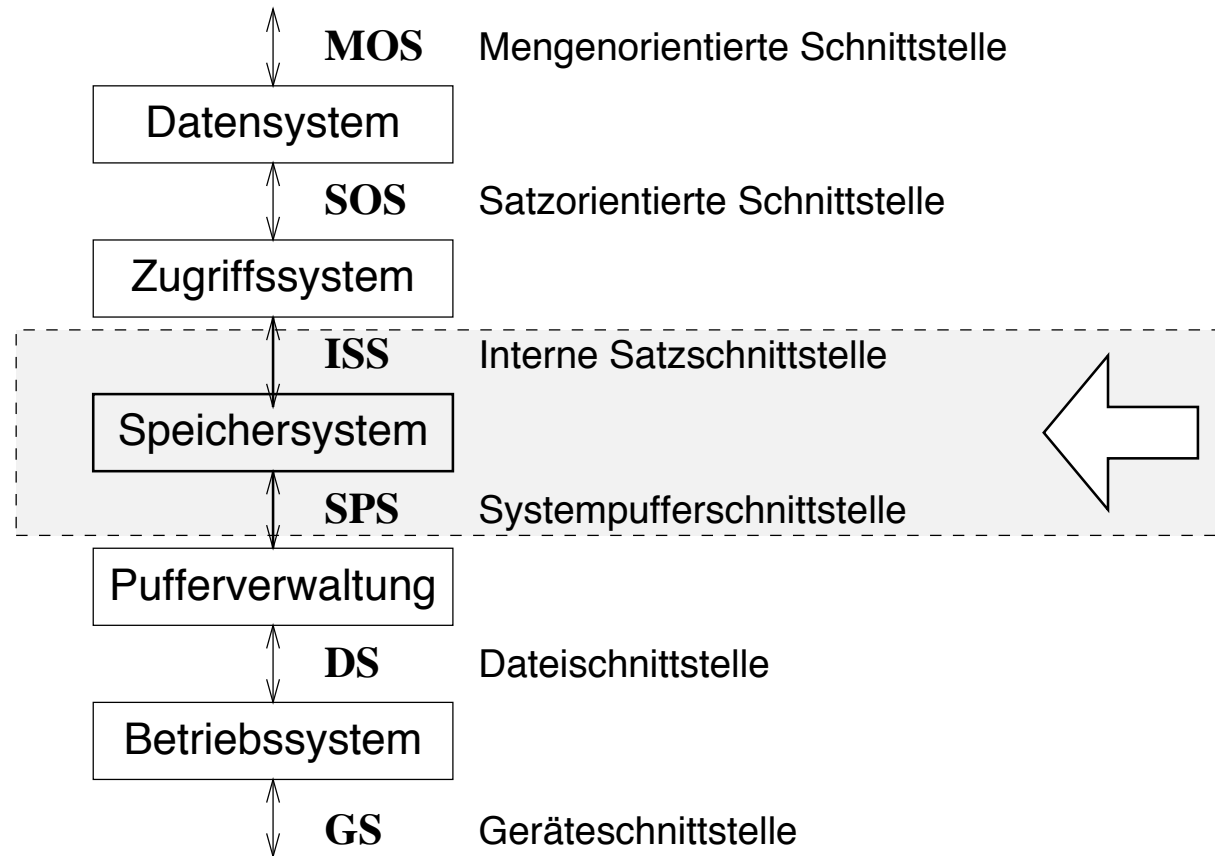


Einordnung in 5-Schichten-Architektur

- *Speichersystem* fordert über Systempufferschnittstelle Seiten an
- interpretiert diese als *interne Datensätze*
- interne Realisierung der logischen Datensätze mit Hilfe von Zeigern, speziellen Indexeinträgen und weiteren Hilfsstrukturen
- *Zugriffssystem* abstrahiert von der konkreten Realisierung

Einordnung /2



Klassifikation der Speichertechniken

- Kriterien für **Zugriffsstrukturen** oder **Zugriffsverfahren**:
 - ▶ organisiert interne Relation selbst (**Dateiorganisationsform**) oder zusätzliche Zugriffsmöglichkeit auf bestehende interne Relation (**Zugriffspfad**)
 - ▶ Art der Zuordnung von gegebenen Attributwerten zu Datensatz-Adressen
 - ▶ Arten von Anfragen, die durch Dateiorganisationsformen und Zugriffspfade effizient unterstützt werden können

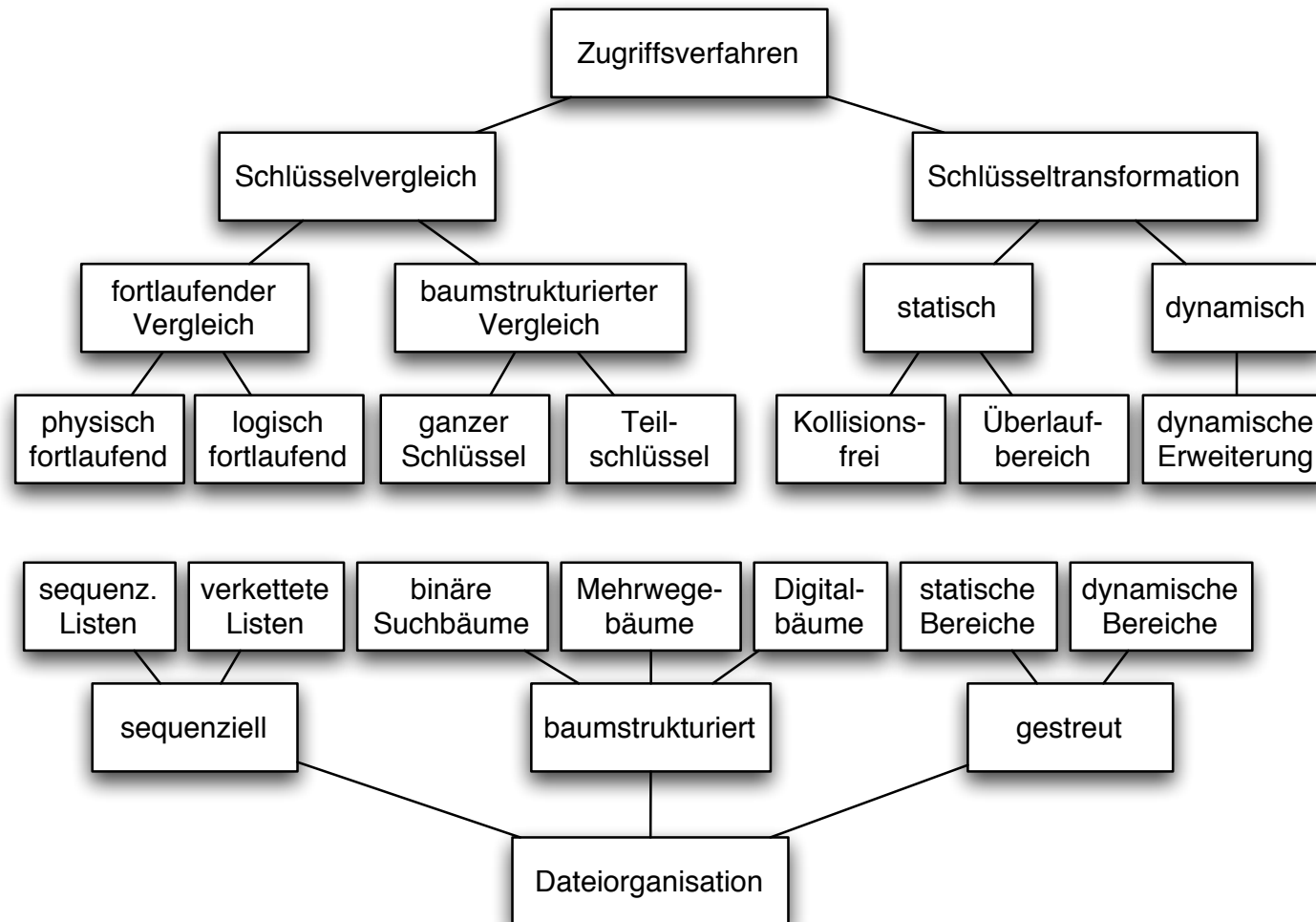
Dateiorganisation vs. Zugriffspfad

- **Dateiorganisationsform**: Form der Speicherung der internen Relation
 - ▶ unsortierte Speicherung von internen Tupeln: **Heap-Organisation**
 - ▶ sortierte Speicherung von internen Tupeln: **sequenzielle Organisation**
 - ▶ gestreute Speicherung von internen Tupeln: **Hash-Organisation**
 - ▶ Speicherung von internen Tupeln in mehrdimensionalen Räumen: **mehrdimensionale Dateiorganisationsformen**
- üblich: Sortierung oder Hashfunktion über Primärschlüssel
- sortierte Speicherung plus zusätzlicher Primärindex über Sortierattributen: **index-sequenzielle Organisationsform**

