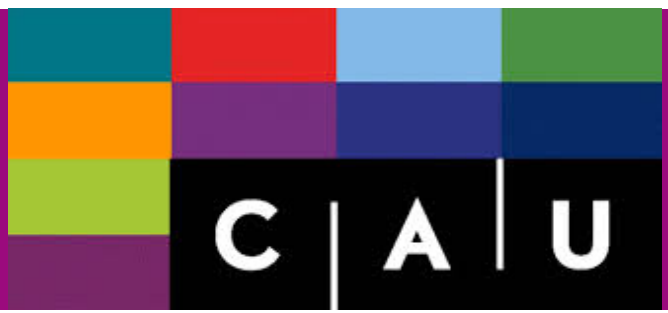
Informationssyteme[1]

Kapitel 7: Physische Datenorganisation

Wintersemester 2020/21

Prof. Dr. Peer Kröger
Institut für Informatik
Arbeitsgruppe Informationssysteme und Data Mining



[1] Die folgenden Folien basieren in großen Teilen auf Material zur Vorlesung "Datenbanksysteme", die ich in meiner Zeit an der LMU München mehrmals gehalten habe. Das Material ist von den damaligen Kollegen maßgeblich (mit-)gestaltet worden, insbes. von Prof. Dr. C. Böhm.

Übersicht

1. Einleitung
2. Datenorganisierende Indexstrukturen
3. Raumorganisierende Indexstrukturen
4. Verschiedenes (Potporri)

Agenda

1. Einleitung

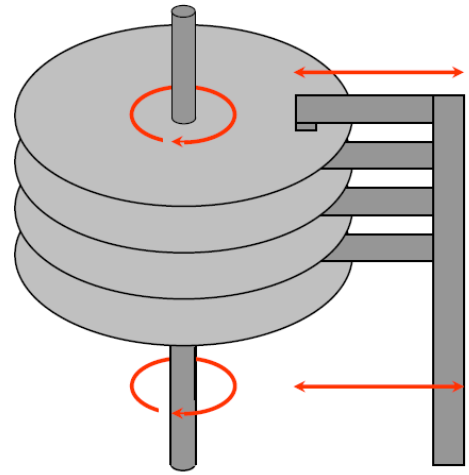
2. Datenorganisierende Indexstrukturen

3. Raumorganisierende Indexstrukturen

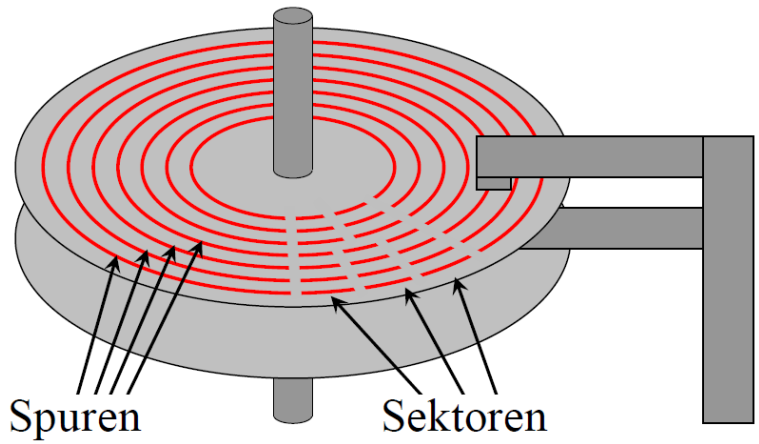
4. Verschiedenes (Potporri)

Wiederholung: 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



- Einteilung der Plattenoberflächen (oben/unten) in Spuren und Sektoren:



Wiederholung: Physische Datenunabhängigkeit

- Trennung zwischen konzeptioneller Ebene (logisches Schema) und physischer Ebene (internes Schema) zur Implementierung von physischer Datenunabhängigkeit
- Das interne Schema beschreibt die systemspezifische Realisierung der DB-Objekte (physische Speicherung), z.B.
  - Aufbau der gespeicherten Datensätze und Art der Speicherung (z.B. zeilenorientiert vs. spaltenorientiert)
  - 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)
- Wir schauen uns im folgenden genauer Indexstrukturen an

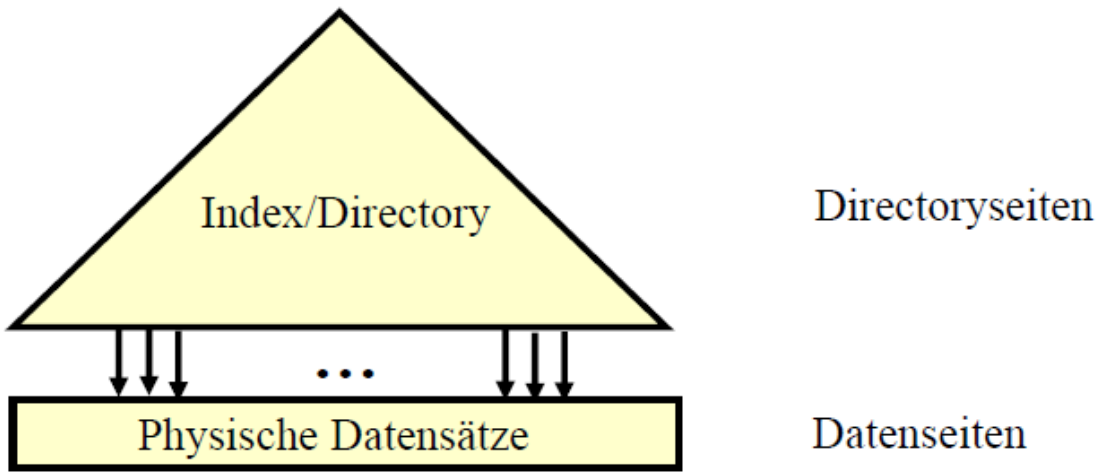
Indexstrukturen

- 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
  - Organisation der Seiten unter dynamischen Bedingungen (also z.B. Überlauf einer Seite führt zum Aufteilen der Seite auf zwei Seiten ...)
- Das interne Schema bestimmt maßgeblich das Leistungsverhalten des gesamten DBS
- Die Anwendungen sind von Änderungen des internen Schemas nicht betroffen (physische Datenunabhängigkeit)

