Datenbankanwendung

Wintersemester 2014/15

Prof. Dr.-Ing. Sebastian Michel
TU Kaiserslautern

smichel@cs.uni-kl.de

Indexstrukturen insbesondere Mehrdimensionale Indexstrukturen, Ausgedehnte Objekte, Ähnlichkeitssuche

Grundidee eines Index

Abbildung: Schlüssel $\rightarrow$ Menge von Einträgen

Beispiele:

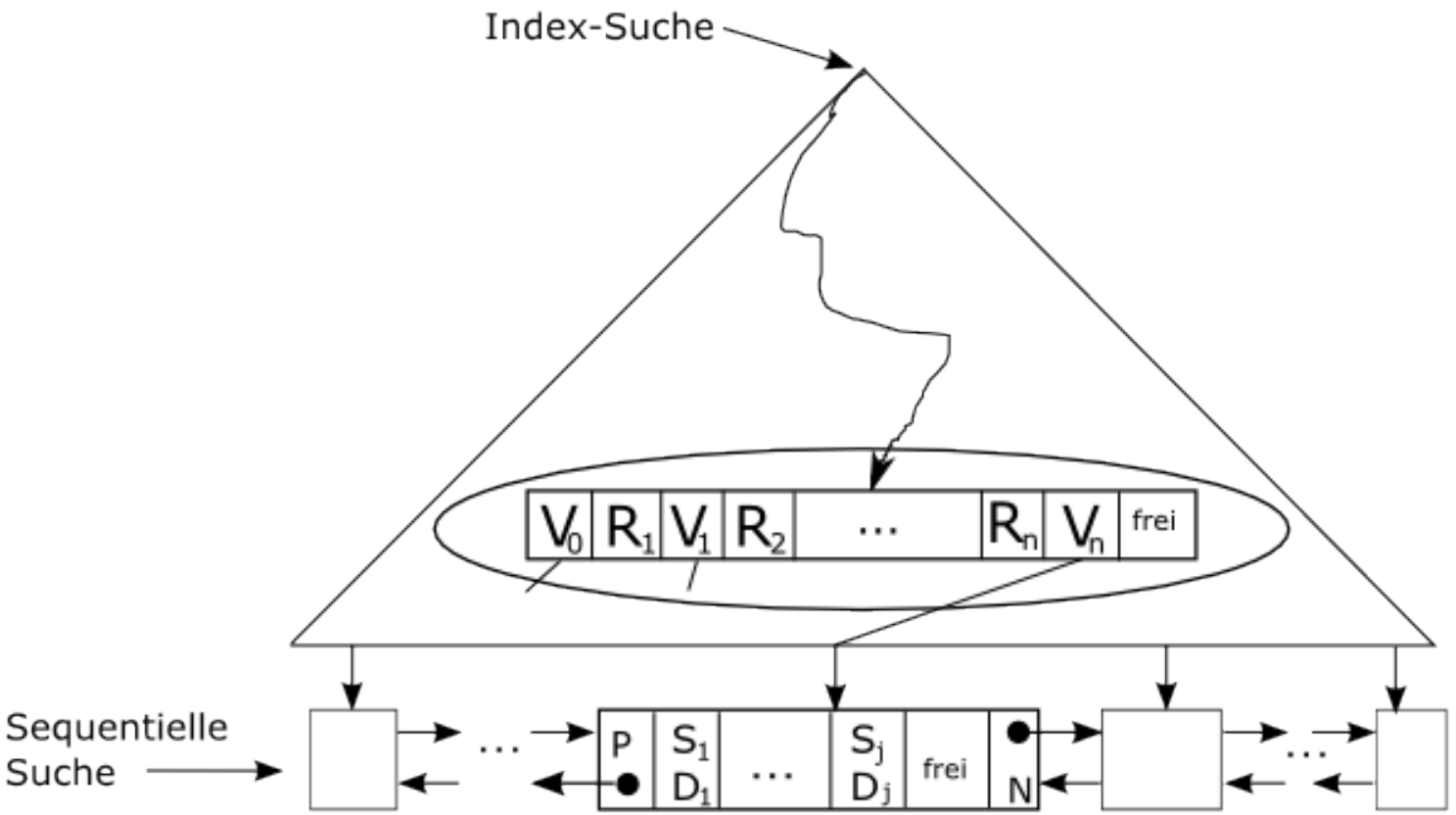
- MatrikelNummer $\rightarrow$ persönliche Daten des Studenten
- PLZ $\rightarrow$ Name und andere Informationen einer Stadt
- Term $\rightarrow$ alle Dokumente in denen diesesr Term enthalten ist

Natürlich möchte man bei diesen häufig auftretenden Anfragen die Antwortzeit gering halten.

Dafür wird ein Index angelegt: Ein Index materialisiert diese Abbildung!

B+ Baum

- "Hohler" Baum: Daten nur in den Blättern
- Suche muss also immer bis zu den Blättern laufen
- Aufbau: Referenzschlüssel R_j, Schlüssel S_k, Daten D_i, Zeiger V_m
- Blattknoten sequentiell verbunden!



B+ Baum: Eigenschaften

Ein B+ Baum vom Typ $(k, k*)$ hat folgende Eigenschaften:

1. Jeder Weg von der Wurzel zu einem Blatt hat die gleiche Länge
2. Jeder Knoten - außer Wurzeln und Blättern- hat mindestens k und höchstens $2k$ Einträge. Blätter haben mindestens $k*$ und höchstens $2k*$ Einträge. Wurzel hat entweder maximal $2k$ oder sie ist ein Blatt mit maximal $2k*$ Einträgen.
3. Jeder Knoten mit n Einträgen, außer den Blättern, hat $n + 1$ Kinder.

