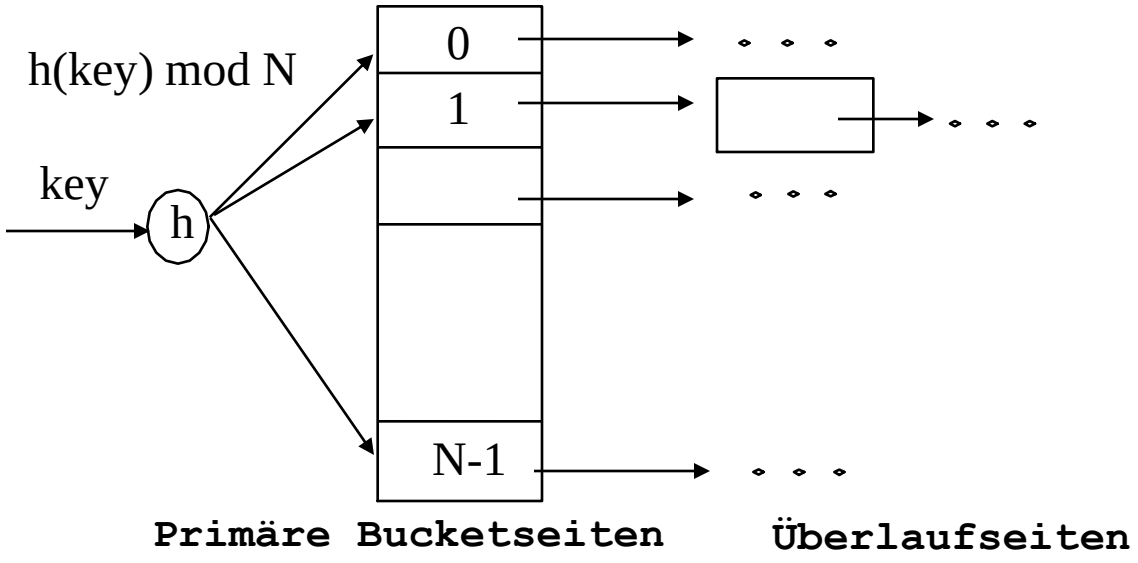
Hash-Verfahren

Einführung

- Drei Alternativen, wie Dateneinträge **k*** im Index aussehen können:
  1. Datensatz mit Schlüsselwert **k**
  2. <**k**, ID des Datensatzes mit dem Wert des Suchschlüssels **k**>
  3. <**k**, Liste von Datensatz-IDs mit Suchschlüssel **k**>
- Auswahl einer Alternative ist orthogonal zur Indexierungstechnik, die genutzt wird, um Dateneinträge mit einem gegebenen Schlüsselwert **k** zu plazieren.
- Hash-basierte Verfahren am besten geeignet für Lookup-Operationen (Suche nach Daten mit einem bestimmten Wert), ungeeignet für Bereichssuche (Range Query)
- Statische und dynamische Hashing-Techniken: Trade-Offs analog zur Bewertung ISAM vs. B+ Bäume

Statisches Hashing

- # Primärseiten fest, sequentiell allokiert, niemals de-allokiert; Überlaufseiten wenn benötigt
- **h**(*k*) mod M = Bucket, zu dem der Dateneintrag mit dem dem Schlüsselwert *k* gehört. (M = # Buckets)