Indexstrukturen

- 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



Indexstrukturen: Anforderungen

- Effizientes Suchen
  - Häufigste Operation in einem DBS: Suchanfragen
  - Insbesondere Suchoperationen müssen mit wenig Seitenzugriffen auskommen
  - 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
- 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
  - Das Einfügen, Löschen und Verändern von Datensätzen darf daher nur lokale Änderungen bewirken

Indexstrukturen: Anforderungen

Problem: Trade-Off zwischen Suchen vs. Einfügen/Löschen

- Unsortierte sequentielle Datei
  - Einfügen und Löschen von Datensätzen werden effizient durchgeführt
  - Suchanfragen müssen ggf. die gesamte Datei durchsuchen
- Sortierte sequentielle Datei
  - Binäre Suche (schneller als lineare Suche)
  - Das Einfügen/Löschen eines Datensatzes erfordert im schlechtesten Fall, dass alle Datensätze um eine Position verschoben werden müssen
  - Folge: auf alle Seiten der Datei muss zugegriffen werden

Indexstrukturen: Anforderungen

- 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

Indexstrukturen: Arten

- 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)

Agenda

1. Einleitung

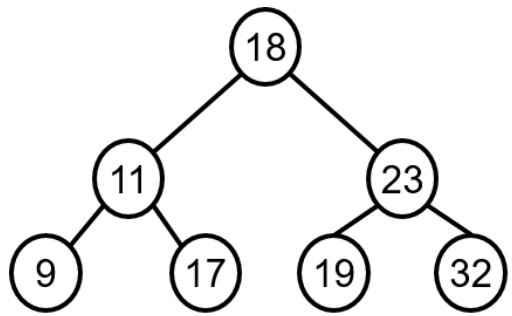
2. Datenorganisierende Indexstrukturen

3. Raumorganisierende Indexstrukturen

4. Verschiedenes (Potporri)

Suchbäume

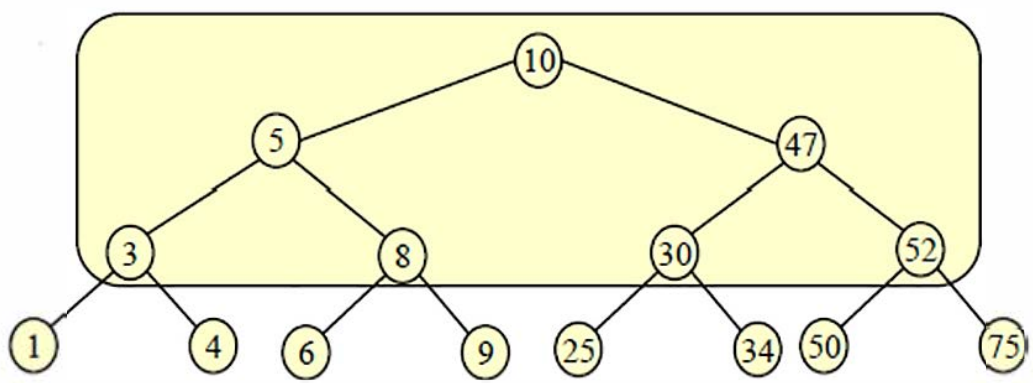
- Balancierte (binäre) Suchbäume (z.B. AVL-Bäume) erlauben effizientes Suchen sowie Einfügen und Löschen (jeweils in $O(\log n)$ für n Suchschlüssel) im Hauptspeicher
- Suchbaumeigenschaft:
  Wenn v_l im linken Teilbaum von v_k liegt und v_r im rechten Teilbaum von v_k liegt, dann $v_l \leq v_k \leq v_r$
- Balanciertheit: Teilbäume aller Knoten "ungefähr" gleich hoch (damit Höhe in $O(\log n)$)



- Operationen sind auf einen Pfad von der Wurzel bis maximal zu einem Blatt beschränkt (Länge des Such-Pfades $\leq$ Höhe des Baumes)

B-Baum: Idee

- Daten auf der Festplatte sind in Blöcken organisiert (z.B. 4 KB pro Block)
- Problem: Bei Organisation der Schlüssel mit einem binärem Suchbaum (ein Knoten pro Seite) entsteht pro Knoten, der erreicht wird, ein Seitenzugriff auf der Platte (sehr teuer!)
- Lösung: Weiterhin ein Knoten pro Seite aber fasse mehrere Knoten zu einem zusammen, so dass ein Knoten im Baum (vom Speicherbedarf) einer Seite auf der Platte entspricht



B-Baum: Definition

B-Baum der Ordnung m

Ein B-Baum der Ordnung m ist ein Suchbaum mit folgenden Eigenschaften:

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

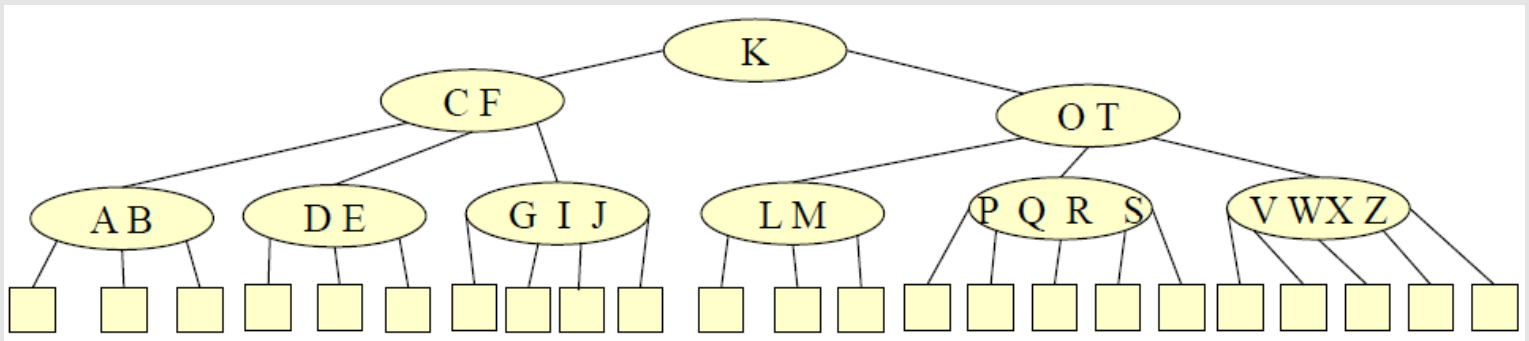
- Es gilt: ein B-Baum mit n Schlüsseln hat eine max. Höhe von

$$\lfloor \log_{m+1}(\frac{n+1}{2}) \rfloor + 1 \quad \text{also} \quad O(\log n)$$

