Datenbanksysteme

Wintersemester 2016/17

Prof. Dr.-Ing. Sebastian Michel
TU Kaiserslautern

smichel@cs.uni-kl.de

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)

Literatur: Witold Litwin: Linear Hashing: A New Tool for File and Table Addressing. VLDB, 1980.

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

$h_j(k) = k$ **mod** $(2^j N)$ erfüllt beide Bedingungen

Belegungsfaktor

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

Mit

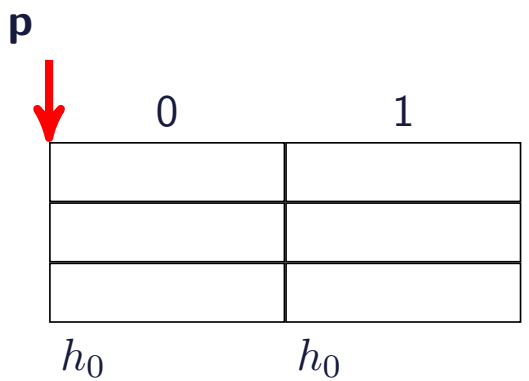
- x: Anzahl der Datensätze
- b: Kapazität eines Buckets (Seite) in Anzahl der Datensätze die hinein passen
- M: Anzahl der Buckets der Primärdatei

