Hasso Plattner Institut
IT Systems Engineering | Universität Potsdam

Datenbanksysteme II

Indexstrukturen

(Kapitel 13)

30.4.2007

Felix Naumann

Motivation

Platzierung der Tupel in Blöcke

- Naiv: Beliebig verteilen
  - SELECT * FROM R
  - Jeden Block untersuchen (Header-Datei)
- Besser: Tupel eine Relation zusammenhängend speichern
  - SELECT * FROM R WHERE a=10
  - Alle Datensätze betrachten
- Noch besser: Index
  - Input: Eigenschaften von Datensätzen (z.B: Feldwert)
    - „Suchschlüssel"
  - Schneller Output: Die entsprechenden Tupel
  - Nur wenige Datensätze werden betrachtet

Primärschlüssel
Sortierschlüssel
Suchschlüssel

Überblick

3

- Indizes auf sequenziellen Dateien
- Sekundärindizes auf nicht-sequenziellen Dateien
- B-Bäume
- Hash-Tabellen

Einfachste Form eines Index

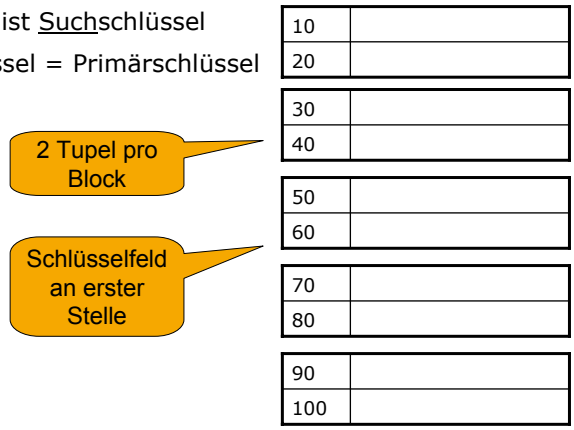
4

- Gegeben sortierte Datei (*data file*)
- Indexdatei enthält Schlüssel-Pointer Paare
  - Schlüsselwert K ist mit einem Pointer verbunden
  - Pointer zeigt auf Datensatz, der den Schlüsselwert K hat.
- Dichtbesetzter Index
  - Ein Eintrag im Index für jeden Datensatz
- Dünnbesetzter Index
  - Nur einige Datensätze sind im Index repräsentiert.
  - Z.B. ein Eintrag pro Block

Sequenzielle Dateien

5

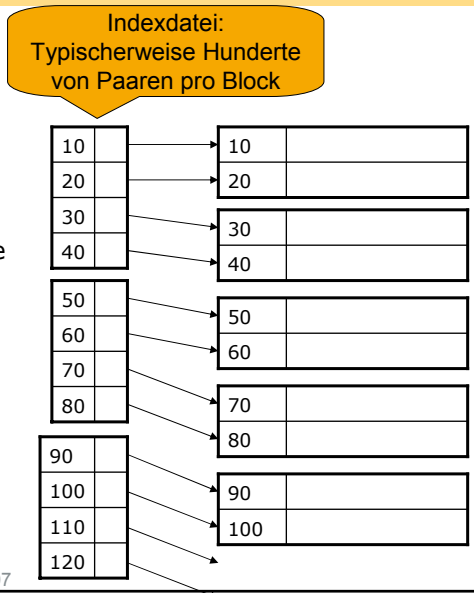
- Index kann sich auf Sortierung des Schlüsselattributs verlassen
  - Hier: Schlüssel ist Suchschlüssel
  - Oft: Suchschlüssel = Primärschlüssel



Dichtbesetzte Indizes

6

- Blocksequenz mit Schlüssel-Pointer Paaren
- Jeder Schlüssel der Daten ist durch ein Paar repräsentiert
  - Aber: Wesentlich kleinere Datenmenge
  - Passt womöglich in den Hauptspeicher
  - Nur ein I/O pro Zugriff
- Sortierung der Paare = Sortierung der Daten



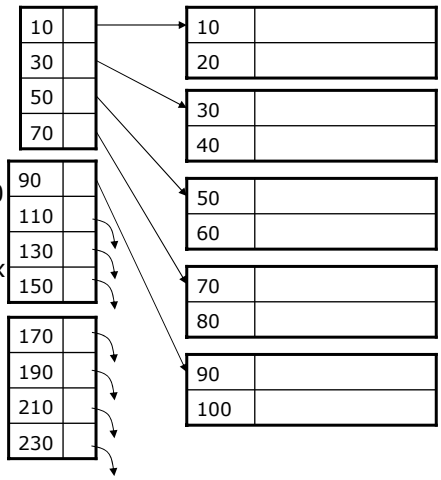
Anfragebearbeitung

- Gegeben Suchschlüssel K
1. Durchsuche Indexdatei nach K
2. Folge Pointer
3. Lade Block aus Datendatei
- Beschleunigung
  - Indexdatei hat nur wenige Blocks
  - Binäre Suche um K zu finden
  - Indexdatei im Hauptspeicher

- Beispiel: 1.000.000 Tupel
  - Block: 4096 Byte = 10 Tupel
  - Gesamtgröße 400 MB
  - Schlüsselfeld hat 30 Byte
  - Pointer hat 8 Byte
  - => 100 Paare pro Block
  - Dichtbesetzter Index: 10.000 Blöcke für Index
    - 40 MB => vielleicht OK im Hauptspeicher
  - Binäre Suche: $\log_2(10.000) \approx 13$
    - => 13-14 Blocks pro Suche
    - Wichtigsten Blöcke im Hauptspeicher reichen (1/2, ¼, ¾, …)

Dünnbesetzte Indizes

- Weniger Speicherbedarf
- Aber höherer Suchaufwand
- Nur ein Pointer pro Block
- Beispiel
  - 100.000 Datenblöcke, 100 Indexpaare pro Block
  - => 1.000 Blocks für Index = 4MB



Anfragebearbeitung mit dünnbesetzten Indizes

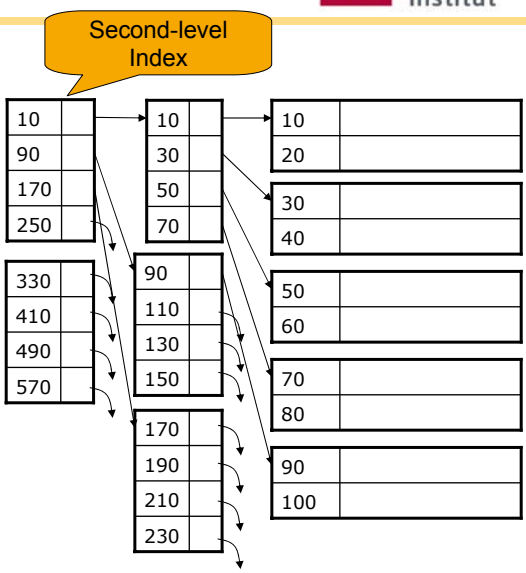
1. Suche im Index größten Schlüssel, der kleiner als Suchschlüssel ist.
   - Binäre Suche (leicht modifiziert)
2. Hole assoziierten Datenblock
3. Durchsuche Block nach Datensatz

- Nachteil?
  - „SELECT `TRUE‘ FROM R WHERE a=10“
  - Kann nicht ausschließlich mit Index beantwortet werden

Mehrstufiger Index

Auch ein Index kann unangenehm groß sein.

- Nimmt viel Speicher ein
- Kostet viel I/O
  - Auch bei binärer Suche
- Idee: Index auf den Index
  - Zweiter Index macht nur als dünn-besetzter Index Sinn.
- Theoretisch auch dritte, vierte, … Ebene
  - B-Baum aber besser

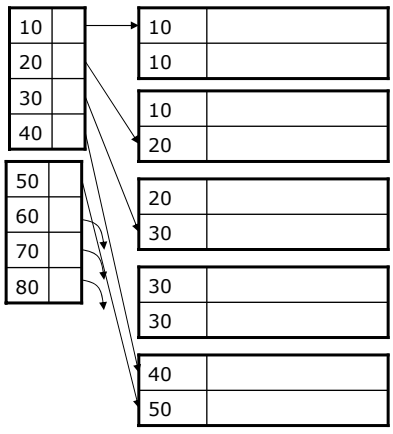


Mehrstufiger Index – Beispiel

- 100.000 Datenblöcke, 100 Indexpaare pro Block
- => 1.000 Blocks für Index erster Stufe
  = 4MB
- => 10 Blocks für Index zweiter Stufe
  = 40KB
- Kann mit Sicherheit im Hauptspeicher verbleiben
- Vorgehen
  1. Suche im Index zweiter Stufe größten Schlüssel, der kleiner als Suchschlüssel ist.
  2. Hole entsprechenden Block im Index erster Stufe.
     - Eventuell schon im Hauptspeicher
  3. Suche in dem Block größten Schlüssel, der kleiner als Suchschlüssel ist.
  4. Hole entsprechenden Datenblock.
  5. Suche Datensatz (falls Index erster Stufe dünnbesetzt ist).
- Zusammen: 2 I/Os

Indizes für Nicht-eindeutige Suchschlüssel

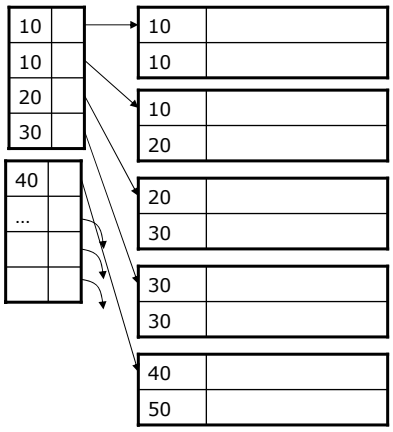
- Annahme bisher: Suchschlüssel ist auch Schlüssel der Relation
- Annahme weiter: Relation ist nach Suchschlüssel sortiert
- Idee 1: Dichtbesetzter Index: ein Paar im Index für jeden Datensatz
  - Anfragebearbeitung:
    - Suche erstes Paar mit K.
    - Wähle alle weiteren mit K (direkt dahinter)
    - Hole entsprechende Datensätze.
- Idee 2: Nur ein Indexpaar pro Schlüsselwert. Der zeigt auf ersten Datensatz mit K.
  - Weitere Datensätze mit K folgen direkt.
  - Wichtig: Blöcke haben Pointer auf jeweils nächsten Block



Indizes für Nicht-eindeutige Suchschlüssel

13

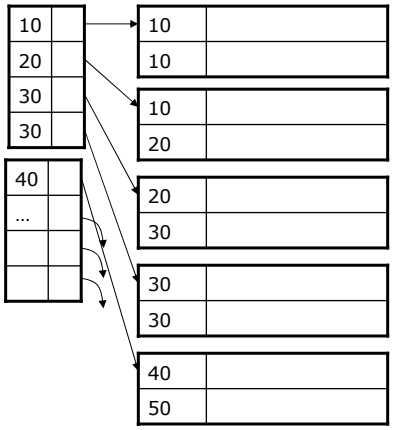
- Idee 3: Dünnbesetzter Index wie gehabt
  - Pointer jeweils auf Datensatz am Blockanfang
- Anfragebearbeitung
  - Suche letzten Eintrag E1 im Index, dessen Datenwert ≤ K
  - Suche von dort im Index nach vorn bis zu einem Eintrag E2 mit Datenwert < K
  - Hole alle Datenblöcke zwischen und inklusive E1 und E2.
- Beispiel: K = 20



Indizes für Nicht-eindeutige Suchschlüssel

14

- Idee 4: Dünnbesetzter Index; aber
  - Datenwert im Index ist der kleinste neue Wert im entsprechenden Datenblock.
  - Falls kein neuer Wert im Block, dann den existierenden Wert.
- Anfragebearbeitung einfacher
  - Suche im Index nach Paar mit Datenwert = K oder < K aber nächste Wert ist > K.
  - Hole Datenblock und gegebenenfalls folgende Datenblöcke.



Änderungsoperationen

15

Daten ändern sich (Insert, Update, Delete).

