

Skript zur Vorlesung
Informationssysteme
Wintersemester 2019/20

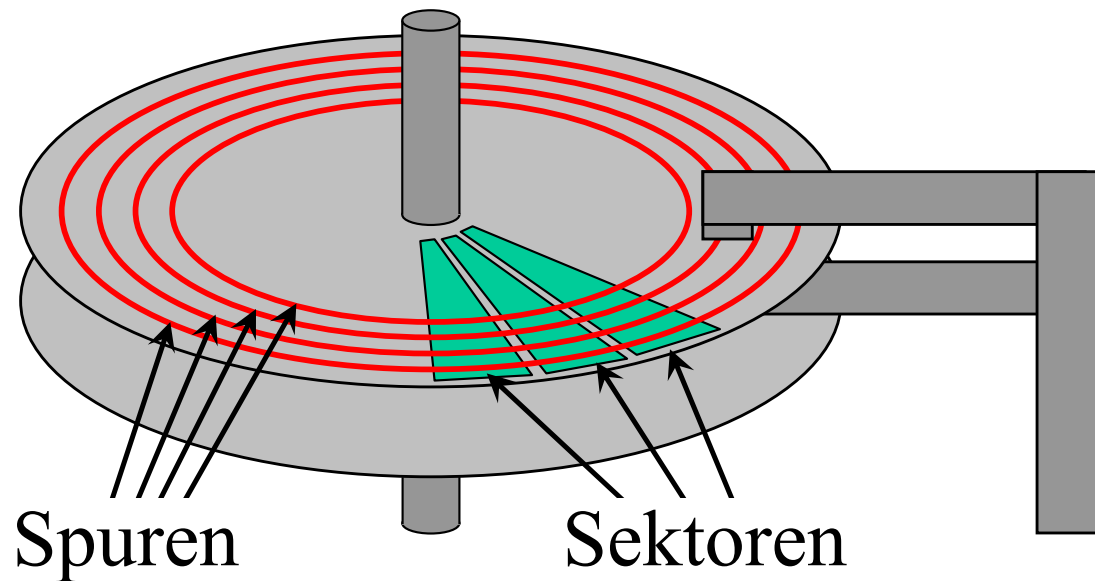
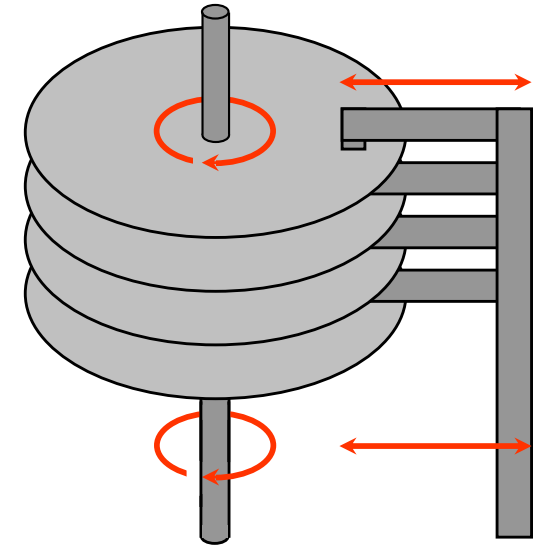
Kapitel 8: Physische Datenorganisation

Skript © 2019 Matthias Renz, CAU Kiel
(basiert auf dem Skript der LMU München)

Wiederholung (1)

Aufbau einer Festplatte

- Mehrere magnetisierbare **Platten** rotieren um eine gemeinsame Achse
- Ein Kamm mit je zwei **Schreib-/Leseköpfen** pro Platte (unten/oben) bewegt sich in radialer Richtung.



Wiederholung (2)

Intensionale Ebene vs. Extensionale Ebene

- Datenbankschema:

Name (10 Zeichen)	Vorname (8 Z.)	Jahr (4 Z.)
<input type="text"/>	<input type="text"/>	<input type="text"/>

- Ausprägung der Datenbank:

F	r	a	n	k	l	i	n			A	r	e	t	h	a			1	9	4	2
R	i	t	c	h	i	e				L	i	o	n	e	l			1	9	4	9

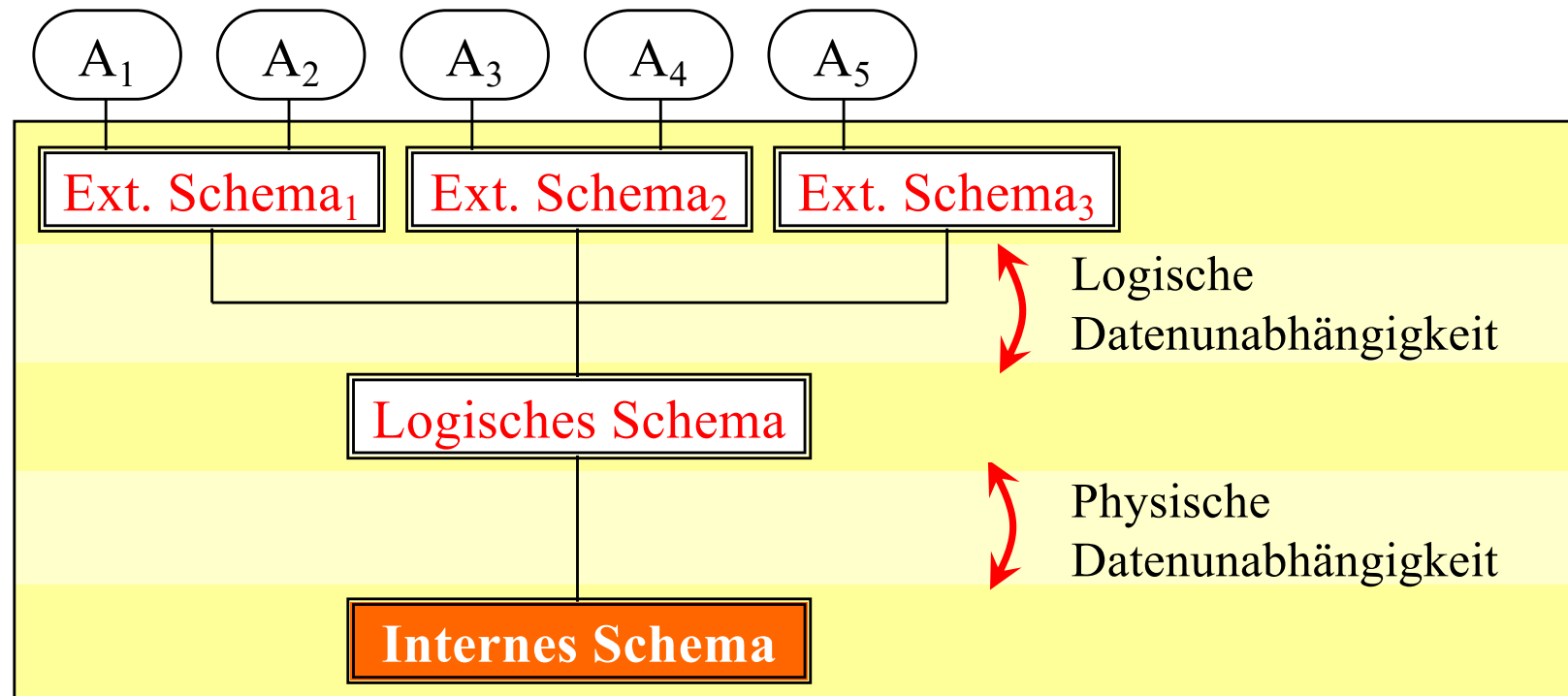
- Nicht nur DB-Zustand, sondern auch DB-Schema wird in DB gespeichert.
- Vorteil: Sicherstellung der Korrektheit der DB

Wiederholung (3)

Drei-Ebenen-Architektur zur Realisierung von

- **physischer**
- **und logischer**

Datenunabhängigkeit (nach ANSI/SPARC)



Wiederholung (4)

- Das interne Schema beschreibt die systemspezifische Realisierung der DB-Objekte (physische Speicherung), z.B.
 - Aufbau der gespeicherten Datensätze
 - Indexstrukturen wie z.B. Suchbäume
- Das interne Schema bestimmt maßgeblich das Leistungsverhalten des gesamten DBS
- Die Anwendungen sind von Änderungen des internen Schemas nicht betroffen (physische Datenunabhängigkeit)