B+ Baum: Eigenschaften

4. Seien $R_1, ..., R_n$ die Referenzschlüssel eines inneren Knotens (d.h. auch der Wurzel) mit $n+1$ Kindern. Seien $V_0, V_1, ..., V_n$ die Verweise auf diese Kinder.
   (a) V_0 verweist auf den Teilbaum der Schlüssel kleiner R_1
   (b) V_i $(i = 1, ..., n-1)$ verweist auf einen Teilbaum, dessen Schlüssel zwischen R_i und R_{i+1} liegen (einschließlich R_i).
   (c) V_n verweist auf den Teilbaum mit Schlüsseln größer oder gleich R_n ist.

B* Baum

Statt wie im B+ Baum eine volle Seite auf zwei Seiten aufzuteilen, werden im B* Baum die Datensätze aus m Geschwisterknoten gleichmässig auf $(m+1)$ Seiten aufgeteilt.

Dies verbessert die Speicherplatzausnutzung (SPAN):

SPAN	$m=1$	$m=2$	m
worst case:	$\frac{1}{1+1}$	$\frac{2}{2+1}$	$\frac{m}{m+1}$
avg. case:	$ln2\ (=69\%)$	81%	$m * ln(\frac{m+1}{m})$

Allerdings werden die Kosten für das Einfügen erhöht, so dass in der Praxis $m \leq 3$.

Präfix B-Baum

Beobachtung

- Die Einträge in den inneren Knoten eines B+ Baums werden nur zur Navigation während der Suche benutzt.
- Einträge bestehen aus Verweisen (Pointern) und aus Schlüsseln.
- Pointer sind nur ein paar Bytes groß, aber Schlüssel können sehr lang sein (z.B. Donaudampfschifffahrtsgesellschaftskapitän)
- Wenn Platz gespart sind, passen mehr Einträge in einen Knoten. D.h. Baum wird flacher!

Idee

- Verwende nicht die gesamten Schlüssel, wie sie im Fall eines B+ Baums anfallen würden, sonder geeignete Präfixe.
- Ein Präfix B Baum ist ein B+ Baum, bei dem der Index-Teil des B+ Baums durch einen Index-Teil Präfix B-Baum ersetzt wurde.

Literatur: R. Bayer und K. Unterauer. Prefix B-trees. TODS, 1977.

Beispiel Split und Separator

Gegeben folgender volle Blattknoten eines B+ Baums:

Bigbird, Burt, Cookiemonster, Ernie, Snuffleopogus

Um den Schlüssel "Grouch" in diese Seite einzufügen, muss sie wie folgt in zwei Seiten aufgespalten werden:

Bigbird, Burt, Cookiemonster

Ernie, Grouch, Snuffleopogus

Statt nun "Ernie" als Schlüssel im Index zu benutzen, würde es auch ausreichen "D" oder "E" für den gleichen Zweck zu benutzen.

Separator

Im Allgemeinen kann man jeden String s benutzen, falls er die Eigenschaft hat, dass

$$\text{Cookiemonster} < s \leq \text{Ernie}$$

s kann dann im Index gespeichert werden, um die obigen beiden Seiten zu trennen.

Idee

Wenn man den möglichst kürzesten Separator benutzt, geht nur ein Minimum an Platz innerhalb einer Seite verloren. Der Baum wird flacher (weil höherer "Fan-Out")

Präfix-Eigenschaft

- Sind die Schlüssel Worte über einem Alphabet und ist die Ordnung der Schlüssel in alphabetischer (lexikographischer) Ordnung, dann gilt folgende Eigenschaft:

Präfix-Eigenschaft

Seien x und y zwei Schlüssel, so dass $x < y$. Dann gibt es einen eindeutigen Präfix $\hat{y}$ von y mit:

(a) $\hat{y}$ ist ein Separator zwischen x und y und

(b) kein anderer Separator zwischen x und y ist kürzer als $\hat{y}$.

Es kann auch Sinn machen Index-Knoten nicht genau in der "Mitte" zu splitten, sondern leichte Abweichung davon zuzulassen. Wieso? Was ist ein guter Separator zwischen Donaudampfschifffahrtsdampferhersteller und Donaudampfschifffahrtsgesellschaft?

Präfix-Suffix-Komprimierung

- Fortlaufende Kompression der Schlüssel innerhalb einer Seite, so dass nur der Teil des Schlüssels
  - vom Zeichen, in dem er sich vom Vorgänger unterscheidet (Position V),
  - bis zum Zeichen, in dem er sich vom Nachfolger unterscheidet (Position N)

  zu übernehmen ist.
- Eintrag enthält
  - Anzahl $F = V - 1$ der Zeichen des Schlüssels, die mit Vorgänger übereinstimmen (beim ersten Eintrag ist $F = 0$)
  - Länge $L = max(N - V, 0) + 1$ des komprimierten Schlüssels
  - dazugehörige Zeichenfolge
- Suchalgorithmus benutzt Stack
- Schlüssel können nicht vollständig, sondern nur noch bis zur Eindeutigkeitslänge rekonsstruiert werden.

Beispiel

Schlüssel	V	N	F	L	Komprimat
City_of_New_Orleans	1	6	0	6	City_o
City_to_City	6	2	5	1	t
Closet_Chronicles	2	2	1	1	l
Cocaine	2	3	1	2	oc
Cold_as_Ice	3	6	2	4	ld_a
Cold_Wind_to_Walhalla	6	4	5	1	w
Colorado	4	5	3	2	or
Colours	5	3	4	1	u
Come_Inside	3	12	2	10	me_Inside
Come_Inside_of_my_Guitar	12	6	11	1	_
Come_on_over	6	6	5	1	o
Come_together	6	1	5	1	t

Bulkloading

Problemstellung

- Gegeben eine große Menge an Datensätzen
- Wie kann ein B+ Baum darüber aufgebaut werden?
- Das Problem ist natürlich allgemeiner und tritt auch bei anderen Bäumen/Indexstrukturen auf.