B-Baum

Beispiel: B-Baum der Ordnung 2

Graphisch:

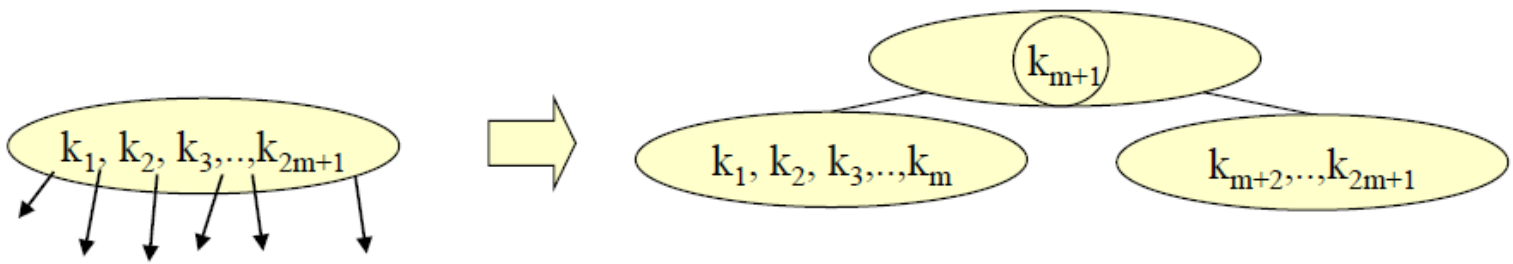


- Ordnung in realen B-Bämen: 600-900 Schlüssel pro Seite
- Schlüssel in einem Knoten sortiert, d.h. binäre Suche innerhalb eines Knoten möglich

B-Baum: Einfügen

Einfügen eines Schlüssels k:

- Suche Knoten B in den k eingeordnet werden würde (endet in 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 durch Split von B



- Split kann sich über mehrere Ebene fortsetzen bis zur Wurzel

B-Baum: Löschen

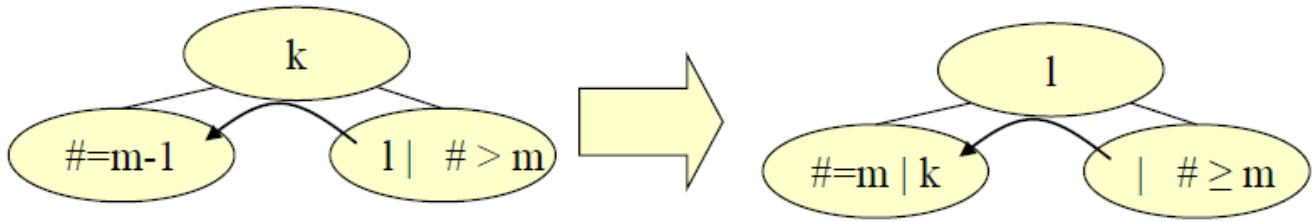
Löschen eines Schlüssels k aus dem Baum:

- Suche Schlüssel
- Falls Schlüssel in innerem Knoten, vertausche Schlüssel mit dem größten Schlüssel im linkem Teilbaum (Rückführung auf Fall mit Schlüssel in Blattknoten)
- Falls Schlüssel im Blattknoten B: "Overflow" Behandlung durch Split von B (wie beim Einfügen)
  - 1. Fall: B hat noch mehr als m Schlüssel: lösche Schlüssel
  - 2. Fall: B hat genau m Schlüssel: "Underflow"

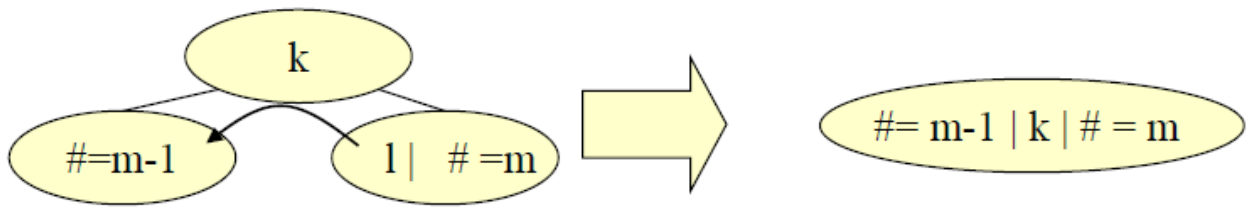
B-Baum: Löschen

Underflow-Behandlung:

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



  - 2. Fall: Bruder hat genau *m* Knoten: verschmelzen der Brüder



  - Verschmelzen kann sich über mehrere Ebenen fortsetzen bis zur Wurzel

B+-Baum

- 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

- Grundidee:
  - Trennung der Indexstruktur in Directory (Metainformationen für die Suche) und Datei (eigentliche Daten)
  - 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+-Baum