- Annahme bisher: Daten füllen Blöcke perfekt und ändern sich nicht
- Änderungen im Datenblock: Siehe voriger Foliensatz
  - Overflow Blocks
    - In dünnbesetzten Indizes nicht repräsentiert
  - Neue Blöcke in der Sequenz
    - Benötigen neuen Indexeintrag
    - Indexänderungen bergen dieselben Probleme wie Datenänderungen.
      - » Platzierung der Blocks
      - » In Indizes höherer Stufe
  - Tupel verschieben
    - Index muss angepasst werden.
- Generelle Regel: Indizes können wie normale data files behandelt werden. Gleiche Strategien können angewendet werden.

Änderungsoperationen

16

Index für Datensätze

Index für Blöcke

Aktion	Dichtbesetzter Index	Dünnbesetzter Index
Erzeugung eines leeren Overflow Blocks		
Löschen eines leeren Overflow Blocks		
Erzeugen eines leeren sequenziellen Blocks		
Löschen eines leeren sequenziellen Blocks		
Datensatz einfügen		
Datensatz löschen		
Datensatz verschieben		

Änderungsoperationen

Index für Datensätze

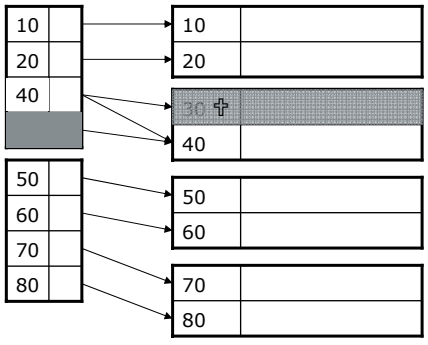
Index für Blöcke

Aktion	Dichtbesetzter Index	Dünnbesetzter Index
Erzeugung eines leeren Overflow Blocks	%	%
Löschen eines leeren Overflow Blocks	%	%
Erzeugen eines leeren sequenziellen Blocks	%	Insert
Löschen eines leeren sequenziellen Blocks	%	Delete
Datensatz einfügen	Insert	Update?
Datensatz löschen	Delete	Update?
Datensatz verschieben	Update	Update?

Nur falls Datensatz erster im Block ist

Änderungsoperationen – Beispiele

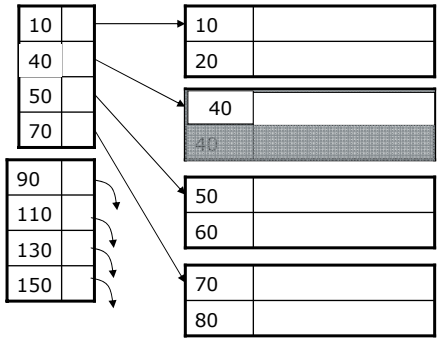
- Datensatz mit K = 30 wird gelöscht.
- Annahme: Block kann/soll nicht reorganisiert werden.
  - Ersatz durch *tombstone*
- Datensatz 40 wird nicht verschoben.
- Index kann reorganisiert werden.
  - Main memory



Änderungsoperationen – Beispiele

19

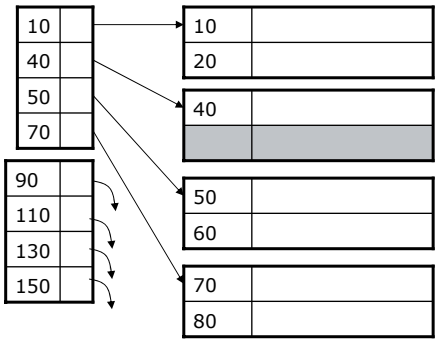
- Datensatz mit K = 30 wird gelöscht.
  - Annahme: Block kann reorganisiert werden.
  - Datensatz 40 wird verschoben
  - Index wird aktualisiert
- Nun auch Datensatz mit K=40 Löschen
  - Leerer Block entsteht
  - Index wird aktualisiert (löschen)
  - Index wird reorganisiert



Änderungsoperationen – Beispiele

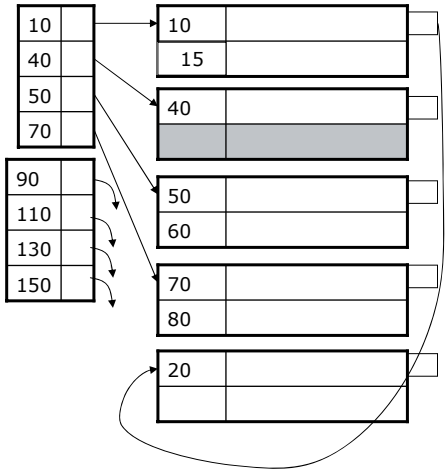
20

- Einfügen eines Datensatzes 15
  - Block 1 ist voll.
  - Datensatz 20 wird in nächsten Block verschoben.
    - Block wird reorganisiert.
  - Datensatz 15 wird eingefügt.
  - Index wird aktualisiert.
    - 20 statt 40



Änderungsoperationen – Beispiele

- Wieder Datensatz 15 einfügen
  - Diesmal mit Overflow Blocks
  - Block 1 ist voll.
  - Datensatz 20 wird in Overflow Block verschoben.
  - Datensatz 15 wird eingefügt.
  - Index bleibt gleich



Überblick

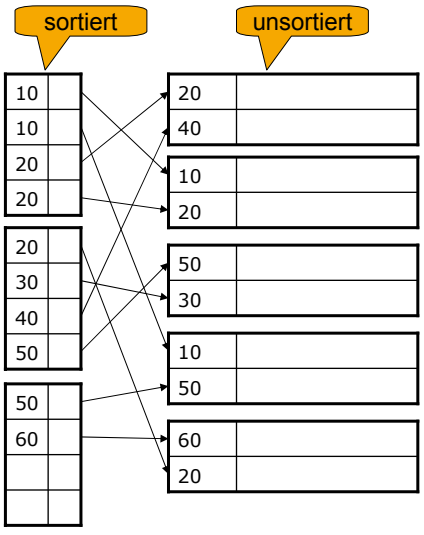
- Indizes auf sequenziellen Dateien
- Sekundärindizes
- B-Bäume
- Hash-Tabellen

Motivation

- Annahme bisher: Datensätze sind nach Schlüssel sortiert
  - „Primärindex"
- Oft sinnvoll: Mehrere Indizes pro Relation
- **Schauspieler(Name, Adresse, Geschlecht, Geburtstag)**
  - **Name** ist Primärschlüssel => Primärindex
  - ```
    SELECT Name, Adresse
    FROM Schauspieler
    WHERE Geburtstag = DATE `1952-01-01'
    ```
- Sekundärindex auf **Geburtstag** beschleunigt Anfragebearbeitung.
  - ```
    CREATE INDEX GEBIndex ON Schauspieler(Geburtstag)
    ```
- Sekundärindizes bestimmen nicht Platzierung der Datensätze, sondern geben Speicherort an.
  - Dünnbesetzte Sekundärindizes sind sinnlos.
  - => Sekundärindizes sind immer dichtbesetzt.

Aufbau von Sekundärindizes

- Dichtbesetzt; mit Duplikaten
- Schlüssel-Pointer Paare
- Schlüssel sind sortiert
- Index zweiter Stufe wäre wiederum dünnbesetzt
- Suche kostet idR mehr I/O
  - Beispiel: Suche nach „20" muss 5 Blöcke lesen.
  - Ist nicht zu ändern: Daten sind halt nach einem anderen Schlüssel sortiert.



Anwendungen

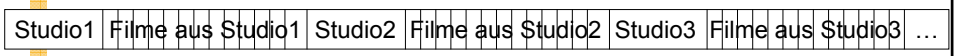
25

- Unterstützung von Selektionsbedingungen auf Nicht-Primärschlüssel
- Datensätze liegen nicht sortiert vor.
  - Sekundärindex auf Primärschlüssel
- Datensätze aus zwei Relationen werden geclustered gespeichert.
  - N:1 Beziehung zwischen R und S
  - Speicher Datensätze aus R direkt beim entsprechenden Datensatz aus S.

