Datenbankanwendung

Wintersemester 2014/15

Prof. Dr.-Ing. Sebastian Michel
TU Kaiserslautern

smichel@cs.uni-kl.de

Hashverfahren

Idee

- Partitionierung des Raums
- Hashfunktion $h : S \rightarrow \{0, 1, 2, \ldots, n-1\}$
- Interpretation: Abbildung von Daten(raum) S auf "Buckets"
- Im DB Kontext werden diese Buckets auch Seiten genannt.
- $|S| >> n$

Bereits besprochene Anwendungen

- Zum effizienten Auffinden von Join-Partnern
- Als Indexstruktur, d.h. zur Organisation von Daten

Statische Hashverfahren

- Idealfall: h ist injektiv! Das heisst es gibt keine Kollissionen.
  $\rightarrow$ Jeder Datensatz kann mit nur einem Seitenzugriff gefunden werden.
- Leider nur in Ausnahmefällen realisierbar (wenn Schlüsselmenge dicht ist)

- Beispiel einer Seite (mit Nummer id), der Größe 4 und bereits 3 Einträgen (dargestellt durch deren Schlüssel, 15, 50 und 20).

id	
	15
	50
	20

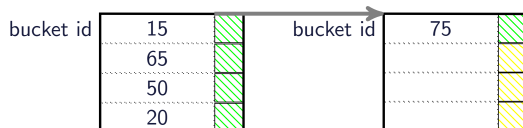
- Weiteres Einfügen in diese Seite wird sie komplett füllen

id	
	15
	50
	20
	85

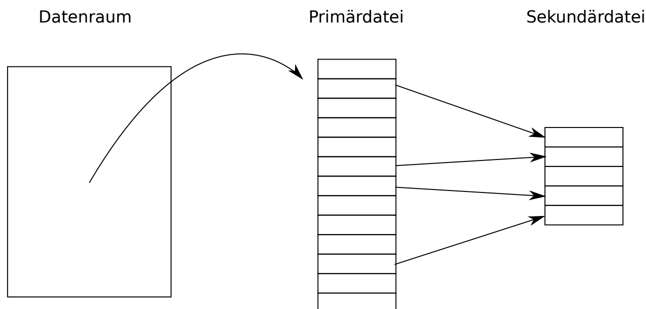
- Was geschieht bei weiteren Datensätzen, die laut Hashfunktion in diese Seite gehören?

Getrennte Behandlung von Überläufern

- Datensätze, die nicht mehr in eine bereits volle Seite eingefügt werden können, sogenannte Überläufer, werden in einer Seite einer Sekundärdatei abgelegt
- Eine Referenz von Primär- auf Sekundärseite wird eingefügt
- Entsprechend werden Überläufer der Überlaufsseiten behandelt

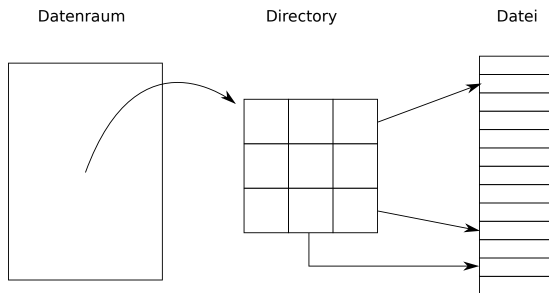


Klassifizierung von Ansätzen: Ohne Directory



- Wie zuvor
- Wenn Seite auf Primärdatei voll ist muss eine Überlaufseite in der Sekundärdatei angelegt und der entsprechende Datensatz dort gespeichert werden.
- Problem: Wenn es viele Datensätze in den Überlaufsseiten gibt, kann die Suchzeit stark zunehmen.

Klassifizierung von Ansätzen: Mit Directory



- Directory (Verzeichnis) wächst mit Anzahl der Datensätze
- Directory ist "Hilfsstruktur" zur Abbildung der Datensäte auf die Seiten der eigentlichen Datei
- D.h. es sind mindestens zwei Zugriffe notwending (einen auf Directory und einen auf die eigentliche Seite).
- Directory passt i.d.R. nicht in Hauptspeicher!

