Vorlesung Implementierung von Datenbanksystemen

4. Schlüsselzugriff – Teil 1

Prof. Dr. Klaus Meyer-Wegener

Wintersemester 2019/20

Lehrstuhl für Informatik 6 (Datenmanagement)
Friedrich-Alexander-Universität Erlangen-Nürnberg

4. Satzzugriff über Schlüsselwerte

- **Satzadressen haben nichts mit Anwendungsdaten zu tun.**
  - Vom System vergeben, künstlich (wie Telefonnummer)
- **Ziel statt dessen:**
  - Über **Inhalt** (bestimmte Felder) eines Satzes zugreifen zu können
    - Z.B. Kundennummer, Name, Wohnort
  - Auch "assoziativer" Zugriff genannt
- **Diese Felder nennt man (Such-) Schlüssel.**
- **Besonders sinnvoll in interaktiven Anwendungen:**
  - Für den Schlüssel wird ein Wert vorgegeben
    - Z.B. vom GUI-Formular eingelesen
  - Datenhaltungssystem (Zugriffsmethode) soll den Satz der Datei liefern, der diesen Schlüsselwert enthält
    - Eigentlich alle Sätze, die ihn enthalten

Schlüsselzugriff

- **Bisher nicht unterstützt**
  - Sequenzielle und direkte Satzdatei können das nicht.
- **Einzige Möglichkeit:**
  - Alle Sätze der Reihe nach geben lassen (sog. **Scan**)
    und dann selbst auf gewünschten Schlüsselwert abprüfen – ineffizient!
    - Wird deshalb auch nicht als Operation angeboten.
- **Also: neue Hilfsstrukturen erforderlich**
  - Zugriff über bestimmten Schlüssel
    muss im voraus geplant
    und vom System speziell unterstützt werden
  - D.h. Sätze gleich so abspeichern,
    dass man sie gut wiederfinden kann
    - Macht ein bisschen mehr Arbeit

4.1 Gestreute Speicherung (Hashing)

- **Ziel:**
  - **Berechnung der Speicherposition** (Adresse) aus dem Schlüsselwert
  - Dann direktes Lesen des Blocks, der den gesuchten Satz enthält
    - Eine einzige E/A-Operation!
- **Umsetzung:**
  - Berechnung der Blocknummer genügt
    - Suchaufwand innerhalb des Blocks vernachlässigbar
- **(Fast) keine Hilfsstrukturen erforderlich**
  - Nur eine geeignete Berechnungsfunktion
- **Voraussetzungen:**
  - Maximale Anzahl der Sätze abschätzbar → zulässige Blocknummern
  - Hinreichend viele Blöcke für alle Sätze von Anfang an bereitstellen
    - Mit etwas "Luft"
      - Z.B. bei durchschnittlich 10 Sätzen pro Block und 10.000 Sätzen: 1.100 Blöcke bereitstellen

Gestreute Speicherung (2)

- **Blöcke, die mit gestreuter Speicherung gefüllt werden:**
  - Bildhaft auch "**Buckets**" (Eimer) oder "Bins" (Kästen) genannt
  - Spezielle Art von Blöcken
    - Neben denen für Sätze mit Zugriff über Satzadresse und denen für Freispeicherverwaltung
      - Es wird noch mehr Arten geben …
- **Vergabe einer Satzadresse**
  - weiterhin möglich, aber umständlich
  - Sollte ersetzt werden durch eindeutigen Schlüssel
    - Siehe unten

Hash-Funktion

- **Aufgabe: Umrechnung Schlüsselwert in Blocknummer**
- **Ziel dabei:**
  - Möglichst gleichmäßige Verteilung der Sätze auf die Buckets
- **Kollisionen**
  - Gleiche Blocknummer bei verschiedenen Schlüsselwerten
  - Bis zu einem gewissen Grad sogar erwünscht:
    Es passen ja mehrere Sätze in ein Bucket.
- **Auswahl der geeigneten Hash-Funktion**
  - hängt vom Typ des Schlüssels und der Verteilung der Schlüsselwerte ab
  - Je genauer die Verteilung der Schlüsselwerte bekannt ist,
    desto besser kann die Hash-Funktion darauf abgestimmt werden.
  - Es konnte (mathematisch) bewiesen werden:
    Wenn nichts bekannt ist, ist das **Divisions-Rest-Verfahren** am besten:
    - **h(k) = (k modulo q)**
    - mit k Schlüsselwert (ggf. in Zahl umgerechnet oder als Zahl interpretiert),
      q Anzahl der Buckets und h(k) errechnete Blocknummer (0 bis q–1)

Überlaufbehandlung

- **Völlig gleichmäßige Verteilung gelingt selten**
  - D.h. einige Buckets bekommen mehr Sätze ab, andere weniger.
- **Bucket kann "überlaufen":**
  - Zu klein für alle Sätze, die ihm durch die Hash-Funktion zugeteilt werden

- **Erste Lösung: Open Addressing**
  - Ausweichen auf Nachbar-Buckets in festgelegter Reihenfolge
    - \+ Kein zusätzlicher Speicherplatz benötigt
    - – Beim Suchen findet man in einem Bucket nicht nur Treffer, sondern auch "Überläufer".
    - – Beim Löschen muss man Überläufer "zurückholen".
    - – In den Nachbar-Buckets werden u.U. zusätzliche Überläufe erzeugt.