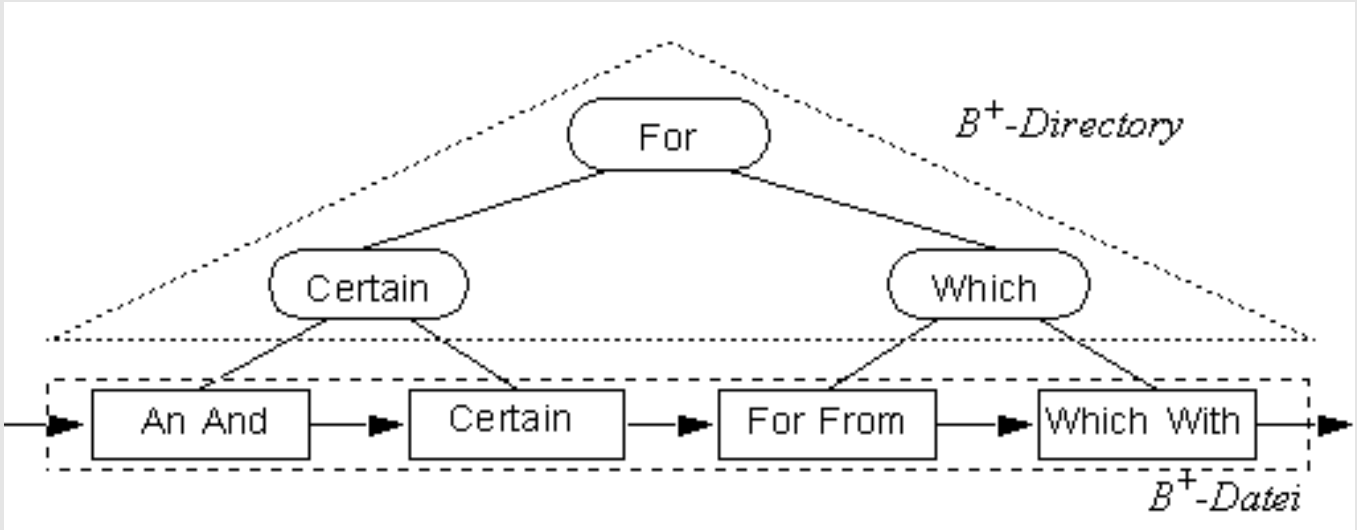
- B+-Directory:
  - Die inneren Knoten des B+-Baumes heißen Directoryknoten oder Directoryseiten
  - Directoryknoten enthalten nur noch Separatoren s
  - Für jeden Separator $s(u)$ eines Knotens u gelten folgende Separatoreneigenschaften
    - $s(u) > s(v)$ für alle Directoryknoten v im linken Teilbaum 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

Beispiel: B+-Baum

B+-Baum für die Zeichenketten:

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



Agenda

1. Einleitung

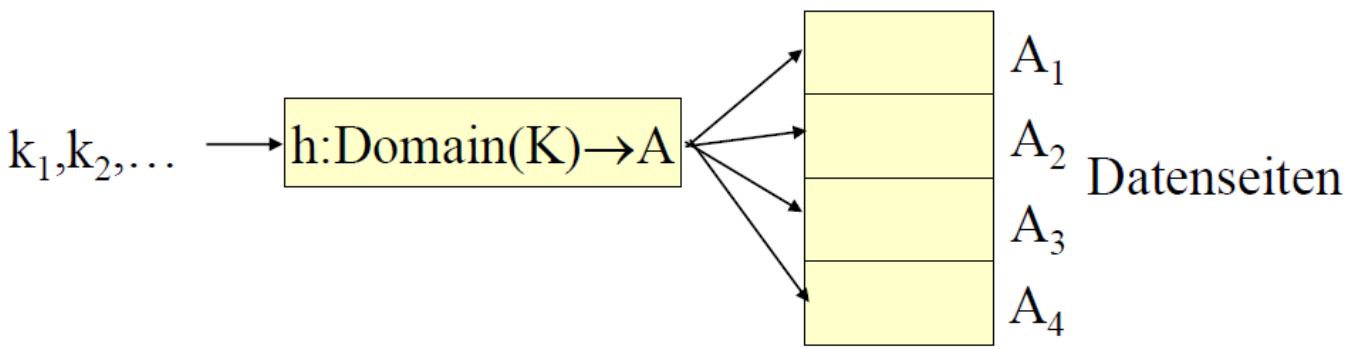
2. Datenorganisierende Indexstrukturen

3. Raumorganisierende Indexstrukturen

4. Verschiedenes (Potporri)

Hash-Verfahren

- Hash-Verfahren sind raumorganisierende Verfahren
- Idee: Verwende Funktion, die aus den Schlüsseln K die Seitenadresse A berechnet (Hashfunktion)
- Vorteil: Im besten Fall konstante Zugriffszeit auf Daten (vergleiche: Arrays im RAM)
- Probleme / Herausforderungen
  - Gleichmäßige Verteilung der Schlüssel über A
  - $|Domain(K)| >> |A|$, d.h. es wird Kollisionen geben