Statische Hashverfahren

Notation

- $N = \#$Sätze
- $n = n_p = \#$Seiten
- $n_s = \#$Sekundärseiten
- $b =$ Kapazität der Primärseiten
- $c =$ Kapazität der Sekundärseiten
  (Annahme: $c = b$)
- Speicherplatzausnutzung:

$$\alpha = \frac{N}{n_p \times b + n_s \times c}$$

Einfügen von 78, 84, 4, 7, 64, 9, 15, 0, 43, 50, 88, 73, 33, 92, 19, 6, 17, 20, 79, 26

Seite	Einträge
0	15, 0, 50
1	
2	7
3	78, 43
4	84, 4, 64, 9

- $S = \{0, \ldots, 99\}$
- #Seiten: $n = 5$
- $h(k) = k \bmod 5$
- $b = c = 4$

Einfügen von 78, 84, 4, 7, 64, 9, 15, 0, 43, 50, 88, 73, 33, 92, 19, 6, 17, 20, 79, 26

Bucket	Einträge
0	15, 0, 50
1	
2	7
3	78, 43, 88
4	84, 4, 64, 9

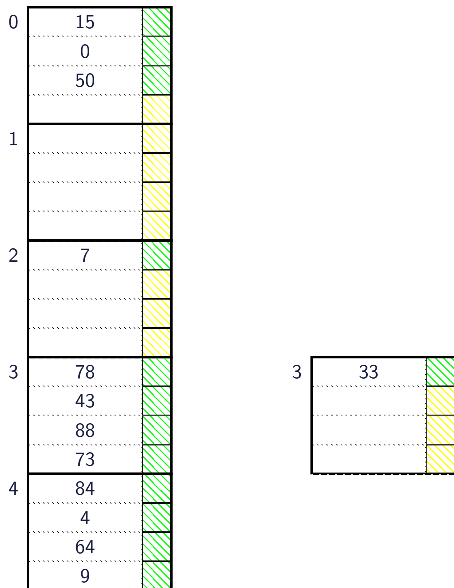
Einfügen von 88 macht keine Probleme: Bucket 88 mod 5 = 4 hat noch Platz

Einfügen von 78, 84, 4, 7, 64, 9, 15, 0, 43, 50, 88, 73, 33, 92, 19, 6, 17, 20, 79, 26

0	15
	0
	50
1	
2	7
3	78
	43
	88
	73
4	84
	4
	64
	9

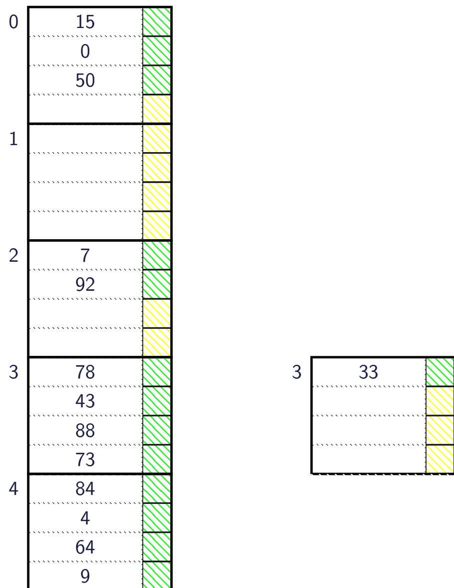
Einfügen von 73 macht keine Probleme: Bucket 73 mod 5 = 3 hat noch Platz

Einfügen von 78, 84, 4, 7, 64, 9, 15, 0, 43, 50, 88, 73, 33, 92, 19, 6, 17, 20, 79, 26



Einfügen von 33 macht Probleme: Bucket 33 mod 5 = 3 ist voll! Datensatz wird also in Seite in Sekundärdatei für Bucket 3 platziert.

Einfügen von 78, 84, 4, 7, 64, 9, 15, 0, 43, 50, 88, 73, 33, 92, 19, 6, 17, 20, 79, 26



Einfügen von 92 macht keine Probleme: Bucket 92 mod 5 = 2 hat noch Platz

Einfügen von 78, 84, 4, 7, 64, 9, 15, 0, 43, 50, 88, 73, 33, 92, 19, 6, 17, 20, 79, 26

Bucket				
0	15	0	50	
1				
2	7	92		
3	78	43	88	73
4	84	4	64	9

Bucket				
3	33			
4	19			

Einfügen von 19 macht Probleme: Bucket 19 mod 5 = 4 ist voll. Also Einfügen in Sekundärdatei für Bucket 4

Einfügen von 78, 84, 4, 7, 64, 9, 15, 0, 43, 50, 88, 73, 33, 92, 19, 6, **17, 20, 79, 26**

Bucket	Einträge
0	15, 0, 50
1	6
2	7, 92
3	78, 43, 88, 73
4	84, 4, 64, 9

Überlauf-Bucket	Einträge
3	33
4	19

Einfügen von 6 macht keine Probleme: Bucket 6 mod 5 = 1 hat noch Platz

Einfügen von 78, 84, 4, 7, 64, 9, 15, 0, 43, 50, 88, 73, 33, 92, 19, 6, 17, 20, 79, 26

Bucket	Primärbucket	Überlaufbucket
0	15, 0, 50	
1	6	
2	7, 92, 17	
3	78, 43, 88, 73	33
4	84, 4, 64, 9	19

Einfügen von 17 macht keine Probleme: Bucket 17 mod 5 = 2 hat noch Platz

Einfügen von 78, 84, 4, 7, 64, 9, 15, 0, 43, 50, 88, 73, 33, 92, 19, 6, 17, 20, 79, 26

Bucket	Einträge
0	15, 0, 50, 20
1	6
2	7, 92, 17
3	78, 43, 88, 73
4	84, 4, 64, 9

Überlauf-Bucket	Einträge
3	33
4	19

Einfügen von 20 macht auch keine Probleme.

