

Architektur und Implementierung von Datenbanksystemen WS 05/06

Dr. Jens-Peter Dittrich

jens.dittrich@inf

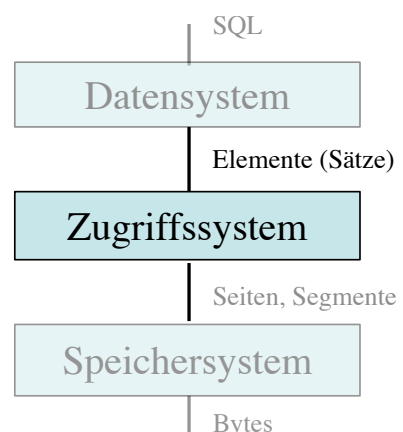
www.inf.ethz.ch/~jensdi

Institut für Informationssysteme



Zugriffssystem

3. Eindimensionale Zugriffspfade (Teil 2)



Motivation

- Anfragen:
 - Wie ist die Adresse von Student mit Matrikelnummer 424342?
 - Welche Studenten besuchen weniger als 2 Vorlesungen dieses Semester?
 - Welche Studenten wohnen nicht im Kanton Zürich?
- Wie beantwortete das DBMS diese Anfragen?
 - Alle Studenten-Datensätze anschauen
(Sequentieller Zugriffspfad, siehe letzte VL-Stunde)
 - Daten geschickt organisieren, so dass die benötigten Sätze schnell gefunden werden
(Baumstrukturierte Zugriffspfade, heute)

Primäre vs. Sekundäre Zugriffspfade

- Primärer Zugriffspfad:
Zugriff über Index, der den Primärschlüssel der Relation als Schlüssel des Index nutzt
- Sekundärer Zugriffspfad:
Zugriff über Index dessen Schlüssel nicht dem Primärschlüssel der Relation entspricht

Beispiel:

Index von...

Key \longrightarrow TID = Primärer Zugriffspfad

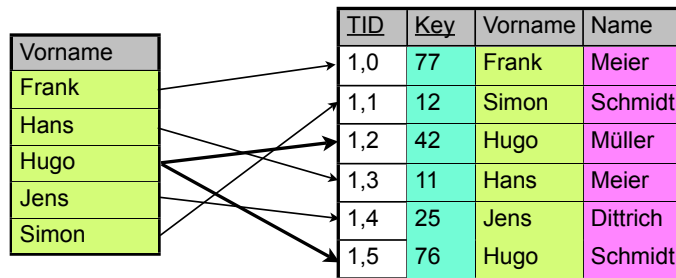
Vorname \longrightarrow TID = Sekundärer Zugriffspfad

Name \longrightarrow TID = Sekundärer Zugriffspfad

Key	Vorname	Name
77	Frank	Meier
12	Simon	Schmidt
42	Hugo	Müller
11	Hans	Meier
25	Jens	Dittrich
76	Hugo	Schmidt

Sekundäre Zugriffspfade und Invertierung

- Suche über Sekundären Zugriffspfad kann mehr als ein Ergebnis liefern (1:n Beziehung zwischen Schlüsseln und Ergebnissen)



Indexierung in Google™ (ohne Ranking)

- Jedes Dokument (html, pdf, doc, etc.) bekommt eine eindeutige Document ID (DocID)
- Der Sekundäre Zugriffspfad indiziert nun die Abbildung

keyword --> {(DocID, {pos})}

{pos} = Liste von Positionen, an denen das keyword im Dokument vorkommt.

- Diese Datenstruktur heisst invertierte Liste bzw. invertiertes File.
- Bsp.: ...

```
jens    -> (42,{3,500,900,1000}),
          (88,{3,300}),
          (4025,{1,20,5000}),
dittrich -> (12,{2,450,600}),
           (78,{1,4300,7000}),
           (2123,{30}),
...

```

Keyword Anfrage in Google™

[Web](#) [Bilder](#) [Groups](#) [Verzeichnis](#) [News](#) [Mehr >](#)

[Erweiterte Suche](#)
[Einstellungen](#)
[Sprachtools](#)

Suche: Das Web Seiten auf Deutsch Seiten aus der Schweiz

- Simplem Nachschlagen in invertiertem File mit keyword "jens"!
- Rückgabe der ersten 10 Elemente der Liste

- Was passiert wenn ich nach mehreren Keywords frage?

Google™ Mehrere Keywords (ohne Ranking)

- Beispiel:

[Web](#) [Bilder](#) [Groups](#) [Verzeichnis](#) [News](#) [Mehr >](#)

[Erweiterte Suche](#)
[Einstellungen](#)
[Sprachtools](#)

Suche: Das Web Seiten auf Deutsch Seiten aus der Schweiz

- Suchalgorithmus
 - 3 Zugriffe auf Sekundärindex mit Schlüssel
<eth>, <jens>, <dittrich>
 - Ergebnis: 3 Listen T_1, T_2, T_3 von DocIDs (Document-IDs)
 - Berechne Schnitt $T = T_1 \cap \{t \mid t \in (T_2 \cap T_3) \wedge \text{pos}(t, T_2) = \text{pos}(t, T_3) - 1\}$
 - Gib ersten 10 Elemente von T als Ergebnis aus.
 - **Achtung: die ersten 10 Elemente müssen nicht die Wichtigsten sein!**
 - Fertig.

Indexierung in Google™ (mit Ranking)

- Problem: keyword-Suche kann Tausende von Ergebnissen liefern
- Für keyword "jens" schätzt Google 15,3 Millionen Dokumente
- Die interessanten Seiten müssen nicht unbedingt unter den ersten 10 Treffern sein.
- Lösung: Versuche Seiten bezüglich Ihrer Relevanz anzuordnen
- Viele Algorithmen, der Wichtigste: Page Rank
- mehr zu Page Rank in der VL "Multimedia Retrieval"

- Aber was für Auswirkungen hat Ranking auf die Indexierung?

Indexierung in Google™ (mit Ranking)

- Grundidee:
 1. nummeriere Dokumente neu mit docID = rank
d.h. Dokument mit docID=1 ist am wichtigsten, docID=2 am zweitwichtigsten, etc.
Achtung: die docIDs müssen weiterhin ein Primärschlüssel sein!
 2. Ordne Ergebnislisten im Invertierten File jetzt nach Rank

- Bsp.: ...

	Rank
jens	-> (7000, {3,500,900,1000}), (8888, {3,300}), (40251, {1,20,5000}),
dittrich	-> (12, {2,450,600}), (78, {1,4300,7000}), (2123, {30}),
....	

Google Mehrere Keywords (mit Ranking)

- Beispiel:

Web Bilder Groups Verzeichnis News Mehr >

jens dittrich* eth [Erweiterte Suche](#)
[Einstellungen](#)
[Sprachtools](#)

Google-Suche Auf gut Glück!