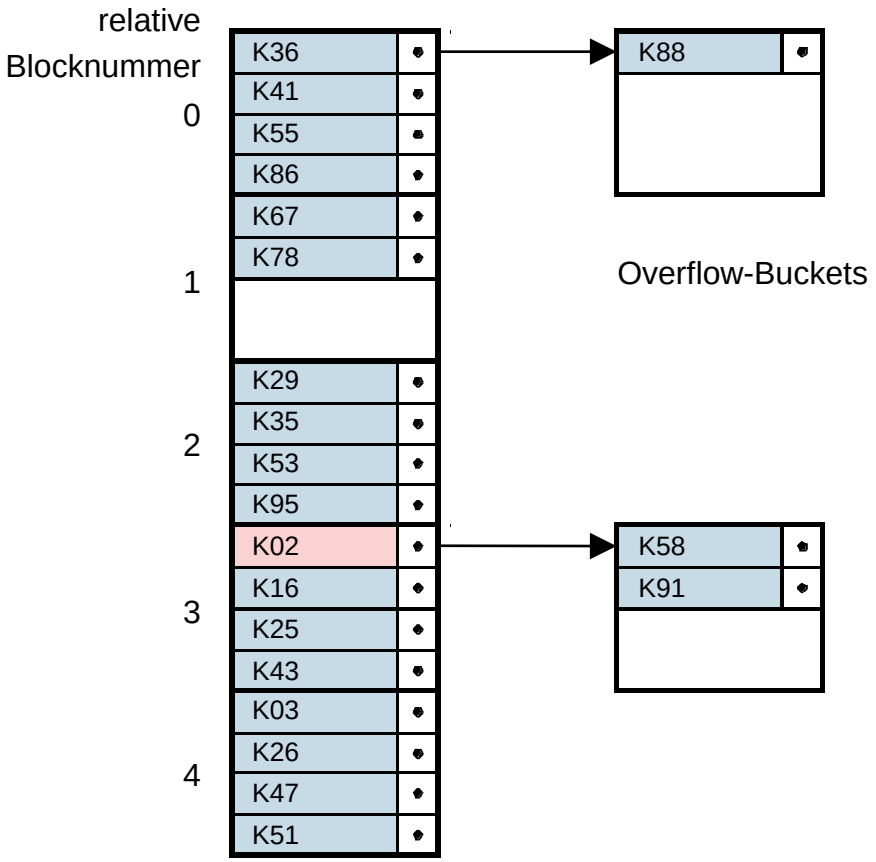
Überlaufbehandlung (2)

- **Zweite Lösung: Overflow-Buckets (mit "Separate Chaining"):**
  - Anlegen spezieller zusätzlicher Überlauf-Buckets
    - Mit Verkettung: Primär-Bucket zeigt auf sein Überlauf-Bucket
      - Pointer = Blocknummer
  - Und zwar für jedes übergelaufene Bucket separat
    - – Speicherplatzbedarf
    - \+ Keine Mischung von Sätzen
    - \+ Keine Beeinträchtigung der Nachbar-Buckets

- **Nachteil in beiden Fällen:**
  - Zwei Blockzugriffe (sonst nur einer)

Überlaufbehandlung (3)

- **Hash-Berechnung für Schlüsselwert K02**



$$
\begin{array}{ll}
 & 1101\ 0010 \\
\oplus & 1111\ 0000 \\
\oplus & 1111\ 0010 \\
\hline
 & 1101\ 0000 = 208_{10}
\end{array}
$$

208 mod 5 = 3

Hash-Funktion:
EBCDIC-Darstellung jedes Zeichens
als Bitkette betrachten,
in 8-Bit-Stücken mit EXOR verknüpfen
("falten")
und Ergebnis als Zahl interpretieren

Anschließend Divisions-Rest-Verfahren

Zugriffsoperationen (Dienste)