Einfügen von 78, 84, 4, 7, 64, 9, 15, 0, 43, 50, 88, 73, 33, 92, 19, 6, 17, 20, 79, 26

Bucket	Primärdatei
0	15
	0
	50
	20
1	6
2	7
	92
	17
3	78
	43
	88
	73
4	84
	4
	64
	9

Bucket	Sekundärdatei
3	33
4	19
	79

Einfügen von 79 macht aber Probleme, denn Bucket 79 mod 5 = 4 ist voll. Also Einfügen in Sekundärdatei.

Einfügen von 78, 84, 4, 7, 64, 9, 15, 0, 43, 50, 88, 73, 33, 92, 19, 6, 17, 20, 79, 26

Bucket	Einträge
0	15, 0, 50, 20
1	6, 26, ,
2	7, 92, 17,
3	78, 43, 88, 73
4	84, 4, 64, 9

Überlauf-Bucket	Einträge
3	33, , ,
4	19, 79, ,

Einfügen von 26 macht keine Probleme, denn in Bucket 26 mod 5 = 1 ist noch Platz.

Überlaufbehandlung: Lineares Sondieren

aka. Linear Probing

Idee

- Überläufer werden in der Primärdatei abgelegt
- Soll ein Datensatz in eine bereits volle Seiten eingefügt werden, so wird linear nach der ersten nicht vollen Seite mit Adresse

  $(h(k) + 1)$ **mod** n

  gesucht. $i = 0, \ldots, n - 1$

Einfügen von 78, 84, 4, 7, 64, 9, 15, 0, 43, 50, 88, 73, 33, 92, 19, 6, 17, 20, 79, 26

Seite	Einträge
0	15, 0, 50
1	
2	7
3	78, 43
4	84, 4, 64, 9

- $S = \{0, \ldots, 99\}$
- #Seiten: $n = 5$
- $h(k) = k \textbf{ mod } 5$
- $b = c = 4$

Einfügen von 78, 84, 4, 7, 64, 9, 15, 0, 43, 50, 88, 73, 33, 92, 19, 6, 17, 20, 79, 26

Bucket	Einträge
0	15, 0, 50
1	
2	7
3	78, 43, 88
4	84, 4, 64, 9

Einfügen von 88 macht keine Probleme:
Bucket 88 mod 5 = 4 hat noch Platz

Einfügen von 78, 84, 4, 7, 64, 9, 15, 0, 43, 50, 88, 73, 33, 92, 19, 6, 17, 20, 79, 26

Bucket	Einträge
0	15, 0, 50
1	
2	7
3	78, 43, 88, 73
4	84, 4, 64, 9

Einfügen von 73 macht keine Probleme:
Bucket 73 mod 5 = 3 hat noch Platz

Einfügen von 78, 84, 4, 7, 64, 9, 15, 0, 43, 50, 88, 73, 33, 92, 19, 6, 17, 20, 79, 26

Bucket	Einträge
0	15
	0
	50
	33
1	
2	7
3	78
	43
	88
	73
4	84
	4
	64
	9

Einfügen von 33 macht Probleme: Bucket 33 mod 5 = 3 ist voll! Verschoben wird nach Bucket (3+1) mod 5 = 4, der aber auch voll ist. Nächster Versuch ist Bucket (3+2) mod 5 = 0, dort ist noch Platz!

Einfügen von 78, 84, 4, 7, 64, 9, 15, 0, 43, 50, 88, 73, 33, 92, 19, 6, 17, 20, 79, 26

Bucket	Einträge
0	15, 0, 50, 33
1	
2	7, 92
3	78, 43, 88, 73
4	84, 4, 64, 9

Einfügen von 92 macht keine Probleme:
Bucket 92 mod 5 = 2 hat noch Platz

Einfügen von 78, 84, 4, 7, 64, 9, 15, 0, 43, 50, 88, 73, 33, 92, 19, 6, 17, 20, 79, 26

Bucket	Wert
0	15
	0
	50
	33
1	19
2	7
	92
3	78
	43
	88
	73
4	84
	4
	64
	9

Einfügen von 19 macht Probleme: Bucket 19 mod 5 = 4 ist voll. Nächster Versuch ist Bucket (4+1) mod 5 = 0, der aber auch voll ist. In Bucket (4+2) mod 5 = 1 ist aber noch Platz.