Suche: Das Web Seiten auf Deutsch Seiten aus der Schweiz

- Suchalgorithmus (entspricht Algorithmus ohne Ranking)
 - 3 Zugriffe auf Sekundärindex mit Schlüsseln
<eth>, <jens>, <dittrich>
 - Ergebnis: 3 Listen T_1, T_2, T_3 von DocIDs (Document-IDs)
 - Berechne Schnitt $T = T_1 \cap \{t \mid t \in (T_2 \cap T_3) \wedge \text{pos}(t, T_2) = \text{pos}(t, T_3) - 1\}$
 - Gib ersten 10 DocIDs von T' als Ergebnis aus.
 - **Unterschied: Die ersten 10 DocIDs sind gleichzeitig die 10 wichtigsten Dokumente (bezüglich des Ranking)!**
 - Fertig.

Entwurfsziele für Indexstrukturen

- hohe Speicherplatzausnutzung
- geringe E/A-Kosten
- geringe CPU-Kosten
- kurze Antwortzeiten für eine Operation
- hoher Durchsatz von Operationen
- leichte Integration in bestehende DBMS
- leichte Erweiterbarkeit

Bäume: Klassen

- Binärbäume
 - Nicht geeignet für DBMS
 - Knoten lassen sich nur schwer auf Seiten abbilden
- Digitalbäume
 - spezielle Anwendungen
 - wichtig für nicht-relationale Daten
- B-Bäume
 - wichtigste Datenstruktur im Datenbankbereich
 - Vorteil: extrem vielseitig, lässt sich leicht erweitern
 - seit über 30 Jahren Forschung, immer noch wichtige Erweiterungen
 - Viele Indexstrukturen mit ähnlichen Ideen (R-Baum, M-Baum)

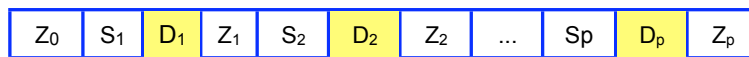
B-Bäume: Agenda

- Grundlagen (Wiederholung)
- ISAM
- Clustered Index
- Indirekte vs. Direkte Speicherung
- Primär vs. Sekundärer Zugriffspfad
- Bulk-Loading
- Präfix B+-Bäume
- Präfix/Suffix-Komprimierung
- Grosse Seiten
- Cache-sensitive B+-Bäume

B-Bäume

- Es gilt: Z_0 weist auf Teilbaum mit Schlüssel $< S_1$
- Für Z_i ($i=1, \dots, p-1$) gilt: $S_i < \text{keys}(Z_i) < S_{i+1}$
- Z_p weist auf Teilbaum mit Schlüssel $> S_p$

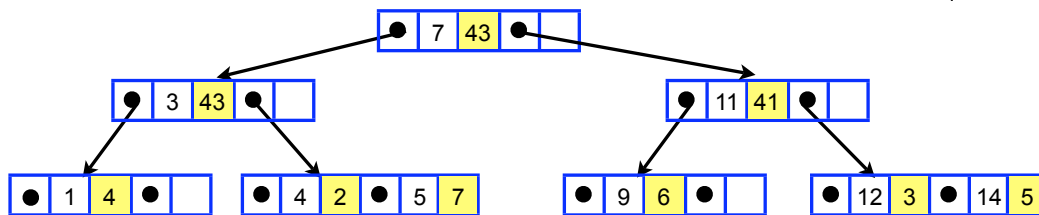
Knotenformat:



Z_i : Zeiger

S_i : Schlüssel

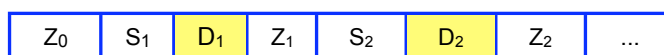
D_i : Daten



B-Bäume: Definition

- Ein B-Baum vom Typ (k, h) hat folgende Eigenschaften:
 1. Jeder Weg von der Wurzel zum Blatt hat die Länge h .
 2. Jeder Knoten hat mindestens $k+1$ Söhne. Die Wurzel ist ein Blatt oder hat mindestens 2 Söhne.
 3. Jeder Knoten hat höchstens $2k+1$ Söhne.

Knotenformat:



Z_i : Zeiger

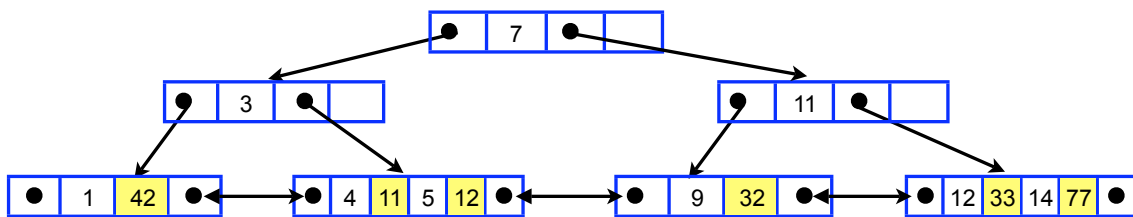
S_i : Schlüssel

D_i : Daten

Literatur: R. Bayer, E. M. McCreight. Organization and Maintenance of Large Ordered Indexes. Acta Informatica, 1:4. 1972. 290-306.

B⁺-Bäume

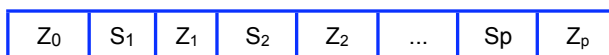
- Wie B-Baum, aber:
 - Daten werden nur in den Blättern gespeichert
 - Effekt: höherer Verzweigungsgrad der Knoten
 - Blätter werden miteinander verkettet:
ISAM (Index Sequential Access Method)



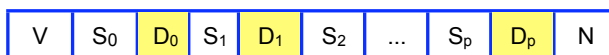
B⁺-Bäume: Definition

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

Knotenformat:



Blattformat:



Z_i : Zeiger

S_i : Schlüssel

D_i : Daten

V : Vorgänger-Zeiger

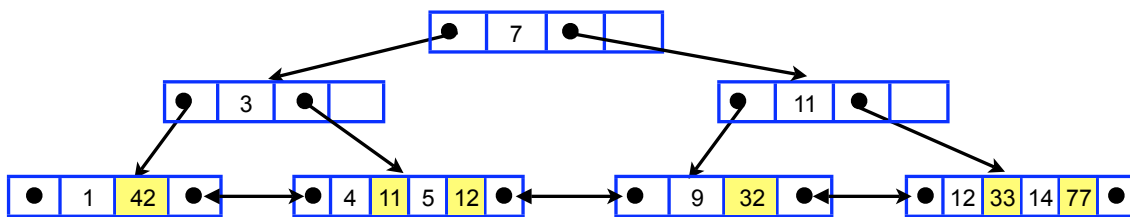
N : Nachfolger-Zeiger

B+-Bäume

- Es gilt: Z_0 weist auf Teilbaum mit Schlüssel $\leq S_1$
- Für Z_i ($i=1, \dots, p-1$) gilt: $S_i < \text{keys}(Z_i) \leq S_{i+1}$
- Z_p weist auf Teilbaum mit Schlüssel $> S_p$
- 2 unterschiedliche Knotentypen: Knoten und Blätter