- **Hash-Funktion nur intern verfügbar!**
  - Ausgewählt vom Systemadministrator, nicht vom Benutzer
  - D.h. Administrator kann den Datenbestand reorganisieren:
    - Neue Hash-Funktion auswählen, Zahl der Buckets erhöhen und alle Datensätze erneut abspeichern ("Rehash"),

    ohne dass die Anwendungen betroffen sind.
  - Operationen nehmen keinen Bezug auf Hash-Funktion.
- **Abspeichern eines Satzes:**

```
void HashedRecordFile::insert ( char *RecordBuffer,
        int RecordLength, char *KeyValue )
```

- **Auffinden eines Satzes über Schlüsselwert:**

```
char *HashedRecordFile::read ( char *KeyValue,
        int *RecordLength )
```

Zugriffsoperationen (2)

- **Operationen modify, delete, read-first und read-next**
  - wie bei direkten satzstrukturierten Dateien,
    nur mit Schlüsselwert statt Satzadresse
  - Satz und Schlüssel jeweils für sich verwaltet,
    daher auch eigene Änderungsoperation:

```
void HashedRecordFile::modify-key ( char *OldKeyValue,
  char *NewKeyValue )
```

- **Wenn Schlüssel *nicht* eindeutig: Sonderbehandlung erforderlich**
  - Gefundene Sätze einzeln abrufen
    - **next** o.ä. mit Leseposition ("Cursor")
  - Anhand anderer Felder den richtigen Satz (zum Ändern oder Löschen) identifizieren
  - Änderungen dann auf diesen Satz beziehen (aktuelle Leseposition)

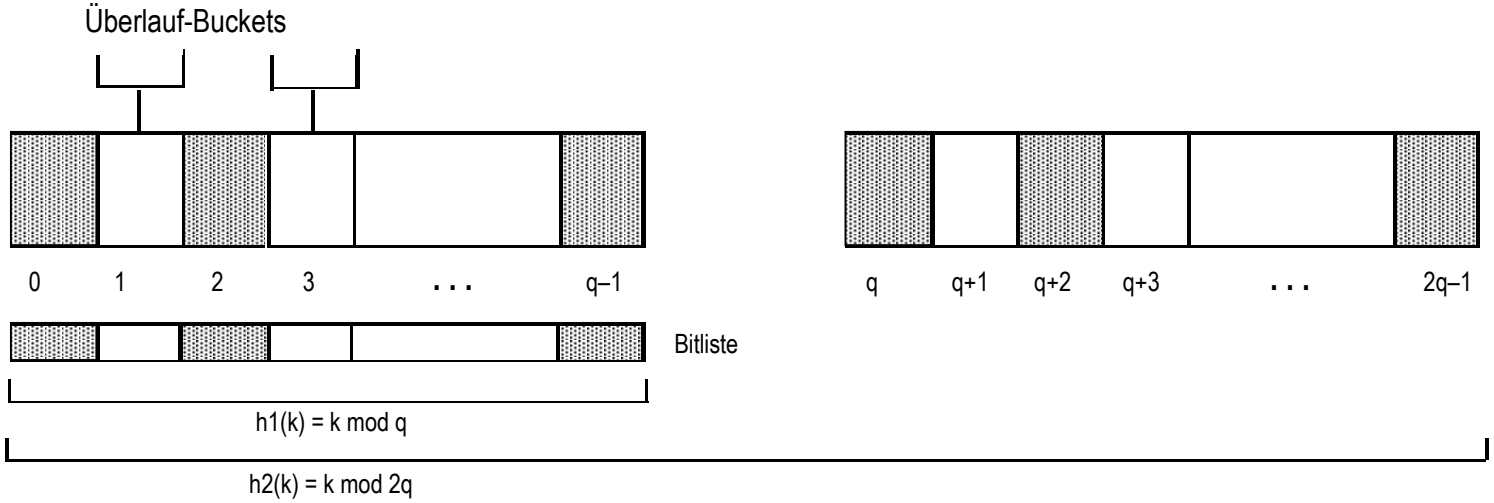
Bewertung

- **Schneller Zugriff über Schlüssel (1 bis 2 Blockzugriffe)**
  - Entscheidender Vorteil
  - Schafft keine andere Schlüssel-Zugriffsmethode
- **Sätze werden durcheinandergewürfelt**
  - Auch nicht nach Schlüsselwert geordnet
- **Speicherplatz muss im voraus belegt werden!**
  - Macht man ihn vorsichtshalber zu groß: Verschwendung
  - Macht man ihn zu klein: aufwändige Reorganisation
    ("offline" – Datei muss vorübergehend aus dem Verkehr gezogen werden)
    - Anmerkung: Es gibt "dynamische" Hash-Verfahren, die Erweiterung um Buckets wie auch Freigeben von Buckets zulassen ohne Reorganisation, siehe unten.
- **Gestreute Speicherung kann nur nach *einem* Schlüssel erfolgen!**
  - Bei Entscheidung z.B. für den Nachnamen
    muss nach Geburtsdatum, Wohnort usw. sequenziell gesucht werden