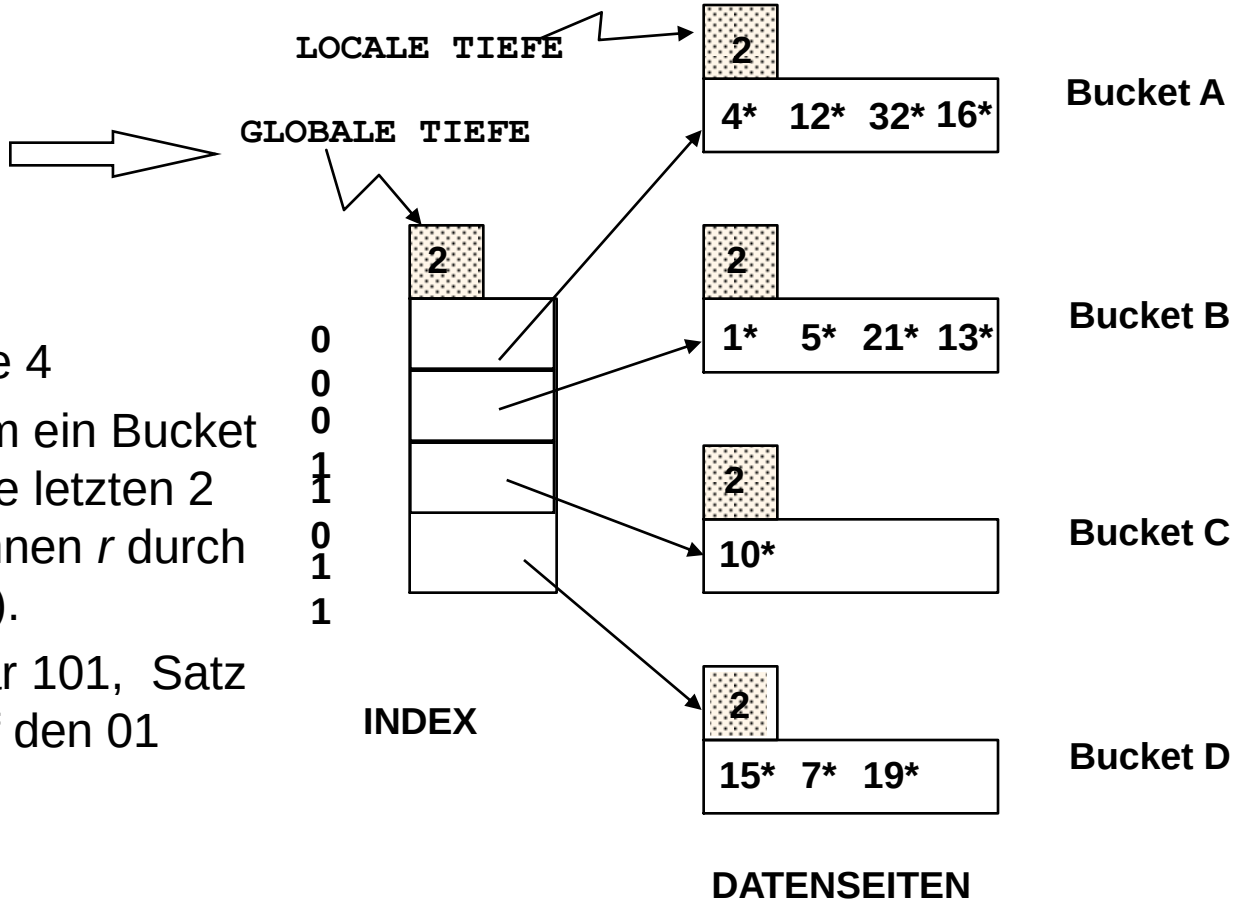
Statisches Hashing (Forts.)

- Buckets enthalten *Dateneinträge* und entsprechen einer Seite
- Hashfunktion auf dem *Suchschlüssel*-Attribut des Datensatzes *r.* Muß eine Verteilung der Werte in einem Bereich 0 ... M-1 gewährleisten (auch Streuen genannt)
  - **h**(*key*) = (a * *key* + b) funktioniert gut
  - Hash-Funktion nach dem Divisionsrestverfahren mit linearem Sondieren (Berechnung von Ersatzadressen)
  - a und b sind Konstanten; viel Kenntnis über Tuning von **h** vorhanden
- Lange Überlaufketten können entstehen und die Performance beeinträchtigen
  - *Extendible* und *Linear Hashing*: Dynamische Techniken, um dieses Problem zu lösen

Extendible Hashing

- Extendible = Erweiterbares Hashing
- Situation: Bucket (Primärseite) voll!
  Reorganisation der Datei durch Verdopplung der Anzahl Buckets nicht sinnvoll!
  - Lesen und Schreiben von Seiten ist teuer!
  - *Idee*: Nutze ein *Verzeichnis von Pointern auf Buckets (Index)*, verdopple die Anzahl Buckets durch Verdopplung des Indexes, Aufsplitten des Buckets, das übergelaufen ist
  - Index viel kleiner als Datei, somit Verdopplung viel billiger. Nur eine Seite mit Dateneinträgen wird gesplittet. *Keine Überlaufseiten*!
  - Hash-Funktion richtig einstellen!