- **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** $(2^0 N)$

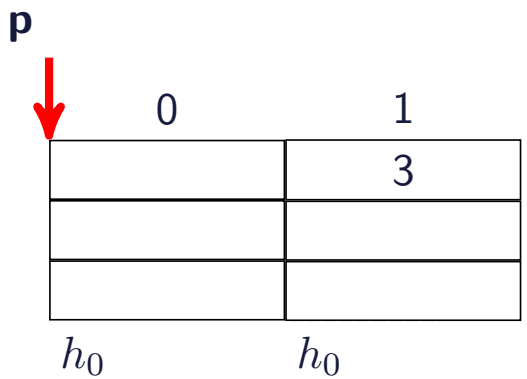
Einfügesequenz: 3, 5, 7, 13, 10



Beispiel

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

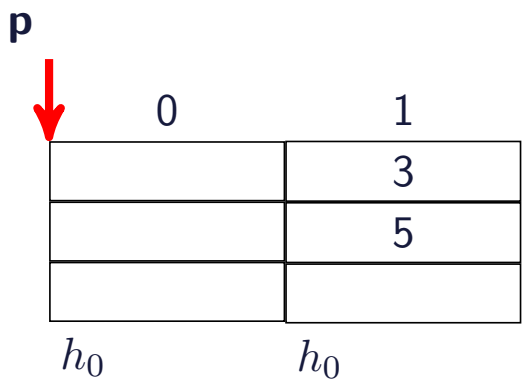
Einfügesequenz: 3, 5, 7, 13, 10



Beispiel

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

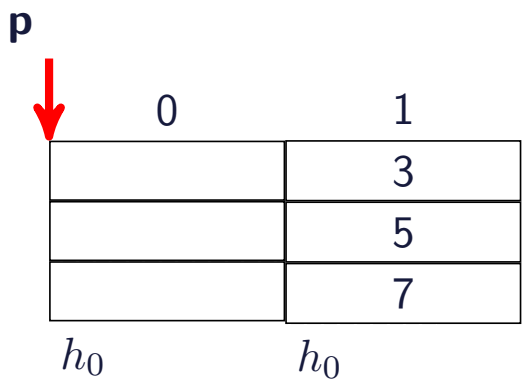
Einfügesequenz: 3, 5, 7, 13, 10



Beispiel

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

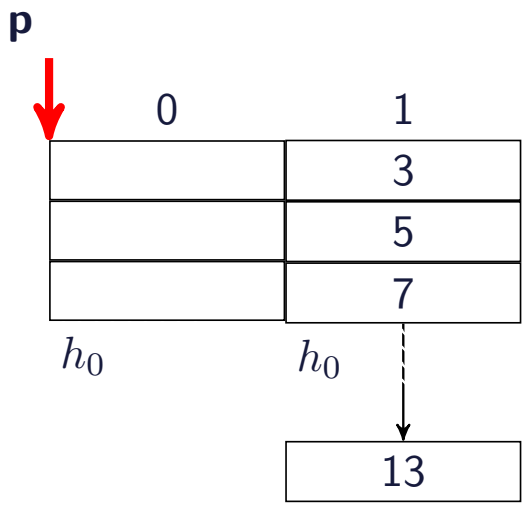
Einfügesequenz: 3, 5, 7, 13, 10



Beispiel

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

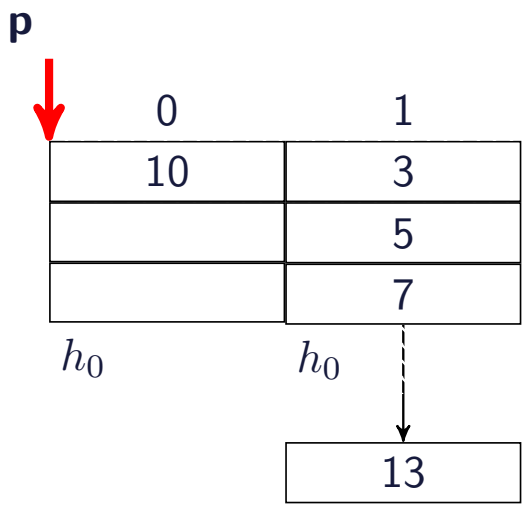
Einfügesequenz: 3, 5, 7, 13, 10



Beispiel

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

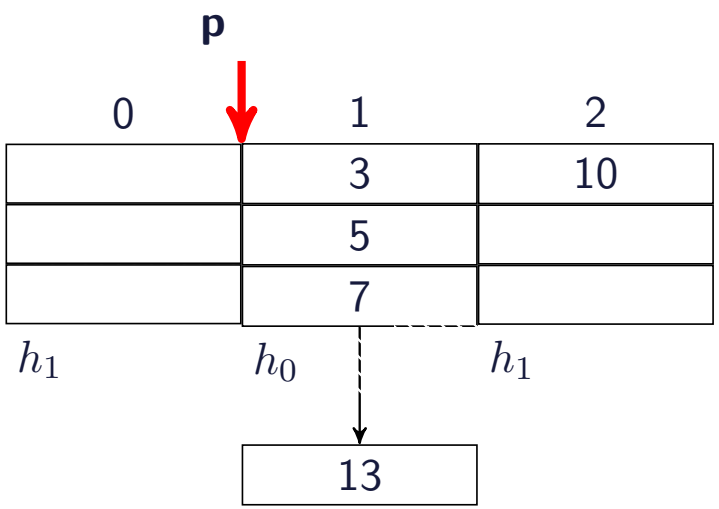
Einfügesequenz: 3, 5, 7, 13, 10



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

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



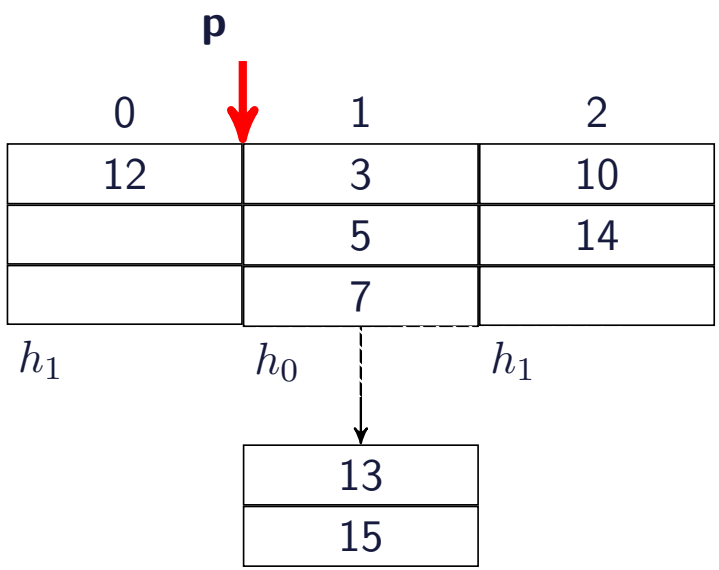
Addressberechnung

- 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

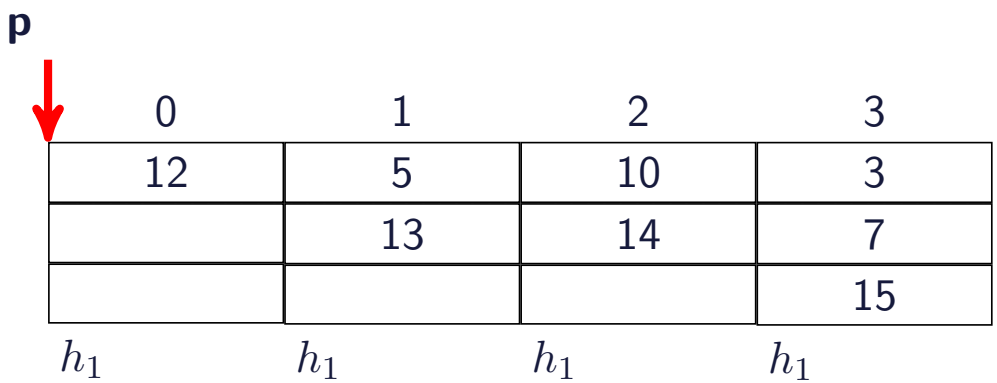


$$\beta = \frac{8}{9} = 0.88$$

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: $h_1(k) = k$ **mod** $(2^1 N) = k$ **mod** (4). Für Bucket 1 auch?

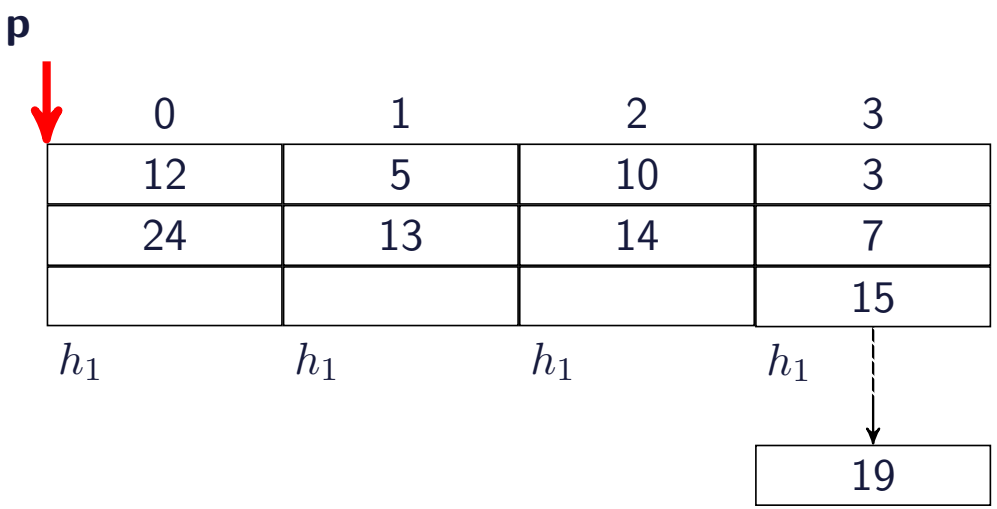
Beispiel (4)

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