Einfügen von 78, 84, 4, 7, 64, 9, 15, 0, 43, 50, 88, 73, 33, 92, 19, 6, 17, 20, 79, 26

Bucket	Einträge
0	15, 0, 50, 33
1	19, 6, ,
2	7, 92, ,
3	78, 43, 88, 73
4	84, 4, 64, 9

Einfügen von 6 macht keine Probleme: Bucket 6 mod 5 = 1 hat noch Platz

Einfügen von 78, 84, 4, 7, 64, 9, 15, 0, 43, 50, 88, 73, 33, 92, 19, 6, 17, 20, 79, 26

Bucket	Wert
0	15
	0
	50
	33
1	19
	6
2	7
	92
	17
3	78
	43
	88
	73
4	84
	4
	64
	9

Einfügen von 17 macht keine Probleme:
Bucket 17 mod 5 = 2 hat noch Platz

Einfügen von 78, 84, 4, 7, 64, 9, 15, 0, 43, 50, 88, 73, 33, 92, 19, 6, 17, 20, 79, 26

Bucket	Eintrag
0	15
	0
	50
	33
1	19
	6
	20
2	7
	92
	17
3	78
	43
	88
	73
4	84
	4
	64
	9

Einfügen von 20 macht Probleme: Bucket 20 mod 5 = 0 ist voll. Eingefügt wird dann in Bucket (0+1) mod 5 = 1, wo noch Platz war.

Einfügen von 78, 84, 4, 7, 64, 9, 15, 0, 43, 50, 88, 73, 33, 92, 19, 6, 17, 20, 79, 26

Bucket	Eintrag
0	15
	0
	50
	33
1	19
	6
	20
	79
2	7
	92
	17
3	78
	43
	88
	73
4	84
	4
	64
	9

Einfügen von 79 macht auch Probleme: Bucket 79 mod 5 = 4 ist voll, ebenso wie im zweiten Versuch für Bucket (4+1) mod 5 = 0. Bucket (4+2) mod 5 = 1 hat aber noch Platz.

Einfügen von 78, 84, 4, 7, 64, 9, 15, 0, 43, 50, 88, 73, 33, 92, 19, 6, 17, 20, 79, 26

Bucket	Wert
0	15
	0
	50
	33
1	19
	6
	20
	79
2	7
	92
	17
	26
3	78
	43
	88
	73
4	84
	4
	64
	9

Einfügen von 26 macht Probleme: Bucket 26 mod 5 = 1 ist voll. Aber in Bucket (1+1) mod 5 = 2 ist noch Platz.

Überlaufbehandlung: Zufälliges Sondieren

Idee

- Wie bei linearem Sondieren: Überläufer werden in der Primärdatei abgelegt.
- Aber hier: Ein Zufallszahlengenerator bestimmt in Abhängigkeit zum Schlüssel k eine Folge von Seitenadressen

Dynamische Hashverfahren

Beobachtung und Ziel

- Performanz (Zugriffszeit, Speicherplatzausnutzung) kann leicht degenerieren
- In diesem Fall müsste Anzahl der Seiten dynamisch angepasst werden
- Dies ist bei den zuvor betrachteten statischen Ansätzen aber nicht möglich

Beispiel:

- $h(k) = k \textbf{ mod } 5$ sei die bisherige Hashfunktion
- wenn wir nun auf $h(k) = k \textbf{ mod } 6$ wechseln würden, also dann 6 statt 5 Buckets zur Verfügung haben, müssen Datensätze neu indexiert werden, d.h. globale Reorganisation (teuer!)
- siehe Illustration/Beispiel auf nächster Folie

Änderung der Hashfunktion/Parameter

Jeweils Primär- und Sekundärdateien für versch. Hashfunktionen

$h(k) = k \ \textbf{mod} \ 5$

Bucket	Primär	Überlauf
0	15, 0, 50, 20	
1	6, 26	
2	7, 92, 17	
3	78, 43, 88, 73	33
4	84, 4, 64, 9	19, 79

$h(k) = k \ \textbf{mod} \ 6$

Bucket	Primär	Überlauf
0	78, 84, 0, 6	
1	7, 43, 73, 19	79
2	50, 92, 20, 26	
3	9, 15, 33	
4	4, 64, 88	
5	17	

Dynamische Hashverfahren

Ziel

- Anpassung der Konfiguration ohne globale Reorganisation

Verschiedene Ansätze

- Erweiterbares Hashing (Extendible Hashing)
- Lineares Hashing
- Spiral-Hashing

Erweiterbares Hashing

- Verfahren mit $b = 0$, d.h. Primärdatei degeneriert zu einem Directory. Eine Primärseite ist ein Eintrag im Directory.