Virtuelles Hashing

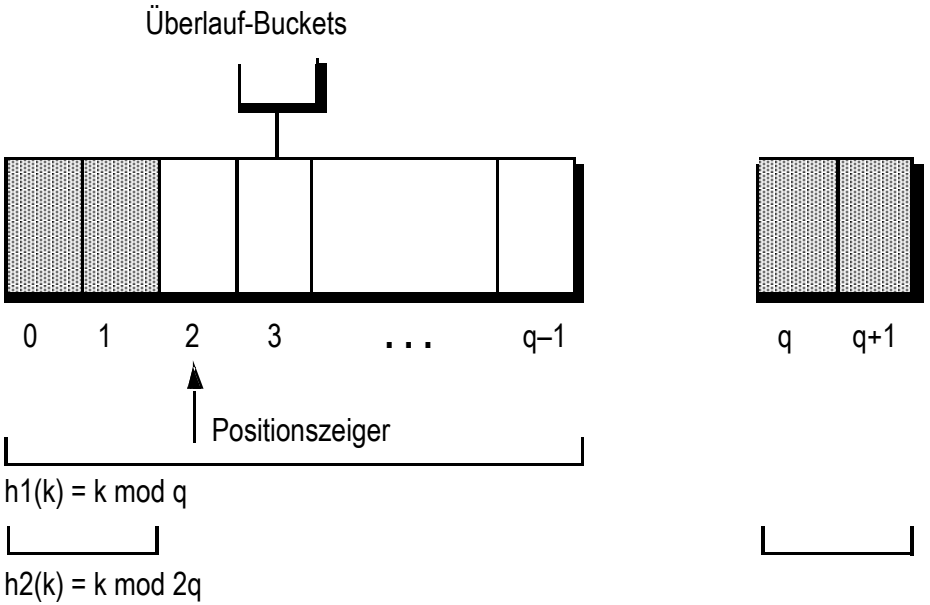
- **Idee: andauernde ("online") Reorganisation der Bucket-Folge während der Einfügungen und Löschungen**
  - Nebeneinander alte Hash-Funktion (für kleinere Zahl von Buckets) und neue Hash-Funktion (für größere Zahl) benutzen
  - Neue Hash-Funktion muss die Schlüssel, die die alte auf einen Bucket abgebildet hat, auf zwei Buckets verteilen (möglichst gleichmäßig)
  - Mit **q** Buckets beginnen;
    jeder mit maximal **b** Sätzen ("Bucket-Faktor")
    - Kapazität also insgesamt $q \times b$ Sätze
  - Belegungsfaktor β := Anzahl gespeicherter Sätze / Kapazität
  - Verwendung eines **Schwellenwerts** α für den Belegungsfaktor,
    $0 \leq \alpha \leq 1$, typischer Wert: 0,8
  - Wenn $\beta > \alpha$:
    Menge der Buckets vergrößern
    - Bis dahin ggf. Überlauf-Buckets wie gehabt

Erster Ansatz: "VH1"



- Wenn α überschritten: auf einen Schlag q, 2q, 4q, … neue Blöcke belegen, direkt hinter den bisherigen Buckets (Datei vergrößern)
- Komplettes Umspeichern aller Sätze verzögern:
  Erst wenn beim nächsten Einfügen eines Satzes in Bucket i (i < q) Überlauf-Bucket gefunden wird, Rehash der Sätze dieses Buckets inkl. Überläufer mit h2; Verteilung der Sätze auf Buckets i und i+q, dann Bit i setzen

Zweiter Ansatz: "Lineares Hashing"



- Wenn α überschritten: *einen* neuen Bucket hinten anfügen und Bucket, auf dem Positionszeiger p steht, aufteilen (auch wenn kein Überlauf-Bucket!)
- Immer zuerst h1 anwenden; falls Ergebnis kleiner als p, auf h2 übergehen
- Nachdem Bucket q–1 aufgeteilt wurde, p wieder auf Null setzen;
  h1 dann nicht mehr benötigt;
  nach Aufteilen von Bucket 0 h2 und h3 im Einsatz

Lineares Hashing (2)

- **Dynamisches Wachsen (und Schrumpfen) des (primären) Hash-Bereichs**
  - Buckets werden in einer fest vorgegebenen Reihenfolge gesplittet.
    ⇒ Einzige Hilfsstruktur: nächstes zu splittendes Bucket (Positionszeiger)
  - D.h. minimale Verwaltungsdaten
  - Keine großen Directories für die Hash-Datei

  - Aber: keine Möglichkeit, Überlaufsätze vollständig zu vermeiden!
    - Auch eine hohe Rate von Überlaufsätzen kann als Indikator dafür genommen werden, dass die Datei eine zu hohe Belegung aufweist und deshalb erweitert werden muss.