Indexstrukturen (1)

- Um Anfragen und Operationen effizient durchführen zu können, setzt die interne Ebene des Datenbanksystems geeignete Datenstrukturen und Speicherungsverfahren (**Indexstrukturen**) ein.
- **Aufgaben**
 - **Zuordnung eines Suchschlüssels** zu denjenigen physischen Datensätzen, die diese Wertekombination besitzen, d.h. Zuordnung zu der oder den Seiten der Datei, in denen diese Datensätze gespeichert sind.
(VW, Golf, schwarz, M-ÜN 40) → (logische) Seite 37
 - **Organisation der Seiten** unter dynamischen Bedingungen.
Überlauf einer Seite → Aufteilen der Seite auf zwei Seiten

Indexstrukturen (2)

- **Aufbau**

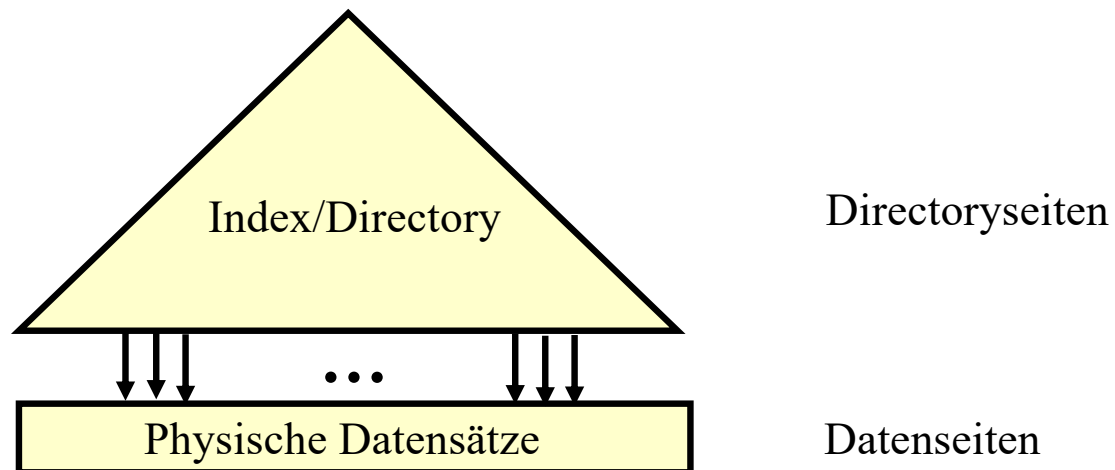
Strukturinformation zur Zuordnung von Suchschlüsseln und zur Organisation der Datei.

- **Directoryseiten:**

Seiten in denen das Directory gespeichert wird.

- **Datenseiten**

Seiten mit den eigentlichen physischen Datensätzen.



Anforderungen an Indexstrukturen (1)

- **Effizientes Suchen**

- Häufigste Operation in einem DBS: Suchanfragen.
- Insbesondere Suchoperationen müssen mit wenig Seitenzugriffen auskommen.

Beispiel: unsortierte sequentielle Datei

- Einfügen und Löschen von Datensätzen werden effizient durchgeführt.
- Suchanfragen müssen ggf. die gesamte Datei durchsuchen.
- Eine Anfrage sollte daher mit Hilfe der Indexstruktur möglichst schnell zu der Seite oder den Seiten geführt werden, auf denen sich die gesuchten Datensätze befinden.

Anforderungen an Indexstrukturen (2)

- **Dynamisches Einfügen, Löschen und Verändern von Datensätzen**
 - Der Datenbestand einer Datenbank verändert sich im Laufe der Zeit.
 - Verfahren, die zum Einfügen oder Löschen von Datensätzen eine Reorganisation der gesamten Datei erfordern, sind nicht akzeptabel.
 - Beispiel: sortierte sequentielle Datei*
 - Das Einfügen eines Datensatzes erfordert im schlechtesten Fall, dass alle Datensätze um eine Position verschoben werden müssen.
 - Folge: auf alle Seiten der Datei muss zugegriffen werden.
 - Das Einfügen, Löschen und Verändern von Datensätzen darf daher nur *lokale Änderungen* bewirken.

Anforderungen an Indexstrukturen (3)

- **Ordnungserhaltung**
 - Datensätze, die in ihrer Sortierordnung direkt aufeinander folgen, werden oft gemeinsam angefragt.
 - In der Ordnung aufeinander folgende Datensätze sollten in der gleichen Seite oder in benachbarten Seiten gespeichert werden.
- **Hohe Speicherplatzausnutzung**
 - Dateien können sehr groß werden.
 - Eine möglichst hohe Speicherplatzausnutzung ist wichtig:
 - Möglichst geringer Speicherplatzverbrauch.
 - Im Durchschnitt befinden sich mehr Datensätze in einer Seite, wodurch auch die Effizienz des Suchens steigt und die Ordnungserhaltung an Bedeutung gewinnt.

Klassen von Indexstrukturen

- **Datenorganisierende Strukturen**
Organisiere die Menge der tatsächlich auftretenden Daten
(Suchbaumverfahren)
- **Raumorganisierende Strukturen**
Organisiere den Raum, in den die Daten eingebettet sind
(dynamische Hash-Verfahren)

Anwendungsgebiete:

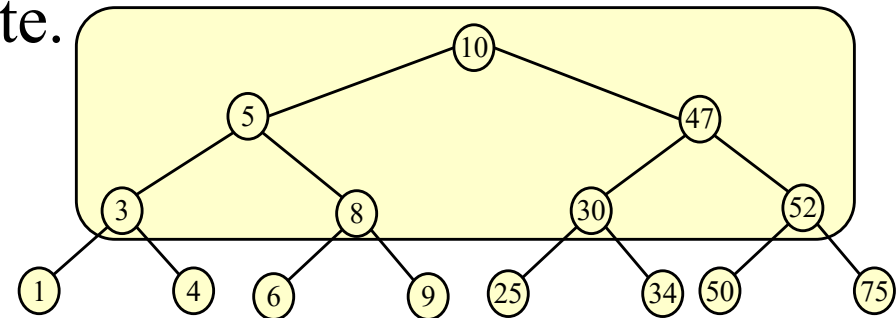
- **Primärschlüsselsuche** (B-Baum und lineares Hashing)
- **Sekundärschlüsselsuche** (invertierte Listen)

B-Baum (1)

Idee:

- Daten auf der Festplatte sind in Blöcken organisiert (z.B. 4 Kb Blöcke)
- Bei Organisation der Schlüssel mit einem binärem Suchbaum entsteht pro Knoten, der erreicht wird, ein Seitenzugriff auf der Platte.

=> sehr teuer



- Fasse mehrere Knoten zu einem zusammen, so dass ein Knoten im Baum einer Seite auf der Platte entspricht.

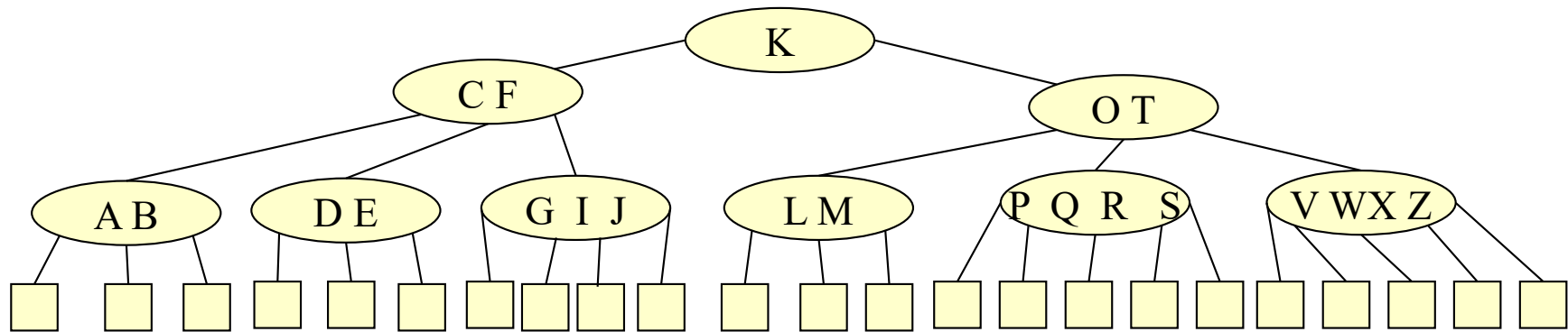
B-Baum (2)

Definition: B-Baum der Ordnung m
(Bayer und McCreight (1972))

- (1) Jeder Knoten enthält höchstens $2m$ Schlüssel.
- (2) Jeder Knoten außer der Wurzel enthält mindestens m Schlüssel.
- (3) Die Wurzel enthält mindestens einen Schlüssel.
- (4) Ein Knoten mit k Schlüsseln hat genau $k+1$ Söhne.
- (5) Alle Blätter befinden sich auf demselben Level.