Anwendung: Clustered file

26

- **Filme(Titel, Jahr, Länge, inFarbe, Studioname, Produzent)**
- **Studio(Name, Adresse, Präsident)**
- Häufige Anfrageform:
  - ```
    SELECT Titel, Jahr
    FROM   Filme, Studio
    WHERE  Präsident = ???
    AND    Filme.Studioname = Studio.Name
    ```

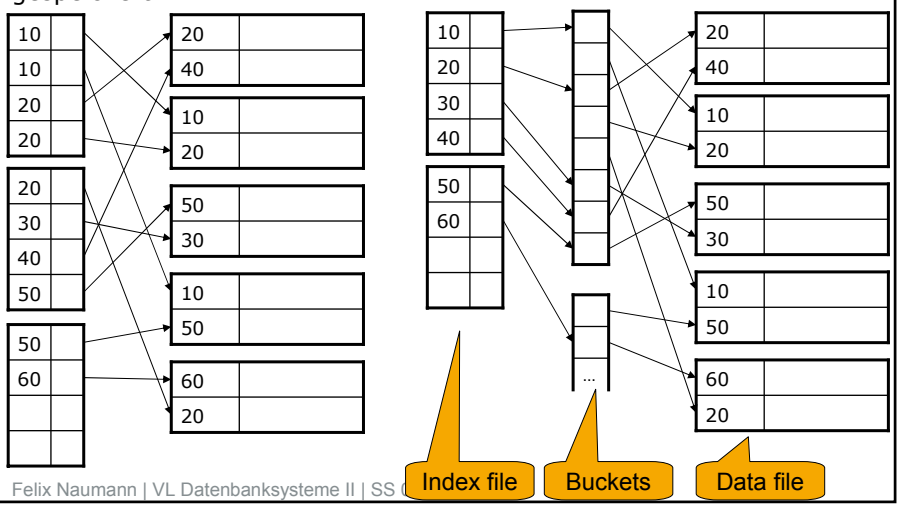


1. Index auf Präsident findet schnell Studio-Datensatz.
2. Entsprechende Filme folgen direkt.
    ```

Indirektion für Sekundärindizes

Können groß sein: Nicht Primärschlüssel

Bisherige Struktur verbraucht Platz: Datenwerte werden mehrfach gespeichert.



Indirektion für Sekundärindizes

- Spart Platz, falls
  - Suchschlüssel sind größer als Pointer.
  - Suchschlüssel im Durchschnitt mindestens zweimal auftauchen.
- Weiterer Vorteil: Bestimmte Anfragen können direkt anhand der Buckets beantwortet werden.
  - Mehrere Selektionsbedingungen, jeweils mit Sekundärindex: Schnittmenge der Pointer in Buckets
  - **`Filme(Titel, Jahr, Länge, inFarbe, Studioname, Produzent)`**
  -
```
SELECT Titel FROM Filme
WHERE  StudioName = `Disney‘
AND    Jahr = 1995
```

Invertierte Indizes

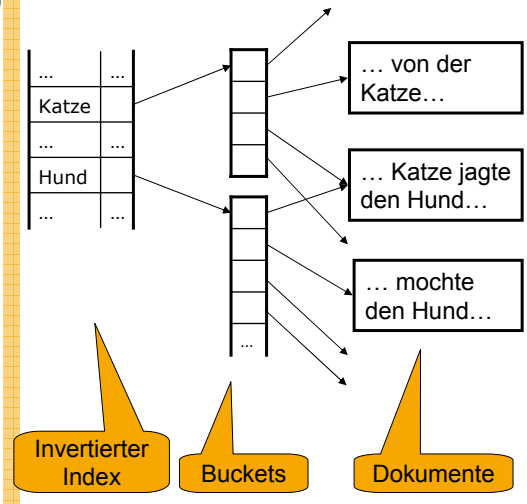
29

Motivation: Dokumenten-Retrieval

- Sichtweise Dokument als Relation
  - **`Dok(hatKatze, hatHund, hatHaus, …)`**
  - Tausende Boolesche Attribute: **`True`** bedeutet das Dokument enthält das Wort.
  - Sekundärindex auf jedes Attribut
    - Aber: Nur die True-Werte werden indiziert
  - Alle Indizes in einen kombiniert: Invertierte Liste
    - Verwendet Indirektion

Invertierte Indizes

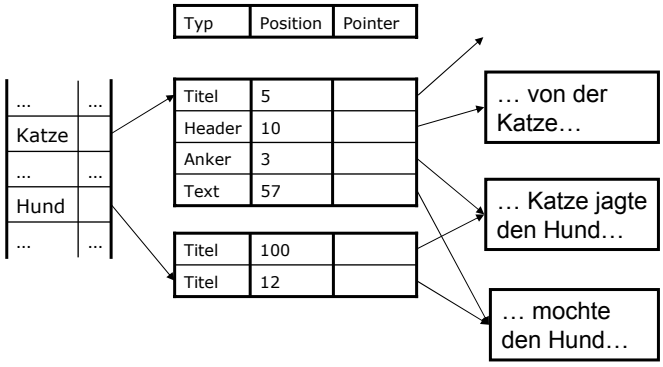
30



- Pointer in den Buckets
  - Auf ein Dokument
  - Auf eine Stelle im Dokument
- Erweiterung: Bucket speichert nicht nur Stelle sondern auch Metadaten
  - Art des Vorkommens (Titel, Abstract, Text, Tabelle, …)
  - Satz (fett, kursiv, …)
  - …
- Anfragen: AND, OR, NOT
  - Durch Schneiden der Pointermengen

Invertierte Indizes

31



Suche nach Dokumenten, die Hunde und Katzen vergleichen

- Dokument erwähnt „Hund" ´im Titel
- Dokument erwähnt „Katze" in einem Anker (Link auf anderes Dokument)

Überblick

32

- Indizes auf sequenziellen Dateien
- Sekundärindizes
- B-Bäume
  - Aufbau
  - Suche
  - Updates
  - Effizienz
  - Varianten
- Hash-Tabellen

B-Bäume

- Bisher: Zweistufiger Index zur Beschleunigung des Zugriffs
- Allgemein: B Bäume (hier B+ Bäume)
  - So viele Stufen wie nötig
  - Blöcke sind mindestens zur Hälfte gefüllt
  - Overflow blocks nicht notwendig

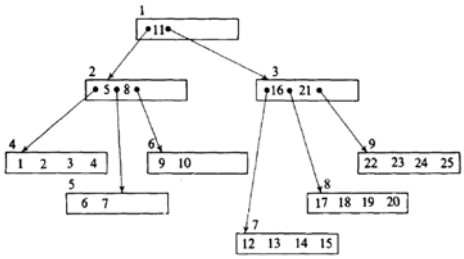
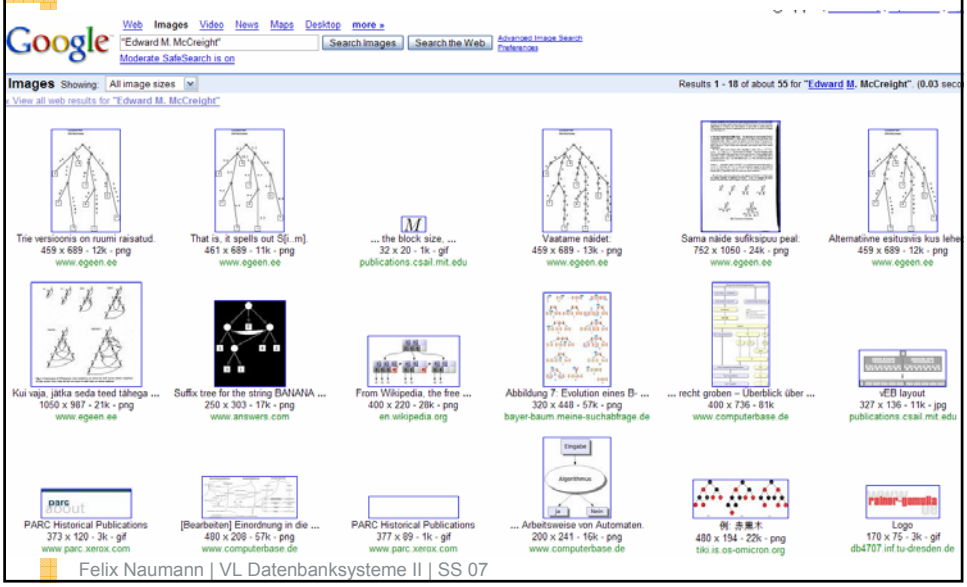


Fig. 2. A data structure in τ(2, 3) for an index

B-Bäume



Struktur

35

- Index-Blöcke in einem Baum organisiert.
- Balanciert
  - Jeder Weg von Wurzel zu Blatt ist gleich lang.
- Parameter *n*
  - Jeder Block enthält bis zu *n* Suchschlüssel
  - Jeder Block enthält bis zu *n*+1 Pointer
  - Also wie Indexblock zuvor, aber ein zusätzlicher Pointer
- Wahl von *n*
  - *n* so groß wie möglich entsprechend der Blockgröße
  - 4096 Byte pro Block; 4 Byte pro Schlüssel; 8 Byte pro Pointer
  - $4n + 8(n+1) \leq 4096 \Rightarrow n = 340$

Regeln im B-Baum

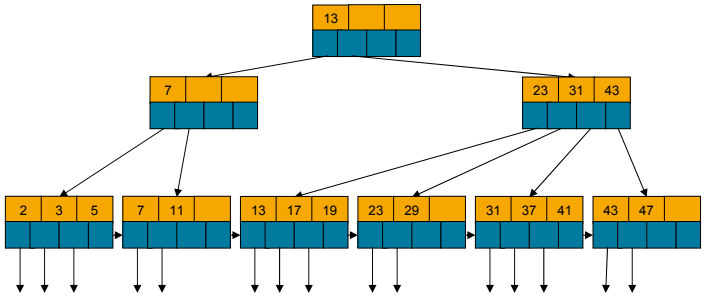
36

- Schlüssel in Blätter sind Schlüssel aus den Daten
  - Sortiert über alle Blätter verteilt (von links nach rechts)
- **Wurzel:** mindestens zwei verwendete Pointer.
  - Alle Pointer zeigen auf B-Baum Block in einer Ebene darunter
- **Blätter:** Der letzte Pointer zeigt auf das nächste Blatt (rechts)
  - Von den übrigen *n* Pointern werden mindestens $\lfloor (n+1)/2 \rfloor$ verwendet.
  - Zeigen auf Datenblöcke