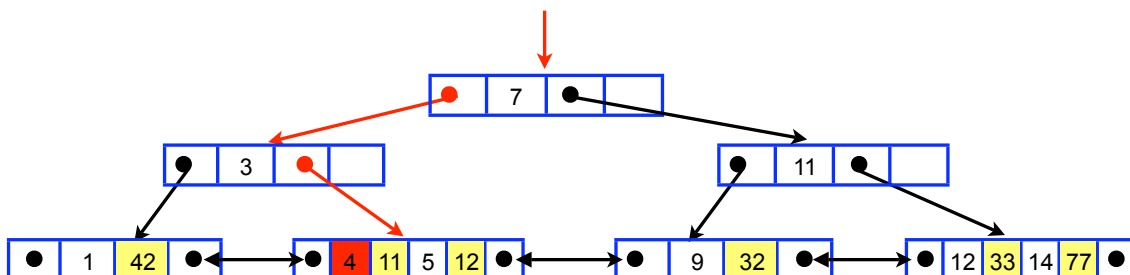


Z_i : Zeiger
 S_i : Schlüssel
 D_i : Daten



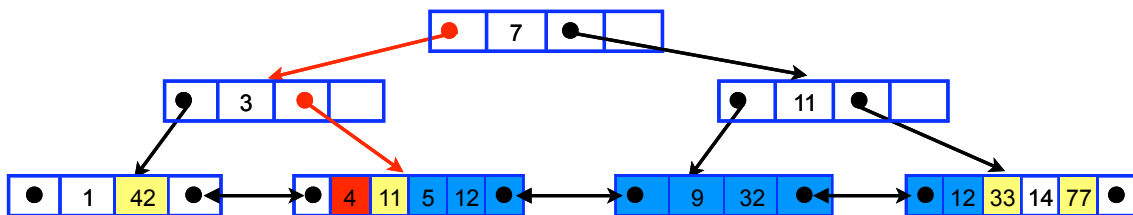
B+-Bäume Punktanfrage

- Rekursive Suche beginnend vom Wurzelknoten
- Innerhalb der Knoten lineare oder binäre Suche
- Es werden genau $h-1$ Knoten und 1 Blatt durchlaufen
- Beispiel: `find_key [4]`



B⁺-Bäume Bereichsanfrage (ISAM)

- Beispiel: find_range [4;12]
- Algorithmus:
 1. Punktanfrage 4 (d.h. find_key [4])
 2. Sequentielle die Blätter lesen bis Eintrag 12 (d.h. ISAM: Index Sequential Access Method)

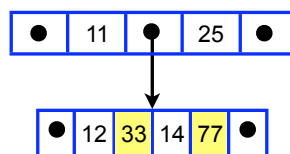


B⁺-Bäume: split

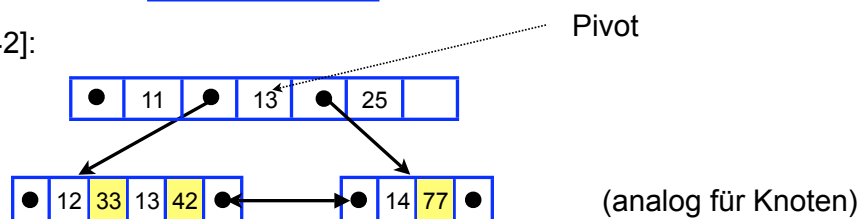
- Falls Blatt $> 2k$ Einträge:
 - Erzeuge neues Blatt
 - Verteile Einträge auf Blätter, so dass jedes Blatt $\geq k$ Einträge
 - Trage neues Blatt + Pivot in Vorgängerknoten ein

Beispiel

Ausgangssituation:



nach insert [13,42]:



B⁺-Bäume: insert (Prozedural)

- insert (42, 12):
 1. node = root
 2. while (node != Leaf)
 - node = choose_subtree(node, 42)
 3. Füge neuen Eintrag in node ein
 4. Falls node mehr als **2k*** Einträge hat:
 - Teile Blatt in zwei neue Blätter auf
 - Verteile Einträge des alten Blattes gleichmässig auf beide Blätter
 - Trage neues Blatt in Vorgängerknoten ein und
 - Ändere Verzweigungsinformation in Vorgängerknoten an
 - Falls Vorgängerknoten mehr als **2k+1** Söhne hat:
 - Teile Vorgängerknoten in zwei neue Knoten auf
 - usw. (bis kein Split mehr notwendig)

...

B⁺-Bäume: insert (Objektorientiert 1/2)

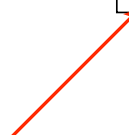
- Node.insert (42, 12):
 1. (split, left, pivot, right) = choose_subtree(42).insert(42, 12)
 2. Falls split:
 - this.insert_node(pivot, right)
 - Falls this.entries > 2k+1:
 - return this.split()
 3. return (false, this, NULL, NULL)
- Leaf.insert (42, 12):
 1. insert_tuple(42, 12)
 2. Falls this.entries > 2k*:
 - return this.split()
 - Sonst
 - return (false, this, NULL, NULL)

Aufruf:
root.insert(42, 12)

B⁺-Bäume: insert (Objektorientiert 2/2)

- Node.insert (42, 12):
 1. (split, left, pivot, right) = choose_subtree(42).insert(42, 12)
 2. Falls split:
 - this.insert_node(pivot, right)
 - Falls this.entries > 2k+1:
 - return this.split()
 3. return (false, this, NULL, NULL)

Aufruf:
root.insert(42, 12)



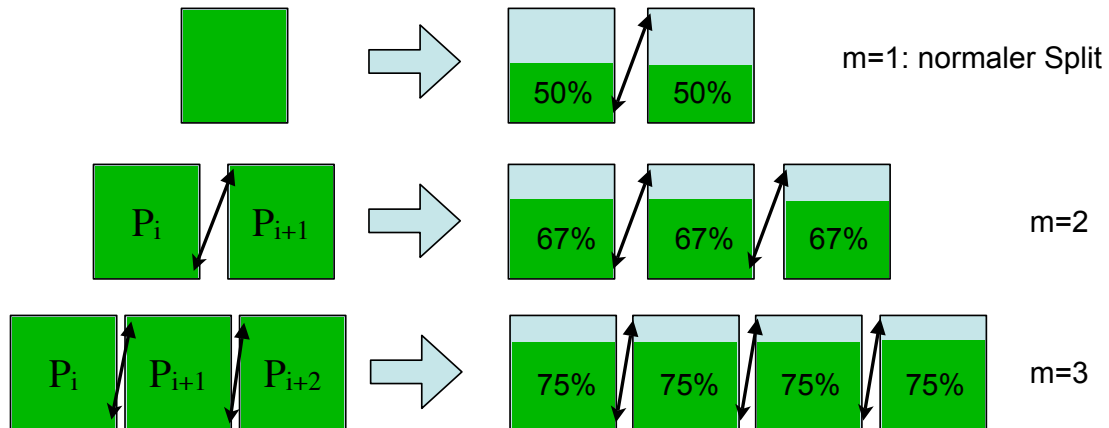
Frage: Was passiert bei einem split der **Wurzel**?