Dateiorganisation vs. Zugriffspfad /2

- **Zugriffspfad**: über grundlegende Dateiorganisationsform hinausgehende Zugriffsstruktur, etwa Indexdatei
 - ▶ Einträge $(K, K \uparrow)$: K der Wert eines Primär- oder Sekundärschlüssels, $K \uparrow$ Datensatz oder Verweis auf Datensatz
 - ▶ K : **Suchschlüssel**, genauer: **Zugriffsattribute** und **Zugriffsattributwerte**
- $K \uparrow$:
 - ▶ **ist Datensatz selbst**: Zugriffspfad wird Dateiorganisationsform
 - ▶ **ist Adresse eines internen Tupels**: Primärschlüssel; Sekundärschlüssel mit $(K, K \uparrow_1), \dots, (K, K \uparrow_n)$ für denselben Zugriffsattributwert K
 - ▶ **ist Liste von Tupeladressen**: Sekundärschlüssel; nachteilig ist variable Länge der Indexeinträge

Klassifikation [Härder Rahm 2001]



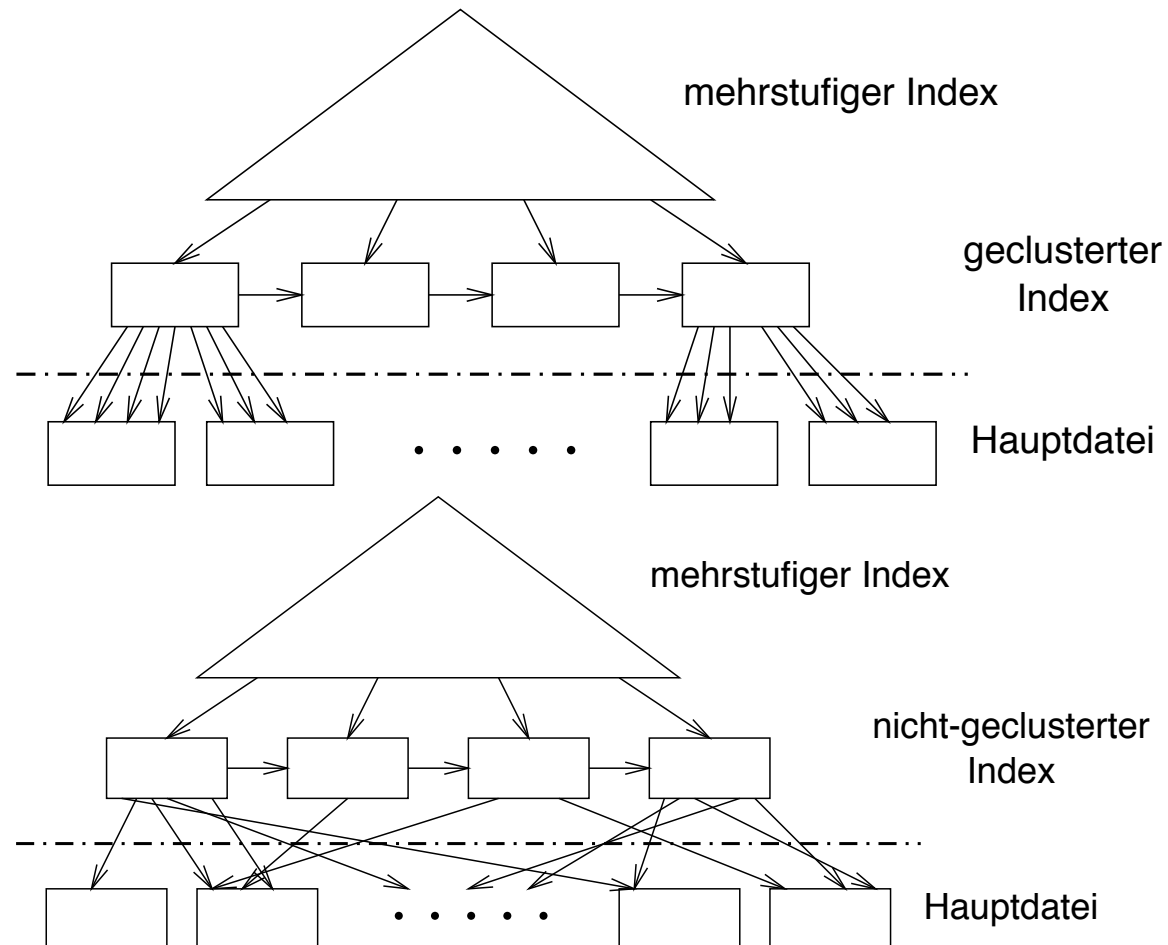
Dünn- vs. dichtbesetzter Index

- **dünnbesetzter Index**: nicht für jeden Zugriffsattributwert K ein Eintrag in Indexdatei
 - ▶ interne Relation sortiert nach Zugriffsattributen: im Index reicht ein Eintrag pro Seite \Rightarrow Index verweist mit $(K_1, K_1 \uparrow)$ auf *Seitenanführer*, nächste Indexeintrag $(K_2, K_2 \uparrow)$
 - ▶ Datensatz mit Zugriffsattributwert $K_?$ mit $K_1 \leq K_? < K_2$ ist auf Seite von $K_1 \uparrow$ zu finden
- *indexsequenzielle Datei*: sortierte Datei mit dünnbesetztem Index als Primärindex
- **dichtbesetzter Index**: für jeden Datensatz der internen Relation ein Eintrag in Indexdatei
- Primärindex kann dichtbesetzter Index sein, wenn Dateiorganisationsform Heap-Datei, aber auch bei Sortierung (**geclusterter Index**)

Geclusterter vs. nicht-geclusterter Index

- **geclusterter Index**: in der gleichen Form sortiert wie zugehörige interne Relation
 - ▶ Bsp.: interne Relation `KUNDEN` nach Kundennummern sortiert \Rightarrow Indexdatei über dem Attribut `KNr` ist dann üblicherweise geclustert
- **nicht-geclusterter Index**: interne Relation ist anders organisiert als der Index
 - ▶ Bsp.: über `Name` von Kunden ein Sekundärindex, Datei selbst ist aber nach `KNr` sortiert (oder auch gar nicht sortiert)
- Primärindex kann dünnbesetzt und geclustert sein
- jeder dünnbesetzte Index ist auch ein geclusterter Index, aber nicht umgekehrt
- Sekundärindex kann nur dichtbesetzter, nicht-geclusterter Index sein (dann auch “invertierte Datei” genannt), da Sortierungen unterschiedlich

Geclusterter vs. nicht-geclusterter Index /2



Schlüsselvergleich vs. -transformation

- **Schlüsselvergleich:** Zuordnung von Primär- oder Sekundärschlüsselwerten zu Adressen in Hilfsstruktur wie Indexdatei
 - ▶ Bsp.: indexsequenzielle Organisation, B-Baum, KdB-Baum, ...
- **Schlüsseltransformation:** berechnet Tupeladresse aufgrund Formel aus Primär- oder Sekundärschlüsselwerten (statt Indexeinträgen nur Berechnungsvorschrift gespeichert)
 - ▶ Bsp.: *Hashverfahren*

Statische vs. dynamische Struktur

- **statische Zugriffsstruktur**: optimal nur bei bestimmter (fester) Anzahl von verwaltenden Datensätzen
- **dynamische Zugriffsstruktur**: unabhängig von der Anzahl der Datensätze optimal
 - ▶ dynamische Adresstransformationsverfahren verändern dynamisch Bildbereich der Transformation
 - ▶ dynamische Indexverfahren verändern dynamisch Anzahl der Indexstufen
⇒ in DBS üblich