- **Innere Knoten:** Pointer zeigen auf B-Baum Blöcke darunterliegender Ebenen
  - Mindestens $\lceil (n+1)/2 \rceil$ sind verwendet
  - Falls j Pointer verwendet werden, gibt es j-1 Schlüssel in dem Block
    - $K_1, \ldots, K_{j-1}$
  - Erster Pointer zeigt auf Teilbaum mit Schlüsselwerten < K_1.
  - Zweiter Pointer auf Teilbaum mit Schlüsselwerten zwischen K_1 und K_2.

Rechenbeispiele

37

- $n = 3$
  - Alle Knoten: Maximal 3 Suchschlüssel und 4 Pointer
  - Wurzel: Mindestens 1 Suchschlüssel und 2 Pointer
  - Innere Knoten: Mindestens 1 Suchschlüssel und 2 Pointer
  - Blätter: Mindestens 2 Suchschlüssel und 3 Pointer



Rechenbeispiele

38

- $n = 4$
  - Alle Knoten: Maximal 4 Suchschlüssel und 5 Pointer
  - Wurzel: Mindestens 1 Suchschlüssel und 2 Pointer
  - Innere Knoten: Mindestens $\lceil (n+1)/2 \rceil$ Pointer = 3 Pointer
    - => Mindestens 2 Suchschlüssel
  - Blätter:
    - 1 Pointer zum nächsten Blatt + mindestens $\lfloor (n+1)/2 \rfloor$ weitere Pointer = 3 Pointer
    - => Mindestens 2 Suchschlüssel
- $n = 5$
  - Alle Knoten: Maximal 5 Suchschlüssel und 6 Pointer
  - Wurzel: Mindestens 1 Suchschlüssel und 2 Pointer
  - Innere Knoten: Mindestens $\lceil (n+1)/2 \rceil$ Pointer = 3 Pointer
    - => Mindestens 2 Suchschlüssel
  - Blätter:
    - 1 Pointer zum nächsten Blatt + mindestens $\lfloor (n+1)/2 \rfloor$ weitere Pointer = 4 Pointer
    - => Mindestens 3 Suchschlüssel

Alternative Definition

39

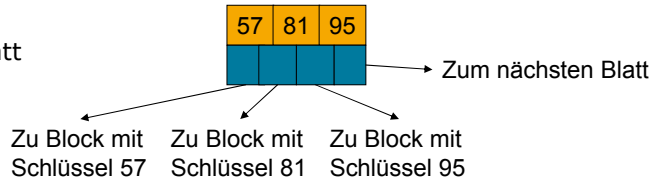
- Bisher: Parameter n
  - Block hat mindestens $\lfloor (n+1)/2 \rfloor$ Suchschlüssel
  - Block hat höchstens n Suchschlüssel
- Alternativ in den Fachbüchern: Parameter k
  - Block hat mindestens k Suchschlüssel
  - Block hat höchstens 2k Suchschlüssel
  - Block hat n + 1 Pointer (wie bisher)
- Immer
  - Ein innerer Block hat immer einen Pointer mehr als Anzahl Suchschlüssel
  - Ein Blatt hat immer ebenso viele Pointer wie Suchschlüssel
    - Plus verkettete Liste

zu Hashtabellen

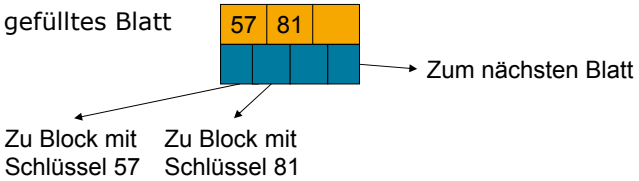
Beispiel Blattknoten

40

- $n = 3$
  - 3 Schlüssel und 4 Pointer
- Volles Blatt



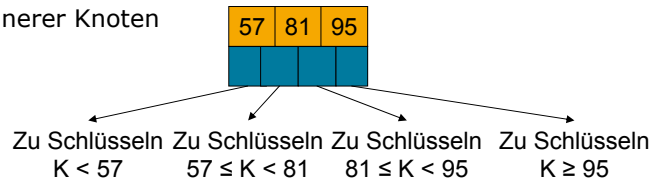
- Teilweise gefülltes Blatt



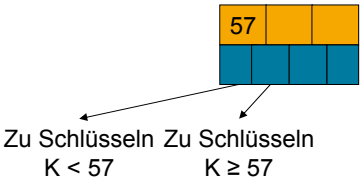
Beispiel innerer Knoten

41

- Voller innerer Knoten



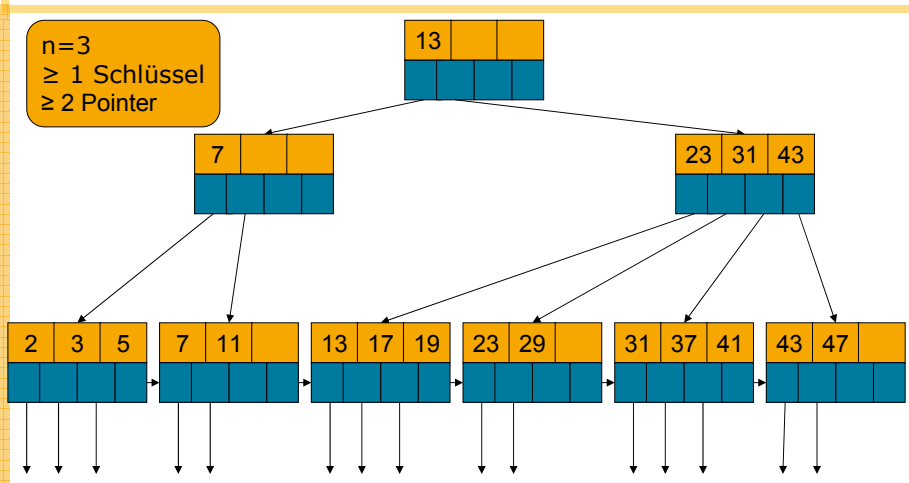
- Teilweise gefüllter innerer Knoten



Beispiel B-Baum

42

n=3
≥ 1 Schlüssel
≥ 2 Pointer



In den Blättern taucht sortiert jeder Schlüssel genau einmal auf.

Anwendungen von B-Bäumen

B-Bäume können verschiedene Index-Rollen übernehmen.

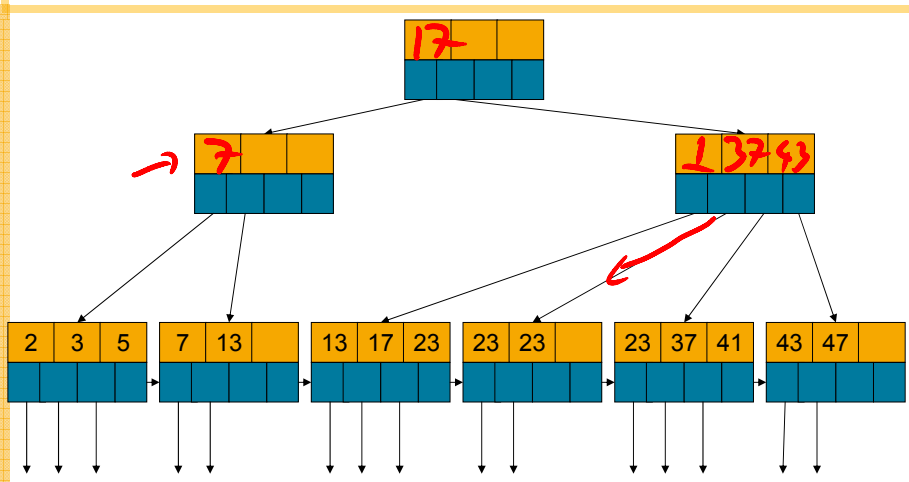
- Suchschlüssel ist Primärschlüssel; dicht-besetzter Index
  - Data file sortiert oder nicht
- Dünn-besetzter Index; data file ist sortiert
- Suchschlüssel ist nicht Primärschlüssel; Dicht-besetzter Index
  - Data file ist nach Suchschlüssel sortiert
  - Pointer zeigen auf jeweils ersten Wert

B-Bäume auf nicht-Primärschlüsseln

- Bedeutung der Pointer auf inneren Ebenen ändert sich.
  - Gegeben: Schlüssel K_1, … , K_j
  - => K_i ist der kleinste neue Schlüsselwert, der vom (i+1)-ten Pointer erreichbar ist.
    - D.h. es gibt keinen Schlüsselwert K_i im linken Teilbaum aber mindestens ein Vorkommen des Schlüsselwertes Teilbaum vom (i+1)-ten Pointer an.
    - Problem: Es gibt nicht immer einen solchen Schlüssel

B-Bäume auf nicht-Primärschlüsseln

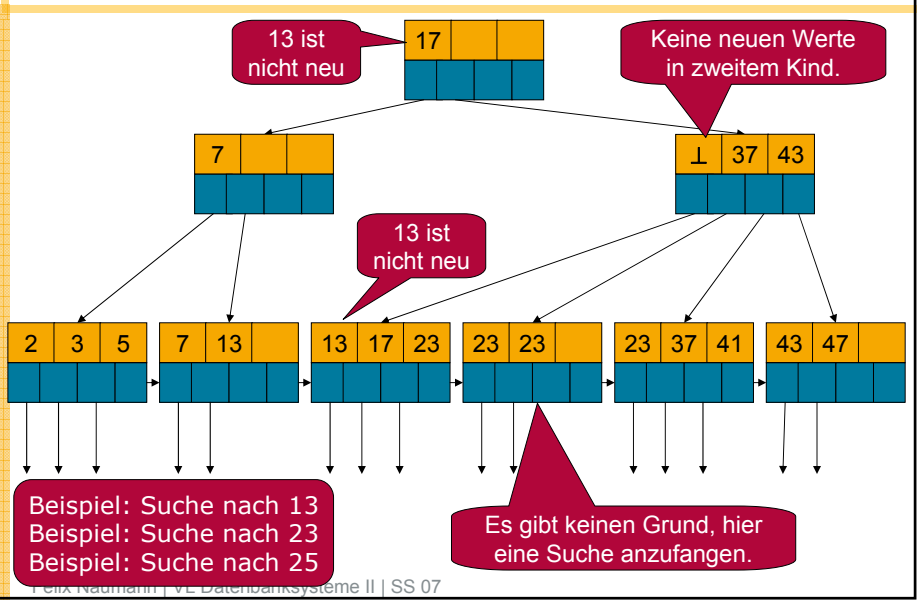
45



K_i ist der kleinste neue Schlüsselwert, der vom (i+1)-ten Pointer erreichbar ist.

B-Bäume auf nicht-Primärschlüsseln

46



Überblick

47

- Indizes auf sequenziellen Dateien
- Sekundärindizes
- B-Bäume
  - Aufbau
  - Suche
  - Updates
  - Effizienz