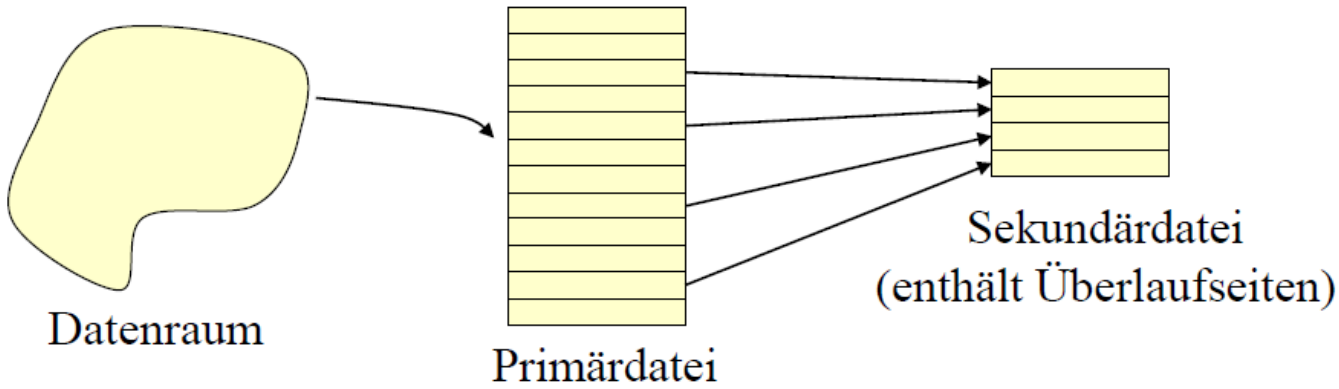


Hash-Verfahren für Sekundärspeicher

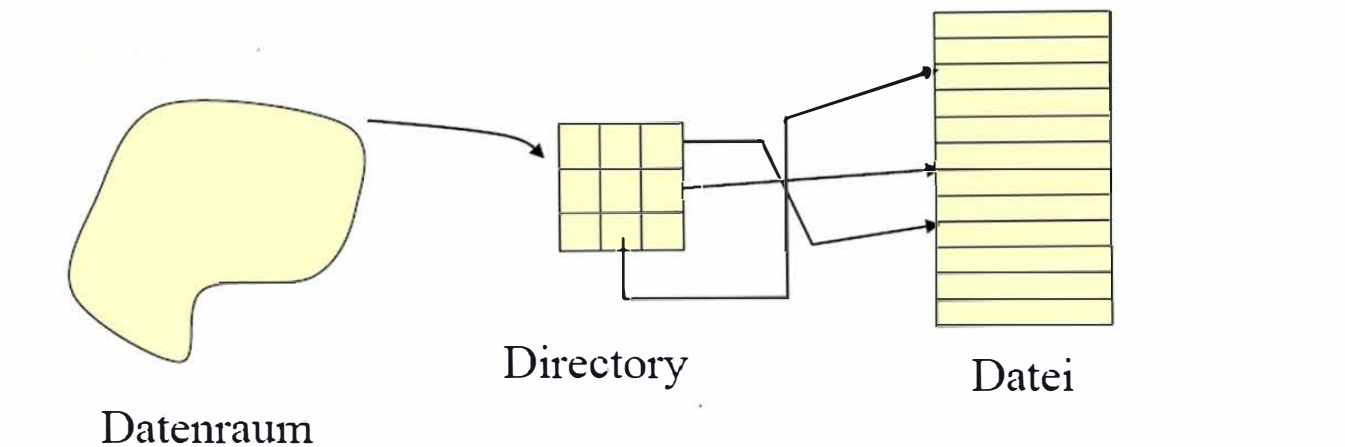
- Für Sekundärspeicher sind weitere Anforderungen von Bedeutung:
  - Hohe Speicherplatzausnutzung (Datenseiten sollten über 50 % gefüllt sein)
  - Gutes dynamisches Verhalten: schnelles Einfügen, Löschen von Schlüsseln und Datenseiten
  - Gleichbleibend effiziente Suche

Arten von Hash-Verfahren für Sekundärspeicher

- Ohne Directory:



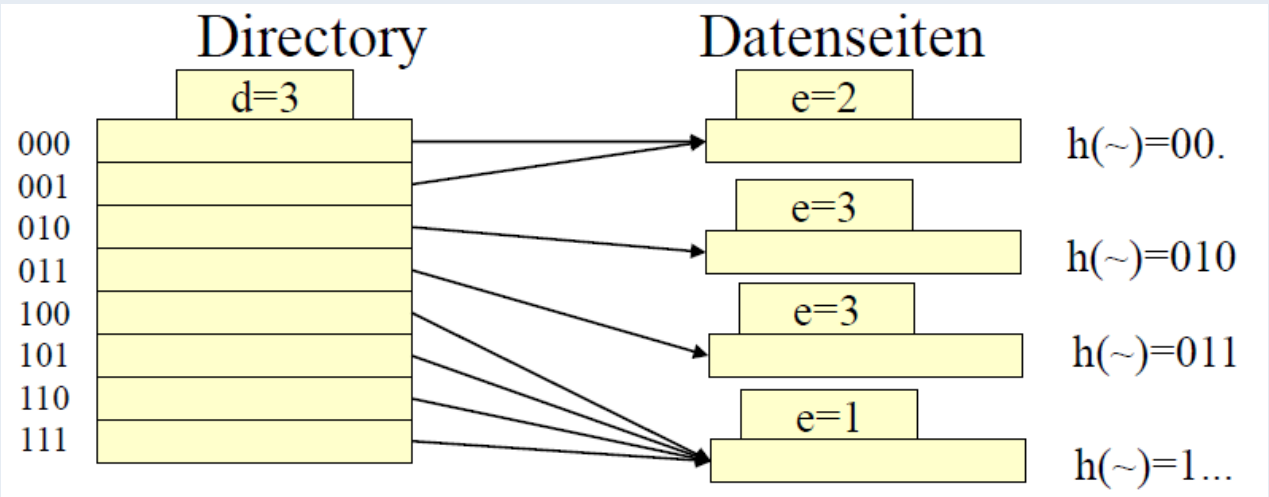
- Mit Directory:



Erweiterbares Hashing

Hash-Verfahren mit Directory: Erweiterbares Hashing

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

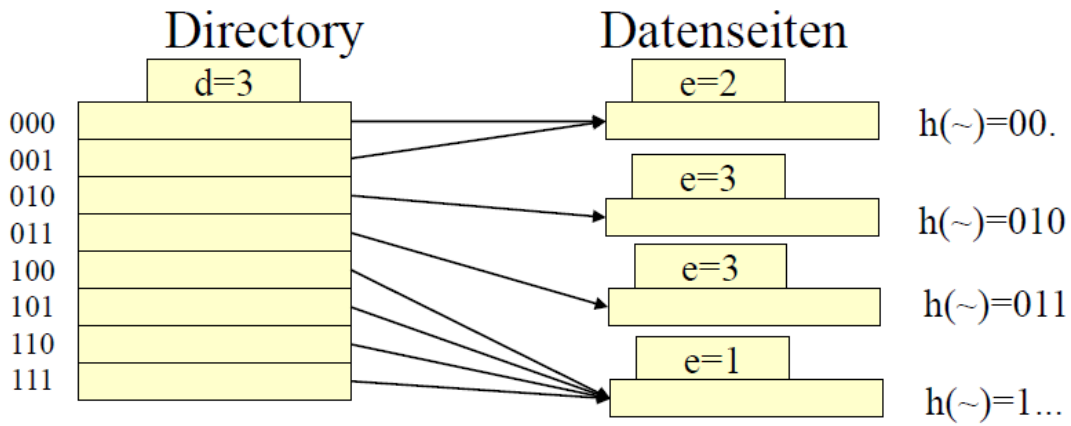


Erweiterbares Hashing: Einfügen

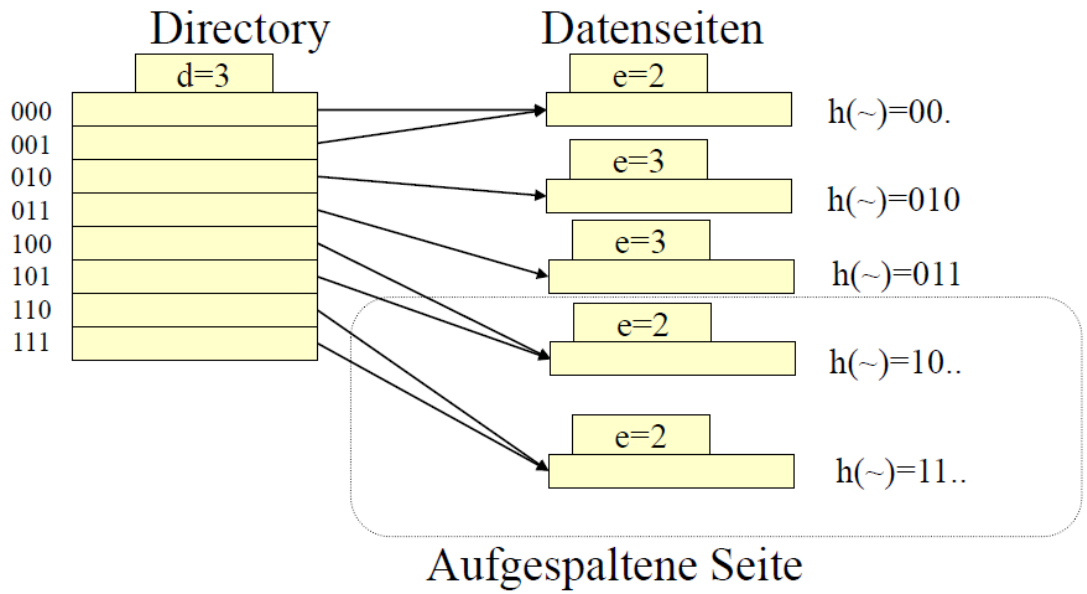
- Gegeben: Datensatz mit Schlüssel k
- Bestimme die ersten Bits des Pseudoschlüssels $h(k) = (b_1, b_2, ..., b_d, ...)$
- Der Directoryeintrag $D[b_1, b_2, .., b_d]$ liefert Seitennummer
- Datensatz wird in berechnete Seite eingefügt
- Falls Seite danach maximal gefüllt (bzw. Füllgrad zu hoch, z.B. $> 90\%$):
  - Aufspalten der Datenseite
  - Verdoppeln des Directory

Erweiterbares Hashing: Einfügen — Überlaufbehandlung

- Vor Split:

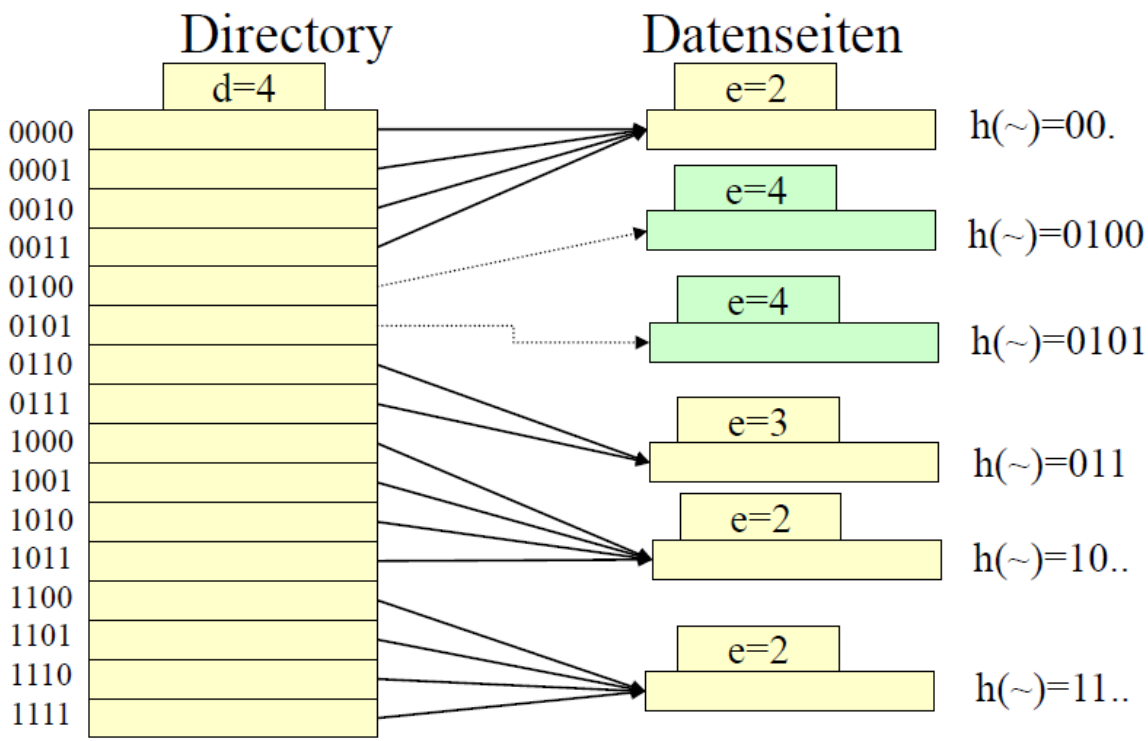


- Nach Split:



Erweiterbares Hashing: Einfügen — Überlaufbehandlung

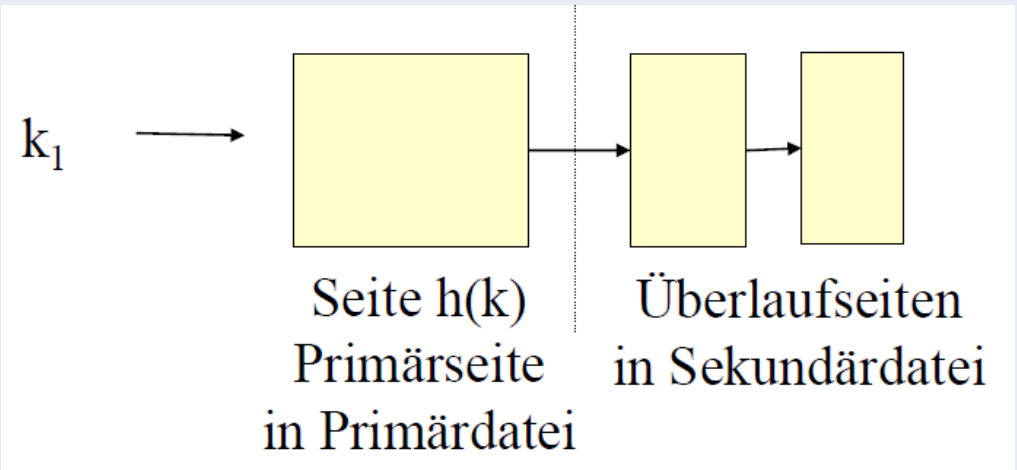
- Irgendwann reichen die Adressen im Directory nicht mehr
- Dann: Verdopplung des Directory:



Lineares Hashing

Hash-Verfahren 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 (verkettet)
- Aber: bei zu vielen Überlaufseiten degeneriert Suchzeit

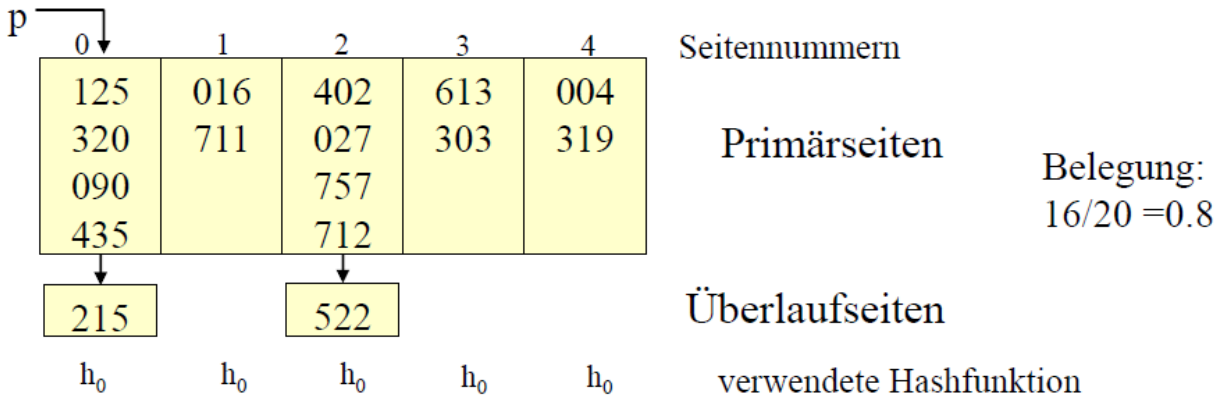


Lineares Hashing

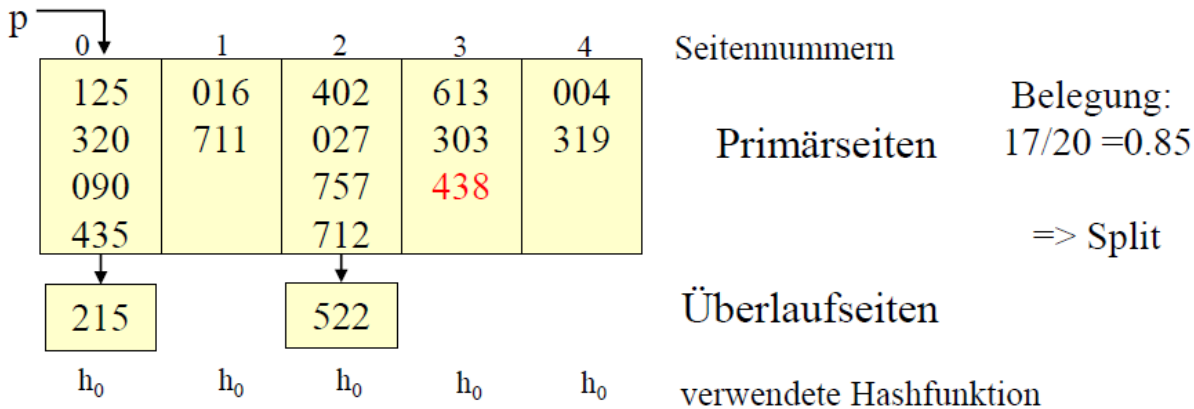
- Verbesserung (lineares Hashing): dynamisches Wachstum der Primärdatei
- Folge von Hash-Funktionen: $h_0, h_1, h_2, \ldots$
- Erweitern der Primärdatei um jeweils eine Seite
- Feste Splitreihenfolge
- Expansionzeiger zeigt an welche Seite gesplittet wird
- Split wenn Belegungsfaktor Schwellwert (typisch: 80%) übersteigt

Lineares Hashing

- Hashfunktionen: $h_0(k) = k \cdot mod(5), h_1(k) = k \cdot mod(10), \ldots$
- p = Expansionzeiger, Schwellwert: 80%