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 Sekundärseite

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

Es wird jeweils nur ein Splitvorgang ausgeführt, unabhängig davon, ob dadurch der Überlauf beseitigt wird. Und zwar wird das Bucket gesplittet, auf das Pointer p zeigt, nicht das Bucket, welches den Überlauf verursacht hat.

Ausblick

- **Invertierter Index** und
- **Top-K Schwellwert Algorithmen**
- **Skyline-Anfragen**

Im Anschluss: **Transaktionsverwaltung**

- **Datenbankwiederherstellung** (Recovery)
- **Mehrbenutzersynchronisation** (Concurrency Control)

Invertierter Index, Top-k Algorithmen und Skylines

Invertierter Index und Indexlisten

- Gegeben Menge von Objekten, z.B. Text-Dokumente
- **Invertierter Index ist "Abbildung" von "Inhalt" oder Eigenschaft auf Objekte mit diesem Inhalt bzw. dieser Eigenschaft**
- **Anwendung:** Suche in Text-Dokumenten:
  - Invertierter Index enthält pro Term (Wort) eine Liste aller Dokumente, die diesen Term enthalten
  - Dann: Für eine Anfrage $q = \{term_1, term_2, \ldots, term_n\}$ kann man sehr effizient alle Dokumente finden, die die Terme der Anfrage enthalten

Sortierung

- Die Listen werden in der Regel sortiert gehalten
- Sortierung nach "Score" absteigend; Score ist z.B. tf*idf aus dem Information Retrieval, Anzahl heruntergeladener Bytes von Webserver

Beispiel

www.server1.com:

ClientIP	Bytes
192.168.1.3	17kB
192.168.1.4	12kB
192.168.1.2	11kB
192.168.1.5	4kB
192.168.1.6	2kB

www.server2.com:

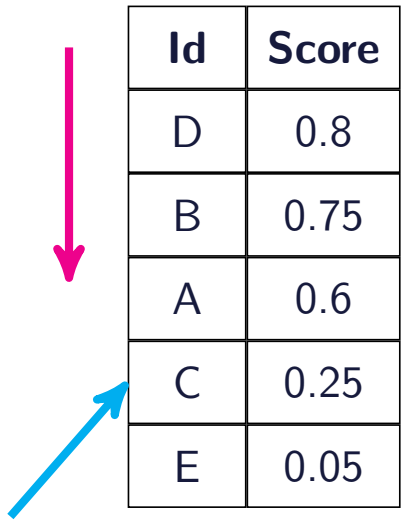
ClientIP	Bytes
192.168.1.1	9kB
192.168.1.3	7kB
192.168.1.2	2kB
192.168.1.6	1kB
192.168.1.7	1kB

www.server3.com:

ClientIP	Bytes
192.168.1.1	19kB
192.168.1.4	15kB
192.168.1.3	12kB
192.168.1.5	5kB
192.168.1.7	2kB

Zugriffsmethoden

- **Sequenzieller Zugriff**: Lese Inhalt der Indexliste von oben nach unten.
- **Wahlfreier Zugriff**: Schaue Score für ein bestimmtes Objekt nach



Id	Score
D	0.8
B	0.75
A	0.6
C	0.25
E	0.05

Notation und Konvention bei fehlenden Einträgen

- $score(i,a)$ bezeichnet Wert für Objekt a in Indexliste L_i.
- Falls a nicht in Indexliste i auftritt wird $score(i,a) = 0$ angenommen.
- D.h. **Werte sind positiv (und 0 der "schlechteste" Wert).**

Top-k Anfrage

- Anfrage: Gegeben eine Menge von Indexlisten $Q = \{L_i\}$
- Eine **Aggregationsfunktion aggr()**
- Und einen **Parameter k; die Größe der Ergebnismenge**

Top-k Ergebnismenge

- Ziel: **Berechne die k Objekte mit den höchsten aggregierten Scores**
- Bei gleichen Score-Werten (Englisch: ties) wird i.d.R. beliebig ausgewählt

Veranschaulichung: In SQL

select id, aggr($L_1.score$,$L_2.score$,...) as overallScore
from L_1, L_2, L_3, ...
where $L_1.id = L_2.id$ **and** $L_2.id = L_3.id$ **and** ...
order by overallScore **DESC**
limit k

Beispiel

- **Listen mit visuellen Eigenschaften (Farbe und Form) von Objekten: rot, blau, rund, rechteckig, ...**
- Informationen sind ungenau (fuzzy), in $[0,1]$, also nicht genau bekannt.
- **Anfrage: Suche die top-2 roten und rechteckigen Objekte**

farbe=rot

Id	Score
E	0.8
B	0.6
D	0.3
A	0.25
C	0.19

form=rechteckig

Id	Score
D	0.8
B	0.75
A	0.6
C	0.25
E	0.05

Ergebnis

Id	$\sum$Score
B	1.35
D	1.10

Top-k Algorithmen

Idee bzw. Ziel

- **Berechne die besten k (i.e., top-k) Ergebnisse ohne die Indexlisten komplett zu lesen**
- Familie der Threshold-Algorithmen (Threshold ist Englisch für Schwellwert)
- **Berechnen den frühestmöglichen Zeitpunkt, an dem die Berechnung abgebrochen werden kann und trotzdem das Ergebnis korrekt ist**

Aggregationsfunktion aggr()

- Aggregiert die Werte der einzelnen Objekte in den beteiligten Indexlisten; meistens die Summe

Ronald Fagin, Amnon Lotem, Moni Naor: Optimal aggregation algorithms for middleware. J. Comput. Syst. Sci. 66(4), 2003.