  - Varianten
- Hash-Tabellen

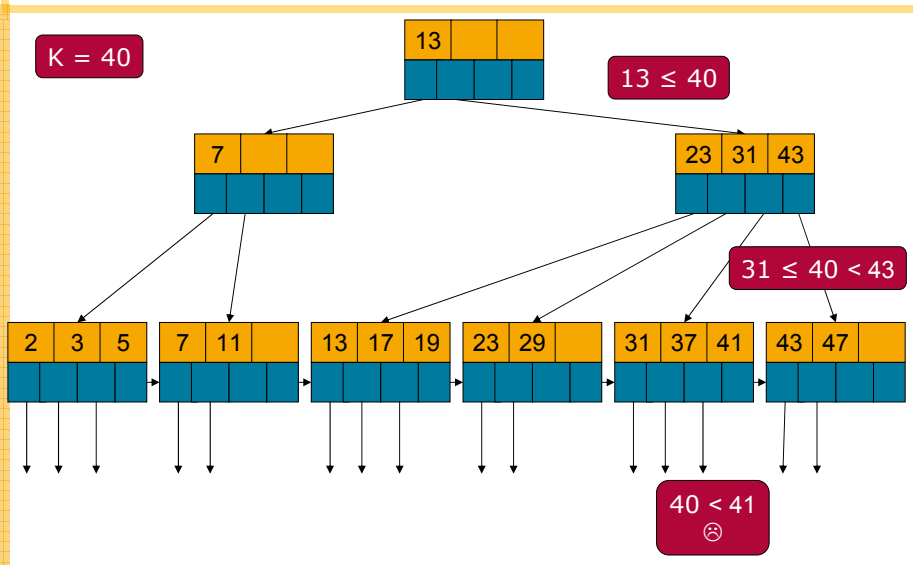
Suche in B-Bäumen

48

- Jetzt wieder: Suchschlüssel = Primärschlüssel
- Dicht-besetzter Index
  - Operationen für dünn-besetzte Indizes ähnlich
- Gesucht sei *K*.
  - Falls wir an einem Blattknoten sind:
    - Suche K auf dem Knoten.
  - Falls wir an einem inneren Knoten mit K_1, K_2, … K_n sind:
    - Falls $K < K_1$ gehe zu erstem Kind
    - Falls $K_1 \leq K < K_2$ gehe zu zweitem Kind
    - …
    - Falls $K_n \leq K$ gehe zu letztem Kind

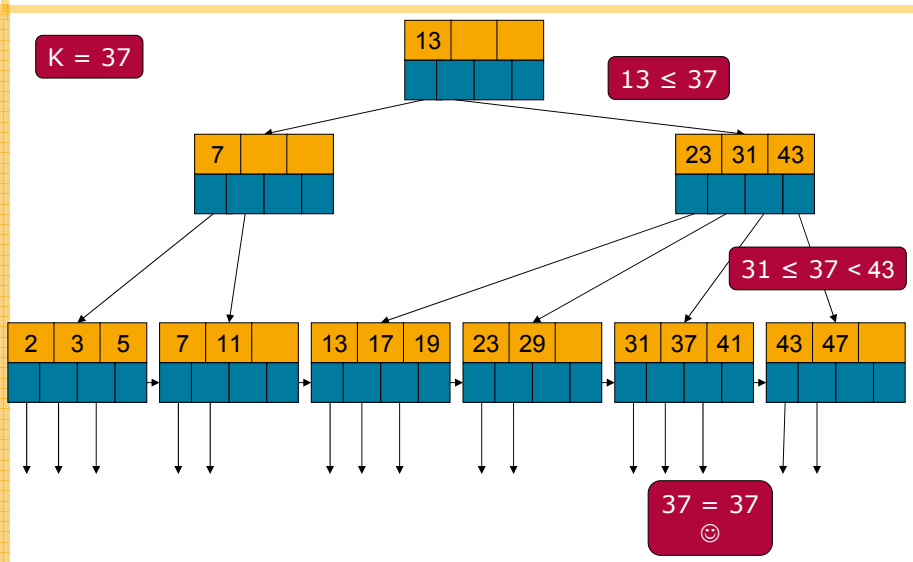
Beispiel Suche im B-Baum

49



Beispiel Suche im B-Baum

50



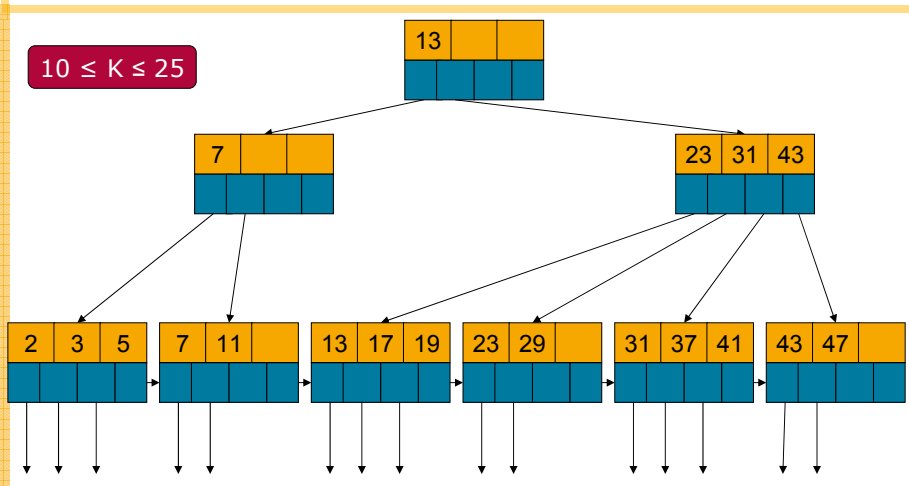
Bereichsanfragen (range queries)

51

- Anfragen mit Ungleichheit in WHERE Klausel
  - SELECT * FROM R
    WHERE R.k > 40
  - SELECT * FROM R
    WHERE R.k >= 10 AND R.k <= 25
- Suche nach Bereich [a,b]
- Suche in B-Baum nach a.
  - Entsprechendes Blatt könnte a speichern.
  - Suche auf dem Blatt alle relevanten Schlüssel.
- Falls auf dem Blatt kein Wert > b.
  - Folge Pointer zu nächstem Knoten.
- Bei offenen Bereichen [-∞ , b] bzw. [a, ∞] ist Suche ähnlich.

Bereichsanfragen (range queries)

52



Überblick

- Indizes auf sequenziellen Dateien
- Sekundärindizes
- B-Bäume
  - Aufbau
  - Suche
  - Updates
  - Effizienz
  - Varianten
- Hash-Tabellen

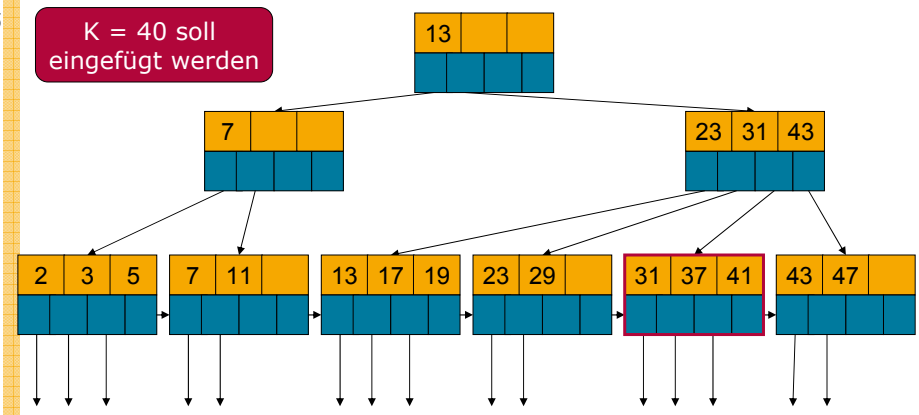
Einfügen in B-Bäume

Rekursives Vorgehen:

- Suche entsprechendes Blatt.
  - Falls Platz herrscht, füge Schlüssel und Pointer ein.
- Falls kein Platz: Überlauf
  - Teile Blatt in zwei Teile und verteile Schlüssel gleichmäßig
  - *„split"*
- Teilung macht Einfügen eines neuen Schlüssel/Pointer-Paares im Elternknoten erforderlich
  - Gehe rekursiv vor
- Ausnahme: Falls in Wurzel kein Platz
  - Teile Wurzel in zwei
  - Erzeuge neue Wurzel (mit nur einem Schlüssel)

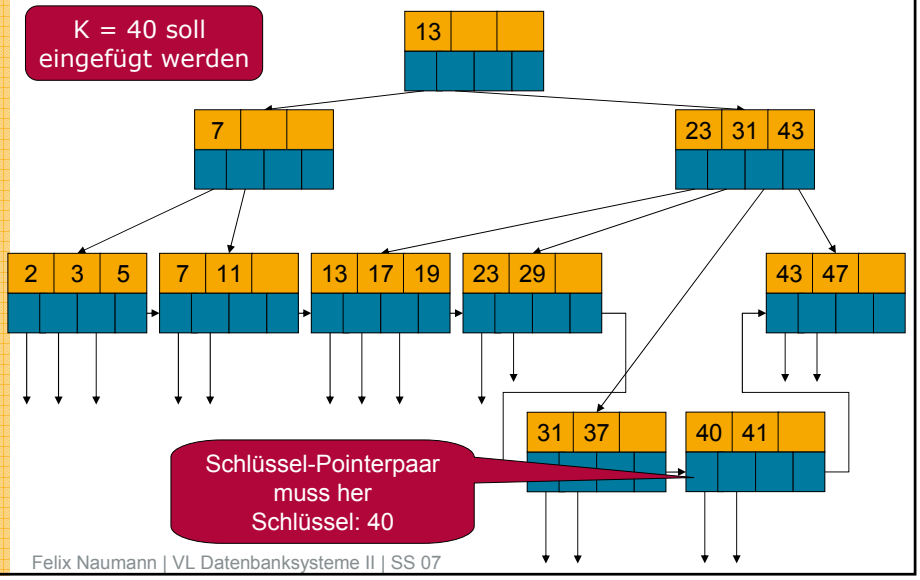
Einfügen in B-Bäume – Beispiel

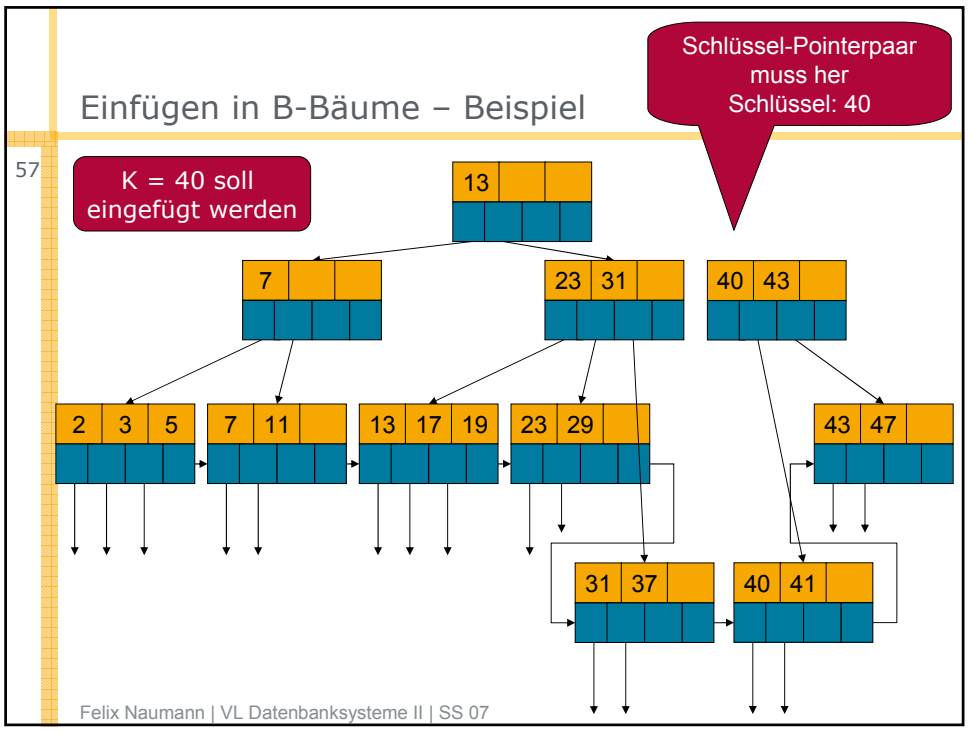
55

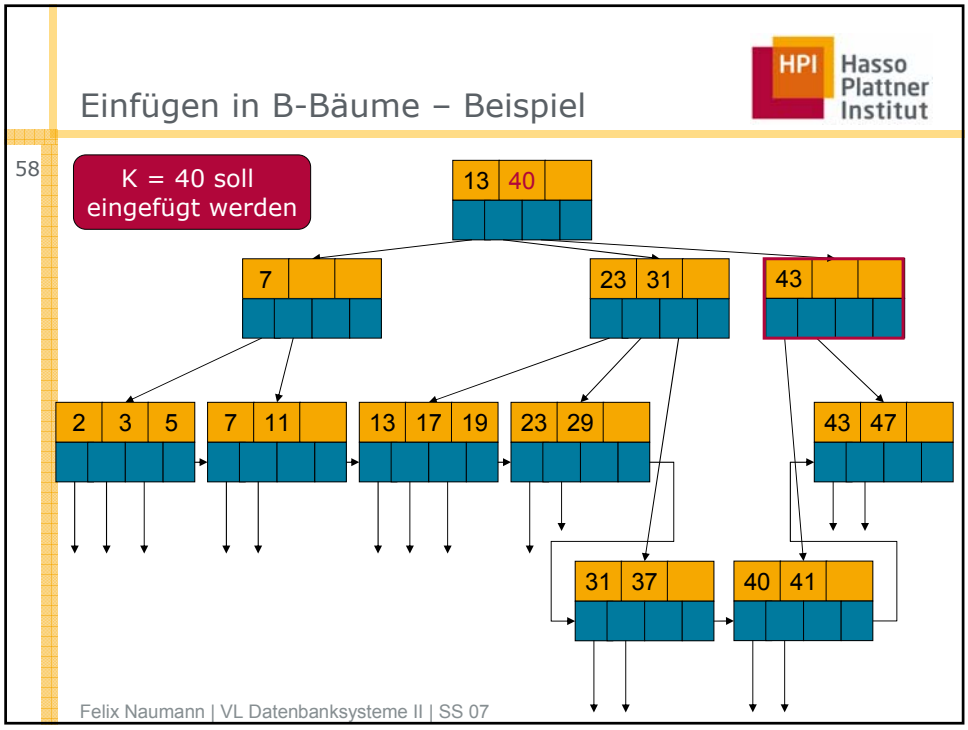


Einfügen in B-Bäume – Beispiel

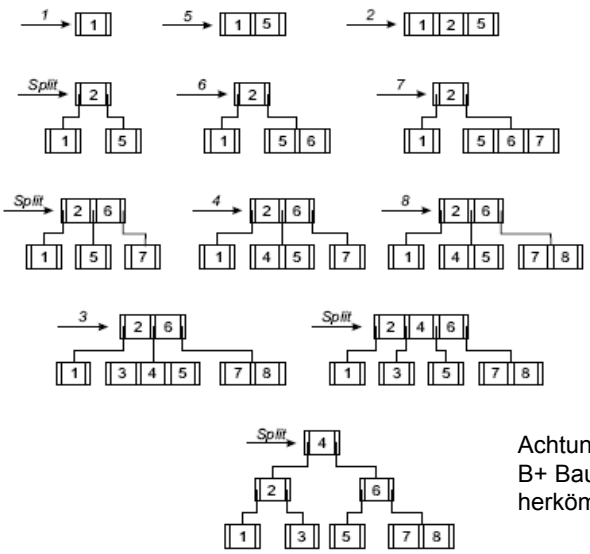
56




Einfügen in B-Bäume – Beispiel
57
K = 40 soll eingefügt werden
Schlüssel-Pointerpaar muss her
Schlüssel: 40
13
7
23 31
40 43
2 3 5
7 11
13 17 19
23 29
43 47
31 37
40 41
Felix Naumann | VL Datenbanksysteme II | SS 07


Einfügen in B-Bäume – Beispiel
HPI Hasso Plattner Institut
58
K = 40 soll eingefügt werden
13 40
7
23 31
43
2 3 5
7 11
13 17 19
23 29
43 47
31 37
40 41
Felix Naumann | VL Datenbanksysteme II | SS 07

Einfügen in B-Bäume – Beispiel



Achtung: Dies ist kein B+ Baum, sondern ein herkömmlicher B-Baum.

Kosten für Einfügen

- Sei h die Höhe des B-Baums
  - Meist $h = 3$
- Suche nach Blattknoten: h
- Falls keine Teilung nötig: Gesamtkosten $h + 1$
- Falls Teilung nötig
  - Worst-case: Bis zur Wurzel
  - Selbst Caching nützt nichts, da Knoten geschrieben werden müssen
  - Insgesamt: $3\ h + 1$
    - Auf jeder Ebene Suche und Überlaufbehandlung

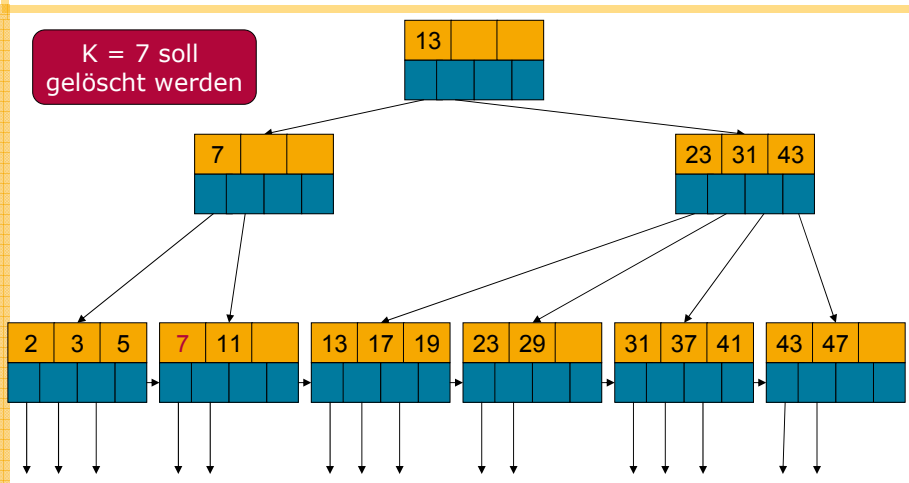
Löschen aus B-Bäumen

61

- Suche entsprechenden Knoten
- Lösche Schlüssel
- Falls immer noch minimale Menge an Schlüsseln im Knoten
  - Nichts tun
- Falls zu wenig Schlüssel im Knoten: *Merge* (Verschmelzen)
  - Falls ein Geschwisterknoten (links oder rechts) mehr als die minimale Knotenmenge hat, verschiebe einen Schlüssel.
    - Gegebenenfalls Schlüsselwerte der Eltern anpassen
  - Falls nicht: Es gibt zwei Geschwisterknoten mit minimaler und sub-minimaler Schlüsselmenge
    - => Diese Knoten können vereinigt werden.
    - Schlüsselwerte der Eltern anpassen: U.a. entsprechenden Schlüssel; gegebenenfalls rekursiv im Baum nach oben löschen