Naive Möglichkeit

- Datensätze einzeln nach und nach in B+ Baum einfügen
- D.h. es wird immer von der Wurzel ab nach der passenden Einfügestelle gesucht
- Nicht sehr effizient (auch wenn die oberen Ebenen in einem DB-Puffer sind)

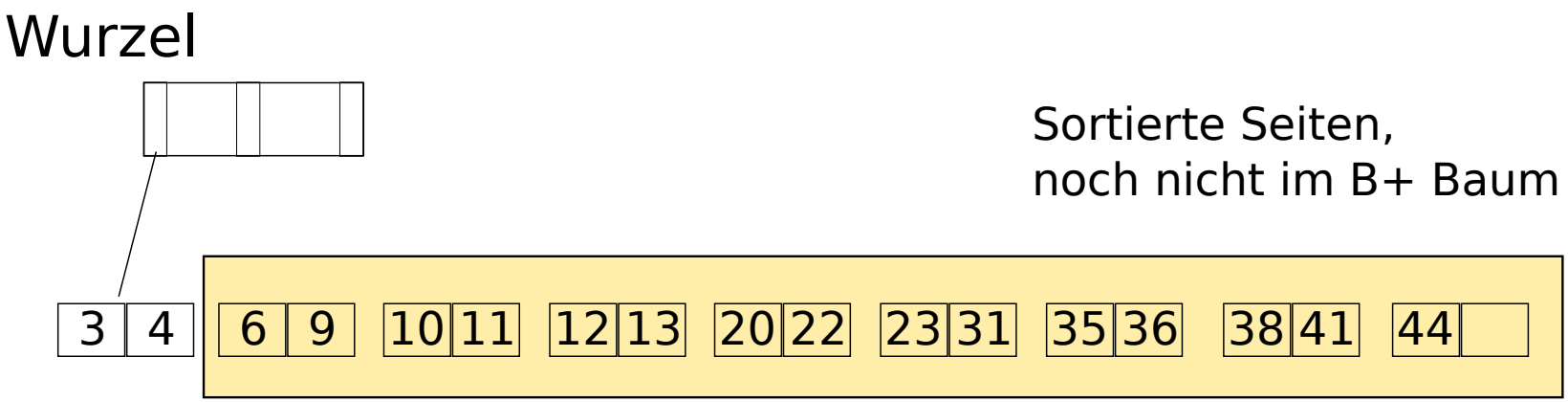
Bulkloading

Idee

- Sortiere Daten vor
- Dann baue Baum auf, indem Datensätze der Reihe nach hinzugeügt werden

Ablauf

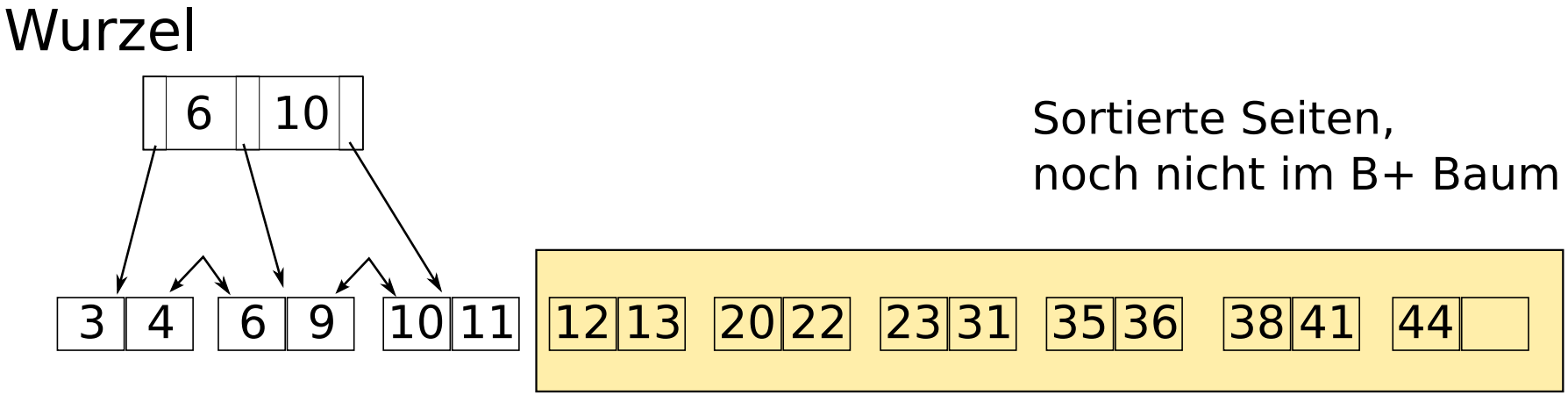
Der B+ Baum kann 2 Einträge pro Seite speichern. Wir fügen je ganze Seiten ein, hier zuerst die Seite mit Einträgen 3 und 4.