B-Baum (3)

Beispiel: B-Baum der Ordnung 2



- max Höhe:
$$h \leq \left\lfloor \log_{m+1} \left(\frac{n+1}{2} \right) \right\rfloor + 1$$

- Ordnung in realen B-Bäumen: 600-900 Schlüssel pro Seite
- Effiziente Suche innerhalb Knoten \Rightarrow **binäre Suche**

Höhe eines B-Baums

Maximale Höhe h_{max} eines
B-Baumes der Ordnung m mit n Schlüsseln:

Level 1 hat $k_1 = 1$ Knoten.

Level 2 hat $k_2 \geq 2$ Knoten.

Level 3 hat $k_3 \geq 2(m + 1)$ Knoten

...

Level $h + 1$ hat $k_{h+1} \geq 2(m + 1)^{h-1}$ (äußere, leere)
Knoten.

Höhe eines B-Baums

Ein B-Baum mit n Schlüsseln teilt den **Wertebereich** der Schlüssel in $n + 1$ Intervalle. Es gilt also

$$k_{h+1} = n + 1 \geq 2(m + 1)^{h-1}, \text{ d.h. } h \leq 1 + \log_{m+1} \left(\frac{n+1}{2} \right).$$

Da die Höhe immer ganzzahlig ist, und da diese Rechnung minimalen Füllgrad annimmt, folgt:

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

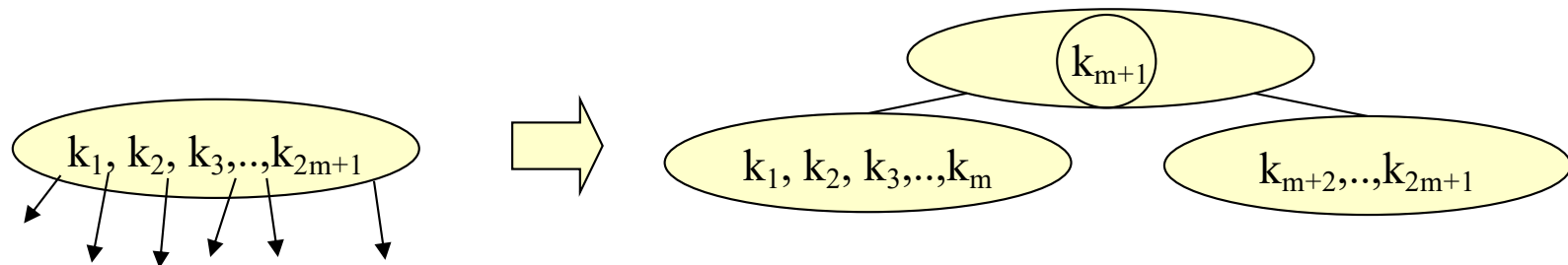
Beobachtung:

Jeder Knoten (außer der Wurzel) ist mindestens mit der Hälfte der möglichen Schlüssel gefüllt. Die Speicherplatzausnutzung beträgt also mindestens 50%!

Einfügen in B-Baum

Einfügen eines Schlüssels k :

- Suche Knoten B in den k eingeordnet werden würde. (Blattknoten bei erfolgloser Suche)
- 1. Fall: B enthält weniger als $2m$ Schlüssel
=> füge k in B ein
- 2. Fall: B enthält $2m$ Schlüssel
=> Overflow Behandlung
 - Split des Blattknotens



- Split kann sich über mehrere Ebene fortsetzen bis zur Wurzel

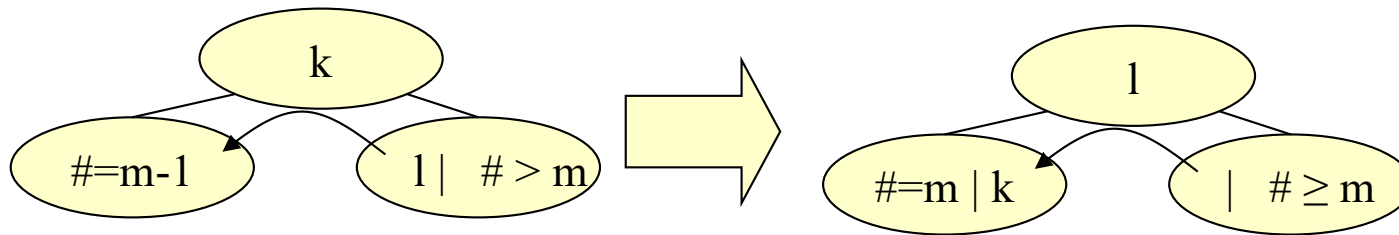
Entfernen aus B-Baum

Lösche Schlüssel k aus Baum:

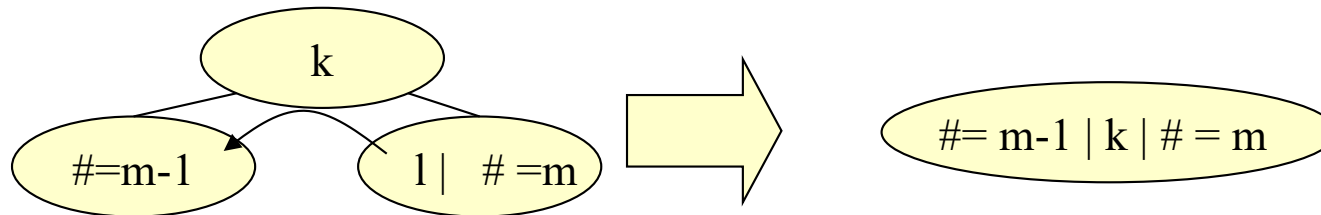
- Suche Schlüssel
- Falls Schlüssel in inneren Knoten, vertausche Schlüssel mit dem größten Schlüssel im linkem Teilbaum (=> Rückführung auf Fall mit Schlüssel in Blattknoten)
- Falls Schlüssel im Blattknoten B :
 - 1. Fall: B hat noch mehr als m Schlüssel,
=> lösche Schlüssel
 - 2. Fall: B hat genau m Schlüssel
=> Underflow

Underflow-Behandlung im B-Baum

- Betrachte Bruderknoten (immer den rechten falls vorhanden)
- 1. Fall: Bruder hat mehr als m Knoten \Rightarrow ausgleichen mit Bruder



- 2. Fall: Bruder hat genau m Knoten \Rightarrow Verschmelzen der Brüder



- Verschmelzen kann sich bis zur Wurzel hin fortsetzen.

B+-Baum (1)

- Häufig tritt in Datenbankanwendungen neben der Primärschlüsselsuche auch sequentielle Verarbeitung auf.
- **Beispiele für sequentielle Verarbeitung:**
 - *Sortiertes Auslesen aller Datensätze*, die von einer Indexstruktur organisiert werden.
 - *Unterstützung von Bereichsanfragen* der Form:
 - “Nenne mir alle Studenten, deren Nachname im Bereich [Be ... Brz] liegt.”
- → Die Indexstruktur sollte die *sequentielle Verarbeitung* unterstützen, d.h. die Verarbeitung der Datensätze in aufsteigender Reihenfolge ihrer Primärschlüssel.

B+-Baum (2)

Grundidee:

- Trennung der Indexstruktur in *Directory* und *Datei*.
- *Sequentielle Verkettung* der Daten in der Datei.

B+-Datei:

- Die Blätter des B+-Baumes heißen *Datenknoten* oder *Datenseiten*.
- Die Datenknoten enthalten alle Datensätze.
- Alle Datenknoten sind entsprechend der Ordnung auf den Primärschlüsseln *verkettet*.