Anforderung an Speichertechniken

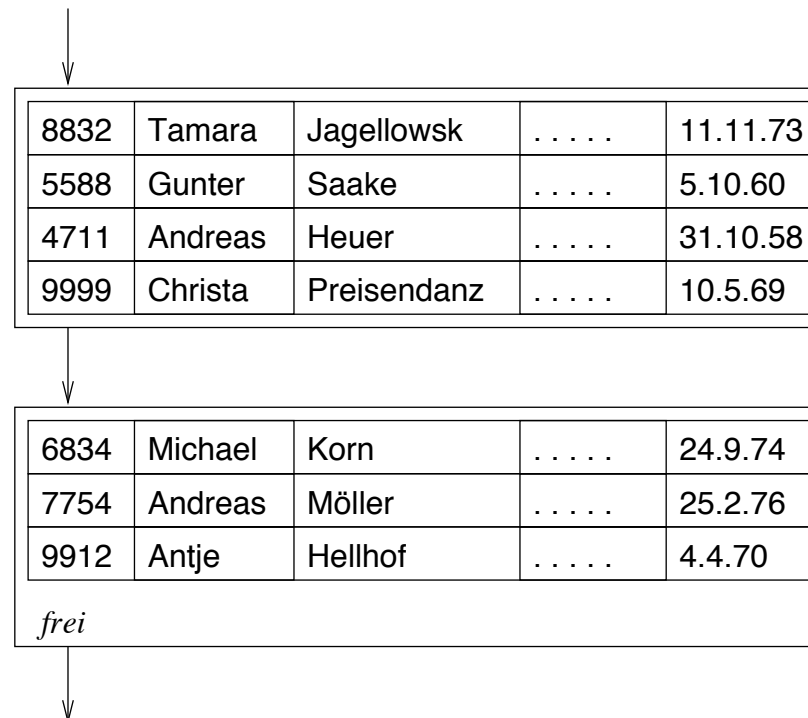
- dynamisches Verhalten
- Effizienz beim Einzelzugriff (Schlüsselsuche beim Primärindex)
- Effizienz beim Mehrfachzugriff (Schlüsselsuche beim Sekundärindex)
- Ausnutzung für sequentiellen Durchlauf (Sortierung, geclusterter Index)
- Clustering
- Anfragetypen: *exact-match*, *partial-match*, *range queries* (Bereichsanfragen)

Statische Verfahren

- Heap, indexsequenziell, indiziert-nichtsequenziell
- oft grundlegende Speichertechnik in RDBS
- direkte Organisationsformen: keine Hilfsstruktur, keine Adressberechnung (Heap, sequenziell)
- statische Indexverfahren für Primärindex und Sekundärindex

Heap-Organisation

- völlig unsortiert speichern
- physische Reihenfolge der Datensätze ist zeitliche Reihenfolge der Aufnahme von Datensätzen

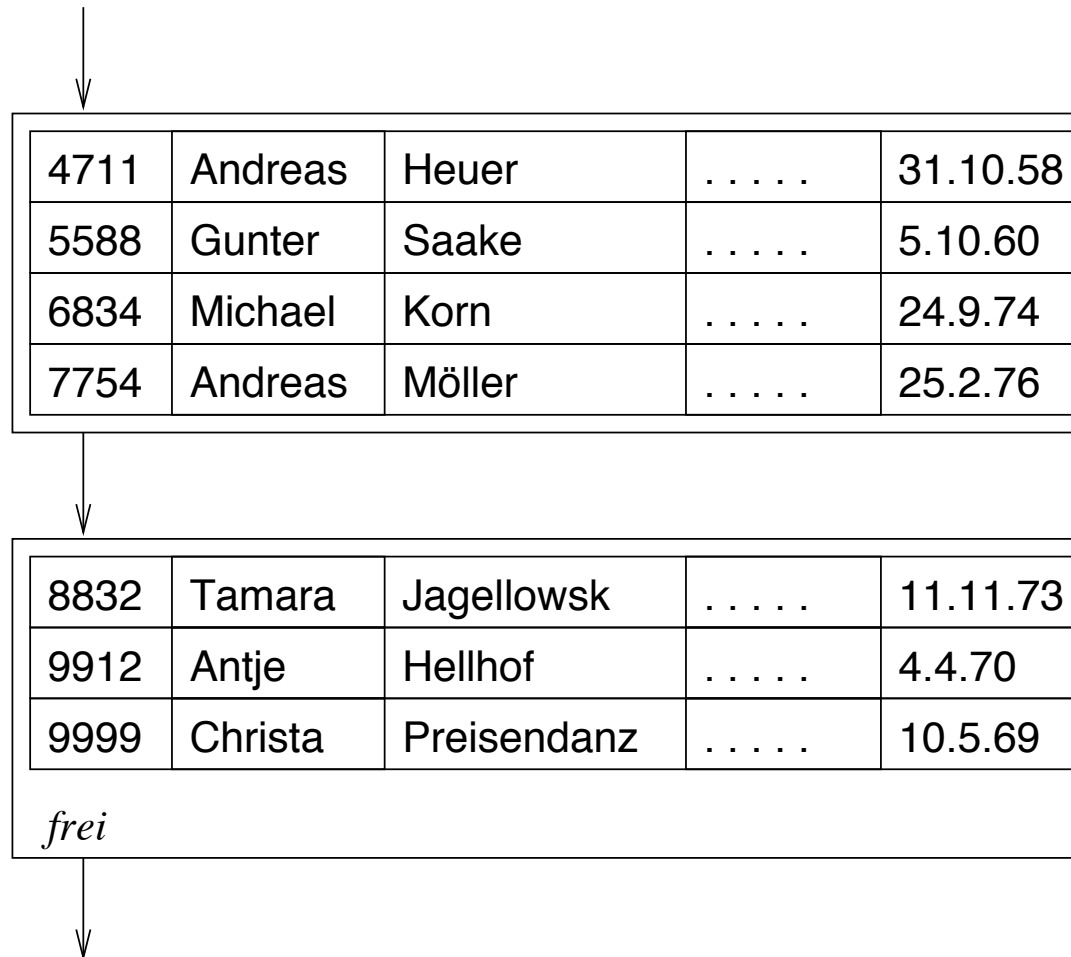


Heap: Operationen

- **insert**: Zugriff auf letzte Seite der Datei. Genügend freier Platz \Rightarrow Satz anhängen. Sonst nächste freie Seite holen
- **delete**: **lookup**, dann Löschbit auf 0 gesetzt
- **lookup**: sequenzielles Durchsuchen der Gesamtdatei, maximaler Aufwand (Heap-Datei meist zusammen mit Sekundärindex eingesetzt; oder für sehr kleine Relationen)
- Komplexitäten:
 - ▶ Neuaufnahme von Daten $O(1)$
 - ▶ Suchen $O(n)$, wobei n Anzahl der Datensätze

Sequenzielle Speicherung

- sortiertes Speichern der Datensätze



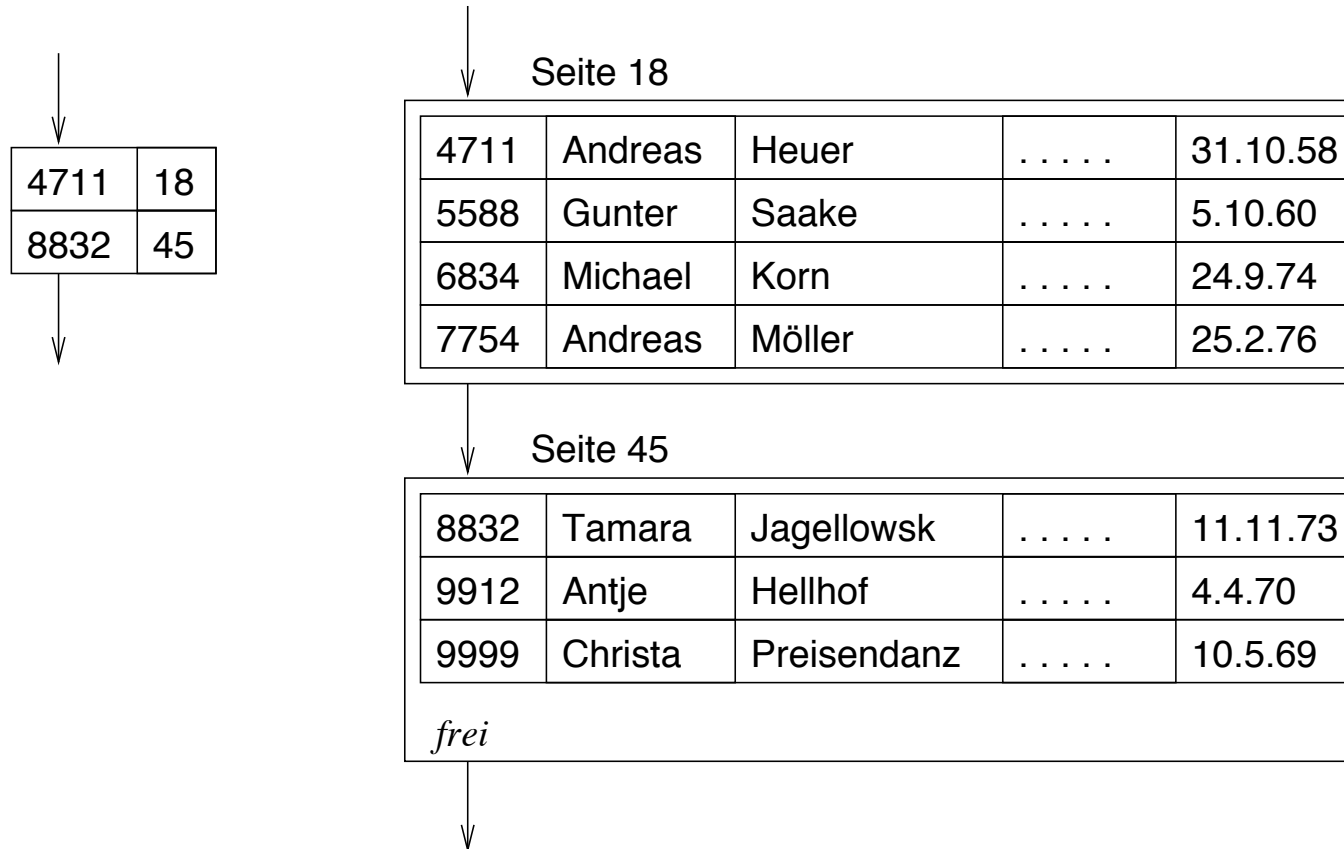
Sequenzielle Datei: Operationen

- **insert**: Seite suchen, Datensatz einsortieren \Rightarrow beim Anlegen oder sequenziellen Füllen einer Datei jede Seite nur bis zu gewissem Grad (etwa 66%) füllen
- **delete**: Aufwand bleibt
- Folgende Dateiorganisationsformen:
 - ▶ schnelleres **lookup**
 - ▶ mehr Platzbedarf (durch Hilfsstrukturen wie Indexdateien)
 - ▶ mehr Zeitbedarf bei **insert** und **delete**
- klassische Indexform: indexsequenzielle Dateiorganisation