B⁺-Bäume: andere insert-Strategien

- Falls Knoten voll:
 - Versuche Einträge auf **d** Vorgänger- bzw. **d** Nachfolgerknoten umzuverteilen
 - Falls Umverteilen möglich:
 - insert ohne split
 - Sonst
 - split
- Bewertung:
 - verbessert Speicherbelegung des Baums
 - Umschichten u.U. aufwendig
 - Verkettung der Nodes notwendig (einfach)

B⁺-Bäume: andere split-Strategien

- Falls Split:
 - Erzeuge neues Blatt
 - Verteile Einträge m benachbarter Knoten/Blätter, so dass jedes Blatt $\geq k^*$ Einträge hat

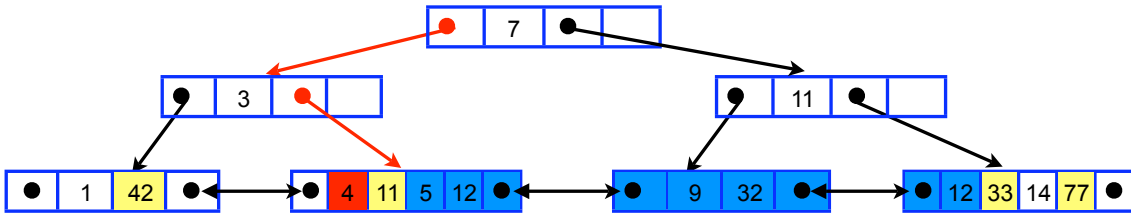


B⁺-Bäume: delete

- Diskussion analog zu split-Operation!
- Bei Unterbelegung von Knoten (Blättern) **merge**
- **merge** = inverse Operation von split
D.h. lege Einträge von m Knoten (Blättern) zusammen.
- Andere Strategie:
 - erlaube kurzzeitig Knoten (Blättern) mit $<k^*$ ($<k+1$) Einträgen.
 - Lösche Knoten (Blättern) nur dann, wenn sie in nächster Zeit nicht wieder gefüllt werden.
 - Verbessert ISAM-Eigenschaft

ISAM und Fragmentierung

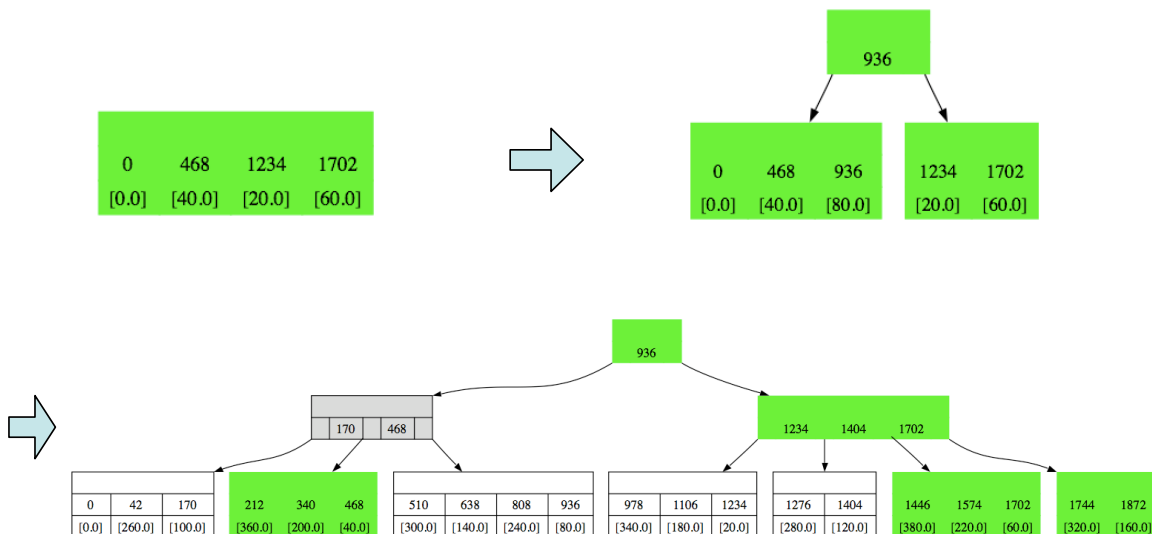
- Verkettung der Blätter erlaubt sequentielles Lesen/Schreiben (ISAM)



- Problem:
 - durch inserts/deletes liegen die Blätter auf Festplatte nicht mehr sequentiell angeordnet
 - je älter der Baum desto grösser die Wahrscheinlichkeit wahlfreier E/A
- Abhilfe:
 - Baum regelmässig defragmentieren (analog zu BS)
 - ggf. Lücken lassen in Bereichen wo ein Anwachsen des Baumes wahrscheinlich ist

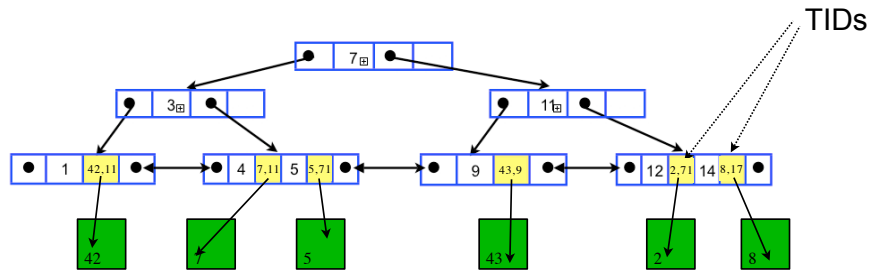
B⁺-Bäume

- Python-Demo

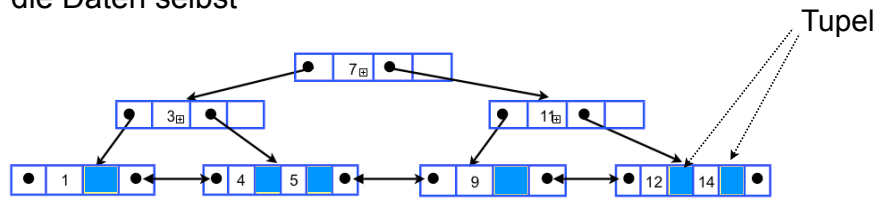


Indirekte vs. Direkte Speicherung

- Was speichert man in den Blättern?
- Entweder: Verweise auf die Sätze (TIDs)