B+-Directory:

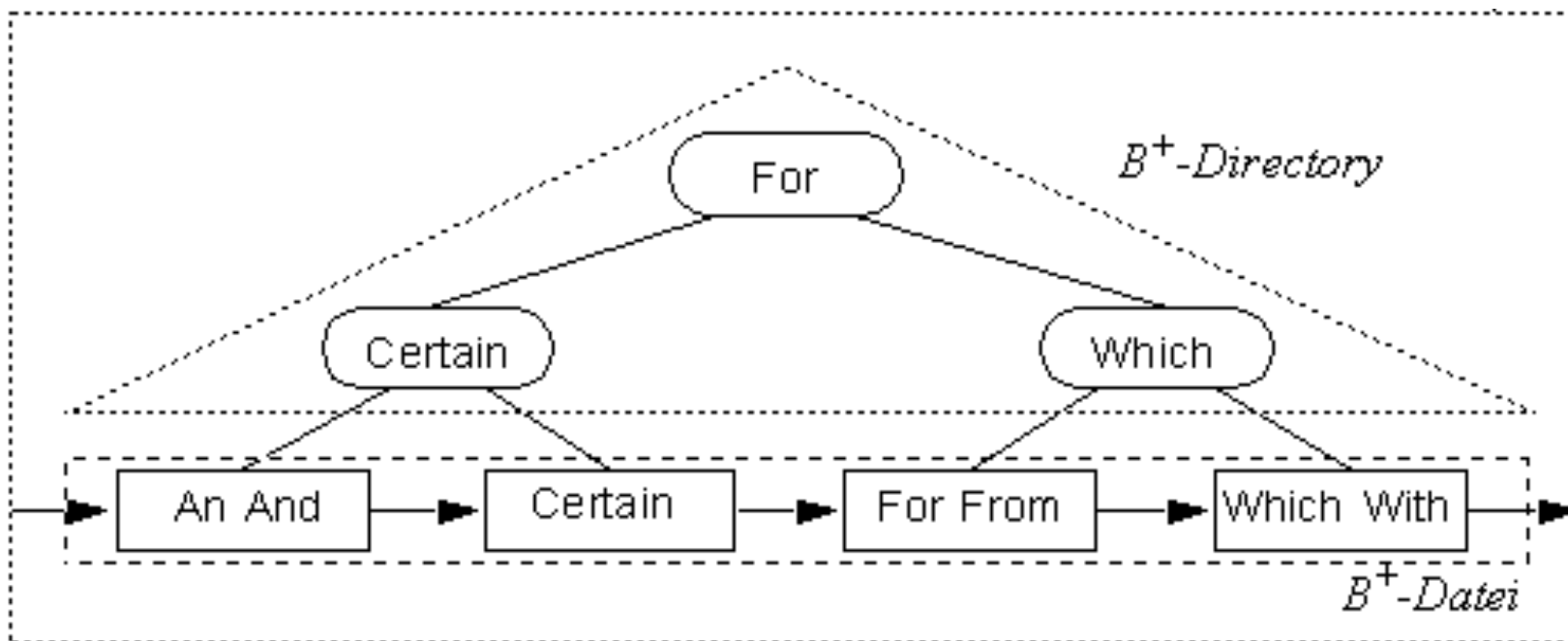
- Die inneren Knoten des B+-Baumes heißen *Directoryknoten* oder *Directoryseiten*.
- Directoryknoten enthalten nur noch *Separatoren* s .
- Für jeden Separator $s(u)$ eines Knotens u gelten folgende *Separatoreneigenschaften*:
 - $s(u) > s(v)$ für alle Directoryknoten v im linken Teilbaum von $s(u)$.
 - $s(u) < s(w)$ für alle Directoryknoten w im rechten Teilbaum von $s(u)$.
 - $s(u) > k(v')$ für alle Primärschlüssel $k(v')$ und alle Datenknoten v' im linken Teilbaum von $s(u)$.
 - $s(u) \leq k(w')$ für alle Primärschlüssel $k(w')$ und alle Datenknoten w' im rechten Teilbaum von $s(u)$.

B+-Baum (3)

Beispiel:

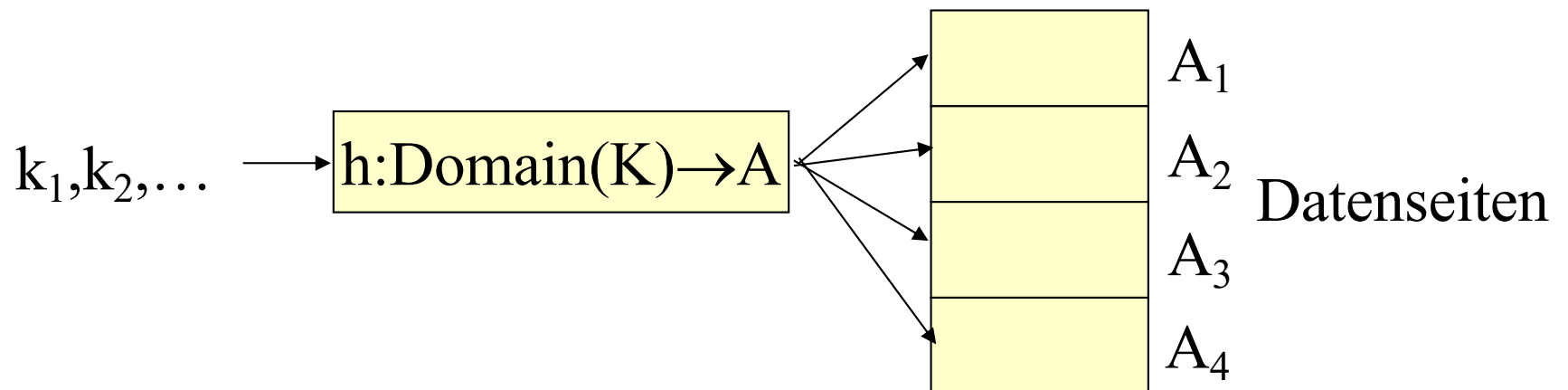
B+-Baum für die Zeichenketten:

An, And, Certain, For, From, Which, With



Hash-Verfahren

- Raumorganisierendes Verfahren
- **Idee:** Verwende Funktion, die aus den Schlüsseln K die Seitenadresse A berechnet. (Hashfunktion)
- **Vorteil:** Im besten Fall konstante Zugriffszeit auf Daten.
- **Probleme:**
 - Gleichmäßige Verteilung der Schlüssel über A
 - $|\text{Domain}(K)| \gg |A| \Rightarrow$ Kollision



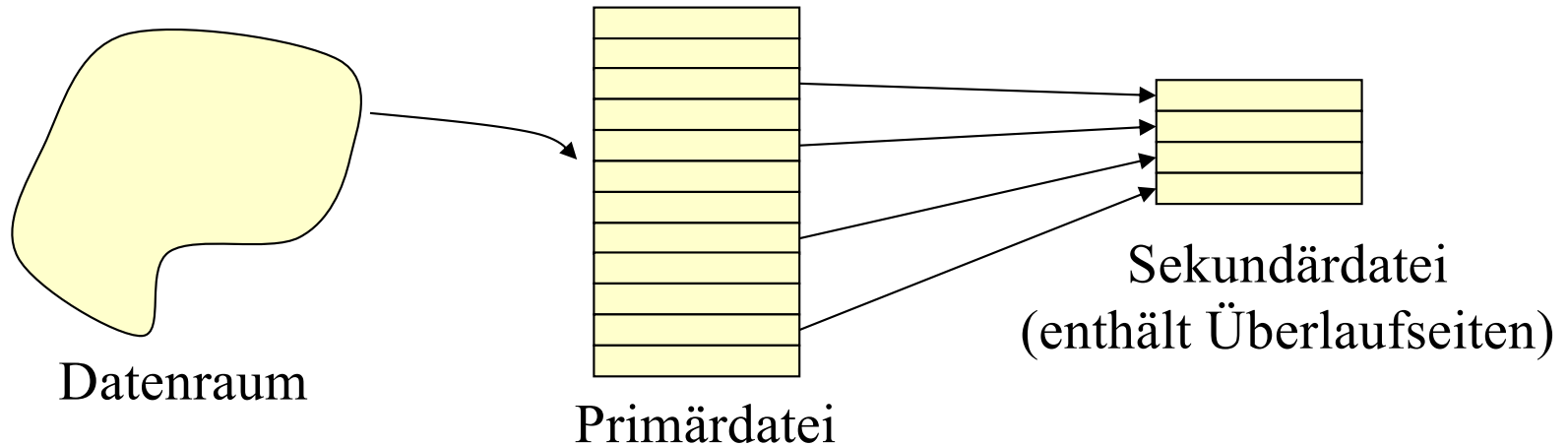
Hash-Verfahren für Sekundärspeicher

Für Sekundärspeicher sind weitere Anforderungen von Bedeutung:

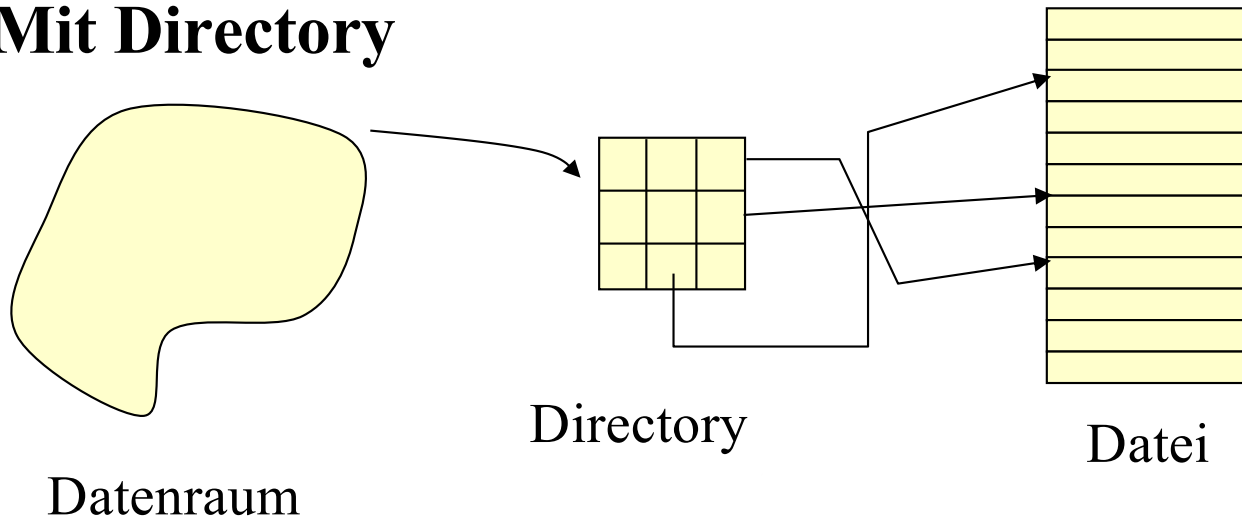
- hohe Speicherplatzausnutzung
(Datenseiten sollten über 50 % gefüllt sein)
- Gutes dynamisches Verhalten:
schnelles Einfügen, Löschen von Schlüsseln und
Datenseiten
- Gleichbleibend effiziente Suche

Klassifizierung von Hash-Verfahren

- **Ohne Directory**



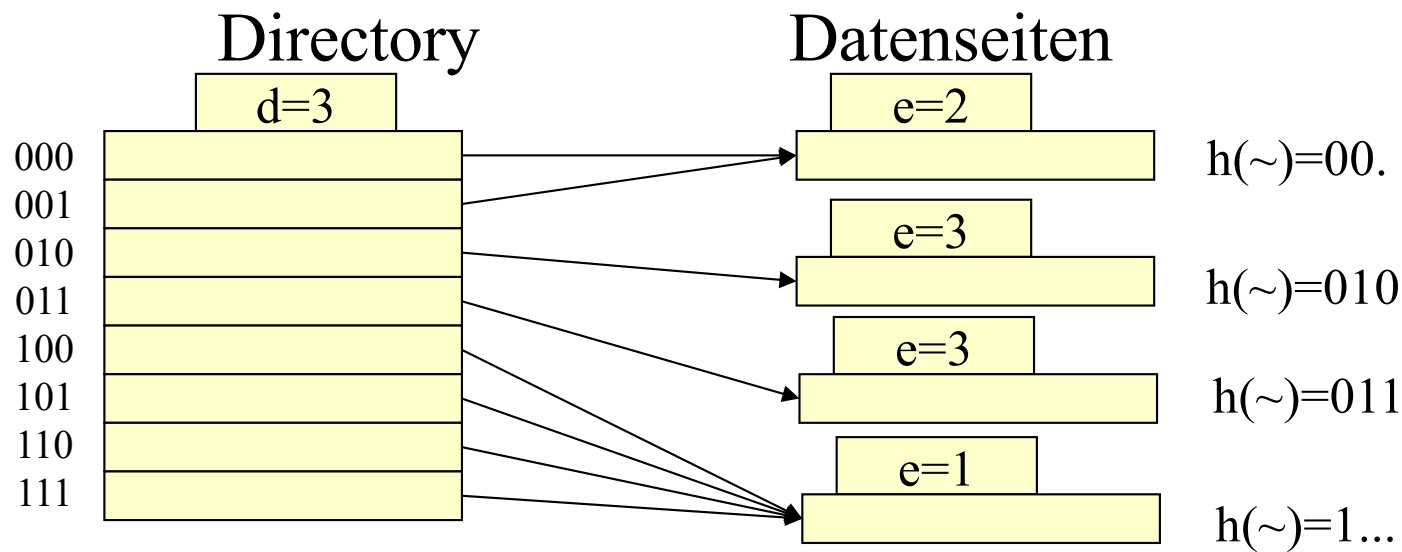
- **Mit Directory**



Hash-Verfahren mit Directory

Erweiterbares Hashing

- Hashfunktion: $h(k)$ liefert Bitfolge $(b_1, b_2, \dots, b_d, \dots)$
- Directory besteht aus eindimensionalen Array $D [0..2^d-1]$ aus Seitenadressen. d heißt Tiefe des Directory.
- Verschiedene Einträge können auf die gleiche Seite zeigen



Einfügen Erweiterbares Hashing (1)

Gegeben: Datensatz mit Schlüssel k

1. Schritt: Bestimme die ersten Bits des Pseudoschlüssels
 $h(k) = (b_1, b_2, \dots, b_d, \dots)$
2. Schritt:
Der Directoryeintrag $D[b_1, b_2, \dots, b_d]$ liefert Seitennummer.
Datensatz wird in berechnete Seite eingefügt.