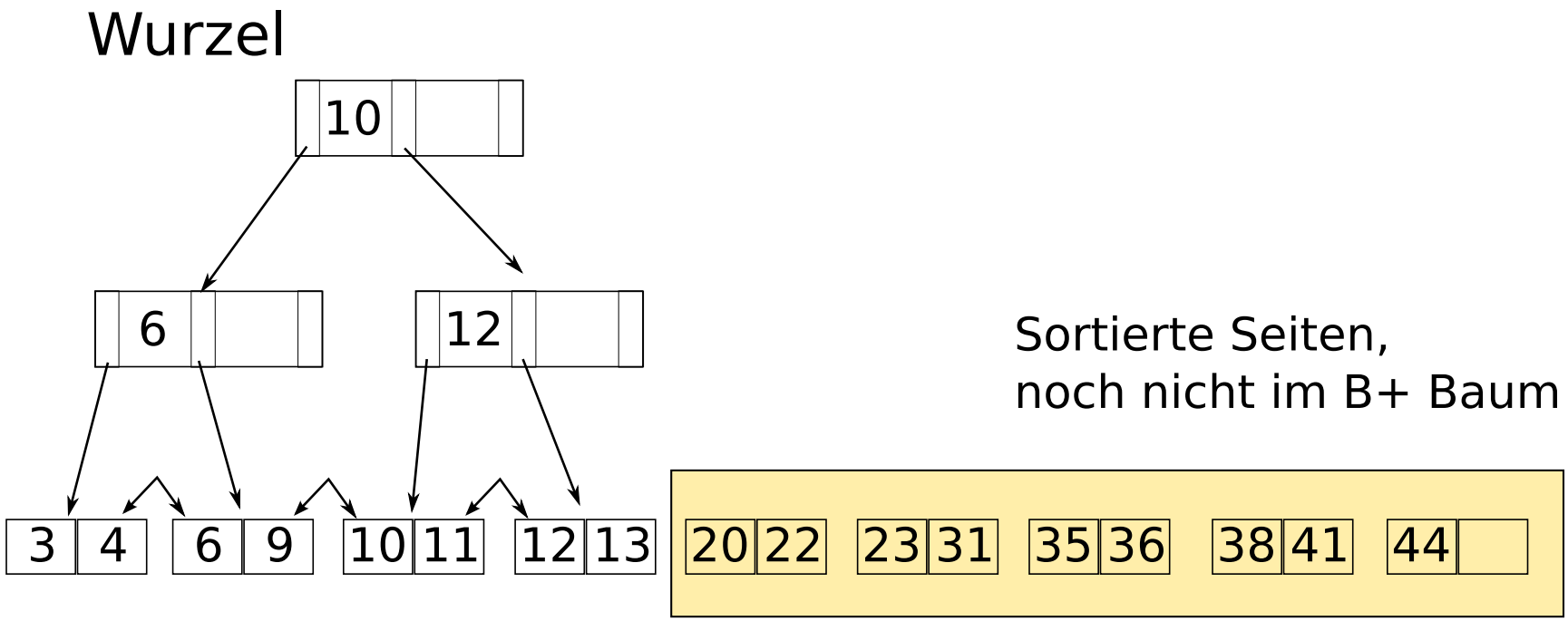
Buch: Ramakrishnan & Gehrke. Database Management Systems.

Bulkloading - Beispiel (2)

- Nachdem die beiden folgenden Seiten eingefügt worden sind ist die Wurzel nun voll.
- Bei dem Einfügen von (12,13) muss die Wurzel also aufgeteilt werden.



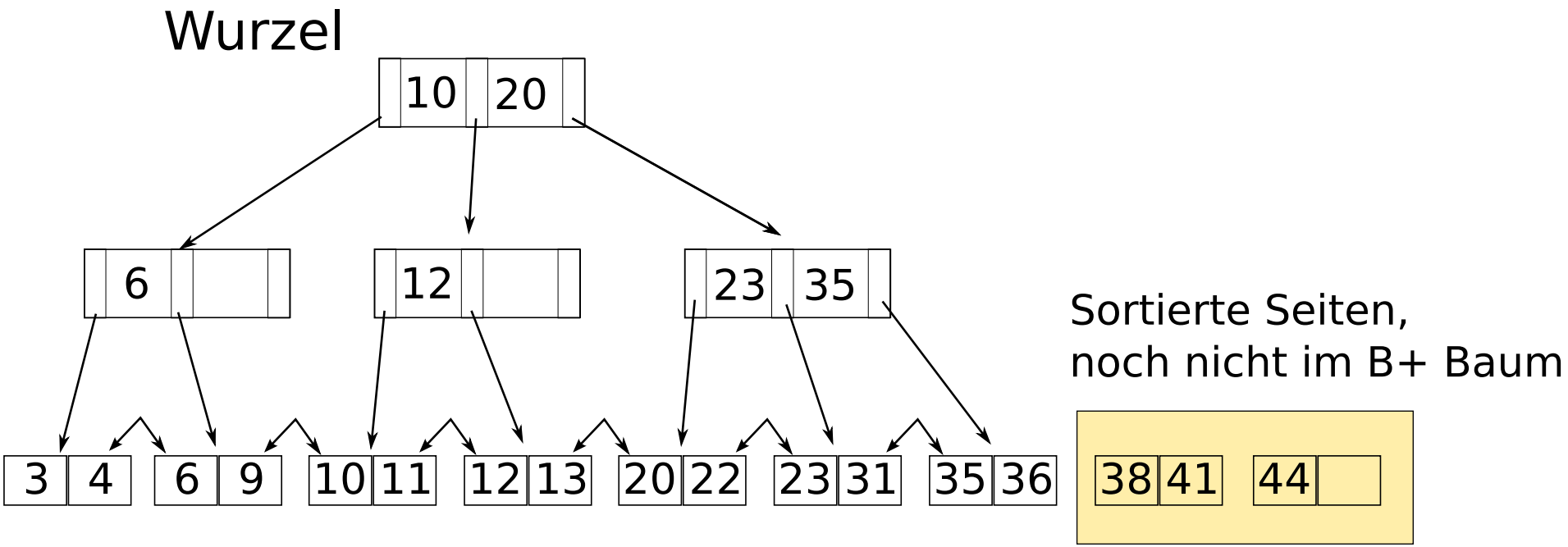
Bulkloading - Beispiel (3)