Anhang

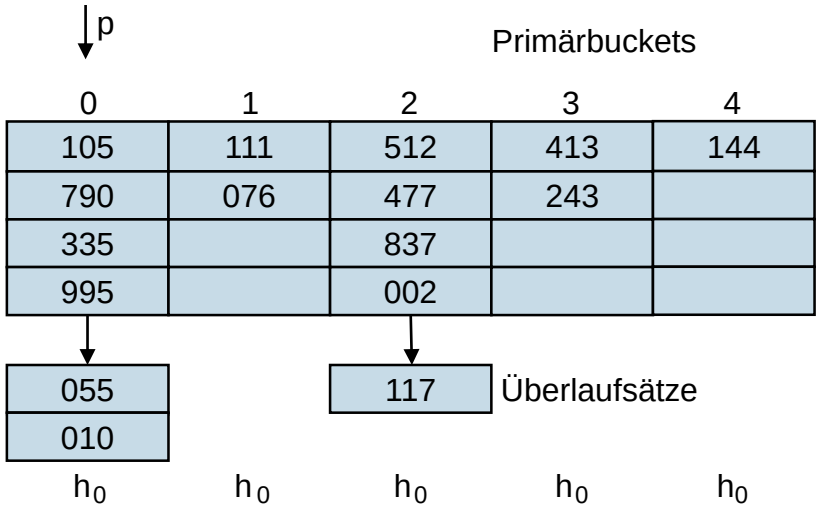
- **Nicht mehr verwendete Folien**
- **Zum Nachschlagen bei Bedarf**

4.A.1 Lineares Hashing

- **Prinzipieller Ansatz**
  - q: Größe der Ausgangsdatei in Buckets
  - Folge von Hash-Funktionen h_0, h_1, ...
    - wobei $h_0(k) \in \{0, 1, ..., q–1\}$ und
      $h_{L+1}(k) = h_L(k)$ oder
      $h_{L+1}(k) = h_L(k) + 2^L \cdot q$
      für alle $L \geq 0$ und alle Schlüssel k gilt
  - Beispiel:
    - $h_L(k) = k \bmod (2^L \cdot q)$, L = 0, 1, ...

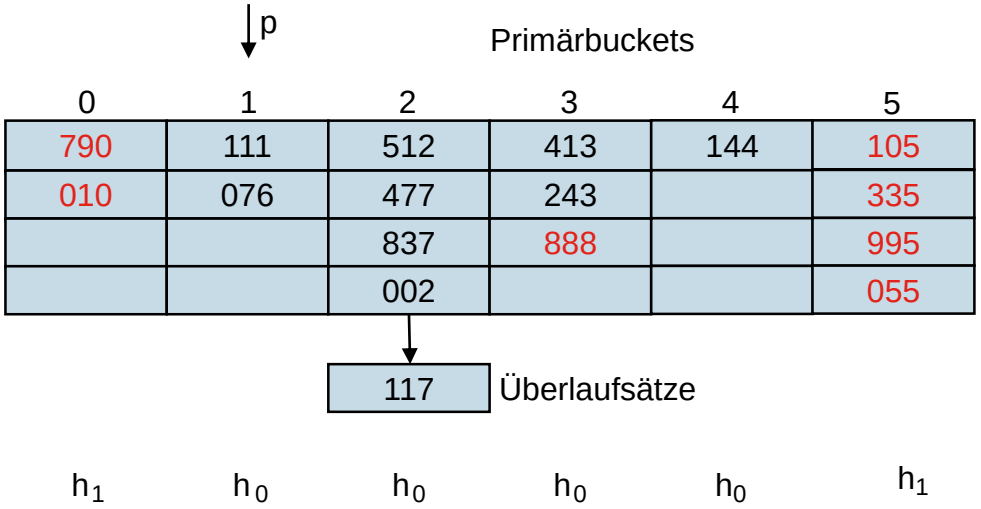
Lineares Hashing (2)

