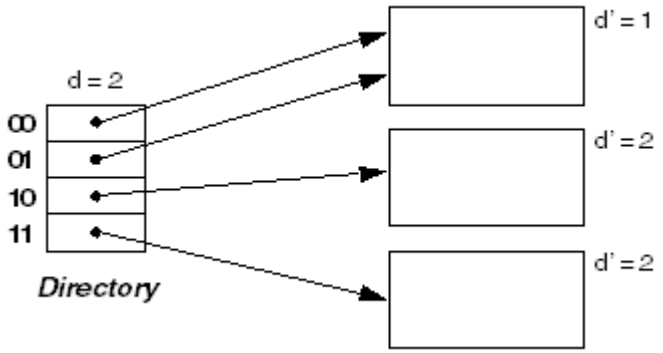
- Sondierungs- und Signaturfolgen von cd und ij seien
  - H (cd) = (18, 27, ...) und S (cd) = (0101, 1011, ...)
  - H (ij) = (18, 99, 8, ...) und S (ij) = (0101, 1110, 0110, ...)

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 = MAX (d')

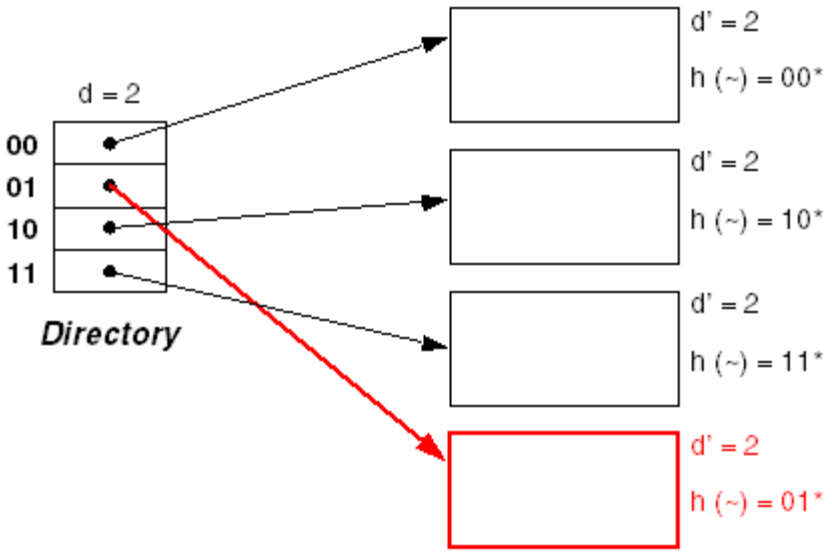


Situation

- Splitting von Buckets

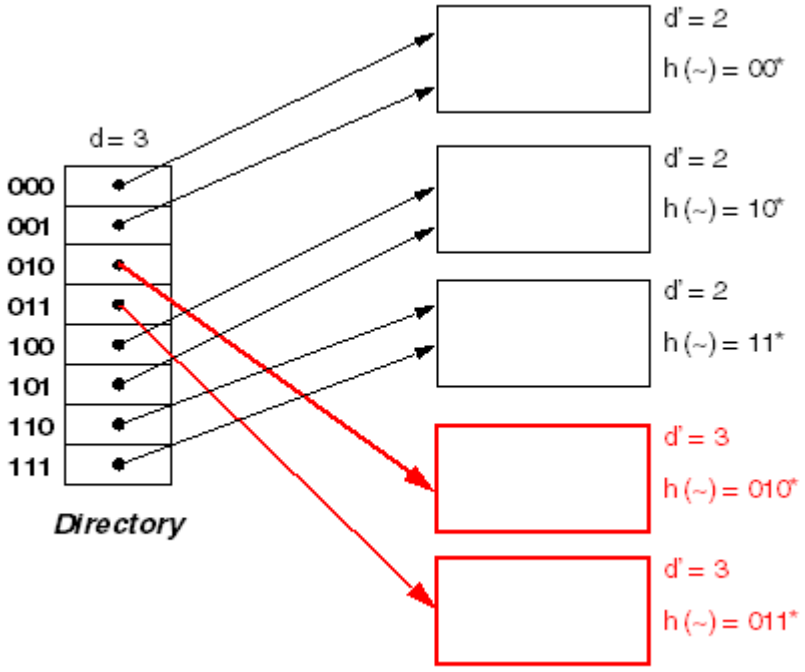
Fall 1

- Überlauf eines Buckets, dessen lokale Tiefe kleiner als die globale Tiefe d ist
  → lokale Neuverteilung der Daten
- Erhöhung der lokalen Tiefe
- lokale Korrektur der Pointer
  im Directory