Indexsequenzielle Dateiorganisation

- Kombination von sequenzieller Hauptdatei und Indexdatei: *indexsequenzielle Dateiorganisationsform*
- Indexdatei kann geclusterter, dünnbesetzter Index sein
- mindestens zweistufiger Baum
 - ▶ Blattebene ist *Hauptdatei* (Datensätze)
 - ▶ jede andere Stufe ist *Indexdatei*

Indexsequenzielle Dateiorganisation /2

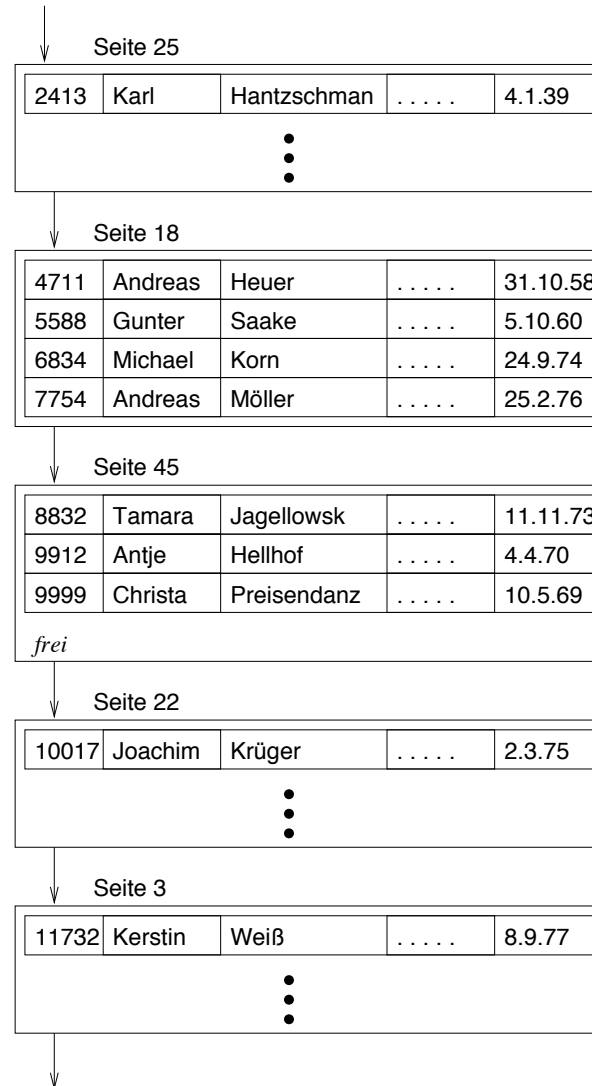


Aufbau der Indexdatei

- Datensätze in Indexdatei:
(Primärschlüsselwert, Seitennummer)
zu jeder Seite der Hauptdatei genau ein Index-Datensatz in Indexdatei
- Problem: „Wurzel“ des Baumes bei einem einstufigen Index benötigt event. nicht nur eine Seite

Aufbau der Indexdatei /2

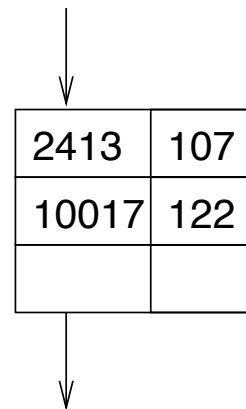
2413	25
4711	18
8832	45
10017	22
11732	3



Mehrstufiger Index

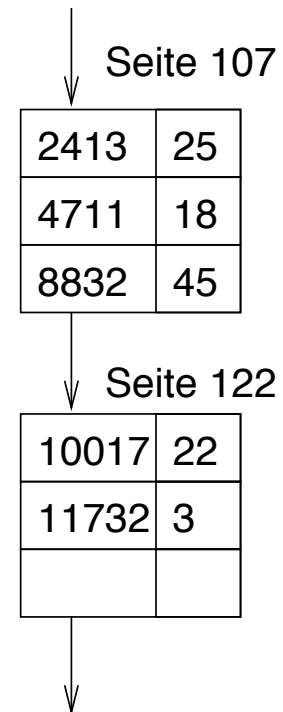
- Optional: Indexdatei wieder indexsequenziell verwalten
- Idealerweise: Index höchster Stufe nur noch eine Seite

Indexdatei 2. Stufe



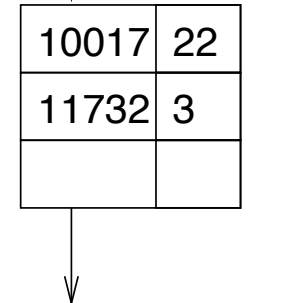
2413	107
10017	122

Indexdatei 1. Stufe



2413	25
4711	18
8832	45

Seite 107



10017	22
11732	3

Seite 122

lookup bei indexsequenziellen Dateien

- **lookup**-Operation sucht Datensatz zum Zugriffsattributwert w
- Indexdatei sequenziell durchlaufen, dabei (v_1, s) im Index gesucht mit $v_1 \leq w$:
 - ▶ (v_1, s) ist letzter Satz der Indexdatei, dann kann Datensatz zu w höchstens auf dieser Seite gespeichert sein (wenn er existiert)
 - ▶ nächster Satz (v_2, s') im Index hat $v_2 > w$, also muß Datensatz zu w , wenn vorhanden, auf Seite s gespeichert sein
- Man sagt dann, dass (v_1, s) den Zugriffsattributwert w *überdeckt*.

insert bei indexsequenziellen Dateien

- **insert**: zunächst mit **lookup** Seite 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
- Alternativ neuen Datensatz auf Überlaufseite zur gefundenen Seite

`delete` bei indexsequenziellen Dateien

- **`delete`**: zunächst mit **`lookup`** Seite finden
- Satz auf Seite löschen (Löschbit auf 0)
- erster Satz auf Seite: Index anpassen
- Falls Seite nach Löschen leer: Index anpassen, Seite an Freispeicherverwaltung zurück

Probleme indexsequenzieller Dateien

- *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)

Indiziert-nichtsequenzieller Zugriffspfad

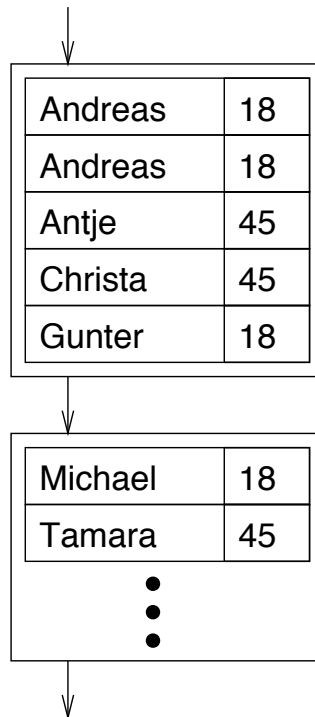
- zur Unterstützung von Sekundärschlüsseln
- mehrere Zugriffspfade dieser Form pro Datei möglich
- einstufig oder mehrstufig: höhere Indexstufen wieder indexsequenziell organisiert