Monotonie der Aggregationsfunktion

Monotonie

- `aggr` **ist monoton falls für alle Objekte a und b gilt:**

$$(\forall_i score(i,a) \leq score(i,b)) \Rightarrow aggr(a) \leq aggr(b)$$

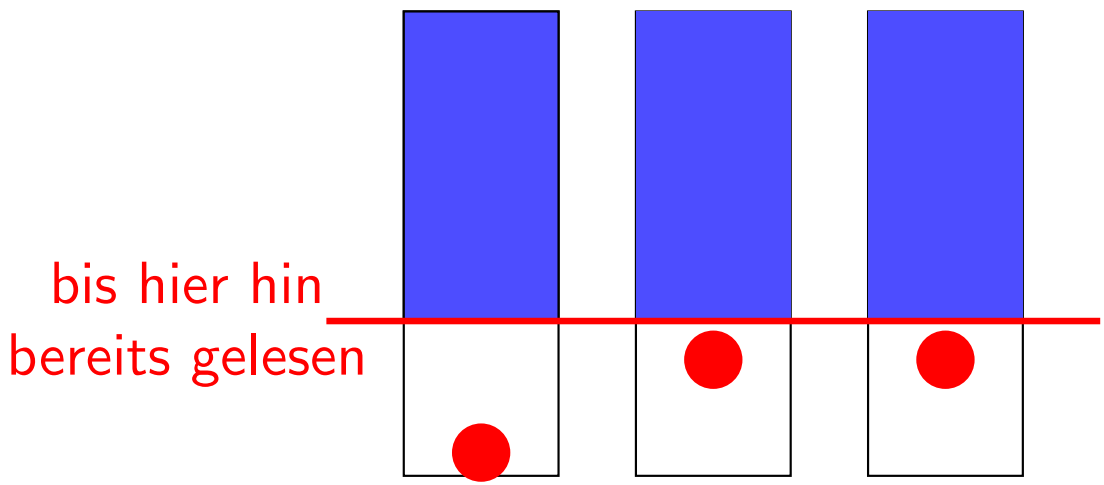
- Im **Beispiel** (rechts) befindet sich Objekt A in beiden Listen "unter" Objekt B. Also kann der aggregierte Wert von A nicht größer sein als der aggregierte Wert von B.

Id	Score
E	0.8
B	0.6
D	0.3
A	0.25
C	0.19

Id	Score
D	0.8
B	0.75
A	0.6
C	0.25
E	0.05

Monotonie der Aggregationsfunktion

- Wenn ein Objekt in allen Indexlisten nicht besser als ein anderes Objekt ist, dann kann es bzgl. finaler Score nicht besser sein.
- Im Beispiel unten ist das Objekt, das durch einen roten Punkt dargestellt ist, noch nicht gesehen worden
- Aber wir wissen bereits, dass es nicht "besser" sein kann als Objekte, die bereits in **allen** Listen gesehen wurden



- **Was bedeutet dies für Algorithmen, die Indexlisten top-down verarbeiten bzw. Kriterium zu stoppen?**

Fagin's Algorithmus (FA)

1. **Lese sequenziell von jeder Liste (round robin) bis es k verschiedene Objekte gibt, die in allen Listen gesehen wurden**

Beispiel für eine top-1 Anfrage

Id	Score
doc3	17
doc4	12
doc2	11
doc5	4
doc6	2

Id	Score
doc1	9
doc3	7
doc2	2
doc6	1
doc7	1

Id	Score
doc1	19
doc4	15
doc3	12
doc5	5
doc2	2

- Mehr muss (erstmal) nicht gelesen werden, da doc3 in allen Listen gesehen wurde (und es eine top-1 Anfrage ist).

Fagin's Algorithmus (FA) (2)

- Haben doc3 gesehen mit Score $17 + 7 + 12 = 36$
- Und partiell auch: doc4 $(12 + 15 = 27)$, doc1 $(9 + 19 = 28)$ und doc2 $(11 + 2 = 13)$
- Reicht dieses Wissen aus um bereits zu diesem Zeitpunkt zu stoppen?

Id	Score
doc3	17
doc4	12
doc2	11
doc5	4
doc6	2

Id	Score
doc1	9
doc3	7
doc2	2
doc6	1
doc7	1

Id	Score
doc1	19
doc4	15
doc3	12
doc5	5
doc2	2

Fagin's Algorithmus (FA) (3)

2. Schaue die fehlenden Scores der Objekte mittels wahlfreiem Zugriff nach (Erinnerung: Fehlende Einträge bedeuten Score 0)
   - Dann haben wir: doc4 (12+0+15=27), doc1 (0+9+19=28) und doc2 (11+2+2=15)
   - Fertig! Wieso?