- Oder: die Daten selbst



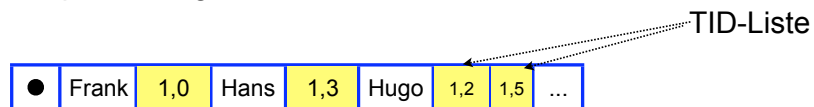
Sekundäre Zugriffspfade und Invertierung

- Suche über Sekundären Zugriffspfad kann mehr als ein Ergebnis liefern (1:n Beziehung zwischen Schlüssel und Ergebnissen)

Vorname	TID	Key	Vorname	Name
Frank	1,0	77	Frank	Meier
Hans	1,1	12	Simon	Schmidt
Hugo	1,2	42	Hugo	Müller
Jens	1,3	11	Hans	Meier
Simon	1,4	25	Jens	Dittrich
	1,5	76	Hugo	Schmidt

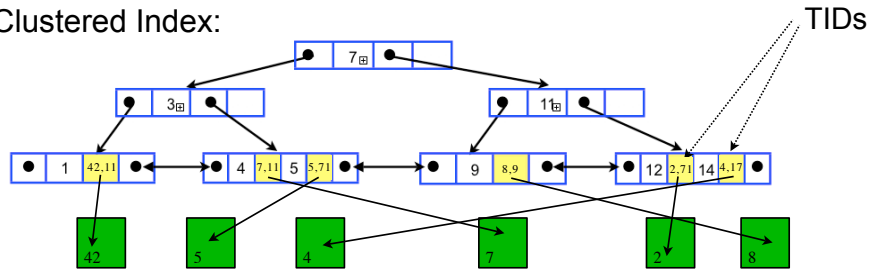
- Es gibt viele Möglichkeiten, diese Invertierung abzubilden.

Beispiel: direkte Speicherung im Blatt:

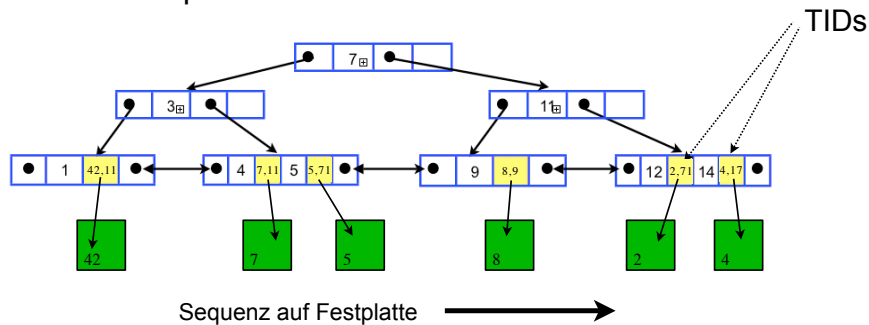


Clustered vs. Non-Clustered Index

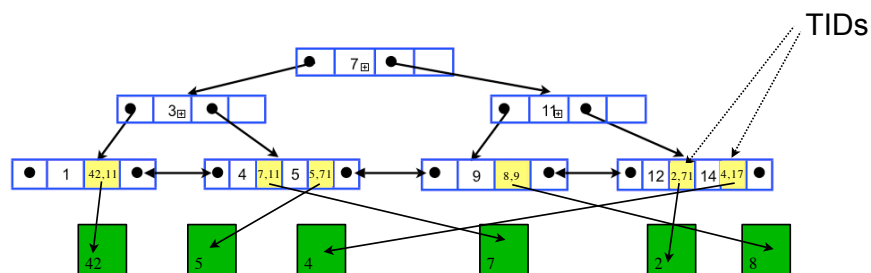
- Non-Clustered Index:



- Clustered Index: Speichere Daten sortiert nach Schlüssel

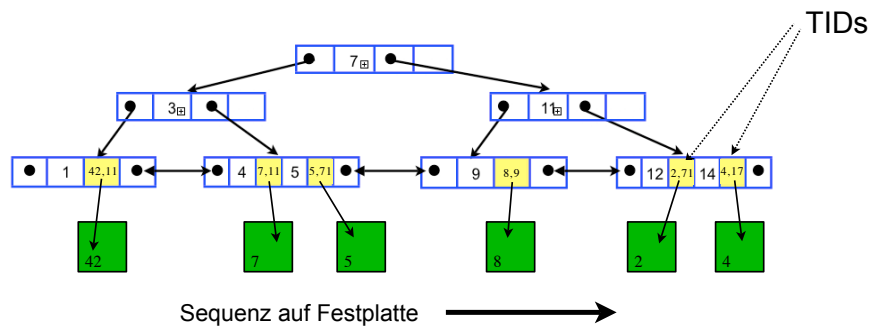


Non-Clustered Index



- mehrere non-clustered Indizes pro Relation möglich
- impliziert indirekten Index
- gut für
 - selektive Anfragen
 - Aggregatfunktionen
 - grosse Sätze

Clustered Index

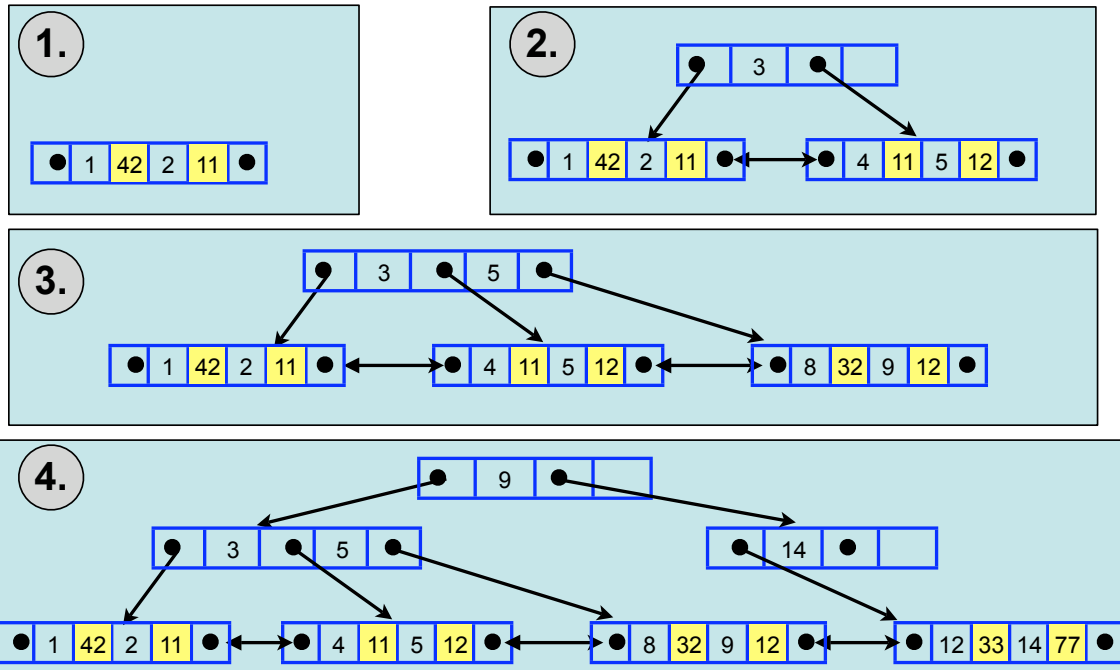


- Maximal 1 clustered Index pro Relation möglich (meistens über Primärschlüssel)
- Achtung:
Clusterung kann auch implizit durch direkte Speicherung erzielt werden