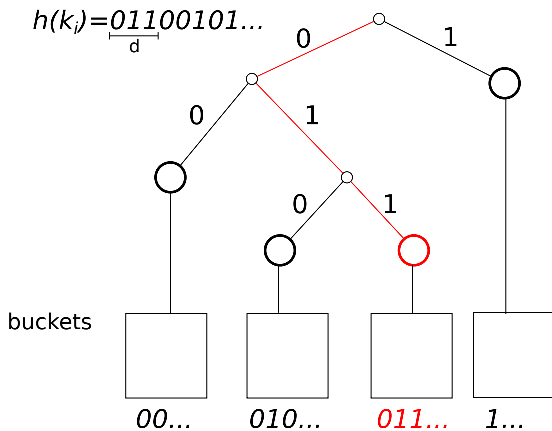
Hashfunktion

- $h_d(k)$, mit $d \geq 0$
- Für Schlüssel k wird ein Pseudoschlüssel $(b_0, b_1, b_2, \ldots)$ generiert, $b_j \in \{0, 1\}$
- z.B. 010001000101111
- Die ersten d Bits des Pseudoschlüssel s werden zur Berechnung der Adresse verwendet.

Paper: R. Fagin et al. Extendible hashing—a fast access method for dynamic files, 1979.

Veranschaulichung

- Interpretation als digitaler Baum der Höhe d.
- d Bits werden benötigt bis ein Bucket den gesamten Teilbaum umfassen kann



Erweiterbares Hashing ist ein digitaler Baum der Ordnung (2^d) der Höhe 1.

Directory

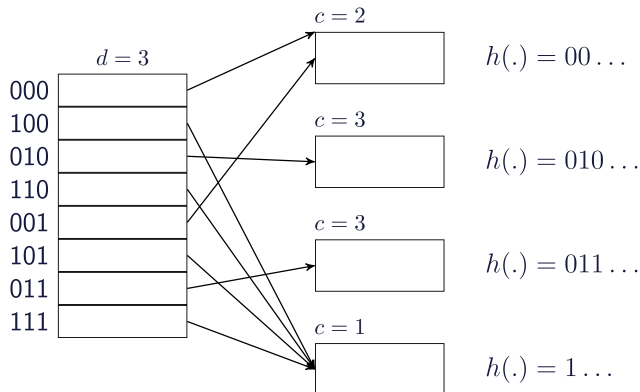
$d = 3$

000	
100	
010	
110	
001	
101	
011	
111	

- Es gibt 2^d Einträge im Direcotry.
- Parameter d wird **globale Tiefe** genannt.
- Eintrag im Directory verweist auf Seite, in der alle dazugehörigen Datensätze gespeichert sind
- Mehrere Einträge können dabei auch auf die gleiche Seite verweisen!

- Es gibt keine Überläufer
- Der Zugriff auf die (exakten) Buckets erfolgt über das Directory (extra Zugriff)

Beispiel



- c wird **lokale Tiefe** genannt
- In einer Seite können nur Datensätze gespeichert werden, die in den ersten c Bits ($c \leq d$) übereinstimmen

Anpassung bei vollen Seiten

Für den Fall, dass eine Seite voll ist oder zu einem bestimmten (vorher spezifizierten) Grad gefüllt ist.

Fall: Seite mit $c < d$ ist voll

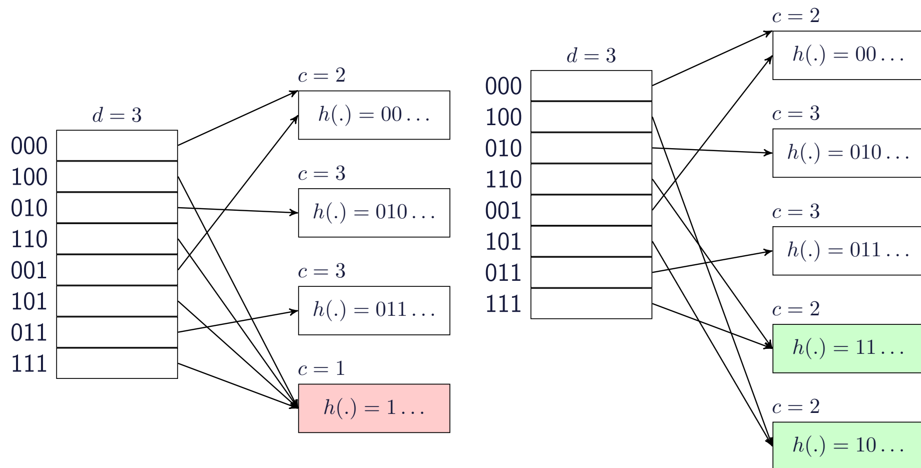
- Seite wird aufgespalten in zwei Seiten mit je $c + 1$ lokaler Tiefe
- Das hinzugekommene Bit teilt die Datensätze der aufzuspaltenen Seite; z.B. Datensätze mit Bit 0 bleiben in Seite, Datensätze mit Bit 1 werden in neue Seite verschoben.