Id	Score
doc3	17
doc4	12
doc2	11
doc5	4
doc6	2

Id	Score
doc1	9
doc3	7
doc2	2
doc6	1
doc7	1

Id	Score
doc1	19
doc4	15
doc3	12
doc5	5
doc2	2

FA - Algorithmus

1. Sequenzielles Lesen in parallel in jeder der m sortierten listen L_i (kann auch in jeder Liste Batches lesen, nicht nur einzeln; siehe Paper). Warte bis es mindestens k Objekte gibt, die jeweils in jeder Liste gesehen wurden.
2. Für jedes Objekt o_i, schaue via wahlfreiem Zugriff fehlende Scores in den Listen nach.
3. Aggregiere scores in $aggr(o_i)$ für alle gesehenen Objekte. Die k Objekte mit den höchsten aggregierten Werten sind das Ergebnis.

Beobachtungen und Analyse von FA

Korrekteit

- Aufgrund der Monotonie der Aggregationsfunktion: Objekte die nach Phase 1 noch gar nicht gesehen wurden können nicht besser sein als die k Dokumente, die in jeder Liste gefunden wurden
- Aber teilweise gesehene Objekte können besser sein, deswegen das Nachschauen der fehlenden Scores (via wahlfreiem Zugriff, aka. random access)

Beobachtung

- Es kann sehr lange dauern bis k verschiedene Objekte in allen Listen gefunden wurden. **Was kann man hier besser machen?**

Threshold Algorithmus (TA)

1. **Lese von jeder Indexliste in sequenzieller Folge (round robin)**
2. **Für jedes gefundene Objekt: schaue in den übrigen Listen die Score nach**

Stop Algorithmus terminiert, sobald mindestens k Objekte gefunden wurden, deren aggregierte Score größer als aggregierten Scores auf der momentanen Scan-Linie

Dieser "Threshold" (Schwellwert) wird mit τ bezeichnet

Id	Score
doc3	18
doc4	12
doc2	11
doc5	4
doc6	2

Id	Score
doc1	9
doc3	7
doc2	2
doc6	1
doc7	1

Id	Score
doc1	19
doc4	15
doc3	12
doc5	5
doc2	2

Threshold Algorithmus (TA) - Schritt 1

- Beginne mit sequenziellem Lesen. Wir sehen zuerst doc3 in Liste 1 und schauen die Score von doc3 in den beiden anderen Listen nach
- Wir erhalten doc3 $(18 + 7 + 12 = 37)$

Id	Score
doc3	18
doc4	12
doc2	11
doc5	4
doc6	2

Id	Score
doc1	9
doc3	7
doc2	2
doc6	1
doc7	1

Id	Score
doc1	19
doc4	15
doc3	12
doc5	5
doc2	2

Threshold Algorithmus (TA) - Schritt 2

- Weiter geht es mit doc1, gesehen in Liste 2
- Nachschauen der Score von doc1 in Liste 1 und Liste 3
- Insgesamt haben wir nun doc3 $(18 + 7 + 12 = 37)$ und doc1 $(0 + 9 + 19 = 28)$

Id	Score
doc3	18
doc4	12
doc2	11
doc5	4
doc6	2

Id	Score
doc1	9
doc3	7
doc2	2
doc6	1
doc7	1

Id	Score
doc1	19
doc4	15
doc3	12
doc5	5
doc2	2

Threshold Algorithmus (TA) - Schritt 3

- Sequenzielles Lesen von Liste 3
- Bleibt dabei: doc3 $(18 + 7 + 12 = 37)$ und doc1 $(0 + 9 + 19 = 28)$
- Scores auf der Scan-Linie: $18 + 9 + 19 = 46$
- **Warum können wir noch nicht aufhören?**

Id	Score
doc3	18
doc4	12
doc2	11
doc5	4
doc6	2

Id	Score
doc1	9
doc3	7
doc2	2
doc6	1
doc7	1

Id	Score
doc1	19
doc4	15
doc3	12
doc5	5
doc2	2

Threshold Algorithmus (TA) - Schritt 4

- doc3 $(18 + 7 + 12 = 37)$, doc1 $(0 + 9 + 19 = 28)$, doc4 $(12 + 0 + 15 = 27)$
- Scores auf der Scan-Linie: $12 + 9 + 19 = 40$
- **Jetzt aufhören? Nein, immer noch nicht, wieso?**