Aufbau der Indexdatei

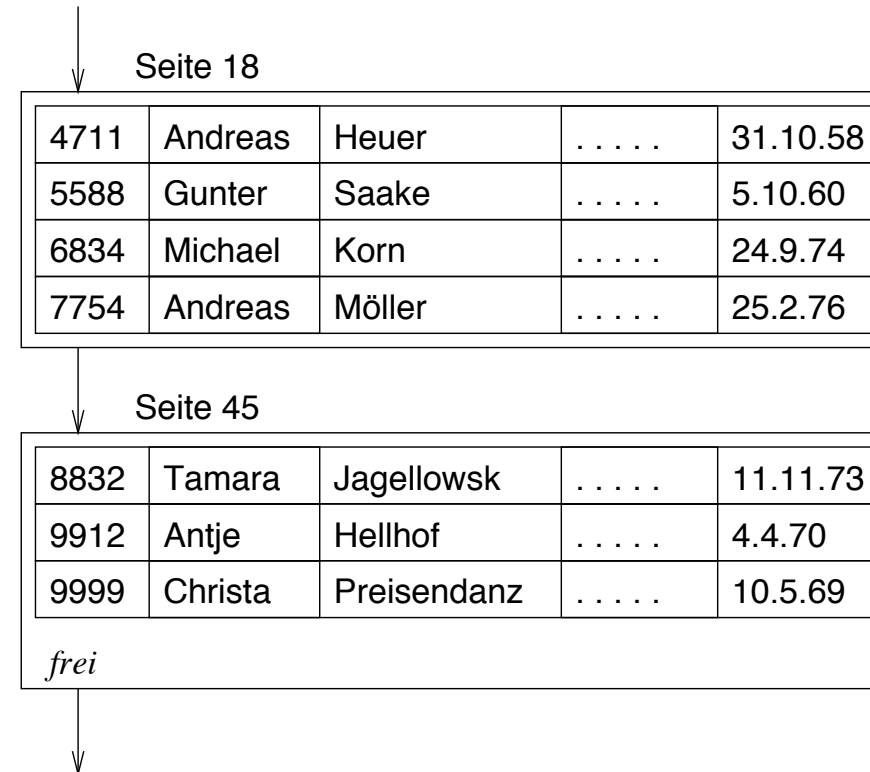
- Sekundärindex, dichtbesetzter und nicht-geclusteter Index
- zu jedem Satz der Hauptdatei Satz (w, s) in der Indexdatei
- w Sekundärschlüsselwert, s zugeordnete Seite
 - ▶ entweder für ein w mehrere Sätze in die Indexdatei aufnehmen
 - ▶ oder für ein w Liste von Adressen in der Hauptdatei angeben

Aufbau der Indexdatei /2

Zugriffspfad Vorname

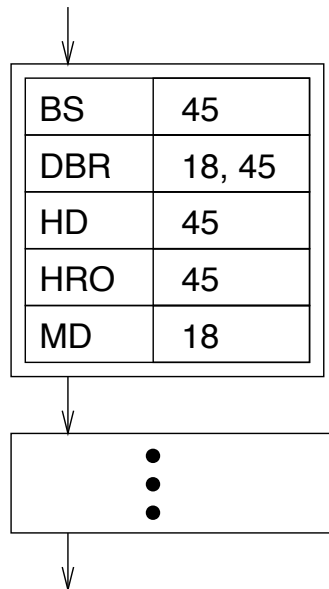


Hauptdatei

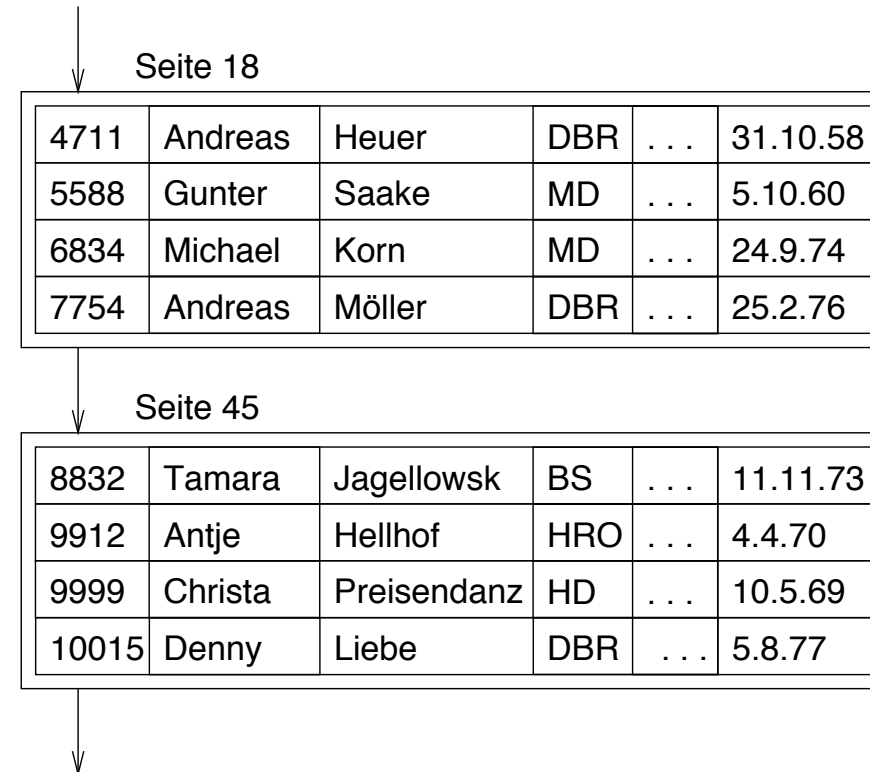


Aufbau der Indexdatei /3

Zugriffspfad Ort



Hauptdatei



Operationen

- **lookup**: w kann mehrfach auftreten, Überdeckungstechnik nicht benötigt
- **insert**: Anpassen der Indexdateien
- **delete**: Indexeintrag entfernen

Baumverfahren

- Stufenanzahl dynamisch verändern
- wichtigste Baumverfahren: *B-Bäume* und ihre Varianten
- B-Baum-Varianten sind noch „allgegenwärtiger“ in heutigen Datenbanksystemen als SQL
- SQL nur in der relationalen und objektrelationalen Datenbanktechnologie verbreitet; B-Bäume überall als Grundtechnik eingesetzt

B-Bäume

- Ausgangspunkt: ausgeglichener, balancierter Suchbaum
- *Ausgeglichen* oder *balanciert*: alle Pfade von der Wurzel zu den Blättern des Baumes gleich lang
- Hauptspeicher-Implementierungsstruktur: binäre Suchbäume, beispielsweise AVL-Bäume von Adelson-Velskii und Landis
- Datenbankbereich: Knoten der Suchbäume zugeschnitten auf Seitenstruktur des Datenbanksystems
- mehrere Zugriffsattributwerte auf einer Seite
- *Mehrweg-Bäume*

Prinzip des B-Baumes

- *B-Baum* von Rudolf Bayer (B für balanciert, breit, buschig, Bayer, **NICHT**: binär)
- dynamischer, balancierter Indexbaum, bei dem jeder Indexeintrag auf eine Seite der Hauptdatei zeigt

Mehrwegebaum ist völlig ausgeglichen, wenn

- 1 alle Wege von Wurzel bis zu Blättern gleich lang
- 2 jeder Knoten gleich viele Indexeinträge

vollständiges Ausgleichen zu teuer, deshalb B-Baum-Kriterium:

Jede Seite außer der Wurzelseite enthält zwischen m und $2m$ Einträge.

Eigenschaften des B-Baumes

- n Datensätze in der Hauptdatei
⇒ in $\log_m(n)$ Seitenzugriffen von der Wurzel zum Blatt
 - ▶ Durch Balancierungskriterium wird Eigenschaft nahe an der vollständigen Ausgeglichenheit erreicht (1. Kriterium vollständig erfüllt, 2. Kriterium näherungsweise)
 - ▶ Kriterium garantiert 50% Speicherplatzausnutzung
 - ▶ einfache, schnelle Algorithmen zum Suchen, Einfügen und Löschen von Datensätzen (Komplexität von $O(\log_m(n))$)