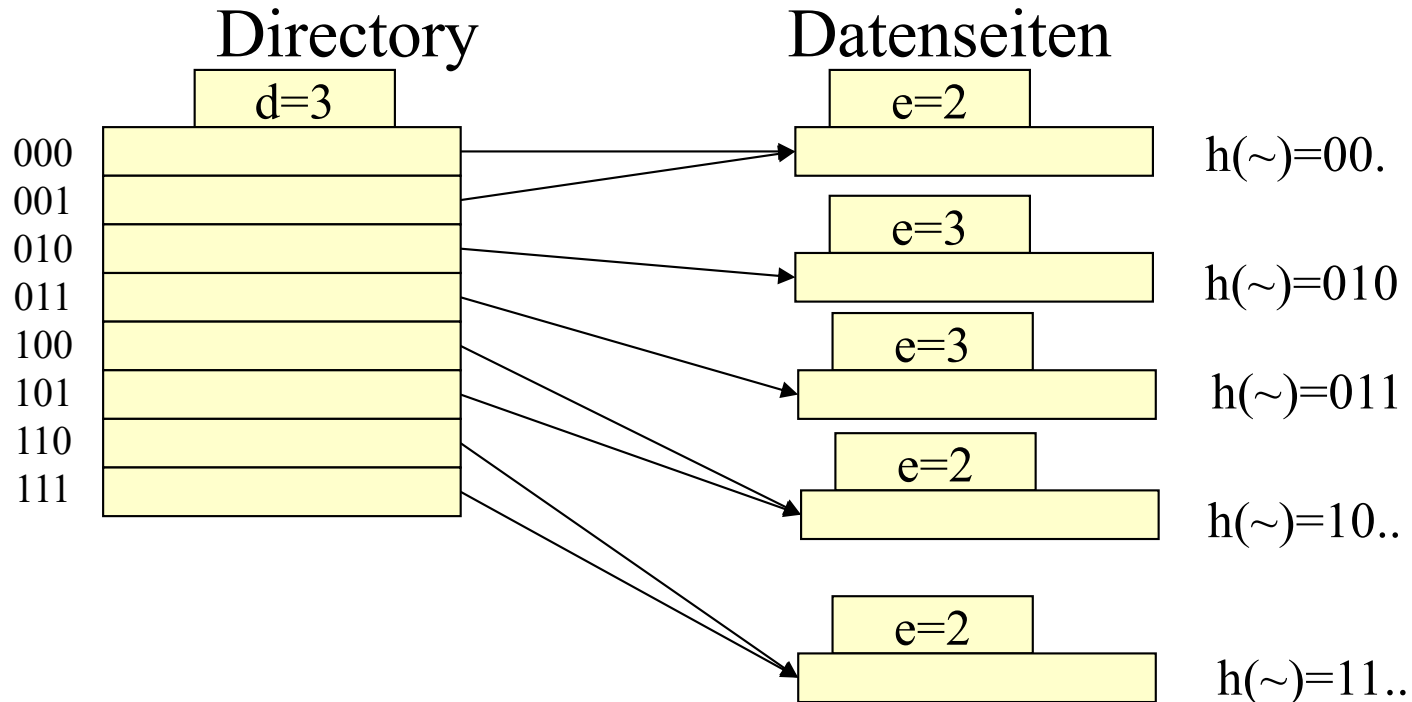
Falls Seite danach max. gefüllt:

1. Aufspalten der Datenseite.
2. Verdoppeln des Directory.

Einfügen Erweiterbares Hashing (2)

Aufspalten einer Datenseite

Aufspalten wenn Füllungsgrad einer Seite zu hoch(>90%).

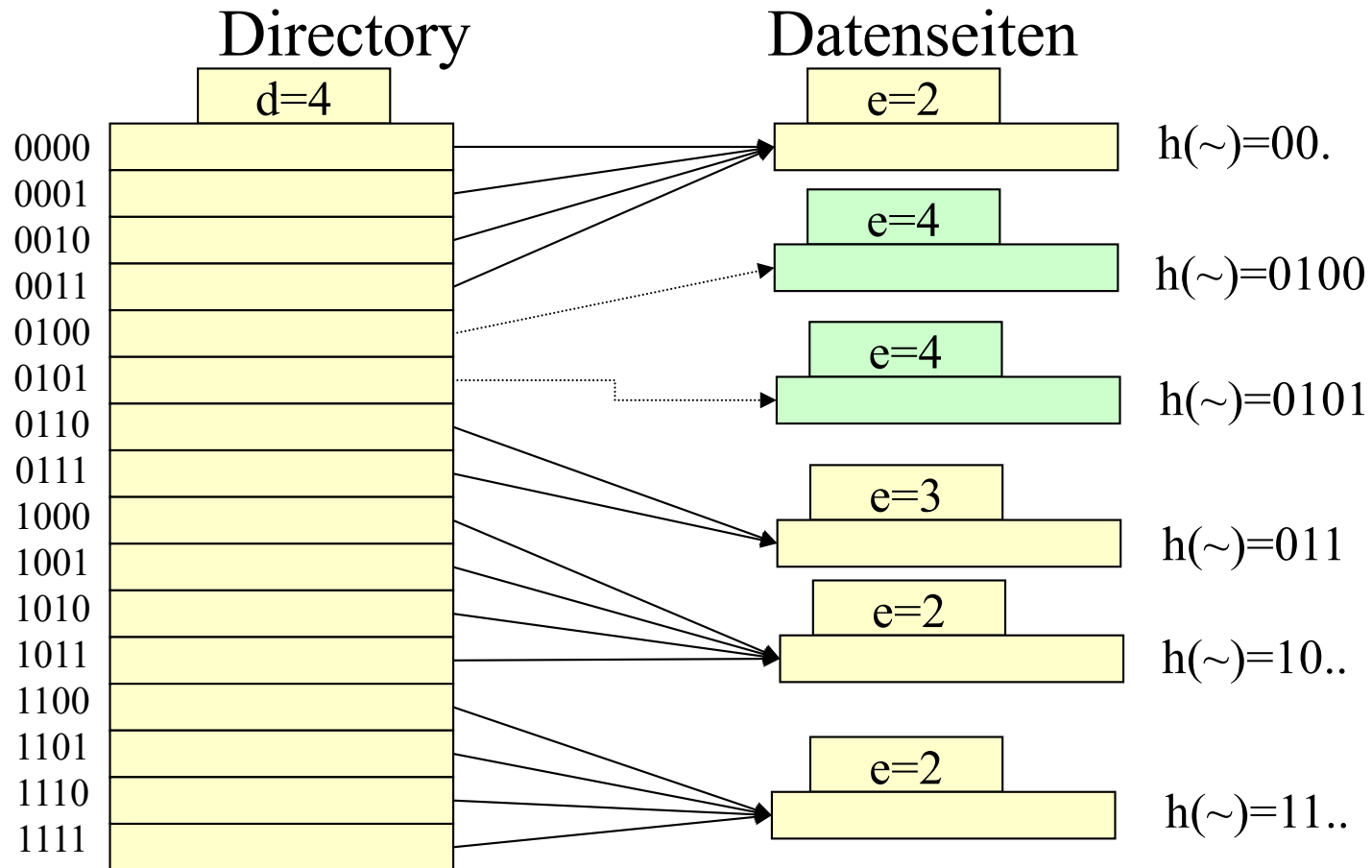


Aufgespaltene Seite

Einfügen Erweiterbares Hashing (3)

Verdopplung des Directory

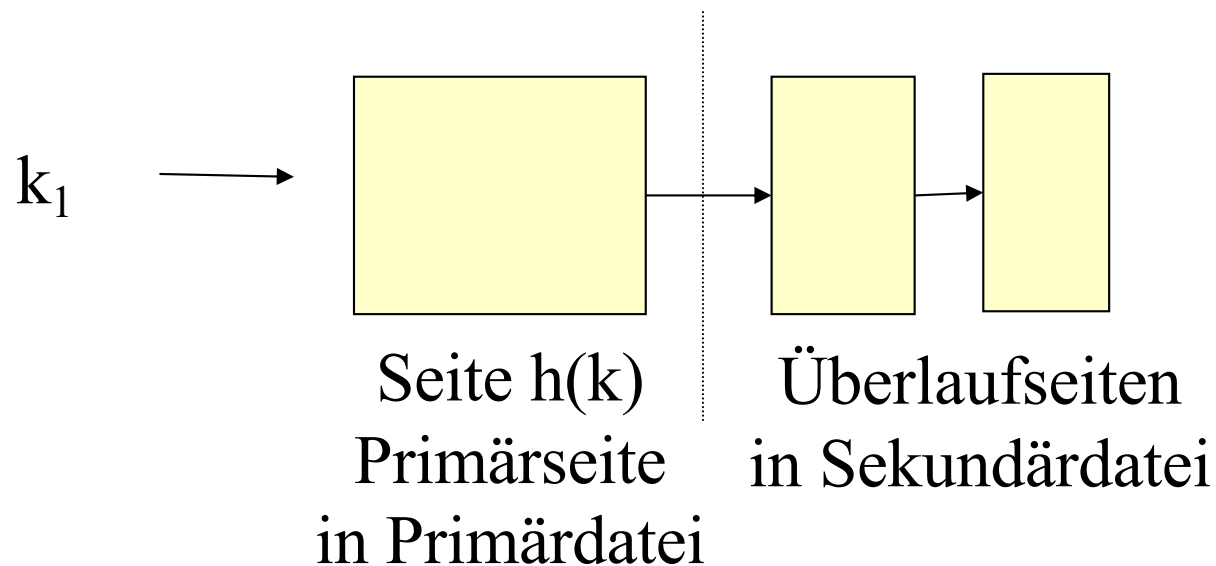
Datenseite läuft über und $d = e$.