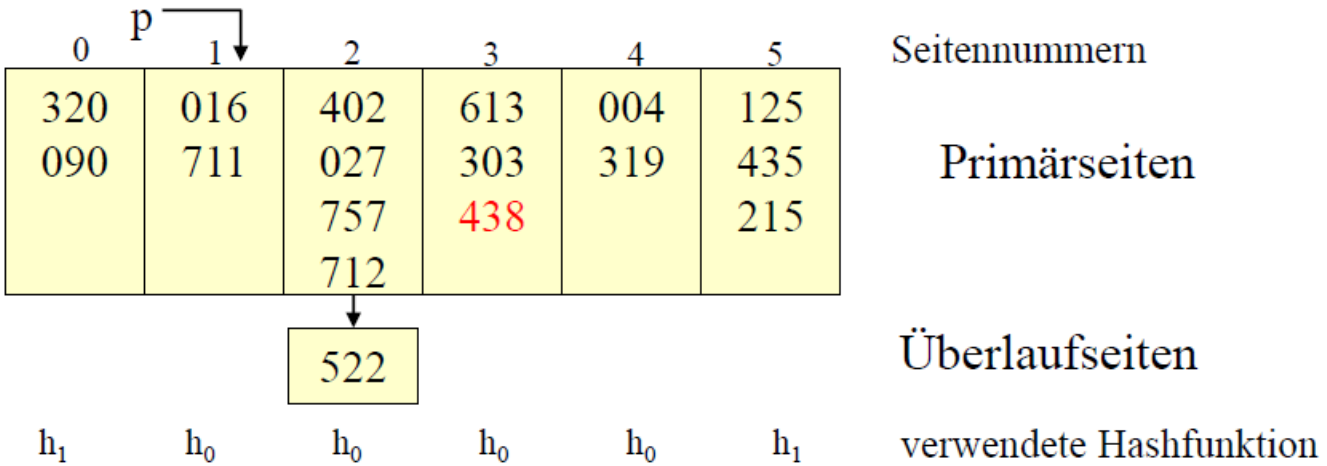


- Einfügen von Schlüssel 438



Lineares Hashing

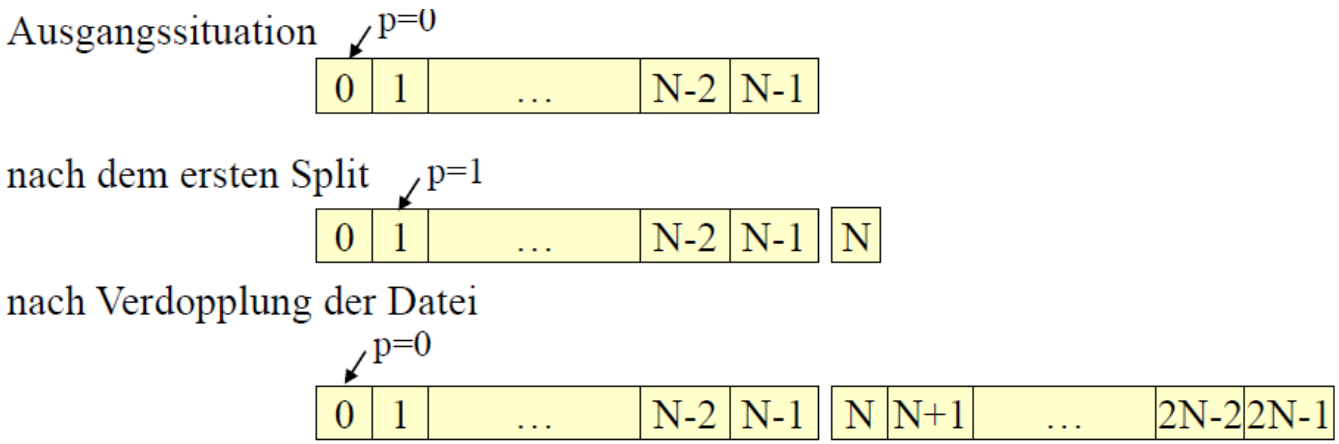
- Expansion der Seite 0 auf die Seiten 0 und 5



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

Lineares Hashing

- Prinzip der Expansion



- Split in fester Ordnung (nicht: Split der vollen Seiten)
- Trotzdem wenig Überlaufseiten
- Gute Leistung für gleich verteilte Daten
- Adressraum wächst linear

Lineares Hashing

- Damit das so funktioniert, müssen einige Anforderungen an die Hashfunktionen $\{h_i\}, i > 0$ gelten
- Bereichsbedingung

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

- Splitbedingung

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

oder

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

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

Agenda

1. Einleitung

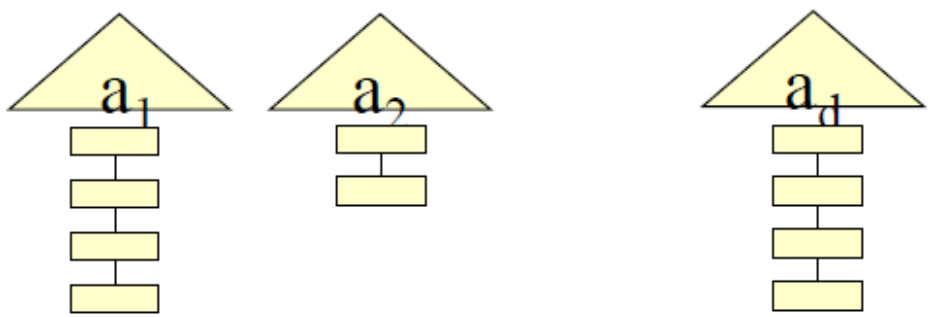
2. Datenorganisierende Indexstrukturen

3. Raumorganisierende Indexstrukturen

4. Verschiedenes (Potporri)

Invertierte Listen

- Häufigste Lösung für Anfragen auf mehreren Attributen
- Jedes relevante Attribute wird mit eindimensionalem Index verwaltet
- Suche nach mehreren Attributen $a_1, a_2, ..., a_d$
- Erstellen von Ergebnislisten mit Datensätzen d bei denen $d.a_i$ der Anfragebedingung genügt
- Bestimmen des Ergebnis über mengentheoretischen Verknüpfung (z.B. Schnitt) der einzelnen Ergebnislisten



Invertierte Listen: Eigenschaften

- Die Antwortzeit ist nicht proportional zur Anzahl der Antworten
- Suchzeit wächst mit Anzahl der Attribute
- Genügend Effizienz bei kleinen Listen
- Sekundärindizes (invertierte Listen) für nicht Primärschlüssel beeinflussen die physikalische Speicherung nicht (keine Lokalität etc.)
- Daher: zusätzliche Sekundärindizes können das Leistungsverhalten bei DB-Updates stark negativ beeinflussen

Indexstrukturen in SQL

- Generierung eines Index

  ```
  CREATE INDEX indexName ON relation (A1, A2, ...  An);
  ```

  die Attributeliste gibt die "Sortier-Priorität" an
- Löschen eines Index

  ```
  DROP INDEX indexName;
  ```
- Verändern eines Index

  ```
  ALTER INDEX index-name ...;
  ```

Index-unterstützte Anfragen

- Exact match query

  `SELECT * FROM t WHERE A1 = ... AND ... AND An =...`

- Partial match query

  `SELECT * FROM t WHERE A1 = ... AND ... AND Ai = ...`

- Range query

  `SELECT * FROM t WHERE ... AND Ai < ... AND ...`

- Pointset query

  `SELECT * FROM t WHERE ... AND Ai IN (7,17,77) AND ...`

- Pattern matching query

  `SELECT * FROM t WHERE ... AND Ai LIKE 'c1c2...ck%'`

  Problem: Anfragen wie `wort LIKE '%system'` (also mit beliebigen Präfix) werden nicht unterstützt (nur Postfix!). Man kann aber z.B. eine Relation `r` aufbauen, in der alle Wörter revers im Attribut `r.a` gespeichert werden und dann effizient nach `a LIKE 'metsys%'` suchen lassen.