Bulk-Loading

- Problem:
Wie geht man vor, wenn man für eine Relation einen neuen Index erzeugen will?
- Einfaches Verfahren (Bottom-up Aufbau):
 - Sortiere Sätze nach Schlüssel
 - Solange Sequenz Sätze enthält:
 - Nimm die ersten $F \cdot k^*$ Sätze der Sequenz und füge sie in einen neuen Blattknoten ein ($0.5 < F \leq 1$, Füllgrad des Knotens/Blattes)
 - Erstelle Eintrag für diesen Satz im Indexknoten

Bulk-Loading Beispiel

 $k=k^*=1$


24. November 2005

Dr. Jens-Peter Dittrich / Institut für Informationssysteme / jens.dittrich@inf

37

Bulk-Loading: Bewertung

- Aufwand:
 1. Kosten fürs Sortieren
 2. Lineare Kosten in der Anzahl der Datenseiten für Baumaufbau
- Vorteil: Verfahren stellt ISAM-Eigenschaft automatisch her
- Es gibt viele andere Verfahren (z.B. Bulk-Loading eines bestehenden Index)
- Literatur:
 - Lars Arge: The Buffer Tree: A New Technique for Optimal I/O-Algorithms. WADS 1995: 334-345 .
 - Jochen Van den Bercken, Bernhard Seeger: An Evaluation of Generic Bulk Loading Techniques. VLDB 2001: 461-470.

24. November 2005

Dr. Jens-Peter Dittrich / Institut für Informationssysteme / jens.dittrich@inf

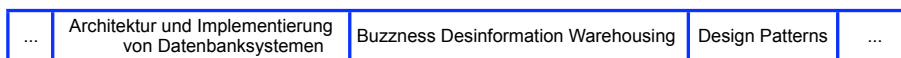
38

Performanzüberlegungen zu B⁺-Bäumen

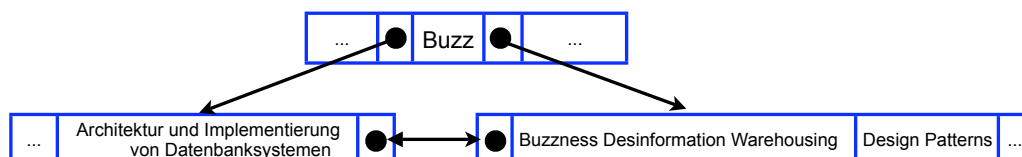
1. Problem: Höhe des Baums hat grossen Einfluss auf Performanz
 - Ziel: fan-out (Anzahl der Söhne) maximieren
 - Techniken:
 - Präfix B⁺-Bäume
 - Präfix/Suffix-Komprimierung
 - Grosse Seiten
2. Problem: B⁺-Bäume nicht optimiert für Hauptspeicher & CPU-Caches
 - Ziel: Cache-Verhalten der Bäume verbessern
 - Techniken:
 - Cache-Sensitive B⁺-Bäume

Präfix-B⁺-Bäume

- Was passiert, wenn man sehr lange Schlüssel im B⁺-Baum speichert?

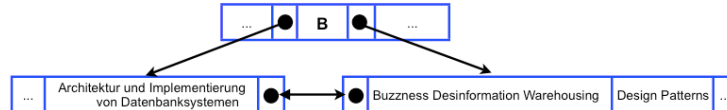


- Viel Platz notwendig für die Schlüssel → weniger Einträge
→ geringerer fan-out → höherer Baum!
- Beobachtung: beliebiger String s zwischen "Archi..." < s <= "Design.." genügt als Separator im Knoten
- Lösung: modifiziere Split-Operation derart, dass Separatoren in den Knoten erzeugt werden



Präfix-Komprimierung

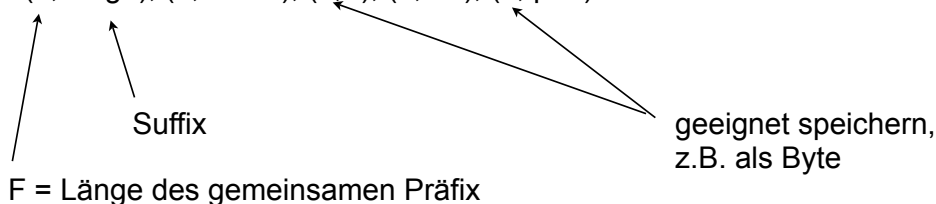
- Beobachtung: beliebiger String s zwischen "Archi..." < s <= "Design.." genügt als Separator im Knoten
- Verbesserung: wähle s minimal, d.h. s ist ein Präfix für die Schlüssel S_i und S_{i+1} , wenn gilt:
 - $S_i < s \leq S_{i+1}$
 - es existiert kein anderer Wert d mit $S_i < d \leq S_{i+1}$ und $\text{len}(d) < \text{len}(s)$



- Folgerungen:
 - Teilbäume haben ein gemeinsames Präfix
 - In den Knoten des Teilbaums muss das Präfix nicht gespeichert werden
 - Aber: Für ISAM muss kompletter Schlüssel auf Blattebene vorliegen!
 - Präfix-B+-Bäume sind eigentlich eine Verallgemeinerung von Digitalbäumen

Suffix-Komprimierung (aka front coding) (1/2)

- Zur Namensgebung:
 - **Präfix**-Komprimierung: das **Suffix** wird weggelassen
 - **Suffix**-Komprimierung: das **Präfix** wird weggelassen
- Idee:
Speichere nur den Teil eines Schlüssels S_i vom Zeichen, in dem er sich vom Vorgänger S_{i-1} unterscheidet
- Beispiel:
 - Hugo, Hummel, Hummer, Hund, Hupen
 - (0, Hugo), (2, mmel), (5, r), (2, nd), (2, pen)



Suffix-Komprimierung (aka front coding) (2/2)

- Idee:
Speichere nur den Teil eines Schlüssels S_i vom Zeichen, in dem er sich vom Vorgänger S_{i-1} unterscheidet
 - Beispiel:
 - Hugo, Hummel, Hummer, Hund, Hupen, ...
 - (0, Hugo), (2, mmel), (5, r), (2, nd), (2, pen), ...
-
- F = Länge des gemeinsamen Präfix
- geeignet speichern,
z.B. als Byte
- Nachteil: keine binäre Suche mehr möglich!
 - Verbesserung: Partial Front Coding