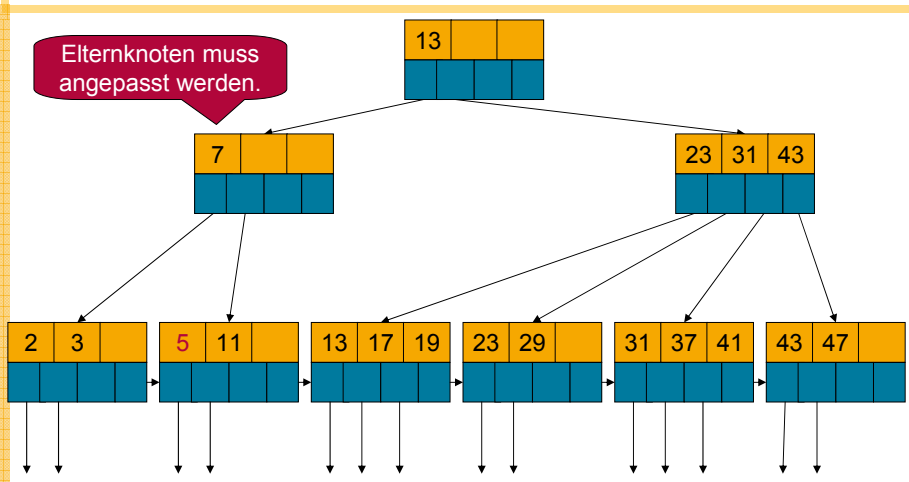
Löschen aus B-Bäumen – Beispiel

62



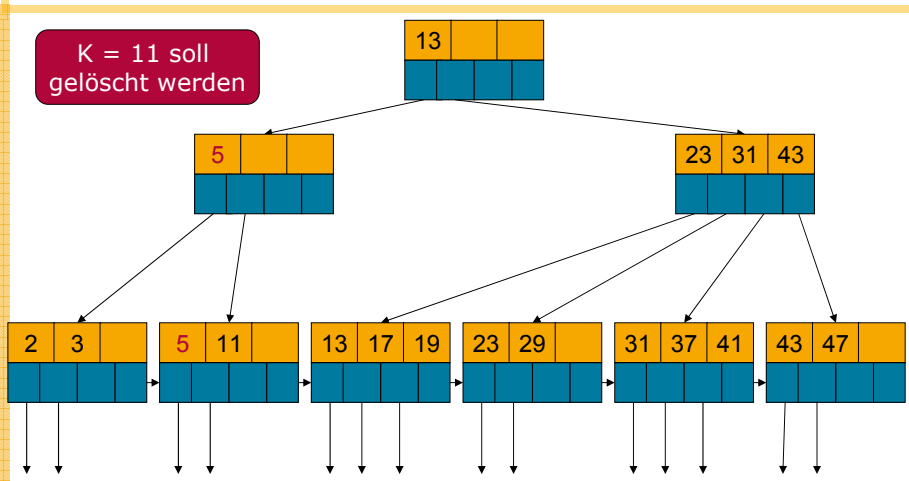
Löschen aus B-Bäumen – Beispiel

63



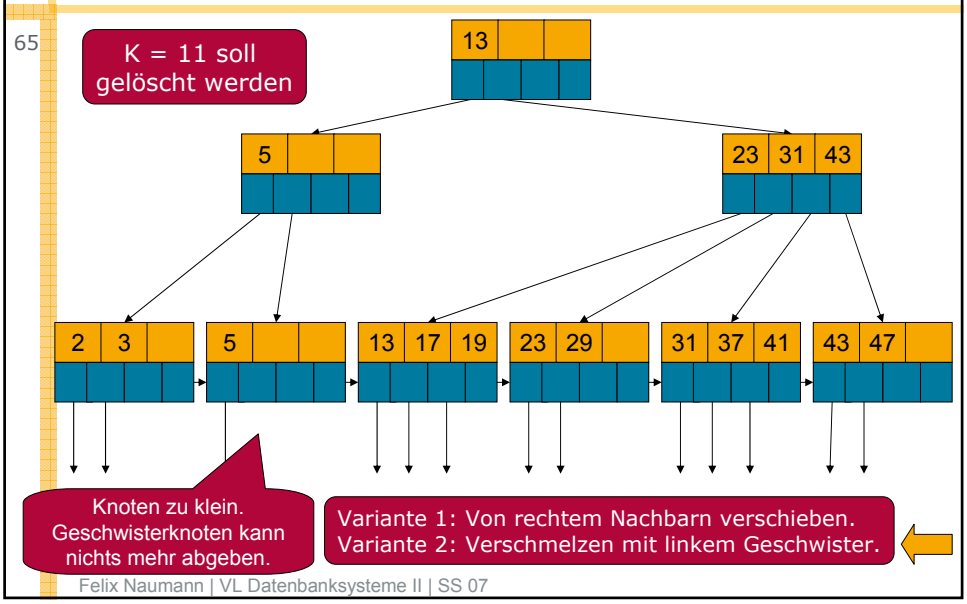
Löschen aus B-Bäumen – Beispiel

64



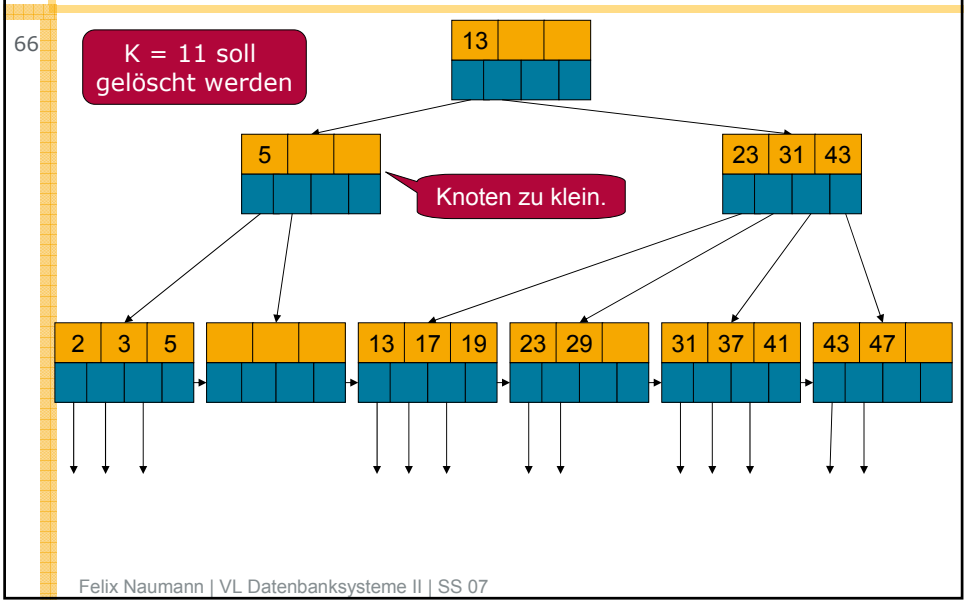
Löschen aus B-Bäumen – Beispiel

65



Löschen aus B-Bäumen – Beispiel

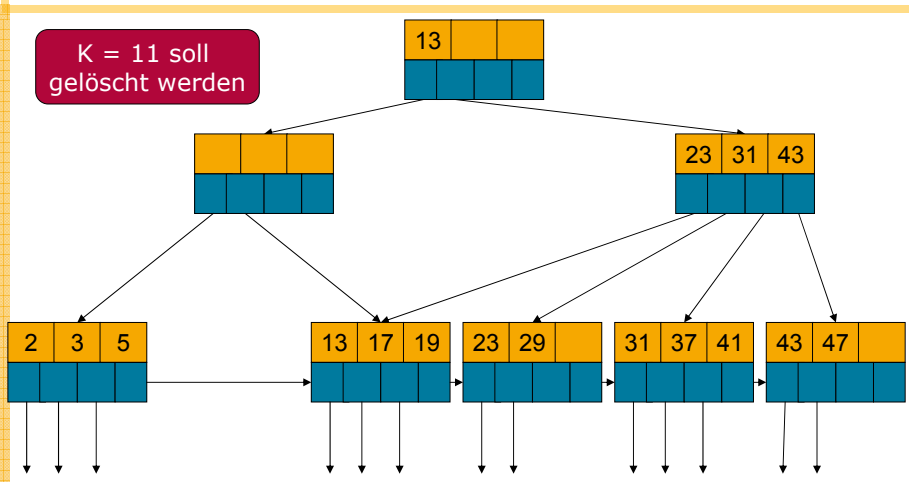
66



Löschen aus B-Bäumen – Beispiel

67

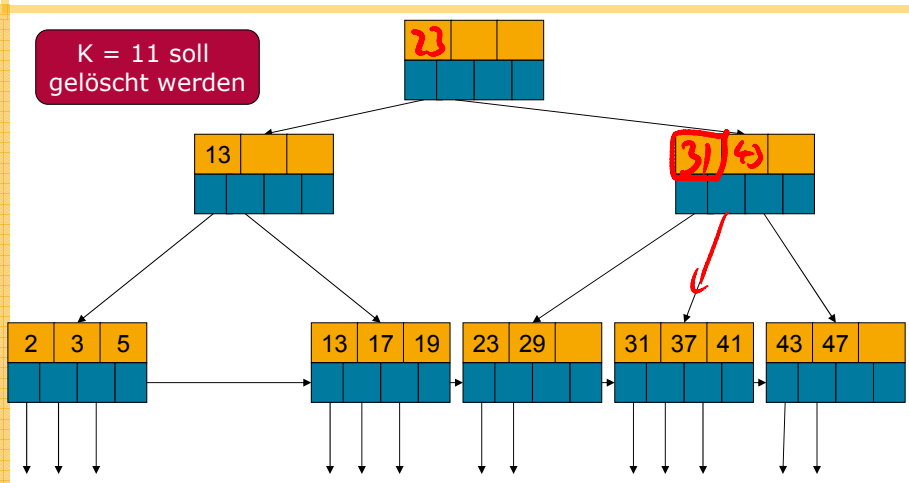
K = 11 soll gelöscht werden



Löschen aus B-Bäumen – Beispiel

68

K = 11 soll gelöscht werden



Kosten für Löschen

- Suche und lokales Löschen: $h + 2$
  - Schreibe Blattknoten
  - Schreibe Datenblock
- Bei merge mit Geschwisterknoten: $h + 6$
  - Prüfe rechts und links
  - Schreibe Block und veränderten Nachbarn
  - Schreibe Elternknoten
  - Schreibe Datenblock
- Bei merge bis zur Wurzel: $3h - 2$

Löschen aus B-Bäumen?

- Annahme: Tendenziell wachsen Datenmengen
- Folgerung: Nie Knoten des B-Baum löschen
  - Knoten, die durch Löschen zu klein werden, werden früher oder später wieder gefüllt.
  - Grabstein muss sowieso erhalten bleiben.
- Verbesserung: Grabstein im B-Baum statt im Block.

Überblick

71

- Indizes auf sequenziellen Dateien
- Sekundärindizes
- B-Bäume
  - Aufbau
  - Suche
  - Updates
  - Effizienz
  - Varianten
- Hash-Tabellen

Effizienz von B-Bäumen

72

Suche, Einfügen und Löschen sollen möglichst wenig I/O-Operationen benötigen.

- Je größer *n* gewählt wird desto seltener müssen Blöcke verschmolzen oder getrennt werden.
  - Meist auf Blattknoten beschränkt
- Suche
  - Anzahl I/Os entspricht der Höhe des Baums
  - + 1 I/O auf den Daten für Lesen
  - + 2 I/O auf den Daten für Einfügen oder Löschen
- Typische Höhe eines B-Baums: 3
  - 340 Schlüssel-Pointer-Paare pro Block
  - Annahme: Füllstand durchschnittlich 255
  - => 255 innere Knoten = > 255^2 = 65025 Blätter = 255^3 Pointer
  - Insgesamt als 16.6 Mio Datensätze
- Zugriff mit 2 I/Os: Nur Wurzel im Hauptspeicher
- Zugriff mit 1 I/O: Wurzel und innere Knoten im Hauptspeicher

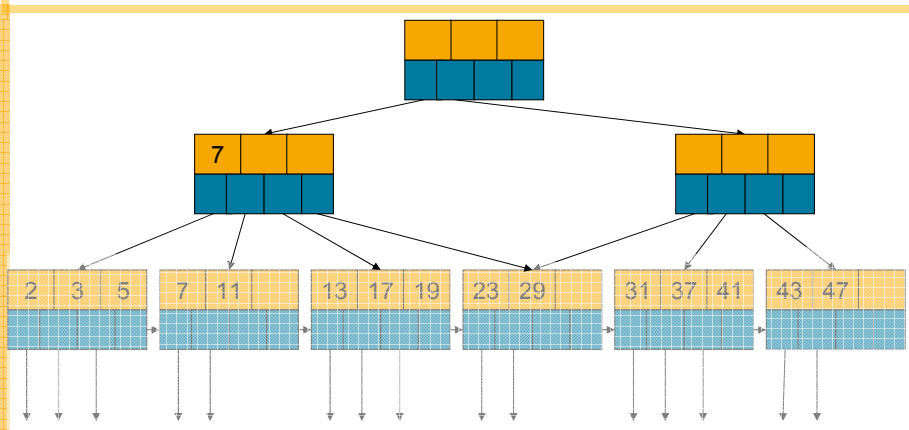
Bulk-loading

73

- Einfügevarianten
  - Einfügen von Daten, die bereits mit existierendem B-Baum indiziert sind.
  - Erstellung eines neuen B-Baums auf existierenden Daten
    - Sukzessives Einfügen jedes Datensatzes ist ineffizient.
    - Immer wieder über die Wurzel suchen
- Besser: Vorsortierung
  - Schritt 1: Erzeuge Schlüssel-Pointer Paare für alle Blöcke
  - Schritt 2: Sortierung der Paare nach Suchschlüssel
  - Schritt 3: Füge nun sukzessive die Paare ein

Bulk-loading

74



Wieso ist dies effizient?

- Nur Zugriff auf Indexblöcke im Cache
- Kosten: Einmal alle Knoten ausschreiben

Geht es noch besser? Besserer Füllstand?

Überblick

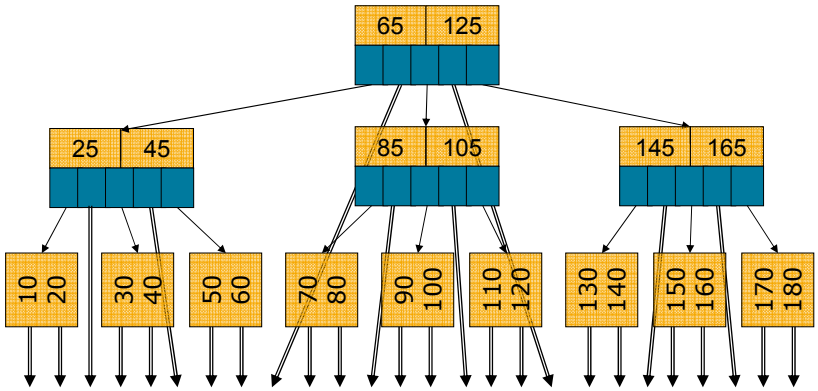
75

- Indizes auf sequenziellen Dateien
- Sekundärindizes
- B-Bäume
  - Aufbau
  - Suche
  - Updates
  - Effizienz
  - Varianten
- Hash-Tabellen

B-Baum Varianten: B-Baum (ohne +)

76

- Bisher: B⁺-Baum
  - Pointer auf Datensätze nur in Blattknoten
- B-Baum
  - Pointer auf Datensätze in allen Knoten



B-Baum Varianten: B-Baum (ohne +)

- Vorteil: Durchschnittlich schnellere Suche als in B^+-Bäumen
- Nachteil: Blätter und innere Knoten haben unterschiedliche Größe
  - Verwaltung schwieriger
- Nachteil: Weniger buschig; Platz wird für Pointer auf Datensätze „verschwendet"
- Nachteil: Löschen ist komplizierter
  - Löschung kann auch in inneren Knoten geschehen.
  - Knoten von einem Blatt muss nach obern wandern.

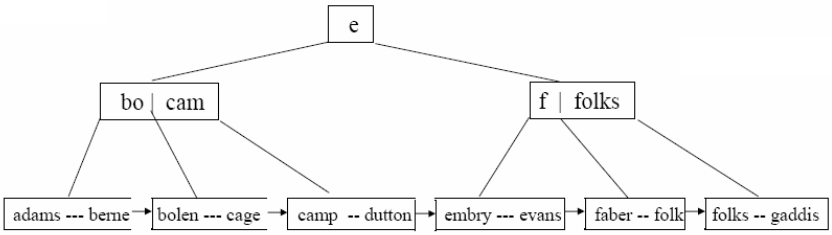
B-Baum Varianten: B^*-Baum