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



Dynamisches Wachsen und Schrumpfen des (primären) Hash-Bereichs

- minimale Verwaltungsdaten
- keine großen Directories für die Hash-Datei

- Aber: es gibt keine Möglichkeit, Überlaufsätze vollständig zu vermeiden!
  - eine hohe Rate von Überlaufsätzen wird als Indikator dafür genommen, dass die Datei eine zu hohe Belegung aufweist und deshalb erweitert werden muss
  - Buckets werden in einer fest vorgegebenen Reihenfolge gesplittet
    → einzige Information: nächstes zu splittendes Bucket

Prinzipieller Ansatz

- n: Größe der Ausgangsdatei in Buckets
- Folge von Hash-Funktionen h_0, h_1, ...
  - wobei $h_0(k)$ Î {0,1,..., n-1} und

    $h_{j+1}(k) = h_j(k)$

    oder

    $h_{j+1}(k) = h_j(k) + n \bullet 2^j$ für alle j ³ 0 und alle Schlüssel k gilt
  - gleiche Wahrscheinlichkeit für beide Fälle von h_{j+1} erwünscht
- Beispiel: $h_j(k) = k(\text{mod } n \bullet 2^j)$, j = 0,1, ...

Beschreibung des Dateizustandes

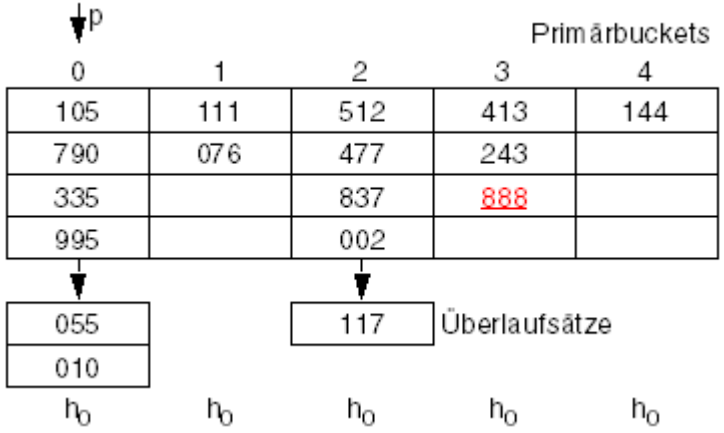
- L: Anzahl der bereits ausgeführten Verdopplungen
- N: Anzahl der gespeicherten Sätze
- b: Kapazität eines Buckets
- p: zeigt auf nächstes zu splittendes Bucket ($0 \leq p < n \times 2^L$)

- β: Belegungsfaktor = $\frac{N}{(n \cdot 2^L + p) \cdot b}$

Beispiel:

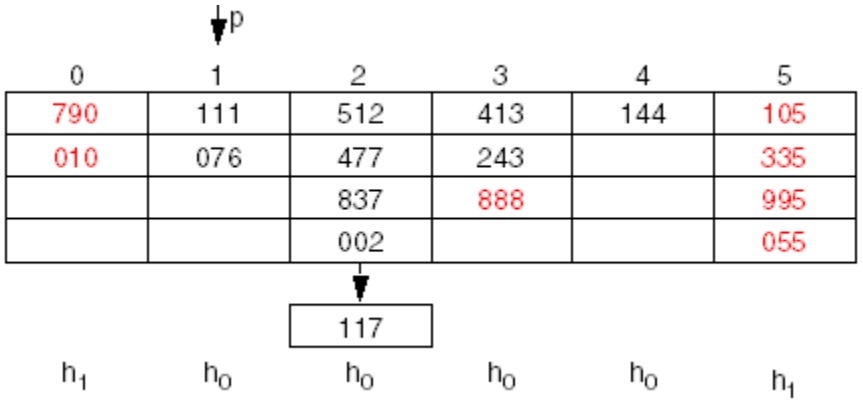
Prinzip des linearen Hashing

- h_0 (k) = k mod 5
- h_1 (k) = k mod 10, ...
- b = 4, L = 0, n = 5
- Splitting, sobald $\beta > \beta_s = 0.8$