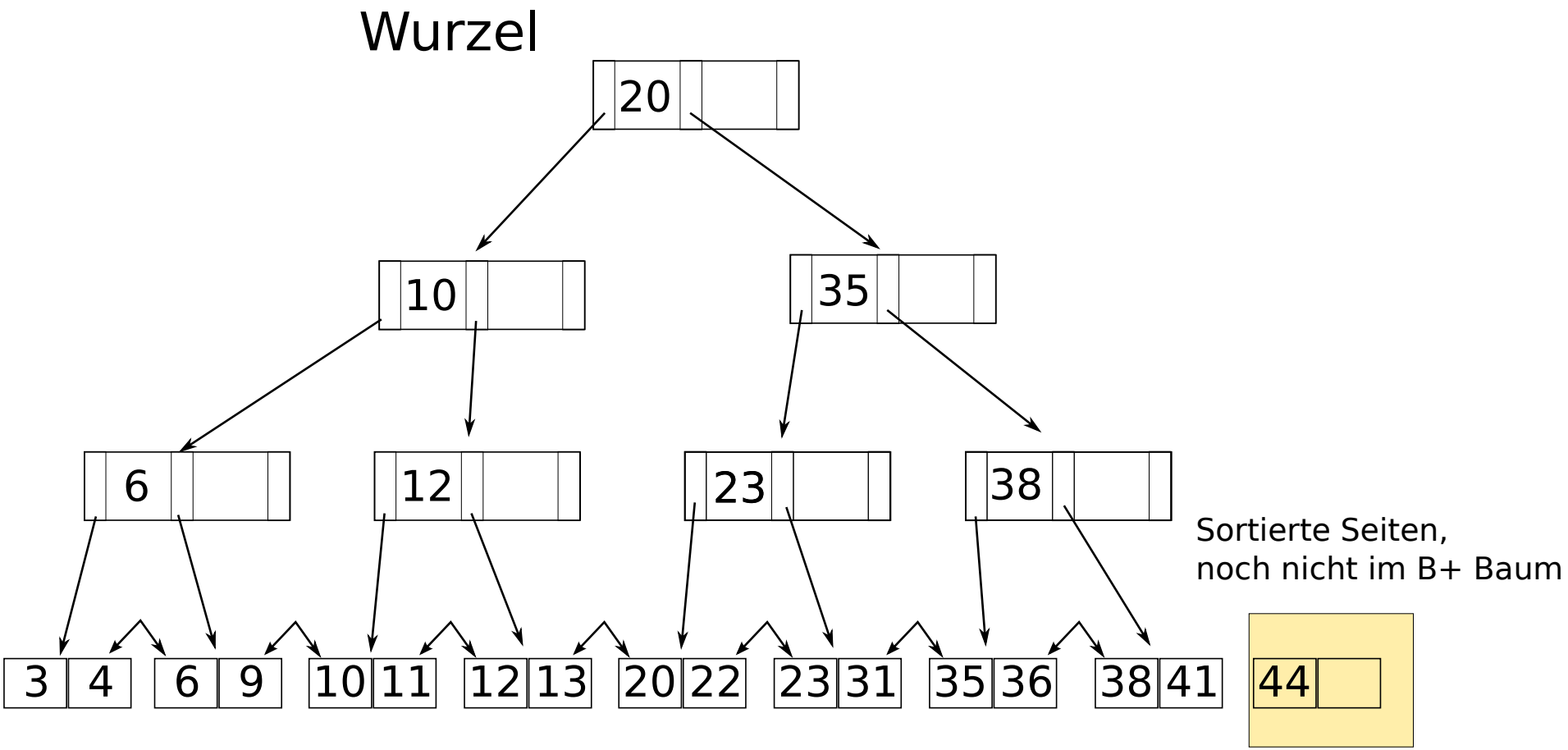
- Beim Aufteilen der Seiten wurden diese hier gleichmässig auf die neuen Seiten unter der Wurzel aufgeteilt
- Im Allgemeinen hätte man auch anders aufteilen können.
- Z.B. nach einem bestimmen Füllgrad (z.B. 80%) oder man hätte auch alle alten Einträge in der linken Seite belassen können.

Bulkloading - Beispiel (4)

- Index-Einträuge für die Blätter werden immer in den am weitesten rechts stehenden Index-Knoten direkt über der Blatt-Ebene eingefügt.
- Sollte dieser voll sein, so muss aufgeteilt werden.



Bulkloading - Beispiel (5)



Ähnlichkeitssuche in hohen Dimensionen

Mehrdimensionale Indexstrukturen

Bislang im B-Baum gesehen: Eindimensionale Schlüssel.

In vielen Anwendungen ist die Anzahl der Dimensionen höher, was tun?