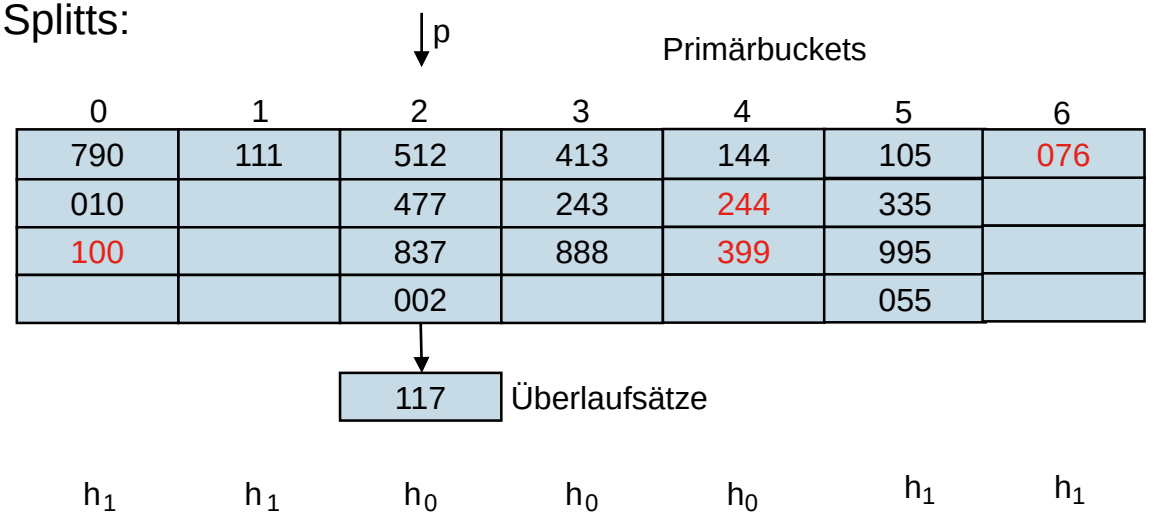
- **Beschreibung des Dateizustands**
  - L: Anzahl der bereits ausgeführten Verdopplungen
  - N: Anzahl der gespeicherten Sätze
  - b: Kapazität eines Buckets
  - p: zeigt auf nächstes zu splittendes Bucket ($0 \le p < q\ 2^L$)
  - β: Belegungsfaktor = $\dfrac{N}{\left(q \cdot 2^L + p\right) \cdot b}$
- **Beispiel:**
  - h_0 (k) = k mod 5
  - h_1 (k) = k mod 10, ...
  - b = 4, L = 0, q = 5
  - Splitt, sobald $\beta > \alpha = 0{,}8$
  - aktuell: N = 16,
    $\beta = 16 / (5 + 0) \cdot 4 = 0{,}8$



Lineares Hashing (3)

- **Splitt**
  - Einfügen von 888
    erhöht Belegungsfaktor auf
    $\beta = 17/20 = 0{,}85$
  - Einfügen von 244, 399 und 100
    erhöht Belegungsfaktor auf
    $\beta = 20/24 = 0{,}83$
  - Auslösen eines weiteren Splitts:





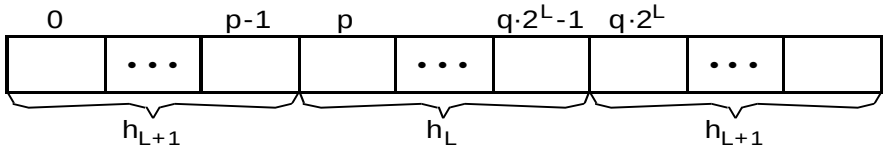
Lineares Hashing (4)

- **Splitt**
  - Auslöser: β
  - Position: p
  - Datei wird um 1 Bucket vergrößert
  - p wird inkrementiert: $p = (p+1) \bmod (2^L \cdot q)$
  - Wenn p wieder auf Null gesetzt wird (Verdopplung der Datei beendet), wird L inkrementiert.

- **Adressberechnung**
  - Wenn $h_0(k) \geq p$, dann ist $h_0(k)$ die gewünschte Adresse.
  - Wenn $h_0(k) < p$, dann ist Bucket bereits gesplittet. $h_1(k)$ liefert die gewünschte Adresse.
  - allgemein:

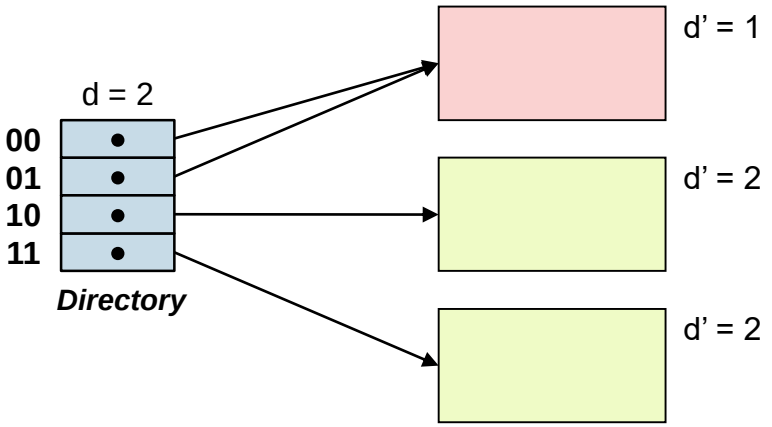
    $h := h_L(k);$

    if $h < p$ then $h := h_{L+1}(k);$



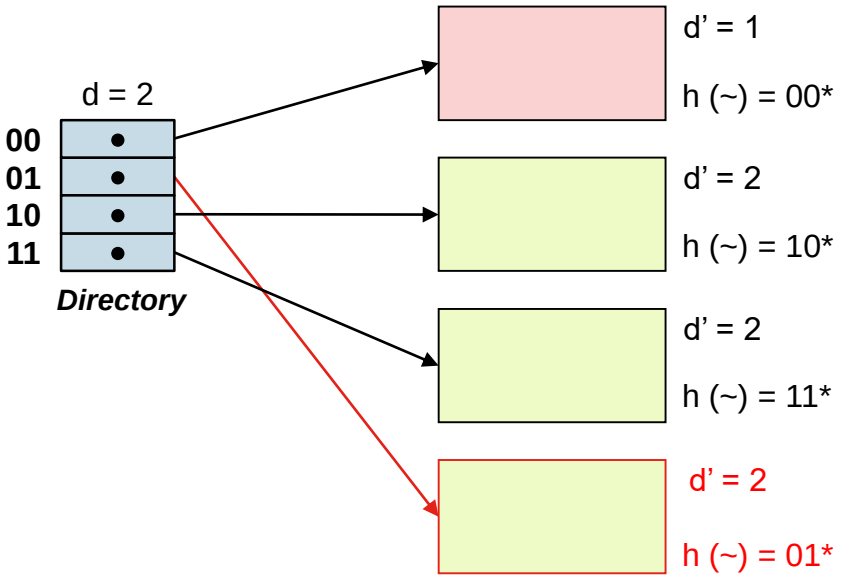
4.A.2 Erweiterbares Hash-Verfahren (Extendible Hashing)

- **Ebenfalls dynamisches Wachsen und Schrumpfen des Hash-Bereichs**
  - Buckets erst bei Bedarf bereitgestellt
  - Hohe Speicherplatzbelegung möglich
- **Keine Überlauf-Bereiche, jedoch Zugriff über Directory**
  - Zwei Blockzugriffe, 1 bis 2 E/A-Operationen
  - Hash-Funktion generiert Pseudoschlüssel zu einem Satz: Folge von Bits
  - Die ersten **d** Bits des Pseudoschlüssels werden zur Identifizierung eines Directory-Eintrags verwendet (d = globale Tiefe).
  - Directory enthält 2^d Einträge; Eintrag verweist auf Bucket.
  - In einem Bucket werden genau die Sätze gespeichert, deren Pseudoschlüssel in den ersten **d'** Bits übereinstimmen (d' = lokale Tiefe).
  - d = MAX (d')



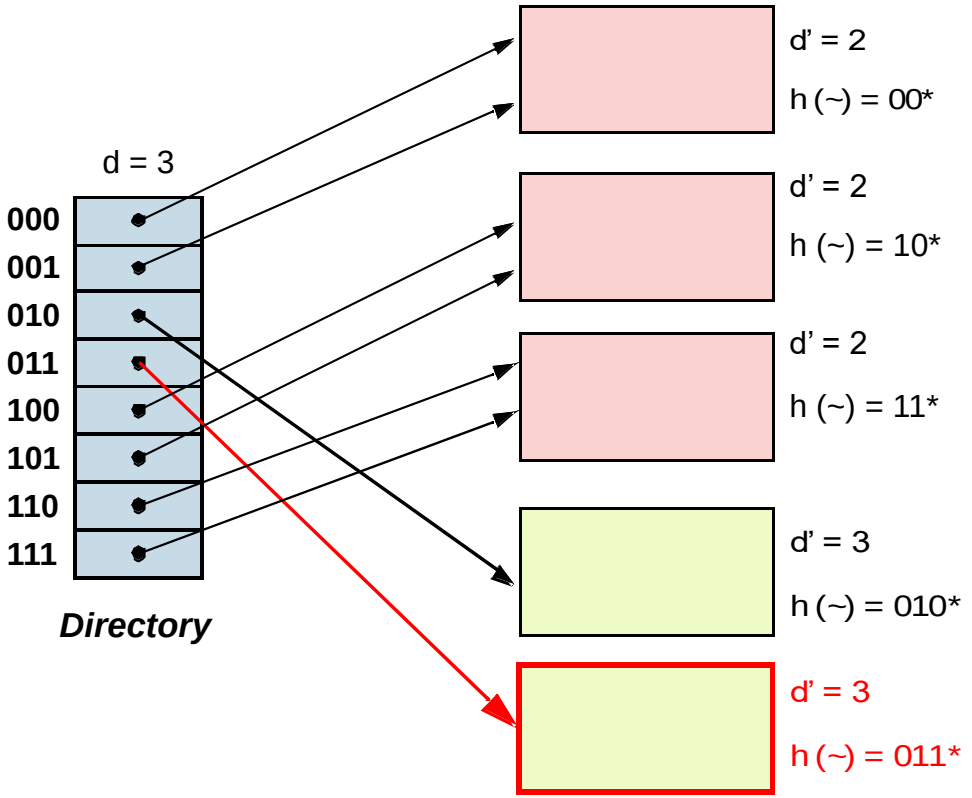
Erweiterbares Hashing (2)