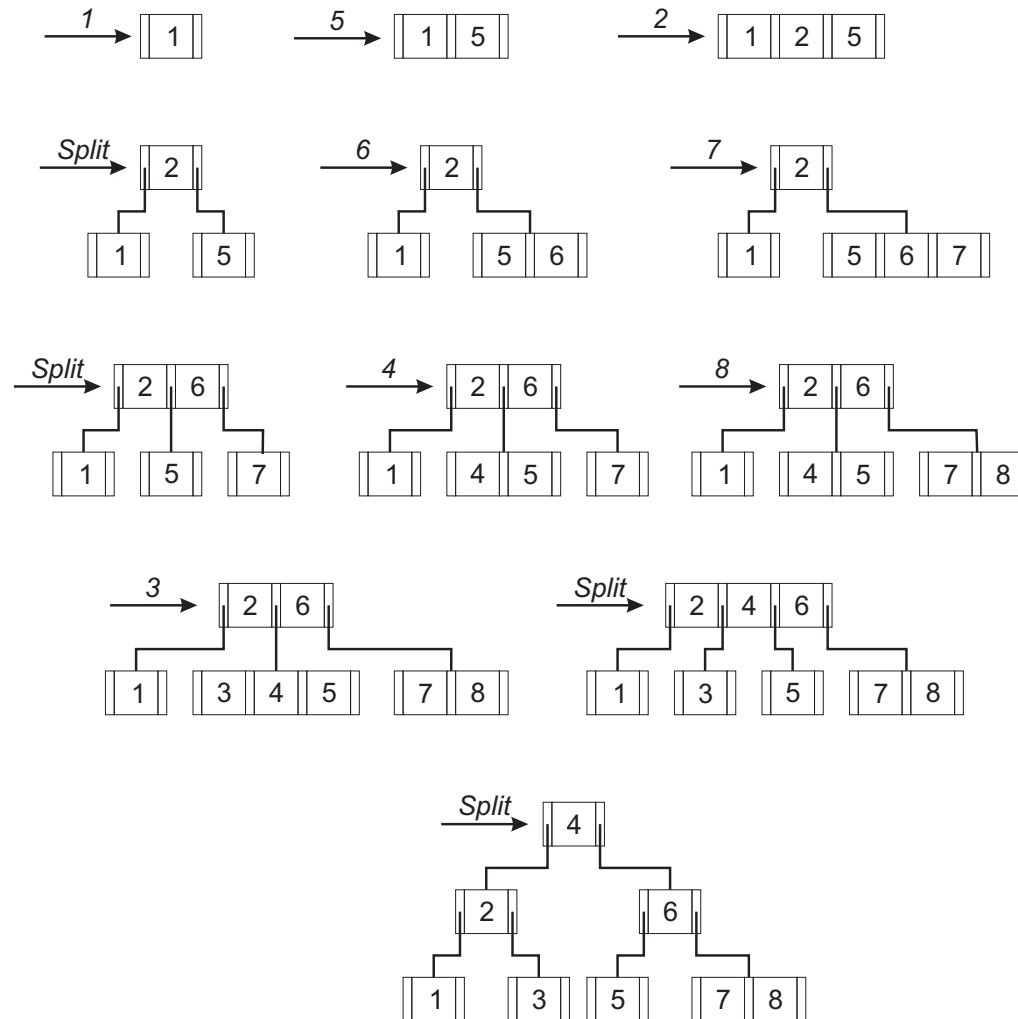
Eigenschaften des B-Baumes /2

- B-Baum als Primär- und Sekundärindex geeignet
- Datensätze direkt in die Indexseiten \Rightarrow Dateiorganisationsform
- Verweist man aus Indexseiten auf Datensätze in den Hauptseiten \Rightarrow Sekundärindex

Definition B-Baum

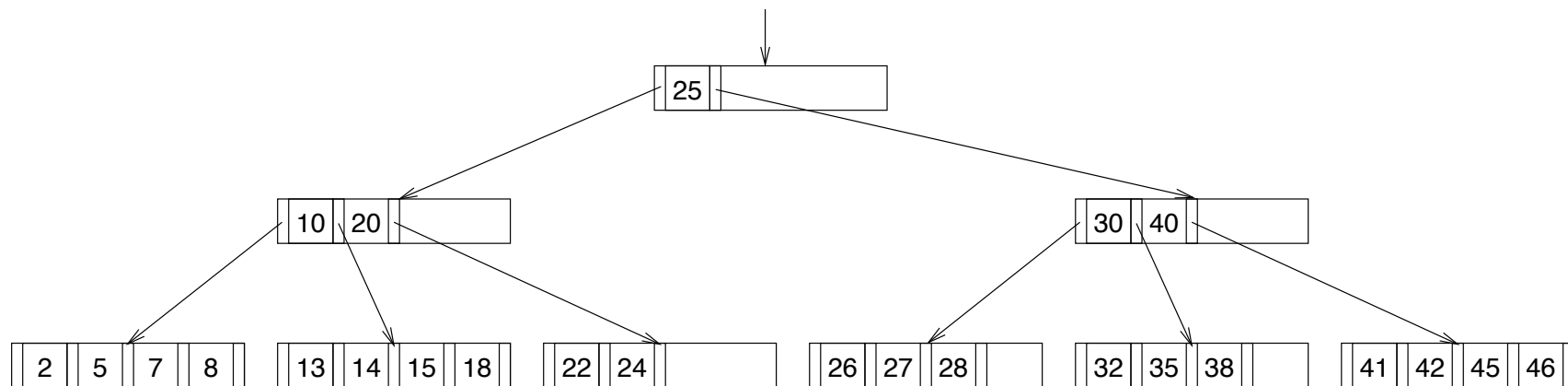
- *Ordnung* eines B-Baumes ist minimale Anzahl der Einträge auf den Indexseiten außer der Wurzelseite
- Bsp.: B-Baum der Ordnung 8 fasst auf jeder inneren Indexseite zwischen 8 und 16 Einträgen
- Def.: Ein Indexbaum ist ein B-Baum der Ordnung m , wenn er die folgenden Eigenschaften erfüllt:
 - 1 Jede Seite enthält höchstens $2m$ Elemente.
 - 2 Jede Seite, außer der Wurzelseite, enthält mindestens m Elemente.
 - 3 Jede Seite ist entweder eine Blattseite ohne Nachfolger oder hat $i + 1$ Nachfolger, falls i die Anzahl ihrer Elemente ist.
 - 4 Alle Blattseiten liegen auf der gleichen Stufe.

Einfügen in einen B-Baum: Beispiel



Suchen in B-Bäumen

- **lookup** wie in statischen Indexverfahren
- Startend auf Wurzelseite Eintrag im B-Baum ermitteln, der den gesuchten Zugriffsattributwert w überdeckt \Rightarrow Zeiger verfolgen, Seite nächster Stufe laden
- Suchen: 38, 20, 6



Einfügen in B-Bäumen

- Einfügen eines Wertes w
 - ▶ mit **lookup** entsprechende Blattseite suchen
 - ▶ passende Seite $n < 2m$ Elemente, w einsortieren
 - ▶ passende Seite $n = 2m$ Elemente, neue Seite erzeugen,
 - ★ ersten m Werte auf Originalseite
 - ★ letzten m Werte auf neue Seite
 - ★ mittleres Element auf entsprechende Indexseite nach oben
 - ▶ eventuell dieser Prozess rekursiv bis zur Wurzel

Löschen in B-Bäumen

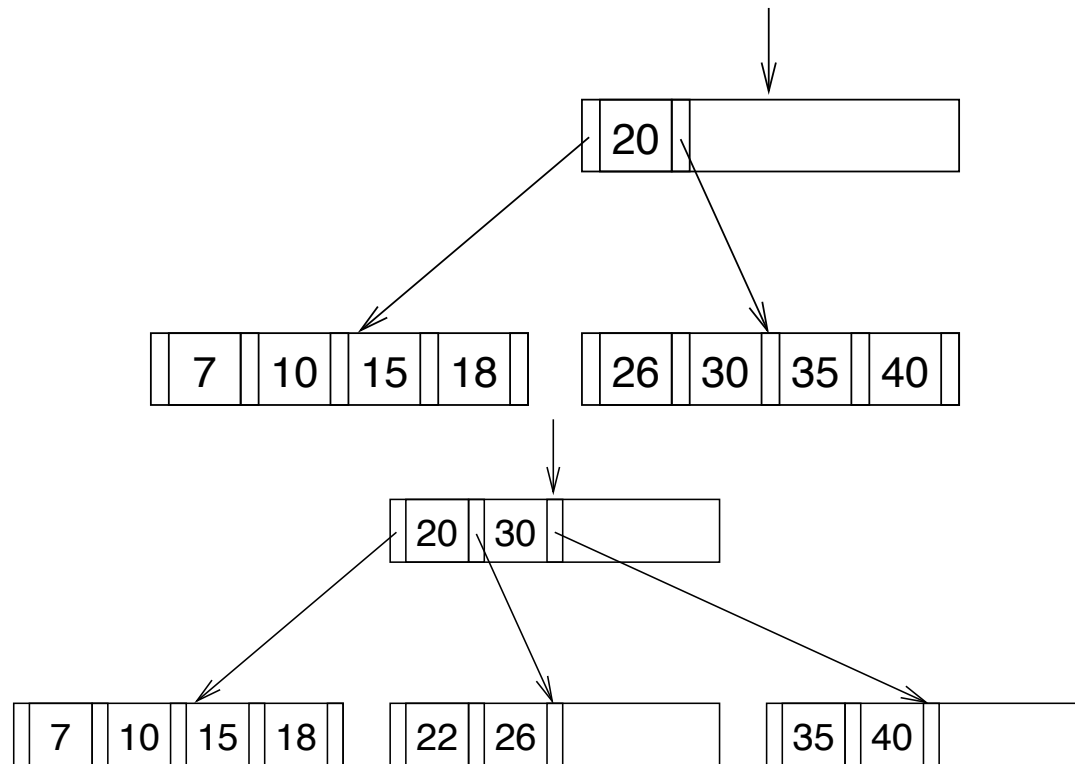
- bei weniger als m Elementen auf Seite: Unterlauf
- Löschen eines Wertes w : Bsp.: 24; 28, 38, 35
 - ▶ mit **lookup** entsprechende Seite suchen
 - ▶ w auf Blattseite gespeichert \Rightarrow Wert löschen, eventuell Unterlauf behandeln
 - ▶ w nicht auf Blattseite gespeichert \Rightarrow Wert löschen, durch lexikographisch nächstkleineres Element von einer Blattseite ersetzen, eventuell auf Blattseite Unterlauf behandeln