Id	Score
doc3	18
doc4	12
doc2	11
doc5	4
doc6	2

Id	Score
doc1	9
doc3	7
doc2	2
doc6	1
doc7	1

Id	Score
doc1	19
doc4	15
doc3	12
doc5	5
doc2	2

Threshold Algorithmus (TA) - Schritt 5

- doc3 $(18 + 7 + 12 = 37)$, doc1 $(0 + 9 + 19 = 28)$, doc4 $(12 + 0 + 15 = 27)$
- Scores auf der Scan-Linie: $12 + 7 + 19 = 38$
- **Wir können immer noch nicht aufhören!**

Id	Score
doc3	18
doc4	12
doc2	11
doc5	4
doc6	2

Id	Score
doc1	9
doc3	7
doc2	2
doc6	1
doc7	1

Id	Score
doc1	19
doc4	15
doc3	12
doc5	5
doc2	2

Threshold Algorithmus (TA) - Schritt 6

- doc3 $(18 + 7 + 12 = 37)$, doc1 $(0 + 9 + 19 = 28)$, doc4 $(12 + 0 + 15 = 27)$
- Scores auf der Scan-Linie: $12 + 7 + 15 = 34$
- **Kann der Algorthmus nun terminieren?**

Id	Score
doc3	18
doc4	12
doc2	11
doc5	4
doc6	2

Id	Score
doc1	9
doc3	7
doc2	2
doc6	1
doc7	1

Id	Score
doc1	19
doc4	15
doc3	12
doc5	5
doc2	2

Threshold Algorithmus (TA) - Schritt 6

- doc3 $(18 + 7 + 12 = 37)$, doc1 $(0 + 9 + 19 = 28)$, doc4 $(12 + 0 + 15 = 27)$
- Scores auf der Scan-Linie: $12 + 7 + 15 = 34$
- **Kann der Algorthmus nun terminieren?** Ja, denn Score von doc3 (37) ist größer als Score auf der Scan-Linie (34). **Aufgrund der sortierten Listen und Monotonie kann kein noch nicht gesehenes Objekt besser als doc3 sein.**

Id	Score
doc3	18
doc4	12
doc2	11
doc5	4
doc6	2

Id	Score
doc1	9
doc3	7
doc2	2
doc6	1
doc7	1

Id	Score
doc1	19
doc4	15
doc3	12
doc5	5
doc2	2

TA - Algorithmus

1. Lese sequenziell in parallel aus jeder der m Listen L_i. Für ein Objekt o_i, das unter diesem sequenziellen Lesen gesehen wurde, schaue in allen anderen Listen die Score des Objekts nach. Dann berechne den aggregierten Wert $aggr(o_i)$. Wenn dieser Wert einer der k höchsten Werte (soweit) ist, behalte Objekt o_i.
2. Für jede Liste L_i sei $high_i$ der Wert des zuletzt gesehenen Objekts unter sequenziellen (!) Lesen. Der Threshold-Wert τ wird definiert als $\tau = aggr(high_1, high_2, \ldots high_m)$. Algorithmus terminiert, sobald es k Objekte gibt, deren aggregierte Score größer ist als τ.
3. Das Ergebnis besteht aus den k Objekten mit der größten aggregierten Score.

TA - Analyse

Korrektheit

- Alle Objekte, die durch sequenziellen Scan gelesen wurden sind komplett bekannt (ihre Scores).
- Sei z ein Objekt, dass nicht gesehen wurde. Das bedeutet, dass für alle Listen L_i gelten muss: $score(i,z) \leq high_i$, also ist die aggregierte Score von $z \leq \tau$. Für unsere k Ergebnisse wissen wir aber, dass sie eine aggregierte Score $\geq \tau$ haben.

Vergleich zu TA

- Das Kriterium, das es erlaubt TA zu terminieren, tritt nicht später ein als im Fall von FA.
- D.h. TA braucht nicht mehr sequenzielle Zugriffe als FA.

Wahlfreie vs. Sequenzielle Zugriffe

- Haben in verschiedenen Szenarien gesehen/diskutiert, dass wahlfreie Zugriffe (random accesses) sehr teuer sind

- Variationen der Threshold-Algorithmen, die zwischen wahlfreien und sequenziellen Zugriffen abwägen
- Oder nur noch sequentielle Zugriffe zulassen

  No Random Access (NRA) Algorithmus

No Random Access (NRA) Algorithmus

- Es sind **nur noch sequenzielle Zugriffe erlaubt.**
- Was bedeutet dies für die Scores eines Objekts?