- B^*-und $B^\#$-Baum
  - Bei Überlauf werden nicht gleich zwei halb-leere Knoten erzeugt.
  - Stattdessen Neuverteilung über alle Blätter
  - Falls nicht möglich, erzeuge 3 neue Blätter aus 2 alten Blättern
  - Dadurch bessere Speicherausnutzung: Mindestens 66%

B-Baum Varianten: Präfix-B⁺-Baum

79

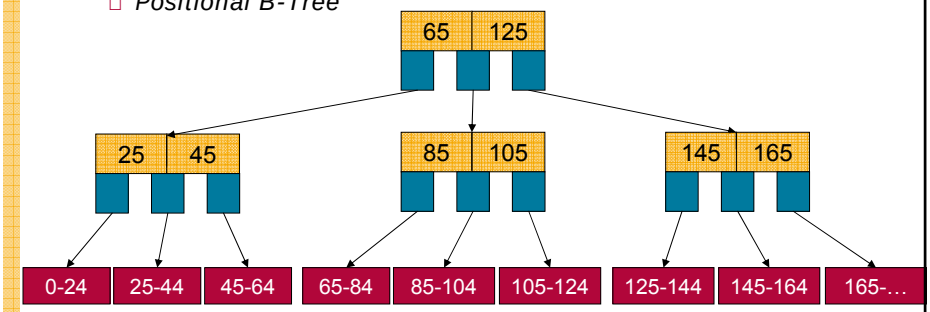
- Präfix-B⁺-Baum
  - Falls Suchschlüssel ein String ist => Hoher Speicherbedarf
  - Besser: Speichere nur „Trennwert"
  - Was trennt „Korn" von „Loeser"?
  - Am besten: Kleinster Trennwert
    - „L"
    - Präfix von „Loeser"



B⁺-Bäume für BLOBs

80

- Idee: Suchschlüssel repräsentiert Offsets im BLOB anstatt Suchschlüssel
  - Blätter zeigen auf Datenseiten des BLOBs
  - *Positional B-Tree*



Überblick

- Indizes auf sequenziellen Dateien
- Sekundärindizes
- B-Bäume
- Hash-Tabellen
  - Allgemeine Hashtabellen
  - Erweiterbare Hashtabellen
  - Lineare Hashtabellen

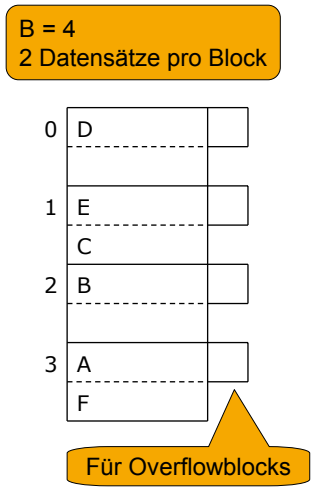
Nachtrag zur Kardinalität von B-Baum Knoten

Hashtabellen – Grundprinzip

- Hashfunktion
  - Input: Suchschlüssel K (Hash-Schlüssel)
  - Output: Integer zwischen 0 und B-1
    - B = Anzahl Buckets
  - Oft z.B. MOD(K/B)
    - Bei Strings: Weise jedem Buchstaben Integer zu und summiere diese.
- Bucketarray
  - Array aus B Headern für B verkettete Listen

Hashtabellen auf Festplatten

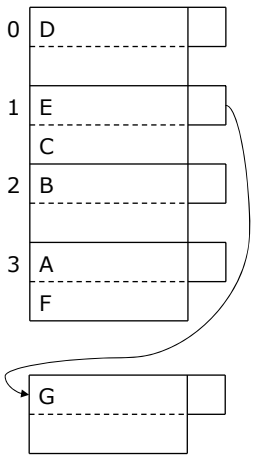
- Bisher (im Studium): Hashtabellen im Hauptspeicher
- Blöcke statt Pointer auf Listen
  - Datensätze, die auf einen bestimmten Wert gehashed werden, werden in dem entsprechenden Block gespeichert.
  - Overflowblocks können ergänzt werden.
- Annahme: Zuordnung eines Hashwerts zur Speicheradresse eines Block möglich
  - Z.B.: Offset



Einfügen in Hashtabelle

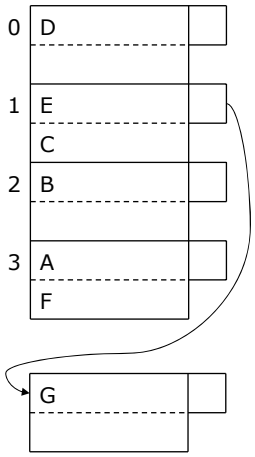
1. Berechne Hashwert
2. Falls Platz, füge Datensatz in entsprechenden Block ein
   - Oder in einen Overflowblock mit Platz
3. Falls nicht, erzeuge neuen Overflowblock und füge dort ein

- Beispiel: Füge G ein
  - h(G) = 1



Löschen aus Hashtabellen

1. Suche Bucket (+ Overflowblocks)
2. Suche Datensatz / Datensätze
3. Lösche sie
4. Gegebenenfalls Reorganisation und Entfernung von Overflowblocks
   - Gefahr der Oszillation
- Beispiel: Lösche C
  - G bewegen



Überblick

- Indizes auf sequenziellen Dateien
- Sekundärindizes
- B-Bäume
- Hash-Tabellen
  - Allgemeine Hashtabellen
  - Erweiterbare Hashtabellen
  - Lineare Hashtabellen

Effizienz statischer Hashtabellen

- Idealerweise: Pro Bucket nur ein Block
  - Zugriffszeit lesen: 1 I/O
  - Zugriffszeit Einfügen / Löschen: 2 I/O
  - Viel besser als Dicht- oder Dünnbesetzte Indizes
  - Besser als B-Bäume
    - Aber Nachteil: Bereichsanfragen
- Weiteres Problem: Lange Listen von Overflowblöcken
  - Statische Hashtabellen
- Lösung: Dynamische Hashtabellen
  - Hashtabellen wachsen
  - B ≈ Anzahl Datensätze / Datensätze pro Block
    - => Ca. 1 Block pro Bucket

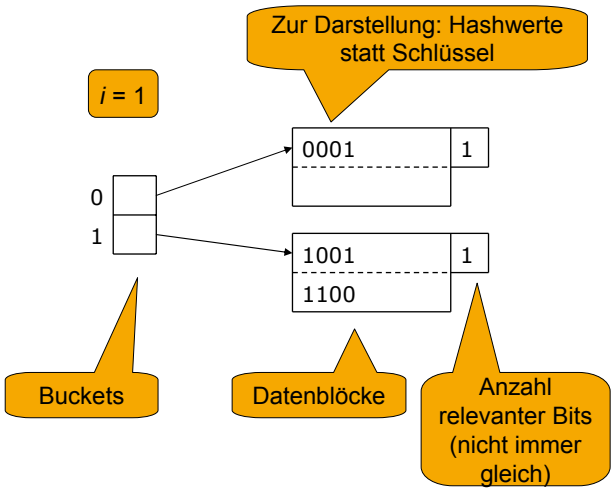
Erweiterbare Hashtabellen

Neuerungen

- Indirektion
  - Bucket besteht aus Pointerarray statt Datenblock(s)
- Wachstum
  - Größe des Pointerarrays verdoppelt sich bei Bedarf
- Sharing
  - Buckets können sich Datenblöcke teilen
  - Wenn es passt
- Hashfunktion
  - Berechnet (zu) großes Bitarray (k bit; z.B. $k = 32$)
  - Bucketarray verwendet nur die ersten i Bits ($i \leq k$)
  - => 2^i buckets

Erweiterbare Hashtabellen

- $k = 4$



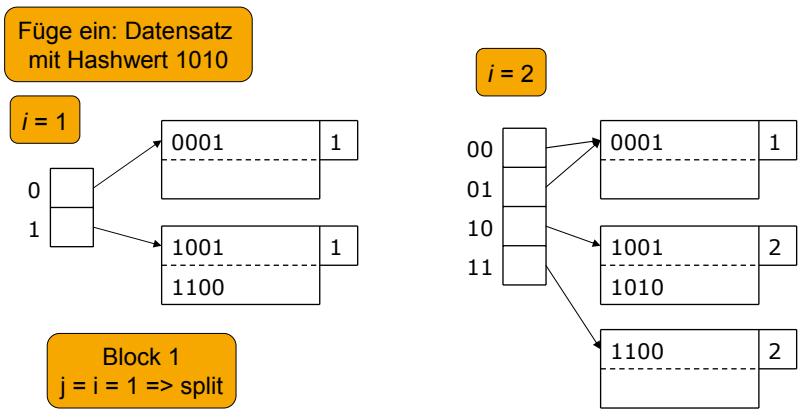
Einfügen in erweiterbare Hashtabellen

- Ähnlich wie normale Hashtabelle
  - Berechne h(K); wähle die ersten i Bits
  - Suche Eintrag im Bucketarray und lade entsprechenden Block