Partielle Suffix-Komprimierung (aka partial front coding)

- Idee:
Speichere jedes k -te Wort vollständig
 - Beispiel: ($k=4$)
 - Hugo, Hummel, Hummer, Hund, Hupen, Husky, Husten, ...
 - (0, Hugo), (2, mmel), (5, r), (2, nd), (0, Hupen), (2, sky), (3, ten), ...
-
- binäre Suche kann hier
aufsetzen
- Literatur: Witten, Moffat, Bell: Managing Gigabytes

Präfix/Suffix-Komprimierung

- Idee: Kombiniere Suffix- und Präfix-Komprimierung
Speichere nur den Teil eines Schlüssels S_i ...
 - vom Zeichen V, in dem er sich vom Vorgänger S_{i-1} unterscheidet
 - bis zum Zeichen N, in dem er sich vom Nachfolger S_{i+1} unterscheidet
- 3 Einträge pro komprimiertem Schlüssel:
 - $F = V-1$ (Länge des gemeinsamen Präfix von S_{i-1} und S_i)
 - $Len = \max(N-V+1, 0)$ (Länge des komprimierten Schlüssels)
 - S_i (Schlüssel)
- Achtung: Verfahren ist verlustbehaftet!

Präfix/Suffix-Komprimierung

Schlüssel (unkomprimiert)	V	N	F	Len	S_i
Hugo	1	3	0	3	Hug
Hummel	3	6	2	4	mmel
Hummer	6	3	5	0	
Hund	3	5	2	3	nd_
Hundertfach	5	8	4	4	ertf
Hunderttausend	8	3	7	0	
Hu ³ pen ⁸	3	1	2	0	

- Achtung: $F = V-1$; $Len = \max(N-V+1, 0)$
 - dies ist verlustbehaftete Komprimierung!
 - es wird nur soviel komprimiert, wie für die Unterscheidung zweier Schlüssel notwendig ist
- Literatur: Robert E. Wagner: Indexing design considerations. IBM Systems Journal 12(4): 351-367 (1973)

Präfix/Suffix-Dekomprimierung

Schlüssel (unkomprimiert)	V	N	F	Len	S _i
Hugo 1 2	1	3	0	3	Hug
Hummel 3 4 5 6 7 8	3	6	2	4	mmel
Hummer 3 4 5 6	6	3	5	0	
Hund 3 4 5	3	5	2	3	nd_
Hundertfach 3 4 5 6 7 8	5	8	4	4	ertf
Hunderttausend 3 4 5 6 7 8	8	3	7	0	
Hu ¹ pen 3 4	3	1	2	0	

- Dekomprimierte Schlüssel:


Hug, Hummel, Humme, Hund_, Hundertf, Hundert, Hu

Performanzüberlegungen zu B⁺-Bäumen

1. Problem: Höhe des Baums hat grossen Einfluss auf Performanz
 - Ziel: fan-out (Anzahl der Söhne) maximieren
 - Techniken:
 - Präfix B⁺-Bäume
 - Präfix/Suffix-Komprimierung
- ➔ Grosse Seiten
2. Problem: B⁺-Bäume nicht optimiert für Hauptspeicher & CPU-Caches
 - Ziel: Cache-Verhalten der Bäume verbessern
 - Techniken:
 - Cache-Sensitive B⁺-Bäume

Verwendung grosser Seiten

Szenario: Verdopplung der Seitengrösse

- Vorteile:
 - Fan-out vergrössert sich  geringere Baumhöhe
 - E/A-Kosten steigen kaum
 - Bei Bereichsanfragen nur halb so viele Zugriffe für ISAM-Zugriff
- Nachteile:
 - mehr Verschnitt durch schlecht gefüllte Seiten
 - mehr unnötige Daten im DB-Puffer
 - Aufwand zum Durchsuchen innerhalb der Seite steigt
- Abhilfe:
 - "logisch grosse Seiten": müssen aber intern sequentiell auf Platte abgelegt werden

Verwendung grosser Seiten

- Andere Idee:
Nutzen unterschiedlich grosser Seiten
- Nachteile:
 - DB-Puffer unterstützt meist nur eine Seitengrösse

Performanzüberlegungen zu B⁺-Bäumen

1. Problem: Höhe des Baums hat grossen Einfluss auf Performanz
 - Ziel: fan-out (Anzahl der Söhne) maximieren
 - Techniken:
 - Präfix B⁺-Bäume
 - Präfix/Suffix-Komprimierung
 - Grosse Seiten
- ➔ 2. Problem: B⁺-Bäume nicht optimiert für Hauptspeicher & CPU-Caches
 - Ziel: Cache-Verhalten der Bäume verbessern
 - Techniken:
 - Cache-Sensitive B⁺-Bäume

Cache-Sensitive B⁺-Bäume

- Problem: B⁺-Bäume wurden als externe Datenstruktur entworfen
- Wie optimiert man B⁺-Bäume so, damit sie bezüglich der CPU-nahen Ebenen der Speicherhierarchie effizient sind?
- 2 prinzipielle Ansätze:
 - cache-oblivious B⁺-Bäume
 - keine genaue Kenntnis (obliviousness) der Speicherhierarchie
 - generischer Ansatz
 - nicht zugeschnitten auf eine bestimmte Architektur(Das Thema 'cache oblivious-Algos.' reicht für eine eigene VL.)
 - cache-aware B⁺-Bäume
 - Kenntnis (awareness) über Speicherhierarchie vorhanden
 - speziellere Ansätze
 - speziell zugeschnitten auf vorherrschende Rechner-Architektur

Cache-Sensitive B⁺-Bäume

- Agenda
 - Graefe
 - CSB+-trees
 - Prefetching
 - Fractal Prefetching

Übersicht über Techniken

- Literatur: Goetz Graefe, Per-Åke Larson: B-Tree Indexes and CPU Caches. ICDE 2001:349-358.

B-tree Indexes and CPU Caches

Goetz Graefe and Per-Åke Larson
Microsoft
{goetzg, palarson}@microsoft.com

Abstract

Since many existing techniques for exploiting CPU caches in the implementation of B-tree indexes have not been discussed in the literature, most of them are surveyed here. Rather than providing a detailed performance evaluation for one or two of them on some specific, contemporary hardware, the purpose here is to survey and to make widely available this heretofore folklore knowledge in order to enable, structure, and hopefully stimulate future research.

1. Introduction

Today, most DBMS installations are “compute bound”, at least after sufficient disk drives and indexes have been added. However, the bottlenecks are actually not CPU cycles for instruction execution but cache faults and pipeline stalls. At the present time, many product developers believe that the traditional goal of reducing instruction path length is much less important than techniques that improve cache faults and branch prediction.