Fall: Seite mit $c = d$ ist voll

- Das globale Directory wird verdoppelt ($d := d + 1$), dann gilt wieder $c < d$ und somit kann die volle Seite dann in zwei Seiten aufgeteilt werden.

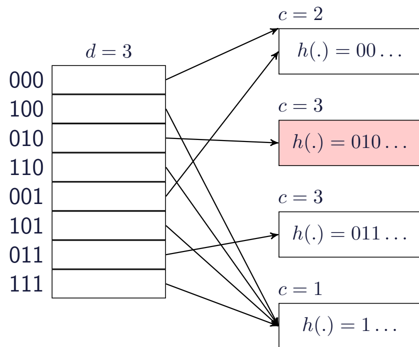
Anpassung bei vollen Seiten(Fall: $c < d$)

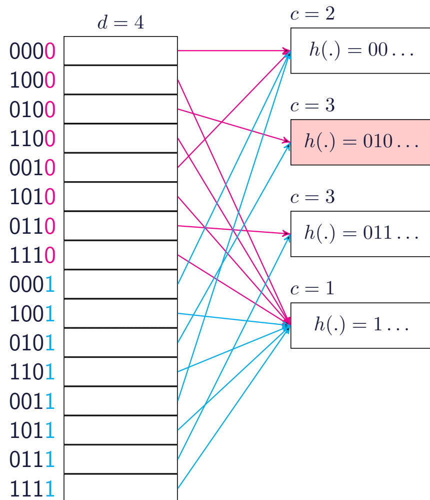
Für den Fall $c < d$. Links: alter Index. Rechts: Index nach Anpassung (Split). Rot=volle Seite. Grün=resultierende Seiten nach Split.



Anpassung bei vollen Seiten (Fall: $c = d$)

Für den Fall $c = d$. Links: alter Index. Rechts: Index nach Verdopplung. Rot=volle Seite. Danach gilt wieder lokale Tiefe < globale Tiefe



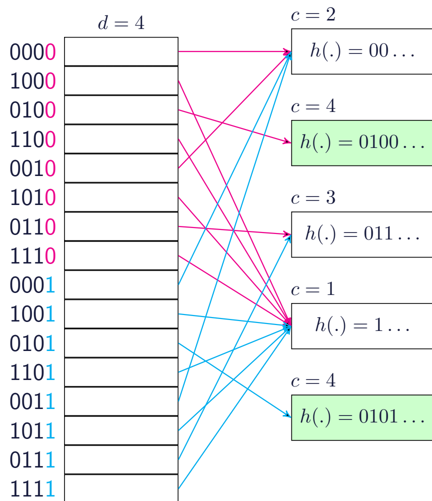


Anpassung bei vollen Seiten (Fall: $c = d$) (Fortsetzung)

Und nun?

- Wir haben das Directory verdoppelt
- Aber keine Seite gesplittet (Wie man sieht gibt es auf jede Seite nun doppelt so viele Zeiger aus dem Directory).
- Die Seite, die voll war, ist immer noch voll!
- Nun haben wir aber (durch die Verdopplung) eine globale Tiefe von $d = 4$ (und nicht mehr $d = 3$),
- Das heißt wir können nun die Seite splitten (da nun $c < d$)

… und nun der darauf folgende Split



Anmerkungen zu Splits

- Abhängig von den Daten (bzw. deren Hashwert (Bits)) kann es passieren, dass obwohl man mehr Bits betrachtet (global bzw. lokal) ein Split nicht dazu führt, dass die Daten der zu teilenden Seite gleichmässig aufgeteilt werden.
- In diesem Fall kann es passieren, dass nach dem Split einer der beiden Seiten immer noch voll ist
- Dann muss ein weiterer Split-Prozess durchgeführt werden.
- Solange bis es keine zu volle Seite mehr gibt.