Löschen in B-Bäumen /2

- Unterlaufbehandlung
 - ▶ Ausgleichen mit der benachbarten Seite (benachbarte Seite n Elemente mit $n > m$)
 - ▶ oder Zusammenlegen zweier Seiten zu einer (Nachbarseite $n = m$ Elemente), das „mittlere“ Element von Indexseite darüber dazu, auf Indexseite eventuell Unterlauf behandeln

Einfügen und Löschungen im B-Baum

- Einfügen des Elementes 22; Löschen von 22



Komplexität der Operationen

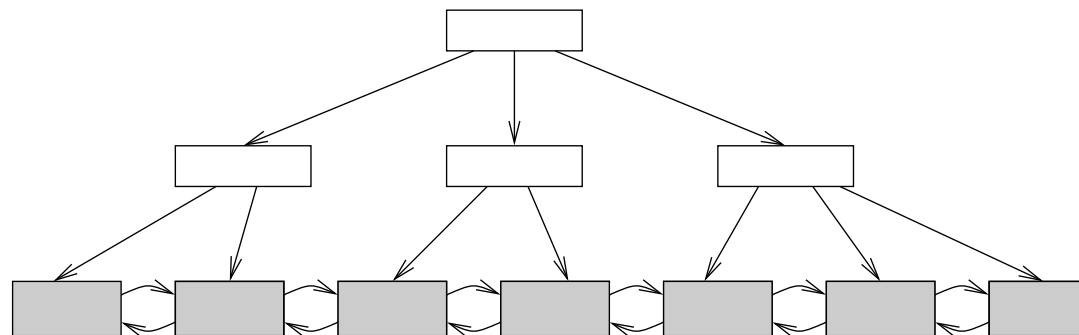
- Aufwand beim Einfügen, Suchen und Löschen im B-Baum immer $O(\log_m(n))$ Operationen
- entspricht genau der „Höhe“ des Baumes
- Beispiel: Seiten der Größe 4 KB, Zugriffsattributwert 32 Bytes, 8-Byte-Zeiger: zwischen 50 und 100 Indexeinträge pro Seite; Ordnung dieses B-Baumes 50
- 1.000.000 Datensätze: $\log_{50}(1.000.000) = 4$ Seitenzugriffe im schlechtesten Fall
- Wurzelseite jedes B-Baumes normalerweise im Puffer: drei Seitenzugriffe

Varianten

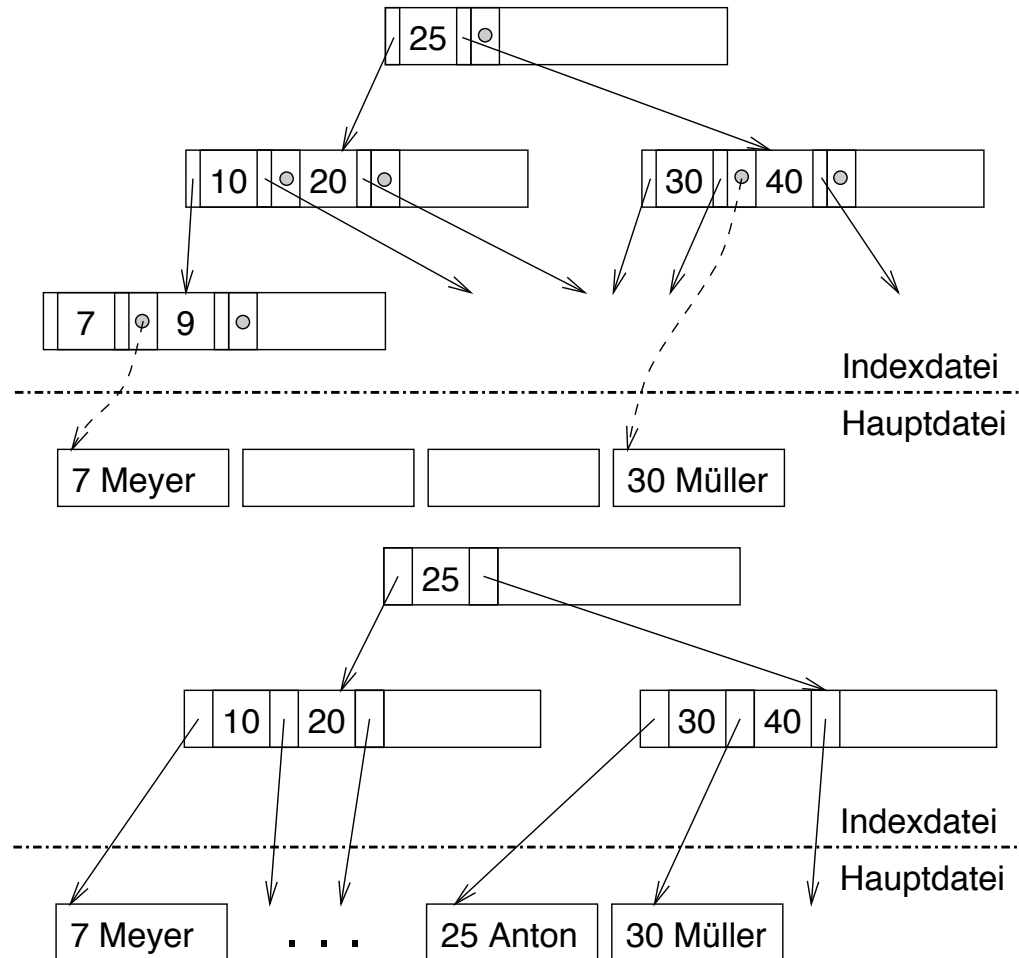
- B^+ -Bäume: Hauptdatei als letzte (Blatt-)Stufe des Baumes integrieren
- B^* -Bäume: Aufteilen von Seiten vermeiden durch „Shuffle“
- Präfix-B-Bäume: Zeichenketten als Zugriffsattributwerte, nur Präfix indexieren

B⁺-Baum

- in der Praxis am häufigsten eingesetzte Variante des B-Baumes: effizientere Änderungsoperationen, Verringerung der Baumhöhe
- integriert Datensätze der Hauptdatei auf den Blattseiten des Baumes
- in inneren Knoten nur noch Zugriffsattributwert und Zeiger auf nachfolgenden Seite der nächsten Stufe



B-Baum und B⁺-Baum im Vergleich

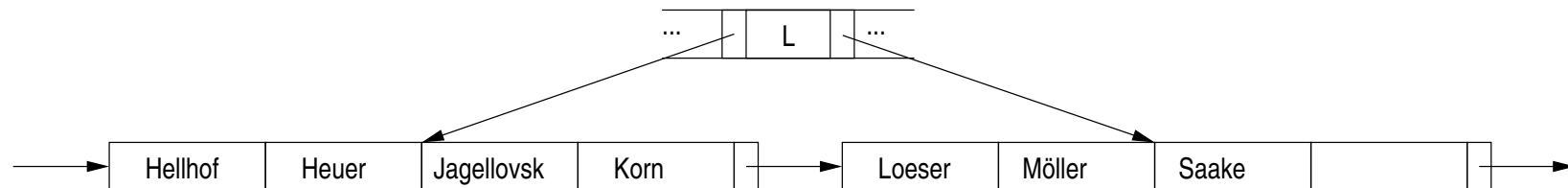


Ordnung; Operationen

- *Ordnung* für B⁺-Baum: (x, y) , x Mindestbelegung der Indexseiten, y Mindestbelegung der Datensatz-Seiten
- **delete** gegenüber B-Baum effizienter („Ausleihen“ eines Elementes von der Blattseite entfällt)
- Zugriffsattributwerte in inneren Knoten können sogar stehenbleiben
- häufig als Primärindex eingesetzt
- B⁺-Baum ist dynamische, mehrstufige, indexsequenziellen Datei

Präfix-B⁺-Baum

- B-Baum über Zeichenkettenattribut
 - ▶ lange Schlüssel in inneren Knoten \rightsquigarrow hoher Speicherplatzbedarf
 - ▶ vollständige Schlüssel eigentlich nicht notwendig, da nur „Wegweiser“
- Idee: **Verwaltung von Trennwerten** \rightsquigarrow **Präfix-B⁺-Baum**
 - ▶ in inneren Knoten nur Trennwerte, die lexikographisch zwischen den Werten liegen
 - ▶ möglichst kurze Trennwerte, z.B. kürzester eindeutiger Präfix