Beispiel

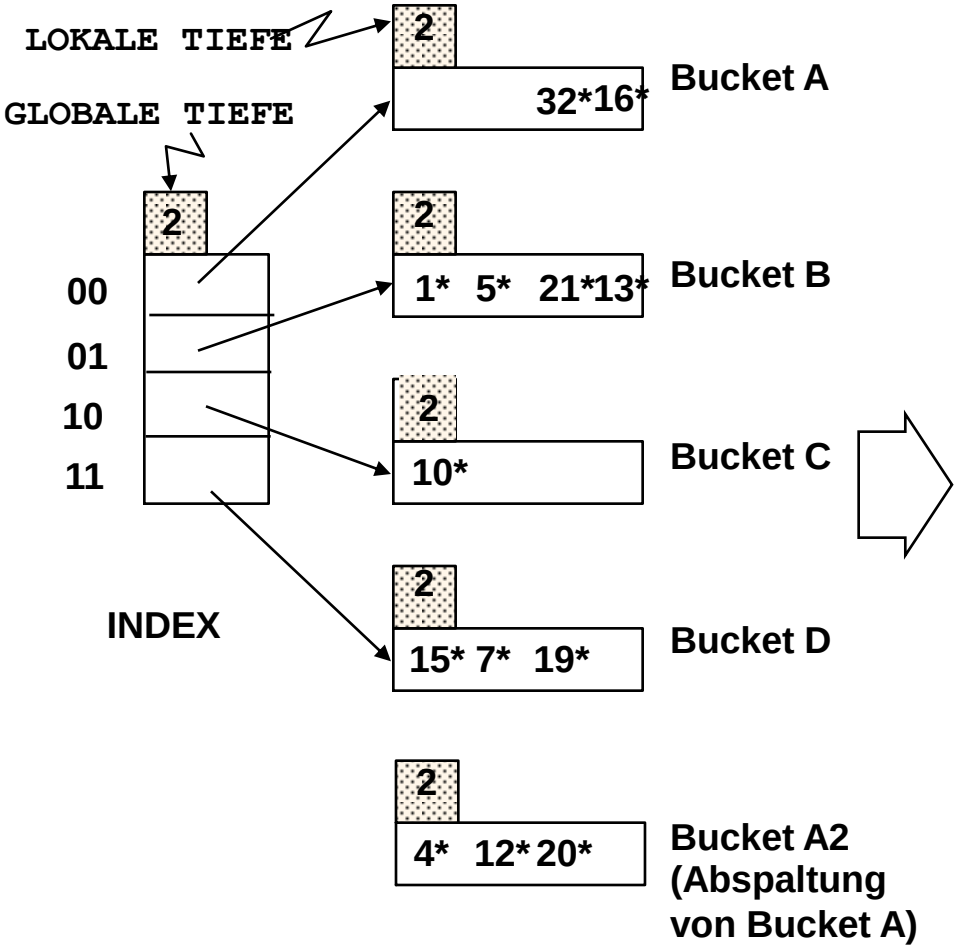


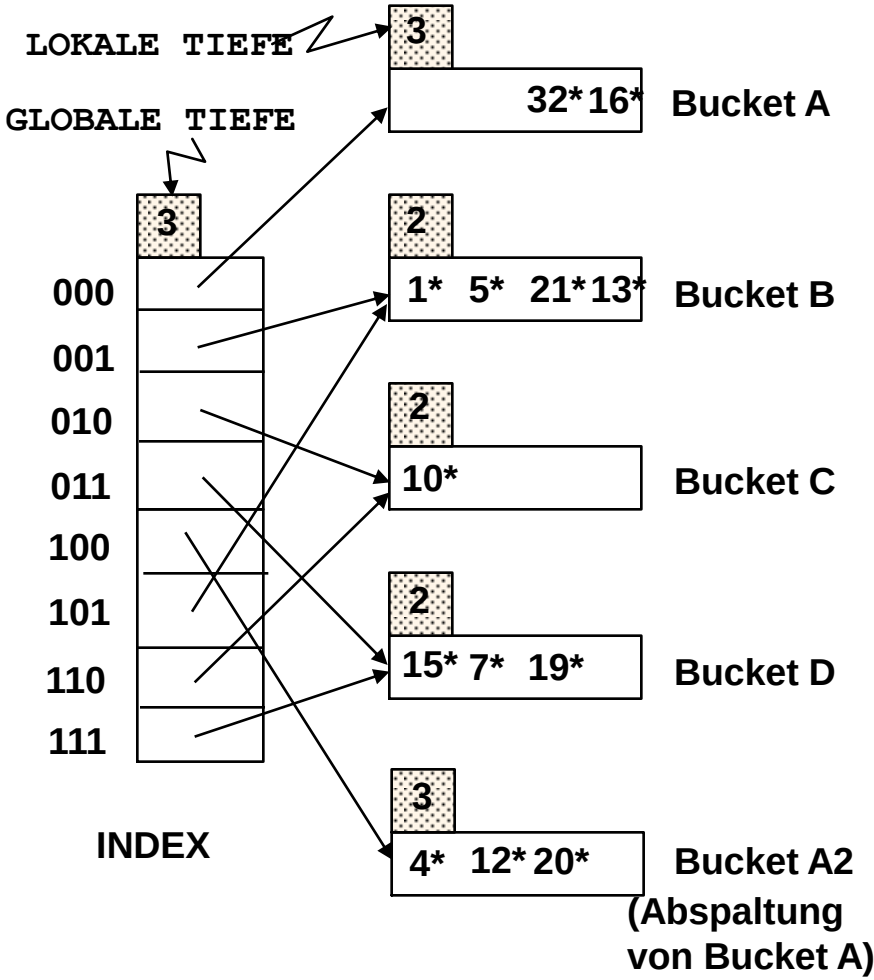
- Index ist Array der Größe 4
- Globale Tiefe 2 heißt: Um ein Bucket für *r* zu finden, nehme die letzten 2 Bits von **h**(*r*); wir bezeichnen *r* durch seine Hash-Funktion **h**(*r*).
  - Wenn **h**(*r*) = 5 = binär 101, Satz gehört in Bucket, auf den 01 verweist.