- **Situation**
  - Neuer Satz soll gespeichert werden, aber Bucket ist voll: Überlauf
- **Fall 1**
  - Überlauf eines Buckets, dessen lokale Tiefe kleiner als die globale Tiefe d ist
    - D.h. mehrere Einträge im Directory zeigen auf diesen Bucket
    - Neuen Bucket anlegen
    - Erhöhung der lokalen Tiefe
    - Umverteilung der Sätze nach ihren Bits in der neuen Tiefe
    - Lokale Korrektur der Pointer im Directory



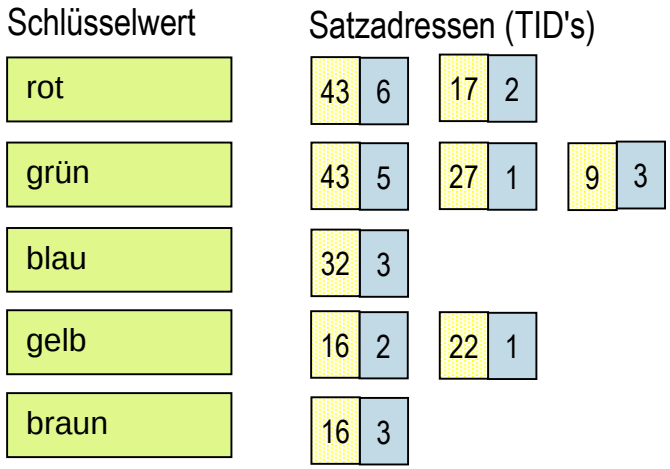
Erweiterbares Hashing (3)

- **Fall 2**
  - Überlauf eines Buckets, dessen lokale Tiefe gleich der globalen Tiefe ist
    - Verdopplung aller Einträge im Directory (Erhöhung der globalen Tiefe)
    - Neuen Bucket anlegen
    - Umverteilung der Sätze nach ihren Bits in der neuen Tiefe
    - Lokale Korrektur der Pointer im Directory



4.A.3 Invertierte Speicherung von Sätzen

- **Verwaltung einer Tabelle**
  - die zu jedem auftretenden Schlüsselwert **Liste von Satzadressen** verwaltet: genau die Sätze, in denen der Schlüsselwert vorkommt



- **Auch Index genannt**
  - Wie in einem Buch: Stichwort und Seitennummern

Invertierte Speicherung (2)

- **Hilfsstruktur**
  - Steht am Anfang/Ende der Datei (neuer Blocktyp)
    oder in eigener Datei
  - Erheblich kompakter als Menge aller Sätze,
    d.h. selbst bei sequenziellem Durchsuchen weniger Blöcke zu lesen
- **Sortierung der Tabelleneinträge nach Schlüsselwert**
  - verbessert Suche weiter
  - Binäre Suche allerdings erschwert durch variabel lange Einträge
- **Abspeicherung der Sätze selbst nicht reglementiert!**
  - Kann also mit gestreuter Speicherung nach einem anderen Schlüssel
    oder auch mit sequenzieller Abspeicherung kombiniert werden
- **Lesen aller Sätze sortiert nach Schlüssel möglich**

Invertierung (3)

- **Nachteile:**
  - Zusätzlicher Speicherplatzbedarf
  - Einfügen oder Löschen eines Satzes
    ändert die Zahl der Satzadressen in einem Eintrag
    - D.h. Verschieben der nachfolgenden Einträge
      (innerhalb des Blocks? über mehrere Blöcke hinweg?)
  - Neuer Satz kann sogar neuen Schlüsselwert mitbringen
    - Sortierordnung erhalten, neuen Eintrag einschieben
- **Also:**
  - Zugriff über Schlüssel effizienter als bei sequenziellem Suchen,
    aber Wartung der Invertierungstabelle bei Änderungen mühsam
- **Lösung:**
  - Hierarchische Strukturierung der Tabelle (Baum)