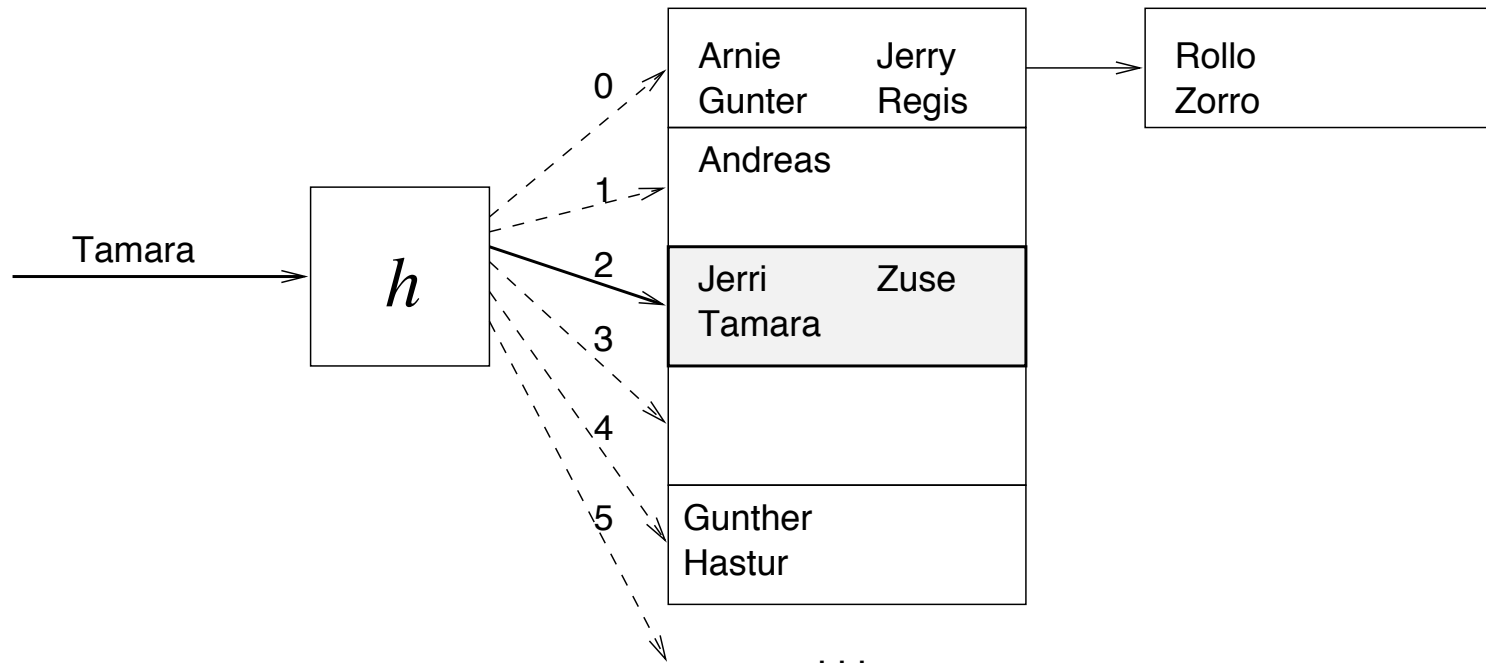
Hashverfahren

- Schlüsseltransformation und Überlaufbehandlung
- DB-Technik: Bildbereich entspricht Seiten-Adressraum
- Dynamik: dynamische Hashfunktionen oder Re-Hashen

Grundprinzipien

- Basis-Hashfunktion: $h(k) = k \bmod m$
- m möglichst Primzahl
- Überlauf-Behandlung
 - ▶ Überlaufseiten als verkettete Liste
 - ▶ lineares Sondieren
 - ▶ quadratisches Sondieren
 - ▶ doppeltes Hashen

Hashverfahren für Datenbanken



Operationen und Zeitkomplexität

- **lookup, modify, insert, delete**
- **lookup** benötigt maximal $1 + \#B(h(w))$ Seitenzugriffe
- $\#B(h(w))$ Anzahl der Seiten (inklusive der Überlaufseiten) des Buckets für Hash-Wert $h(w)$
- Untere Schranke 2 (Zugriff auf Hashverzeichnis plus Zugriff auf erste Seite)

Statisches Hashen: Probleme

- mangelnde Dynamik
- Vergrößerung des Bildbereichs erfordert komplettes Neu-Hashen
- Wahl der Hashfunktion entscheidend; Bsp.: Hash-Index aus 100 Buckets, Studenten über 6-stellige `MATRNR` (wird fortlaufend vergeben) hashen
 - ▶ ersten beiden Stellen: Datensätze auf wenigen Seiten quasi sequenziell abgespeichert
 - ▶ letzten beiden Stellen: verteilen die Datensätze gleichmäßig auf alle Seiten
- Sortiertes Ausgeben einer Relation schlecht

Typische Verfahren:

- Lineares Hashen
- Erweiterbares Hashing

Cluster-Bildung

- Speicherung von logisch zusammengehörigen Datensätzen auf Seiten
- wichtige Spezialfälle:
 - ▶ **Ballung nach Schlüsselattributen**
 - ★ Bereichsanfragen und Gruppierungen unterstützen: Datensätze in der Sortierreihenfolge zusammenhängend auf Seiten speichern \Rightarrow *index-organisierte Tabellen* oder geclusterten, dichtbesetzte Primärindexte
 - ★ **Ballung basierend auf Fremdschlüsselattributen**
Gruppen von Datensätzen, die einen Attributwert gemeinsam haben, werden auf Seiten geballt (Verbundanfragen)

Indexorganisierte Tabellen

- Tupel direkt im Index aufnehmen
- allerdings dann durch häufigen Split TID unsinnig
- weiterer Sekundärindex kann durch fehlenden TID dann aber nicht angelegt werden (Ausnahme: Oracle mit „logischen“ TIDs)
- etwa kein **unique** möglich

Cluster für Verbundanfragen

Verbundattribut: Cluster-Schlüssel

BestellNr

100

Bestelldatum	Kunde	Lieferdatum
15.04.98	Orion Enterprises	01.01.2001

Position	Teil	Anzahl	Preis
1	Aluminiumtorso	2	3145,67
2	Antenne	2	32,50
3	Overkill	1	1313,45
4	Nieten	1000	-.50

BestellNr

123

Bestelldatum	Kunde	Lieferdatum
05.10.98	Kirk Enterpr.	31.12.1999

Position	Teil	Anzahl	Preis
1	Beamer	1	13145,67
2	Energiekristall	2	32,99
3	Phaser	5	1313,45
4	Nieten	2000	−.50

Definition von Clustern

```
create cluster BESTELL_CLUSTER  
  (BestellNr int)  
  pctused 80 pctfree 5;
```

```
create table BESTELLUNG (  
  BestellNr int primary key, ...)  
  cluster BESTELL_CLUSTER (BestellNr);
```

```
create table BESTELL_POSITION (  
  Position int,  
  BestellNr int references BESTELLUNG,  
  ...  
  constraint BestellPosKey  
    primary key (Position, BestellNr)  
  )  
  cluster BESTELL_CLUSTER (BestellNr);
```

Organisation von Clustern

- *Indexierte Cluster* nutzen einen in Sortierreihenfolge aufgebauten Index (z.B. B^+ -Baum) über den Cluster-Schlüssel zum Zugriff auf die Cluster
- *Hash-Cluster* bestimmen den passenden Cluster mit Hilfe einer Hashfunktion
- Indexe für Cluster entsprechen normalen Indexen für den Cluster-Schlüssel
- statt Tupelidentifikatoren Einsatz von Cluster-Identifikatoren oder direkte Speicheradressen (bei Hashverfahren)

Indexierte Cluster

- Verwaltung der Daten in Sortierreihenfolge über Index (B-Baum)
- Speicherung von Cluster-Identifikatoren anstelle von TIDs

```
create index BESTELL_CLUSTER_IDX  
on cluster BESTELL_CLUSTER
```

Hash-Cluster

- Identifikation des betroffenen Clusters über Hashfunktion (Cluster-Schlüssel → Blockadresse)

```
create cluster BESTELL_CLUSTER (  
    BestellNr int)  
    pctused 80  
    pctfree 5  
    size 2k  
    hash is BestellNr  
    hashkeys 100000;
```

Zusammenfassung (2)

- Dateiorganisation vs. Zugriffsverfahren
- indexsequenzielle Organisation
- B- Baum und Varianten
- Hashverfahren
- Clusterbildung