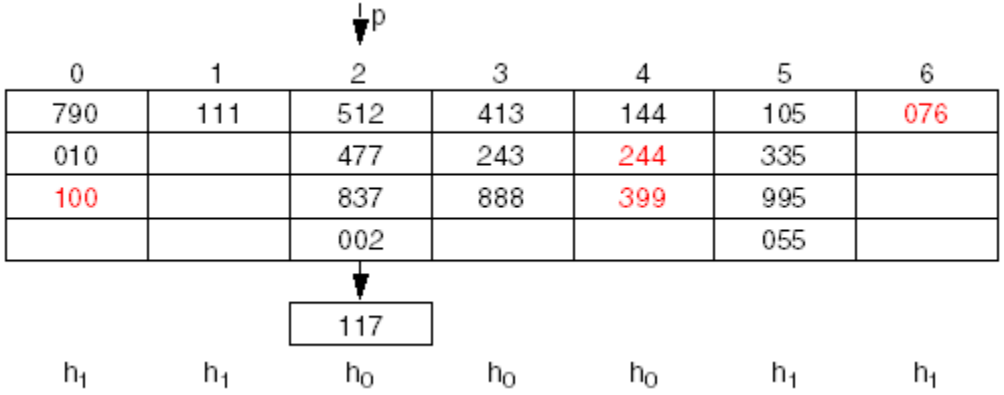


Splitting

- Einfügen von 888 erhöht Belegung auf β = 17/20 = 0.85



- Einfügen von 244, 399 und 100 erhöht Belegung auf β = 20/24 = 0.83
- Auslösen eines Splitting-Vorgangs:



Splitting

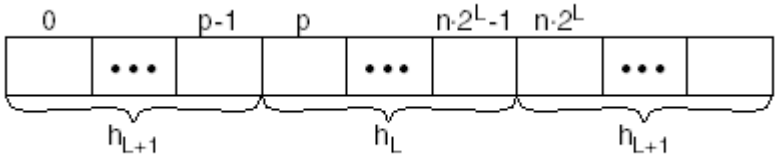
- Auslöser: β
- Position: p
- Datei wird um 1 vergrößert
- p wird inkrementiert: $p = (p+1) \bmod (n \bullet 2^L)$
- Wenn p wieder auf Null gesetzt wird (Verdopplung der Datei beendet), wird L wiederum inkrementiert

Adressberechnung

- Wenn $h_0(k) \geq p$, dann ist h_0 die gewünschte Adresse
- Wenn $h_0(k) < p$, dann war das Bucket bereits gesplittet. $h_1(k)$ liefert die gewünschte Adresse
- Allgemein: $h := H_L(k)$;

  if h < p then

  $h := h_{L+1}(k)$;



Split-Strategien

- Unkontrolliertes Splitting
  - Splitting, sobald ein Satz in den Überlaufbereich kommt
  - $\beta \sim 0.6$, schnelleres Aufsuchen
- Kontrolliertes Splitting
  - Splitting, wenn ein Satz in den Überlaufbereich kommt und $b > b_s$
  - $\beta \sim \beta_s$, längere Überlaufketten möglich

B-Baum / B-Baum*

- selbstorganisierend, dynamische Reorganisation
- garantierte Speicherplatzausnutzung
  - jeder Knoten (bis auf die Wurzel) immer mindestens halb voll, d.h. Speicherausnutzung garantiert >= 50 %
  - bei zufälliger und gleichverteilter Einfügung Speicherausnutzung ln(2), also rund 70 %
- Effizientes Suchen einfach zu realisieren
- Aufwendige Einfüge- und Löschoperationen

Bit-Index

- keine Hierachie, optimal für Attribute mit geringer Ausprägung und logischen Verknüpfungsoperationen

Hashing

- direkte Berechnung der Satzadresse
- Problem: Dynamisches Wachstum der Datenbereiche

Zugriffsverfahren	Speicherungsstruktur	Direkter Zugriff	Sequentielle Verarbeitung	Änderungsdienst (Ändern ohne Aufsuchen)
fortlaufender Schlüsselvergleich	sequentielle Liste gekettete Liste	$O(N) \approx 10^4$ $O(N) \approx 5 \cdot 10^5$	$O(N) \approx 2 \cdot 10^4$ $O(N) \approx 10^6$	$O(1) \leq 2$ $O(1) \leq 3$
Baumstrukturierter Schlüsselvergleich	Balancierte Binärbäume Mehrwegbäume	$O(\log_2 N) \approx 20$ $O(\log_k N) \approx 3 - 4$	$O(N) \approx 10^6$ $O(N) \approx 10^{6a}$	$O(1) = 2$ $O(1) = 2$
Konstante Schlüssel-transformationsverfahren	Externes Hashing mit separatem Überlaufbereich Externes Hashing mit Separatoren	$O(1) \approx 1.1 - 1.4$ $O(1) = 1$	$O(N \log_2 N)^b$ $O(N \log_2 N)^b$	$O(1) \approx 1.1$ $O(1) = 1$ (+D)
Variable Schlüsseltrans-formationsverfahren	Erweiterbares Hashing Lineares Hashing	$O(1) = 2$ $O(1) = 1$	$O(N \log_2 N)^b$ $O(N \log_2 N)^b$	$O(1) \approx 1.1$ (+R) $O(1) < 2$

a. Bei Clusterbildung bis zu Faktor 50 geringer

b. Physisch sequentielles Lesen, Sortieren und sequentielles Verarbeiten der gesamten Sätze, Beispielangaben für $N = 10^6$