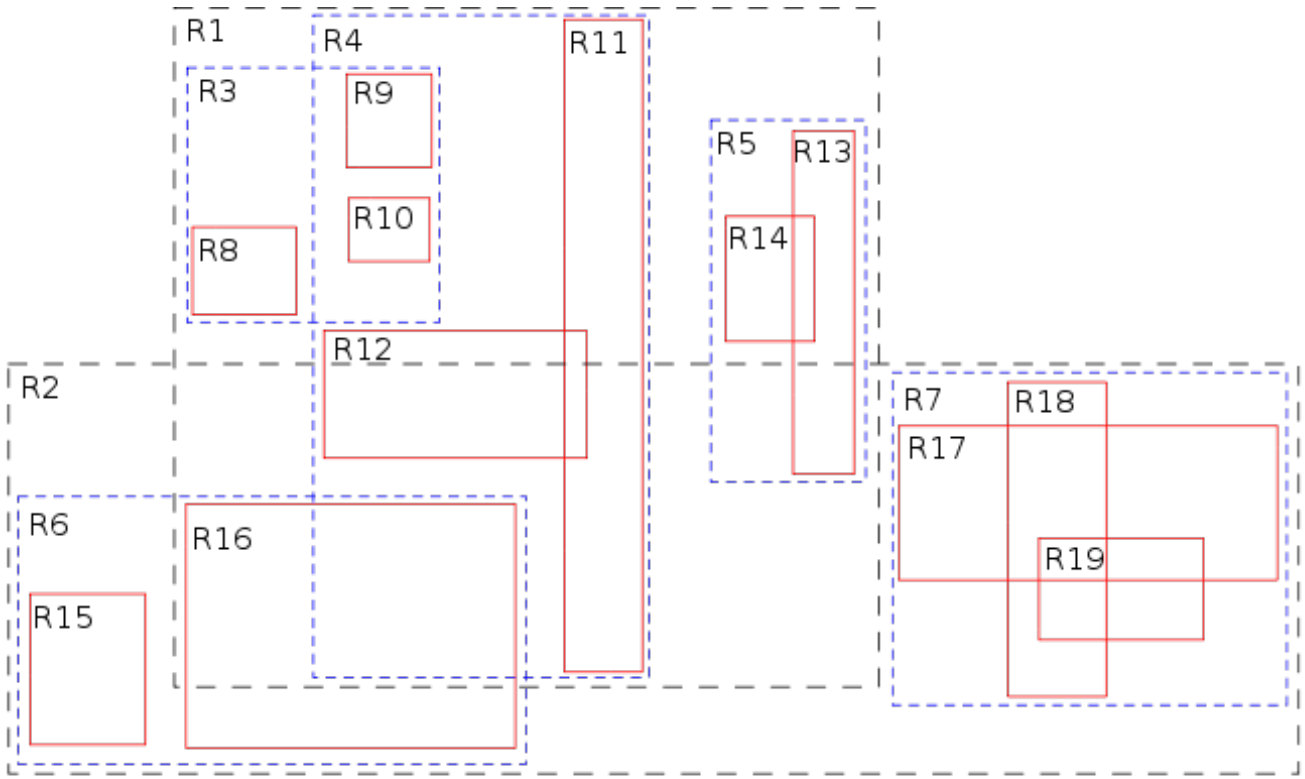
R Baum

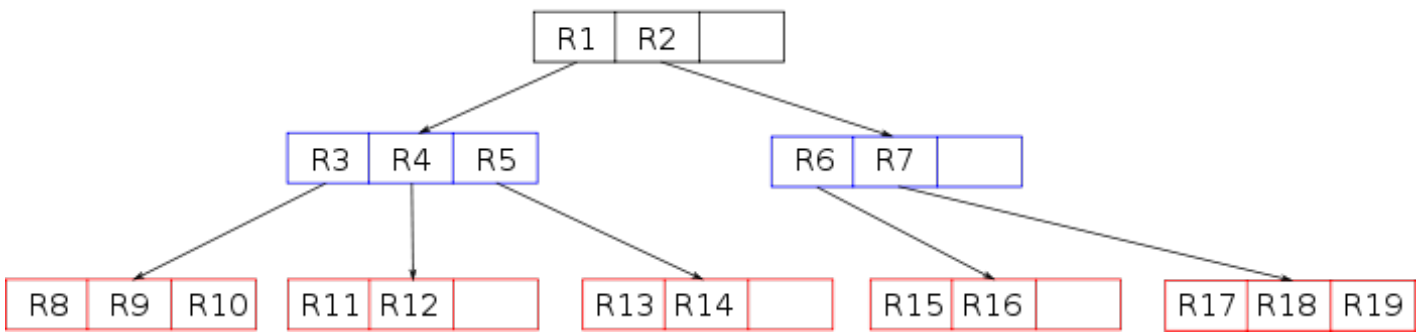
- R steht für Rechteck/Rectangle
- Knoten definieren minimale Rechtecke, die die enhaltenen Rechtecke umschließen
- Balanciert. Aufbau (Algorithmus) ähnlich zum B+ Baum
- Überlappung der MBRs (=Minimum Bounding Rectangles)
- Überlappung führt zu Ineffizienz während der Anfrage.

Wir werden uns den R Baum noch im Detail anschauen.

R Baum: Beispiel

Quelle: wikipedia



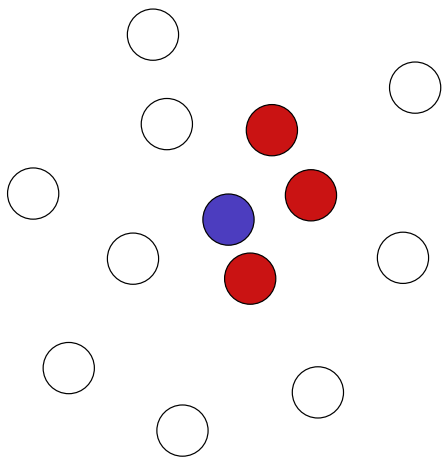


K-Nächste-Nachbarn-Suche (K-Nearest-Neighbor Search)

Objekte werden charakterisiert anhand Menge von relevanten Attributen (Features), z.B. Punkte in einem d-dimensionalen Raum.
Z.B. Städte und ihre X-Y-Positionen im 2-dimensionalen Raum.