- Falls Platz: Füge neuen Datensatz ein.
- Falls kein Platz und j < i (j ist aktuelle Bitanzahl des Blocks)
  - Spalte Block entzwei (*split*)
  - Verteile Datensätze gemäß des (j+1)ten Bits
    - Falls 0: Verbleib
    - Falls 1: Verschieben in neuen Block
  - Setze j+1 als neue Bitanzahl
  - Pointer im Bucketarray aktualisieren
  - Was wäre Pech?
    - Alle Datensätze landen wieder im gleichen Block. Dann wieder j++.

Einfügen in erweiterbare Hashtabellen

- Falls kein Platz und j = i (j ist aktuelle Bitanzahl des Buckets)
  - i++ => Länge des Bucketarrays verdoppelt sich
  - Datenblöcke bleiben unverändert
  - Zwei neue Pointer zeigen auf gleichen alten Block
  - Dann: Spalte relevanten Block entzwei (*split*)
    - Weiter wie zuvor (denn nun j < i)

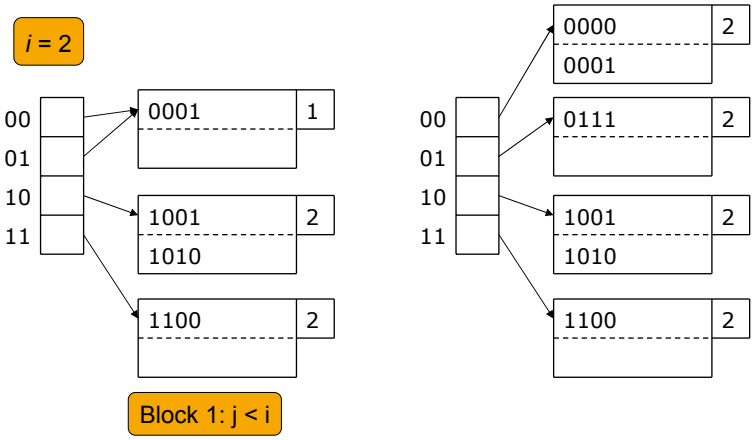
Einfügen in erweiterbare Hashtabellen



Einfügen in erweiterbare Hashtabellen

94

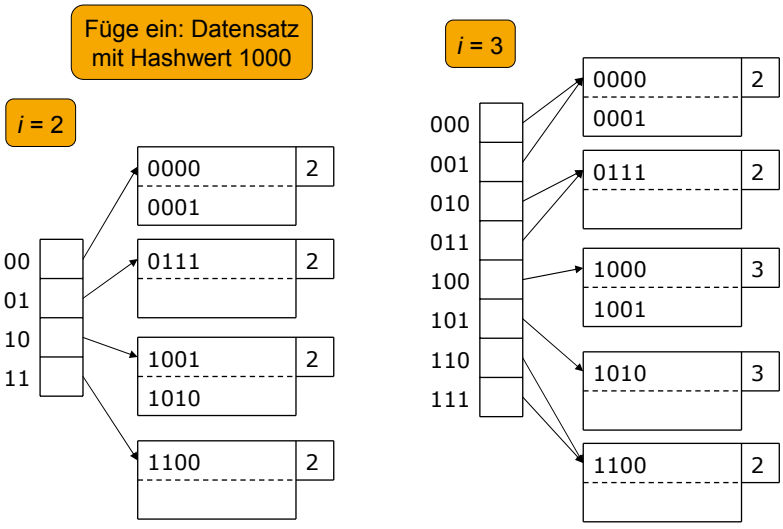
Füge ein: Datensätze mit Hashwerten 0000 und 0111



Einfügen in erweiterbare Hashtabellen

95

Füge ein: Datensatz mit Hashwert 1000



Analyse erweiterbarer Hashtabellen

- Vorteile
  - Bei Suche: Nie mehr als ein Datenblock betrachten
    - Keine Overflow Blocks
  - Bucketarray eventl. im Hauptspeicher
- Nachteile
  - Bei Verdopplung: Viel Arbeit
    - => Ab und zu dauert ein Einfügen sehr lang
  - Bucketarray wächst schnell
    - Passt eventuell nicht mehr in Hauptspeicher
  - Platzverschwendung bei wenigen Datensätzen pro Block
    - Beispiel: 2 Datensätze pro Block
    - Datensatz 1: 00000000000000000001
    - Datensatz 2: 00000000000000000010
    - Datensatz 3: 00000000000000000011
    - => i = 20 (also 2^{20} = 1Mio buckets)

Überblick

- Indizes auf sequenziellen Dateien
- Sekundärindizes
- B-Bäume
- Hash-Tabellen
  - Allgemeine Hashtabellen
  - Erweiterbare Hashtabellen
  - Lineare Hashtabellen

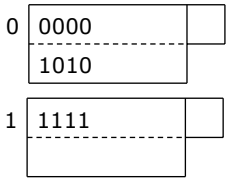
Lineare Hashtabellen

Idee: Anzahl Buckets wächst nur langsam.

- Anzahl *n* der Buckets so gewählt, dass Datenblocks zu ca. 80% gefüllt sind.
- Overflow Blocks sind zugelassen
  - Durchschnittliche Anzahl Overflow Blocks pro Block: <<1
- $\log_2 n$ Bits zur Identifizierung der Buckets
  - Wähle die jeweils letzten Bits des Hashwerts

i = 1 (Anzahl relevanter Bits)
n = 2 (Anzahl Buckets)
r = 3 (Anzahl Datensätze)

Wähle *n*, so dass $r \leq 1{,}7 \cdot n$

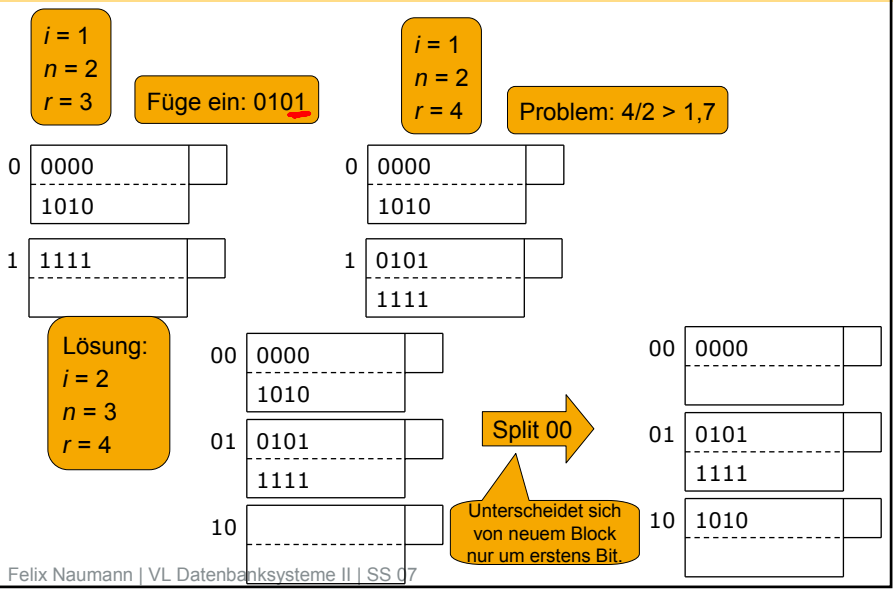


Lineare Hashtabellen – Einfügen

- Berechne h(K)
- Betrachte letzte *i* Bits; interpretiere als Integer *m*
- Falls m < n: Füge Datensatz in Bucket *m* ein.
- Falls m ≥ n:
  - => Bucket *m* existiert noch nicht
- Füge Datensatz in Bucket $m - 2^{i-1}$ ein.
  - D.h. erstes Bit des Schlüssels wird zu 0 (vorher 1).
- Falls kein Platz: Erzeuge Overflow Block
- Berechne r/n
  - Falls zu hoch (z.B. ≥ 1,7): Erzeuge neuen Bucket
  - Neue Bucket hat nichts mit betroffenem Bucket zu tun.
- Falls nun $n > 2^i$: *i*++
  - D.h. alle Bitsequenzen erhalten eine 0 am Anfang
  - Physisch ändert sich nichts

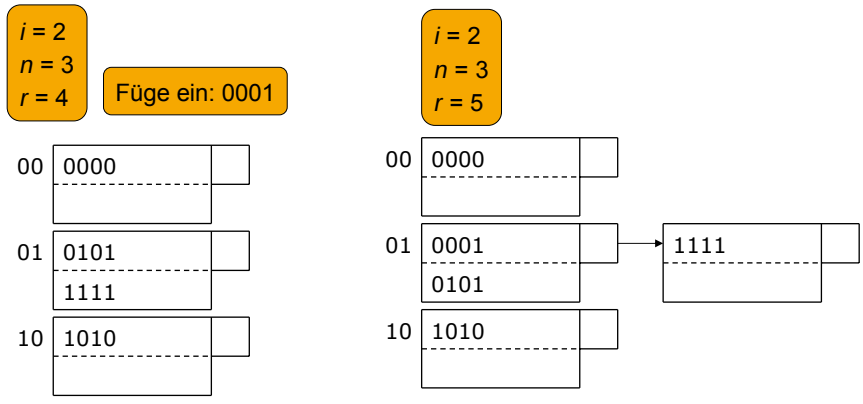
Lineare Hashtabellen – Einfügen

100

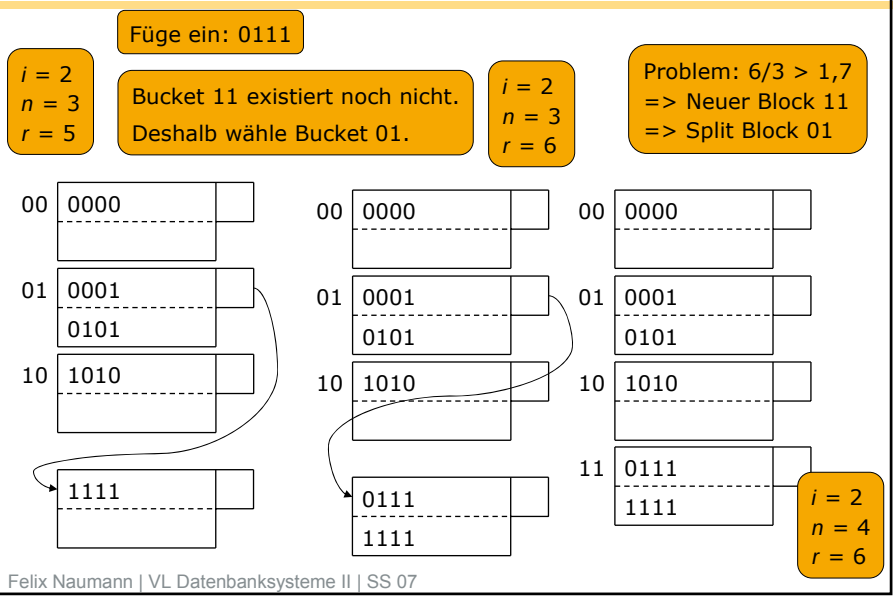


Lineare Hashtabellen – Einfügen

101



Lineare Hashtabellen – Einfügen



Hashing vs. B-Baum

- Hashing effizient zur Feststellung der Existenz eines Wertes
- B-Baum effizient für Bereichsanfragen

- Indexerzeugung in SQL:
  - **CREATE [UNIQUE] INDEX index**
    ON table (column [ASC | DESC]
    [, column [ASC | DESC] ...]
    - UNIQUE erlaubt NULL Werte
  - Keine Angabe über Art des Schlüssels
  - Keine Angabe über Parameter
  - Manchmal Hersteller-spezifische Syntax für Parameter