Hashing ohne Directory

Lineares Hashing

- Hash-Funktion $h:K \rightarrow A$ liefert direkt eine Seitenadresse
- Problem: Was ist wenn Datenseite voll ist ?
- Lösung: Überlaufseiten werden angehängt. Aber bei zu vielen Überlaufseiten degeneriert Suchzeit.



Lineares Hashing (1)

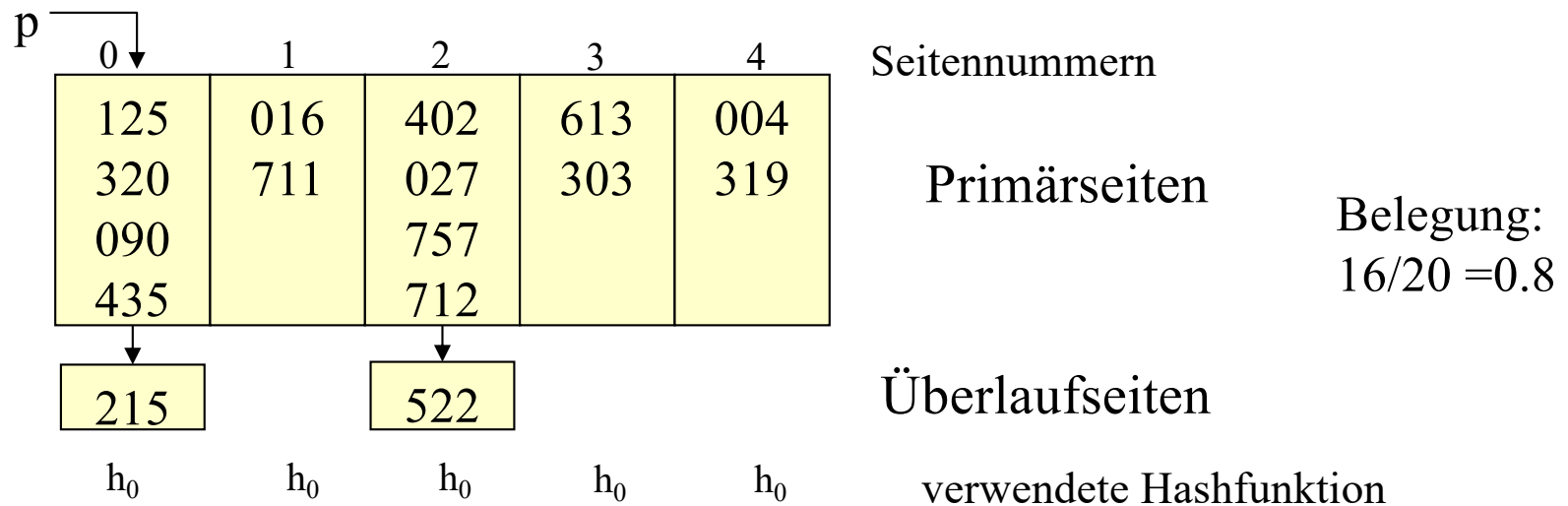
- dynamisches Wachstum der Primärdatei
- Folge von Hash-Funktionen: h_0, h_1, h_2, \dots
- Erweitern der Primärdatei um jeweils eine Seite
- feste Splitreihenfolge
- Expansionzeiger zeigt an welche Seite gesplittet wird
- Kontrollfunktion: Wann wird gesplittet ?
Belegungsfaktor übersteigt Schwellwert:

z.B.

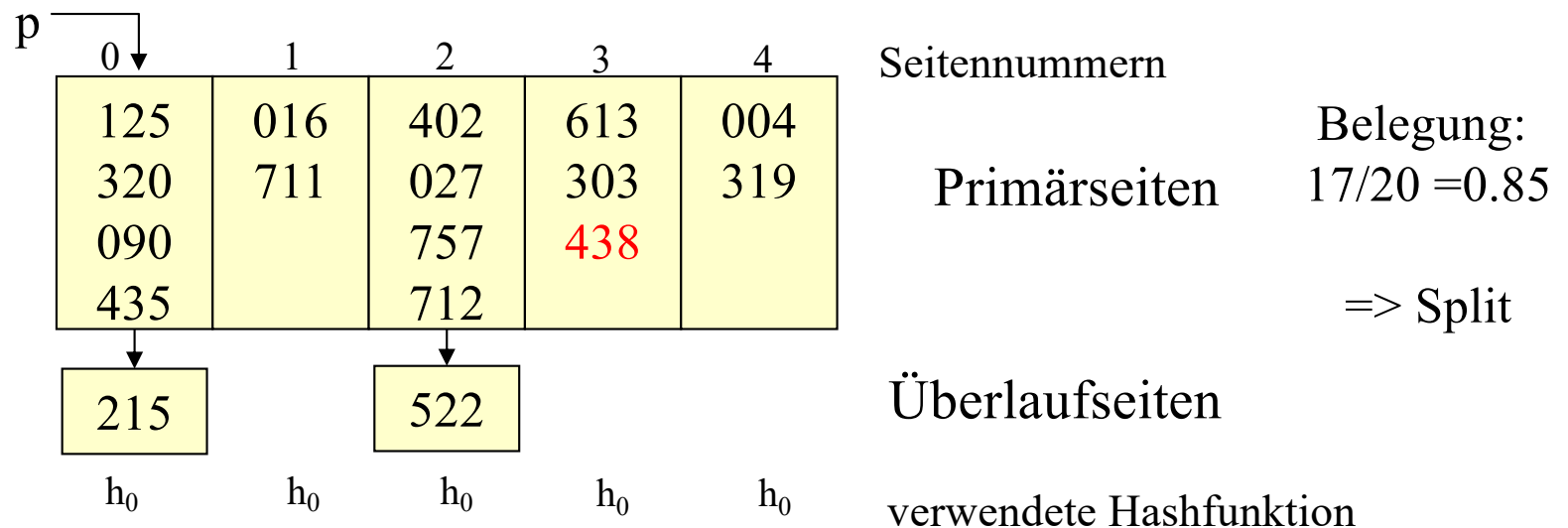
$$80\% < \frac{\# \text{abgespeicherte Datensätze}}{\# \text{mögl. Datensätze in Primärdatei}}$$

Lineares Hashing (2)

- Hashfunktionen: $h_0(k)=k \bmod(5)$, $h_1(k)=k \bmod(10)$, ...

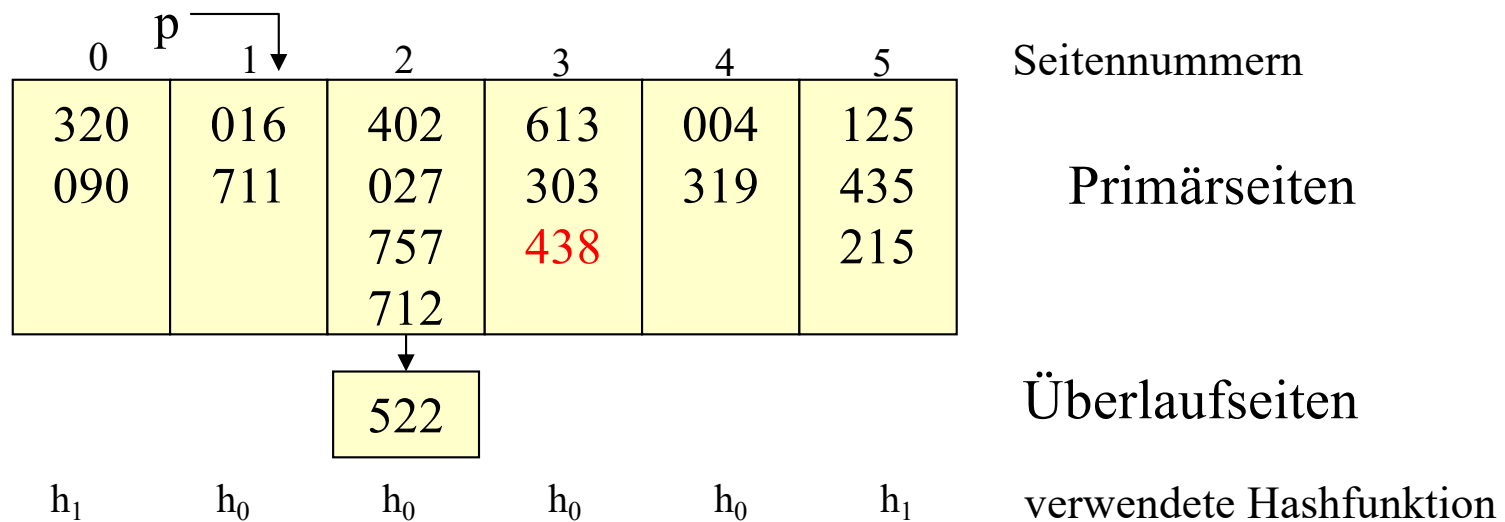


Einfügen von Schlüssel 438:



Lineares Hashing (3)

Expansion der Seite 0 auf die Seiten 0 und 5:

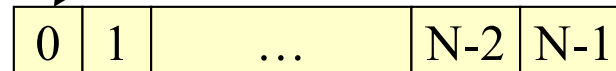


- Umspeichern aller Datensätze mit $h_1(k) = 5$ in neue Seite
- Datensätze mit $h_1(k) = 0$ bleiben

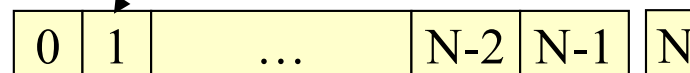
Lineares Hashing (4)

Prinzip der Expansion:

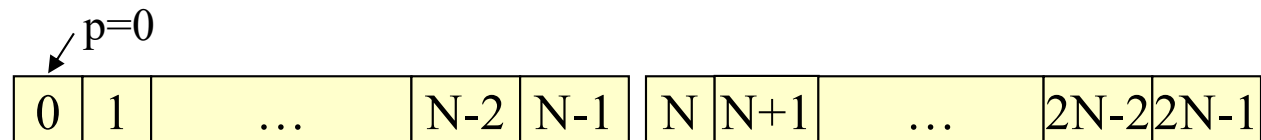
Ausgangssituation $\swarrow p=0$



nach dem ersten Split $\swarrow p=1$



nach Verdopplung der Datei $\swarrow p=0$



- Split in fester Ordnung (nicht: Split der vollen Seiten)
- trotzdem wenig Überlaufseiten
- gute Leistung für gleich verteilte Daten
- Adreßraum wächst linear

Lineares Hashing (5)

Anforderungen an die Hashfunktionen $\{h_i\}$, $i > 0$:

1.) Bereichsbedingung:

$$h_L: \text{domain}(k) \rightarrow \{0, 1, \dots, (2^L * N) - 1\}, L \geq 0$$

2.) Splitbedingung:

$$h_{L+1}(k) = h_L(k) \text{ oder}$$

$$h_{L+1}(k) = h_L(k) + 2^L * N, L \geq 0$$

L gibt an wie oft sich Datei schon vollständig verdoppelt hat.

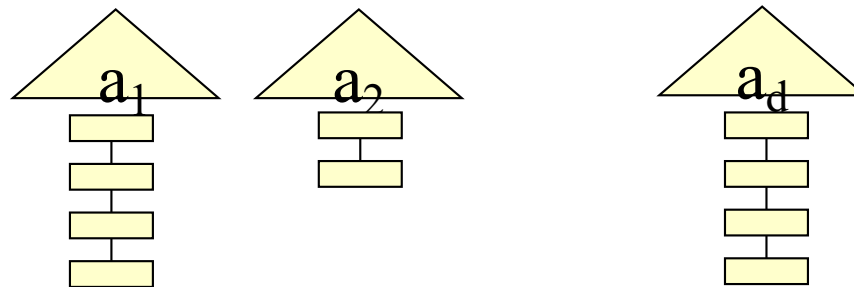
Beispiel:

$$h(k) = k \bmod (2^L * N)$$

Anfragen auf mehreren Attributen (1)

Invertierte Listen (häufigste Lösung)

- jedes relevante Attribute wird mit eindimensionalem Index verwaltet.
- Suche nach mehreren Attributen a_1, a_2, \dots, a_d
- Erstellen von Ergebnislisten mit Datensätzen d bei denen $d.a_1$ der Anfragebedingung genügt.



- Bestimmen des Ergebnis über mengentheoretischen Verknüpfung (z.B. Schnitt) der einzelnen Ergebnislisten.

Anfragen auf mehreren Attributen (2)

Eigenschaften Invertierter Listen:

- Die Antwortzeit ist nicht proportional zur Anzahl der Antworten.
- Suchzeit wächst mit Anzahl der Attribute
- genügend Effizienz bei kleinen Listen
- Sekundärindizes für nicht Primärschlüssel beeinflussen die physikalische Speicherung nicht.
- zusätzliche Sekundärindizes können das Leistungsverhalten bei DB-Updates stark negativ beeinflussen.

Index-Generierung in SQL

- Generierung eines Index:

CREATE INDEX *index-name* ON *table* (*a₁*, *a₂*, ..., *a_n*);

Ein Composite Index besteht aus mehr als einer Spalte. Die Tupel sind dann nach den Attributwerten (lexikographisch) geordnet:

Für den Vergleich der einzelnen Attribute gilt die jeweils übliche Ordnung:

$t_1 < t_2$ *gdw.*

$t_1.a_1 < t_2.a_1$ *oder* $(t_1.a_1 = t_2.a_1$ *und* $t_1.a_2 < t_2.a_2)$ *oder* ...

numerischer Vergleich für numerische Typen, lexikographischer Vergleich bei **CHAR**, Datums-Vergleich bei **DATE** usw.

- Löschen eines Index:

DROP INDEX *index-name*;

- Verändern eines Index:

ALTER INDEX *index-name* ...;

(betrifft u.a. Speicherungs-Parameter und Rebuild)

Durch Index unterstützte Anfragen

- **Exact match query:**

SELECT * FROM t WHERE $a_1 = \dots$ AND ... AND $a_n = \dots$

- **Partial match query:**

SELECT * FROM t WHERE $a_1 = \dots$ AND ... AND $a_i = \dots$

für $i < n$, d.h. wenn die exakt spezifizierten Attribute ein Präfix der indizierten Attribute sind.

Eine Spezifikation von a_{i+1} kann i.a. nicht genutzt werden, wenn a_i nicht spezifiziert ist.

- **Range query:**

SELECT * FROM t WHERE $a_1 = \dots$ AND ... AND $a_i = \dots$ AND $a_{i+1} \leq \dots$

auch z.B. für '>' oder 'BETWEEN'

- **Pointset query:**

SELECT * FROM t WHERE $a_1 = \dots$ AND ... AND $a_i = \dots$ AND $a_{i+1} \text{ IN } (7,17,77)$

auch z.B. ($a_{i+1} = \dots$ OR $a_{i+1} = \dots$ OR ...)

Durch Index unterstützte Anfragen

- **Pattern matching query**

SELECT * FROM *t* WHERE $a_1=...$ AND ... AND $a_i=...$ AND a_{i+1} LIKE ' $c_1c_2... c_k\%$ '

Problem: Anfragen wie *wort* LIKE '*%system*' werden nicht unterstützt.
 Man kann aber z.B. eine Relation aufbauen, in der alle Wörter revers gespeichert werden und dann effizient nach *revers_wort* LIKE '*metsys%*' suchen lassen.

Indexstrukturen in SQL

Index Reihenfolge:

- 1.) **create index** name_geb **on** Dozent
(Name, Geburt)
- 2.) **create index** geb_name **on** Dozent
(Geburt, Name)

Welcher Index unterstützt folgende Anfrage besser:

```
Select Name from Dozent where Name like 'H%'
```

Index „name_geb“ ist besser, da Name ein Praefix von Name, Geburt.

Zusammenfassung

- Um Anfragen und Operationen effizient durchführen zu können, setzt die interne Ebene des Datenbanksystems geeignete Datenstrukturen und Speicherungsverfahren (**Indexstrukturen**) ein.
- Primärindizes verwalten Primärschlüssel
- Sekundärindizes unterstützen zusätzliche Suchattribute oder Kombinationen von diesen.
- Ein Beispiel für eine solche Indexstruktur sind B+-Baum und dynamische Hash-Verfahren.
- Anfragen auf mehreren Attributen werden meist mit invertierten Listen realisiert.