K-Nearest Neighbor (KNN) Anfragen

- Für ein Anfrage-Objekt q aus einem d-dimensionalen Raum: Finde den Punkt, der am nächsten an q liegt, für eine gegebene Distanzfunktion
- K-Nearest Neighbor Search: Suche die K nächsten Nachbar-Punkte



Der Fluch der Dimensionalität

(Englisch: Curse of Dimensionality)

- Mit wachsender Dimensionalität wird es zunehmend schwieriger den Raum geeignet zu indexieren.
- Beispiel: Rechteck (2d) mit Kantenlänge 1.
  - Kleines Rechteck mit Kantenlänge $0,1$ beschreibt 1% des großen Rechtecks
  - Um 1% eines Würfels (3d) mit gleicher Kantenlänge zu beschreiben braucht der Kleine eine Kantenlänge von $0,21$!
  - In 100 Dimensionen: Kantenlänge von $0,95$ erforderlich!
- Abstand zwischen nähestem Objekt und maximal entfernten Objekt nähert sich an.

$$lim_{d\to\infty} \frac{d_{max} - d_{min}}{d_{min}} \to 0$$

Ab einer bestimmten Anzahl von Dimensionen ist der R Baum ineffektiv.

Wirklich hohe Dimensionen

Ähnlichkeitssuche (Similarity Search) in Foto-Kollektion

- Gegeben: Ein Foto
- Gesucht: Die k ähnlichsten Fotos der Kollektion.

Beispiel:

Anfrageobjekt

Treffer

- Eigenschaften (Features) werden aus den Fotos extrahiert.
- Z.B: Farbverteilung, Kanten, ... (MPEG-7 Features)
- **Anzahl der Features wird sehr schnell sehr groß:** $>> 10$

Locality Sensitive Hashing (LSH)

Locality Sensitive Hashing (LSH) ist eine approximative Methode, die es erlaubt in sehr hohen Dimensionen KNN Anfragen zu bearbeiten.

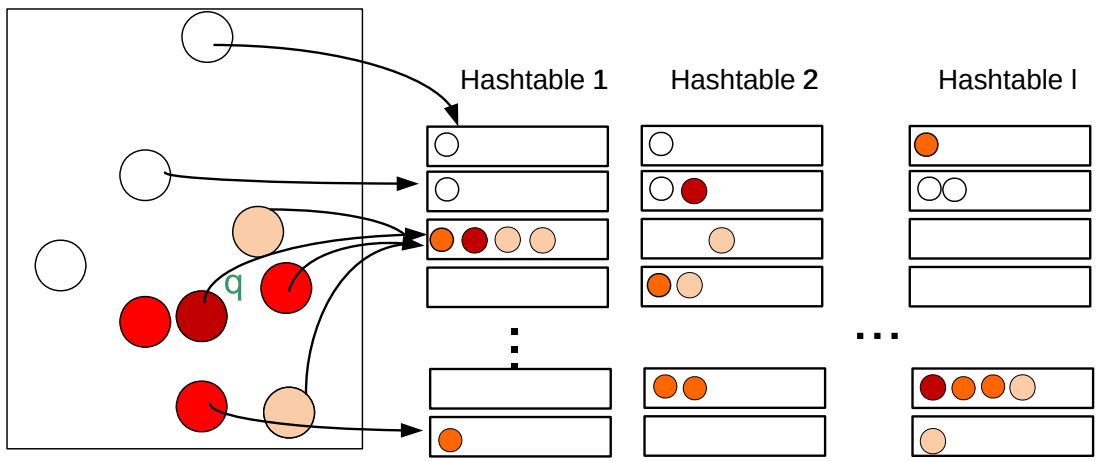
Erinnern Sie sich an Hash-Verfahren:

- Hash-Funktion h platziert Objekte (Tupel) in Hash-Buckets.
- Z.B. basierend auf Schlüssel des Objekts (wie MatrNr für Relation Studenten).
- Schneller Zugriff: Suche nach Objekt x? Schaue in Bucket $h(x)$ nach.
- Wichtig dabei, Kollisionen sind zu reduzieren/vermeiden, wegen (teurer) Handhabung von "überlaufenden" Buckets

Locality Sensitive Hashing (LSH)

Idee:

- Kollisionen: ähnliche Objekte sollen in den gleichen Hash-Bucket fallen.
- Es werden mehrere Hashfunktionen benutzt, gemeinsam ergeben sie den Bucket.
- Um die Wahrscheinlichkeit von Kollisionen zu erhöhen werden mehrere Hashtabellen aufgebaut.



Was bedeutet es Locality Sensitive zu sein?

Eine Familie von Hashfunktionen $H = \{h : S \rightarrow U\}$ heißt (r_1, r_2, p_1, p_2)-sensitive falls die folgenden beiden Bedingungen für beliebige Punkte $\mathbf{q}, \mathbf{v} \in S$ gelten:

- if $dist(\mathbf{q}, \mathbf{v}) \leq r_1$ dann $Pr_H(h(\mathbf{q}) = h(\mathbf{v})) \geq p_1$
- if $dist(\mathbf{q}, \mathbf{v}) > r_2$ dann $Pr_H(h(\mathbf{q}) = h(\mathbf{v})) \leq p_2$

Wichtig: Falls $r_1 < r_2$ und $p_1 > p_2$: ähnliche Objekte bekommen häufiger den gleichen Hash-Wert, im Vergleich zu weniger ähnlichen.

Eine LSH Variante

Für jeden d-dimensionalen Punkt $\mathbf{v}$ betrachten wir k unabhängige Hashfunktionen der Form:

$$h_{\mathbf{a},B}(\mathbf{v}) = \left\lfloor \frac{\mathbf{a} \cdot \mathbf{v} + B}{W} \right\rfloor$$

$\mathbf{a}$: d-dimensionaler Vektor, zufällig anhand Wahrscheinlichkeitsverteilung ausgewählt.
$W \in \mathbb{R}$, und $B \in [0, W]$. $\mathbf{v}$ wird auf $\mathbf{a}$ "projiziert" (Skalarprodukt).