- Insert: Wenn Bucket voll, splitte ihn (allokieren neue Seite - Umverteilung der Einträge)
- Wenn notwendig, verdopple den Index (Splitten eines Bucket erfordert nicht immer Verdopplung des Indexes; hängt ab von der lokalen Tiefe des aufzusplittenden Bucket im Vergleich zur globalen Tiefe)

Einfügen eines neuen Eintrags h(r)=20

Directory verdoppeln!





Bemerkungen

- 20 = binär 10100. Die letzten **2** Bits (00) verraten: *r* gehört in A oder A2. Die letzten **3** Bits werden benötigt, um zu entscheiden wohin genau
  - *Globale Tiefe des Directory*: Maximale Anzahl Bits, die benötigt wird, um zu entscheiden, in welchen Bucket ein Eintrag gehört.
  - *Lokale Tiefe eines Bucket*: Anzahl von Bits, die benötigt wird, um zu entscheiden, ob ein Eintrag in diesen Bucket gehört.
- Wann verursacht das Aufsplitten eines Bucket eine Verdopplung des Indexes?
  - Vor dem Einfügen gilt: lokale Tiefe des Bucket = globale Tiefe. Ein Insert erhöht die lokale Tiefe so daß: lokale Tiefe > globale Tiefe.
  - Index wird verdoppelt durch Kopieren und Fixieren eines Pointers zum neuen, abgespalteten Bucket (*Split Image Page*).
    - Verwendung der hinteren Bits (least significant) gestattet eine effiziente Verdopplung des Indexes durch Kopieren!

Kommentare zum Erweiterbaren Hashing

- Wenn der Index in den Speicher paßt, werden Lookup-Operationen mit einem Plattenzugriff ausgeführt; ansonsten zwei.
  - Beispiel 100 MB File:
    - 100 Bytes/Satz, 4K Seite, somit 40 Einträge pro Seite bzw. Bucket
    - enthält 1,000,000 Sätze (als Dateneinträge)
    - 25,000 Indexeinträge
    - hohe Wahrscheinlichkeit, daß Index in den Speicher paßt
  - Index kann sehr stark wachsen, wenn die Streuung der Hashwerte ungleichmäßig ist
- Kollision: Überlaufseiten anlegen, wenn keine Einträge mehr in Bucket passen!
- Delete:
  - Wenn das Löschen eines Dateneintrages den Bucket vollständig leert, kann dieser mit seinem abgesplitteten Teil (Split Image) wieder zusammengefaßt werden.
  - Wenn jeder Index-Eintrag zum gleichem Bucket wie auf dessen Abspaltung verweist, kann der Index entsprechend halbiert werden.

Lineares Hashing *

- Weiteres dynamisches Hash-Verfahren = Alternative zum Erweiterbaren Hashing.
- LH behandelt das Problem langer Überlaufketten ohne Verwendung eines Index
- *Idee*: Verwende eine Folge von Hash-Funktionen $\mathbf{h}_0$, $\mathbf{h}_1$, $\mathbf{h}_2$, ...
  - $\mathbf{h}_{i+1}$ verdoppelt den Wertebereich von $\mathbf{h}_i$ (ähnlich wie Verdopplung des Index)
- Index wird vermieden in LH durch Verwendung von Überlaufseiten und Auswahl des aufzusplittenden Buckets nach Round-Robin-Methode
  - Überlaufseiten im allgemeinen nicht sehr lang
  - Einfache Behandlung von Duplikaten
  - Speicherplatzauslastung evtl. schlechter als beim erweiterbaren Hashing, weil Splits nicht auf dichten Datenbereichen konzentriert sind.

Zusammenfassung

- Hash-basierte Indexe:
  - am besten für Lookup-Operationen
  - keine Unterstützung für Bereichssuchen
- Statisches Hashing kann zu langen Überlaufketten führen
- Erweiterbares Hashing vermeidet Überlaufseiten durch Splitten voller Buckets, wenn neue Einträge hinzugefügt werden
  (*Duplikate könnten Überlaufseiten erfordern)*
- Index zur Verwaltung der Buckets, verdoppelt sich periodisch
  - Durch unbegrenztes Wachstum des Index-Array paßt es irgendwann nicht mehr in den Hauptspeicher und erhöht I/O-Kosten