Eigenschaften des Erweiterbaren Hashings

- Direktory ist (bzw. kann) sehr groß werden
- Es kann daher in der Regel nicht im Hauptspeicher gehalten werden
- Jeder Datensatz kann garantiert mit 2 Zugriffen gefunden werden
- Keine Unterstützung von Bereichsanfragen (Range Queries)
- Speicherplatzausnutzung ist $ln\,2 \approx 0.69$ (unter der Anname, dass die Pseudoschlüssel gleichverteilt sind)

Lineares Hashing (LH)

- Verfahren mit $b > 0$, d.h. es gibt eine Primärdatei und eine Sekundärdatei, also kein Verfahren mit Directory
- Primärdatei besteht am Anfang aus N Datenseiten

Idee

- Zu Beginn ist eine bestimmte Anzahl N von Hashbuckets gegeben
- Der Reihe nach werden diese gesplittet (round robin),
- Nach einem Durchlauf dieser Splits hat sich die Anzahl der Seiten verdoppelt.
- "Schwierigkeit": Bestimmung des Zeitpunktes der Splits und Berechnung von Seiten(adressen)

Lineares Hashing

Hashfunktionen

- Folge von Hash-Funktionen $\{h_j(k)\}_{j \geq 0}$ mit den folgenden Bedingungen:
- **Bereichsbedingung:**

$$h_j : \mathcal{D} \rightarrow \{0, 1, \ldots, 2^j N - 1\}, \text{ mit } j \geq 0$$

- **Splitbedingung:**

$$h_{j+1}(k) = h_j(k) \text{ oder}$$

$$h_{j+1}(k) = h_j(k) + 2^j N \text{ mit } j \geq 0$$

Durch L wird angegeben wie oft sich die Datei bereits verdoppelt hat. Je nach L und aktuellem Stand (p) der Splits wird dann Hashfunktion h_L oder h_{L+1} benutzt.

Beispiel

$h_j(k) = k \textbf{ mod } (2^j N)$ erfüllt beide Bedingungen

Belegungsfaktor

$$\beta = \frac{x}{b \times M}$$

Mit

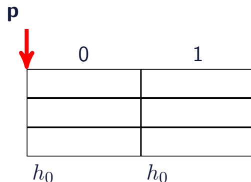
- x: Anzahl der Datensätze
- b: Kapazität eines Buckets (Seite) in Anzahl der Datensätze die hinein passen
- M: Anzahl der Buckets der Primärdatei

- Der Belegungsfaktor kann benutzt werden um Splits zu triggern, durch Schwellwert (threshold).
- Je grösser dieser Schwellwert, desto mehr Überläufer gibt es.

Beispiel

- $N = 2$
- $b = 3$
- $\beta_s = 0.8$
- $h_0(k_i) = k_i$ **mod** $(2^0 N)$

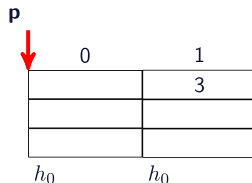
Einfügesequenz: 3, 5, 7, 13, 10



Beispiel

- $N = 2$
- $b = 3$
- $\beta_s = 0.8$
- $h_0(k_i) = k_i$ **mod** $(2^0 N)$

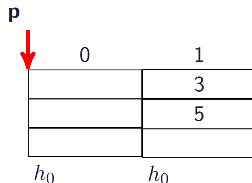
Einfügesequenz: 3, 5, 7, 13, 10



Beispiel

- $N = 2$
- $b = 3$
- $\beta_s = 0.8$
- $h_0(k_i) = k_i$ **mod** $(2^0 N)$

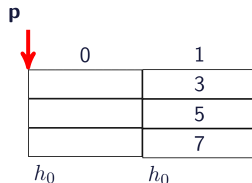
Einfügesequenz: 3, 5, 7, 13, 10



Beispiel

- $N = 2$
- $b = 3$
- $\beta_s = 0.8$
- $h_0(k_i) = k_i$ **mod** $(2^0 N)$

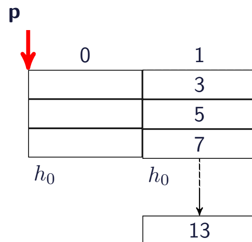
Einfügesequenz: 3, 5, 7, 13, 10



Beispiel

- $N = 2$
- $b = 3$
- $\beta_s = 0.8$
- $h_0(k_i) = k_i$ **mod** $(2^0 N)$

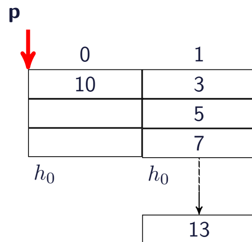
Einfügesequenz: 3, 5, 7, 13, 10



Beispiel

- $N = 2$
- $b = 3$
- $\beta_s = 0.8$
- $h_0(k_i) = k_i$ **mod** $(2^0 N)$

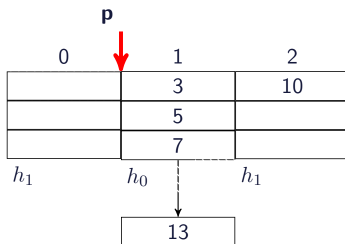
Einfügesequenz: 3, 5, 7, 13, 10



$$\beta = \frac{5}{6} = 0.83$$

Beispiel (2)