"Beschriftung" der LSH Buckets

Mit k Hashfunktionen ist die Beschriftung des Buckets ein Integer-Vektor der Länge k:

$$g(\mathbf{v}) = (h_{\mathbf{a_1},B_1}(\mathbf{v}), ..., h_{\mathbf{a_k},B_k}(\mathbf{v}))$$

Welchen Einfluss hat k auf die Suche?

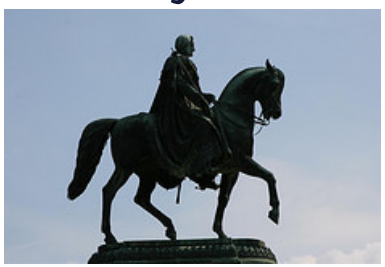
Objekte den Hash-Buckets zuweisen

LSH Bucket "Labels"

Mit k Hashfunktionen ist ein Label ein Integer-Vektor der Länge k:

$$g(\mathbf{v}) = (h_{\mathbf{a_1},B_1}(\mathbf{v}), ..., h_{\mathbf{a_k},B_k}(\mathbf{v}))$$

Objekt:



Anwendung von 4 Hashfunktionen:

$$\begin{aligned} h_1(...) &= 0 \\ h_2(...) &= 1 \\ h_3(...) &= 1 \\ h_4(...) &= 1 \end{aligned}$$

Ergibt das Label: g(...)=(0,1,1,1)

Wie kann man das Problem adressieren, dass ähnliche Objekte in einem anderen Bucket landen und somit nicht gefunden werden?

Suche nach ähnlichen Objekten: Beispiel

		Bucket-Label
Anfrage:		0,1,0,1
Potentieller Treffer:		0,1,1,1

Nicht gefunden!

Idee: Mehrere Hashtabellen!

- Um Trefferwahrscheinlichkeit zu erhöhen werden Objekte nicht nur in einen Bucket gesteckt anhand von g indexiert, sondern anhand von verschiedenen g in mehrere Buckets in verschiedenen Hashtabellen.

Anfrageverarbeitung: Suche der K nächsten Nachbarn

Gegeben ein Anfrage-Punkt q

- Berechne Bucket-Labels $g_i(q)$ für jede Hashtabelle i.
- Hole Objekte aus diesen Buckets
- Berechne echte Distanz und ordne Objekte entsprechend

Trotzdem: LSH ist eine approximative Technik

- Keine harte Garantie, dass alle Treffer (und nur diese) gefunden werden ...
- ... das ist oftmals aber akzeptabel.
- Es gibt theoretische Ansätze die Ergebnisgüte voraus zu sagen

Mehrere Hashtabellen

Benutze mehrere Hashtabellen, um die Wahrscheinlichkeit der Kollisionen zu vergrößern

	Hashtabelle 1	Hashtabelle 2
	0,1,0,1	**1,0,0,1**
	0,1,1,1	**1,0,0,1**
Ergebnis:	Kein Treffer	Treffer!

Beobachtungen

Tuning

- Tradeoff zwischen Größe der Hash-Buckets und der Effektivität
- Mehrere Hashtabellen: Höhere Trefferwahrscheinlichkeit aber größerer Platzverbrauch
- Achtung: auch hier gilt wieder: Ab einem bestimmen Punkt kann ein Full-Scan günstiger sein (und dieser ist sogar noch exakt!)

Erweiterungen

- Schaue in mehrere Hash-Buckets *per* Hashtabelle (aka. multi-probe LSH)
- Auch: verteilte Implementierungen von LSH (z.B. in Peer-to-Peer-Systemen) oder in MapReduce

Min-Hashing

- Gegeben zwei Mengen A und B von Objekten.
- Ein oft benutztes Ähnlichkeitsmaß ist der Jaccard Koeffizient:

$$J(A,B) := \frac{|A \cap B|}{|A \cup B|}$$

Idee hinter Min-Hashing

- Berechne für alle Elemente einer Menge A eine Hashfunktion, die $a \in A$ einem Integer-Wert zuweist.
- Hashfunktion hat hier nichts mit LSH zu tun, es ist eine "normale".
- Betrachte den kleinsten dieser Werte $h_{min}(A)$
- Wie groß ist die Wahrscheinlichkeit, dass zwei Mengen A und B dieser min-Wert identisch ist?

Paper: Andrei Broder. On the resemblance and containment of documents. 1997.

Min-Hashing (2)

- Die Wahrscheinlichkeit $Pr[h_{min}(A) = h_{min}(B)]$ steht in direkter Verbindung zum Jaccard-Koeffizienten:

$$Pr[h_{min}(A) = h_{min}(B)] = \frac{|A \cap B|}{|A \cup B|}$$

Anwendung

- Suche nach ähnlichen Mengen: Betrachte nur Paare von Mengen, deren min-Wert identisch ist
- Bzw., nehme $Pr[h_{min}(A) = h_{min}(B)]$ als Näherung für den Jaccard-Koeffizienten
- Wie gut funktioniert das?

Min-Hashing - Mehrere min-Werte bzw. Hashfunktionen

Mehrere Hash-Funktionen mit je einem min-Wert

- Betrachte k (unabhängige) Hashfunktionen, die jeweils einen min-Wert liefern.
- Approximiere $J(A, B)$ mit Anteil der übereinstimmenden min-Werte.

Mehrere min-Werte & eine Hashfunktionen

- Betrachte nur eine Hashfunktion, aber nehme von dieser die k kleinsten Werte.

Der Fehler ist bei beiden Schemata $O(1/\sqrt{k})$.