This brief paper is a summary of knowledge, much of it folklore, about how to adapt database indexing software, in particular B-tree code, to memory hierarchies with multiple levels of caches. Given the current trends in hardware design that require increasing sophistication from system software and compilers, for example Intel’s 64-bit processor or Transmeta’s innovative processors that employ on-the-fly translation among different machine codes [14], this survey might be useful to both database researchers and implementers of commercial products.

There are just a few principal sources of ideas behind these techniques. Unfortunately, they often conflict, and careful engineering tradeoffs are required, if possible supported by rigorous research and measurements on the specific target architecture if it is known. First, anything that reduces code complexity and code size is good since it reduces cache faults for instructions. In particular, reducing the need for “if”, “case”, and “while” statements as well as virtual functions reduces erroneous predictions of conditional and computed branches. Second, anything helps that compresses or shortens B-tree entries, in leaves or in interior nodes. In fact, compression of database structure is a continuing goal, unaffected by the continually decreasing prices for disk space and memory, because the

higher levels of the memory hierarchy continue to be small and expensive (on-chip and on-board caches) and because the bandwidths between all levels of the memory hierarchy seems eternally insufficient. Third, many of the techniques to deal with the delay (access and transfer time) between memory and disk can be re-applied to CPU caches. Finally, since B-trees and (compression-based) sorting are related in many interesting ways, many of the techniques that make sorting more cache-efficient can be re-applied to B-tree indexes.

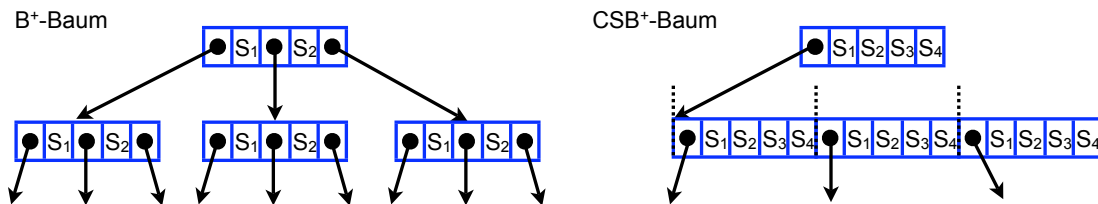
1.1. Scope of this Survey

Most of this paper focuses on exact match index searches as typical in OLTP applications and in index nested loops joins. Inasmuch as materialized views satisfy their purpose, even decision support (DSS) and OLAP applications behave like read-only OLTP applications, i.e., lots of index searches rather than large sort and hash operations, but no issues with respect to concurrency control and recovery. Therefore, materialized views and their effect on OLAP and DSS applications increase the importance of very efficient implementation of B-tree indexes. Range queries typically are combinations of exact match searches for the lower or upper end of the range, combined with sequential scans. This also applies to searches required for existence checks or semi-joins. Considerations specific to insert, update, and delete operations as well as index creation or page compaction are mentioned where warranted. Concurrency control and other transactional issues are generally ignored here since they are largely orthogonal to the issue of cache efficiency.

We only outline the key ideas for each technique, without attempting to list explicitly every thinkable variant and combination. In particular, we do not consider how to tune data structures and algorithms to multiple levels of caches and multiple sizes of cache lines, cache-to-data-out memory and other burst transfer and read-ahead techniques, etc. Moreover, we do not consider in detail which techniques complement or inhibit each other, or how specifically these techniques should be realized in data structures and code. Without doubt, there is plenty of room for future experimental research in this area. The purpose of this paper is to stimulate and to provide a starting point for such advanced research, not to preempt it.

CSB+-Trees

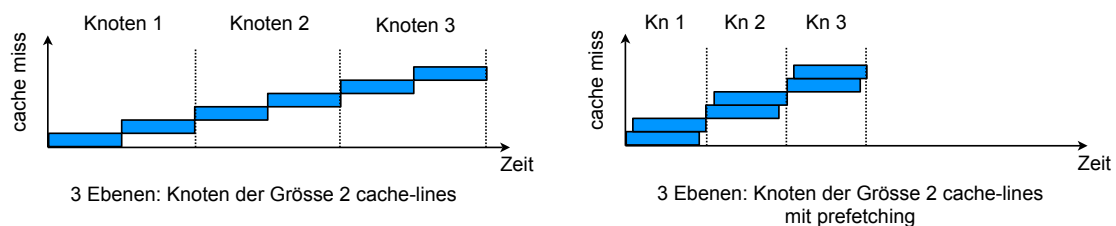
- =Cache Sensitive B⁺-Trees
- Idee:
 - Ebenenweises Layout der Knoten
 - Alle Zeiger bis auf einen eliminieren
 - Verdoppelt Verzweigungsgrad für Knoten



- Suche bis zu 35% besser. Aber: updates 30% langsamer!
- Literatur: Jun Rao, Kenneth A. Ross: Making B+-Trees Cache Conscious in Main Memory. SIGMOD Conference 2000: 475-486

Prefetching: pB+-Trees

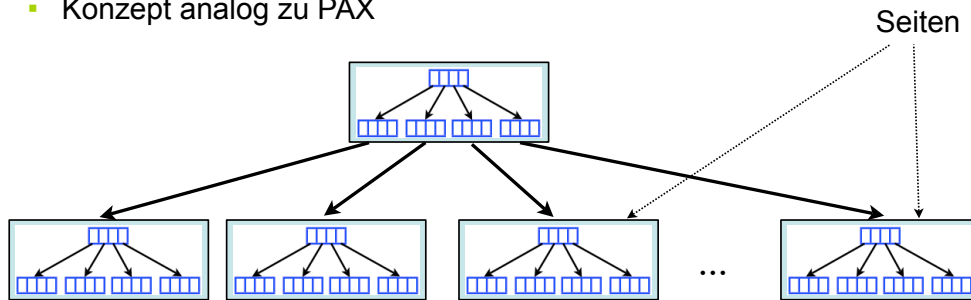
- Idee:
 - Knotengröße ist Vielfaches der Cache Line
 - Alle Cache Lines prefetchen bevor man im Knoten sucht
 - Layout: erst alle Schlüssel hintereinander speichern, dann die Pointer



- >2 Performanz-Steigerung für Suche und Update
- Technik komplementär zu CSB+-Bäumen
- Literatur: Shimin Chen, Phillip B. Gibbons, Todd C. Mowry: Improving Index Performance through Prefetching. SIGMOD Conference 2001

Fractal Prefetching: fpB⁺-Trees

- Idee:
 - Baum von Bäumen
 - Äussere Ebene: Disk-optimierter B+-Baum
 - Innerhalb einer Seite: Cache-optimierter pB⁺-Baum
 - Konzept analog zu PAX



- Literatur: Shimin Chen, Phillip B. Gibbons, Todd C. Mowry, Gary Valentin: Fractal prefetching B⁺-Trees: optimizing both cache and disk performance. SIGMOD Conference 2002: 157-168

Nächste Woche

4. Mehrdimensionale Zugriffspfade