- Split der von p referenzierten Seite wurde ausgeführt
- Der Datensatz mit Schlüssel 10 wurde in Seite 2 verschoben
- Seite 1 bleibt unberührt.
- Wir sehen, dass nun für Seite 0 die Hashfunktion h_1 benutzt werden muss, damit die Hashfunktion auch die neue Seite 2 berücksichtigen kann.
- Für Seite 1 wird weiterhin die Hashfunktion h_0 benutzt



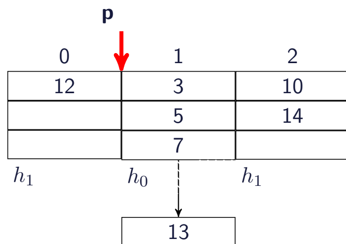
Addressberechnung

- Falls $h_0(k) \geq p$, dann ist $h_0(k)$ die gewünschte Adresse
- Andernfalls (d.h. $h_0(k) < p$), ist die Seite bereits aufgespalten worden und $h_1(k)$ liefert die gewünschte Adresse
- Allgemein:

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

Beispiel (3)

Nach Einfügen von: 12, 14, 15



$$\beta = \frac{8}{9} = 0.88$$

Erklärung zu Mapping der Schlüssel 12 und 14: Beide gehören laut h_0 in Seite 0, aber Achtung, $0 < p$, d.h. wir sehen (offensichtlich), dass diese Seite bereits gesplittet wurde. Wir müssen daher h_1 als Hashfunktion benutzen: $h_1(k) = k \textbf{ mod } (2^1 N) = k \textbf{ mod } (4)$. Für Bucket 1 auch?

Beispiel (4)

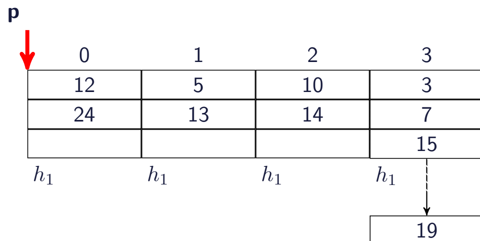
Nun ist die erste komplette Verdopplung vollzogen. D.h. h_1 wird für alle Buckets benutzt.

p ↓ (zeigt auf Bucket 0)

0	1	2	3
12	5	10	3
	13	14	7
			15
h_1	h_1	h_1	h_1

Beispiel (5)

Nach Einfügen von: 19, 24



$$\beta = \frac{10}{12} = 0.83$$

Aufspalten von Seiten

Lineares Hashing löst ein Aufspalten einer Seite aus, wenn eine bestimmte Bedingung (Kriterium) erfüllt ist.

- Auch Kontrollfunktion genannt
- Beispiel: Belegungsfaktor $> 80\%$, dann führe Aufteilen einer Seite aus
- Welche Seite? Es gibt einen **Zeiger** p, der auf die Seite verweist, die als nächstes aufgespaltet werden soll

Parameter / Status

- L: Level (Anzahl bereits ausgeführter Verdopplungen)
- p: Verweise auf die nächste zu teilende Seite
- β: Belegungsfaktor
- N: Anzahl Seiten zu Beginn
- b: Kapazität einer Primärseite
- c: Kapazität einer Sekundärrseite

Split und Splitstrategien

Split

- Bedingung für Split β_s und Belegungsfaktor β
- Aktuelle Position des Zeigers: p
- Datei wird um eine Seite vergrössert
- p wird weiter gesetzt, d.h. $p := (p + 1)$ mod $(N \times 2^L)$
- Falls p wieder auf Null (0) steht, d.h. die Verdoppelung der Datei abgeschlossen ist, so wird L um 1 erhöht

Split und Splitstrategien (2)

Splitstrategien

- **Unkontrolliertes Splitting:**
  - Splitting sobald ein Datensatz in den Überlaufsbereich aufgenommen wird
  - $\beta \sim 0.6$, schnelleres Aufsuchen
- **Kontrolliertes Splitting:**
  - Splitting, wenn ein Datensatz in den Überlaufbereich kommt und $\beta > \beta_s$
  - $\beta \sim \beta_s$, längere Überlaufketten möglich