Worstscore

- **Aktuell bereits gesehene Scores eines Objekts (aggregiert)**

Bestscore

- **Bestmögliche Score (aggregiert) eines Objekts**
- Was heißt bestmöglich? Besser als Scores auf der Scan-Linie kann es nicht mehr werden
- **Aggregation aller Scores der Scan-Linie wird τ genannt**
- **bestscore = worstscore + score der Scan-Linie**
- Score der Scan-Linie natürlich nur für Indexlisten, in denen Objekt noch nicht gesehen wurden

NRA - Bestscore und Worstscore

Id	Score
doc3	18
doc4	12
doc2	11
doc5	4
doc6	2

Id	Score
doc1	9
doc3	7
doc2	2
doc6	1
doc7	1

Id	Score
doc1	19
doc4	15
doc3	12
doc5	5
doc2	2

id	worstscore	bestscore
doc3	18+7=25	+15 = 40
doc1	9+19=28	+11=39
doc4	12 +15 = 27	+7=34
doc2	11	+7+15= 33

NRA - Anfrageverarbeitung Beispiel

ClientIP	Bytes
192.168.1.3	17kB
192.168.1.4	12kB
192.168.1.2	11kB
192.168.1.5	4kB
192.168.1.6	2kB

ClientIP	Bytes
192.168.1.1	9kB
192.168.1.3	7kB
192.168.1.2	2kB
192.168.1.6	1kB
192.168.1.7	1kB

ClientIP	Bytes
192.168.1.1	19kB
192.168.1.4	15kB
192.168.1.3	12kB
192.168.1.5	5kB
192.168.1.7	2kB

Kandidaten:

Id **worstscore** **bestscore**

$min_k = ?$

NRA - Anfrageverarbeitung Beispiel

ClientIP	Bytes
192.168.1.3	17kB
192.168.1.4	12kB
192.168.1.2	11kB
192.168.1.5	4kB
192.168.1.6	2kB

ClientIP	Bytes
192.168.1.1	9kB
192.168.1.3	7kB
192.168.1.2	2kB
192.168.1.6	1kB
192.168.1.7	1kB

ClientIP	Bytes
192.168.1.1	19kB
192.168.1.4	15kB
192.168.1.3	12kB
192.168.1.5	5kB
192.168.1.7	2kB

Status der Anfrageverarbeitung:

Kandidaten:

Id	worstscore	bestscore
192.168.1.3	17	-

$min_k = 17$

NRA - Anfrageverarbeitung Beispiel

ClientIP	Bytes
192.168.1.3	17kB
192.168.1.4	12kB
192.168.1.2	11kB
192.168.1.5	4kB
192.168.1.6	2kB

ClientIP	Bytes
192.168.1.1	9kB
192.168.1.3	7kB
192.168.1.2	2kB
192.168.1.6	1kB
192.168.1.7	1kB

ClientIP	Bytes
192.168.1.1	19kB
192.168.1.4	15kB
192.168.1.3	12kB
192.168.1.5	5kB
192.168.1.7	2kB

Status der Anfrageverarbeitung:

$min_k = 17$

Kandidaten:

Id	worstscore	bestscore
192.168.1.3	17	-
192.168.1.1	9	-

NRA - Anfrageverarbeitung Beispiel

ClientIP	Bytes
192.168.1.3	17kB
192.168.1.4	12kB
192.168.1.2	11kB
192.168.1.5	4kB
192.168.1.6	2kB

ClientIP	Bytes
192.168.1.1	9kB
192.168.1.3	7kB
192.168.1.2	2kB
192.168.1.6	1kB
192.168.1.7	1kB

ClientIP	Bytes
192.168.1.1	19kB
192.168.1.4	15kB
192.168.1.3	12kB
192.168.1.5	5kB
192.168.1.7	2kB

Status der Anfrageverarbeitung:

$min_k = 28$

Kandidaten:

Id	worstscore	bestscore
192.168.1.3	17	45
192.168.1.1	28	45

NRA - Anfrageverarbeitung Beispiel

ClientIP	Bytes
192.168.1.3	17kB
192.168.1.4	12kB
192.168.1.2	11kB
192.168.1.5	4kB
192.168.1.6	2kB

ClientIP	Bytes
192.168.1.1	9kB
192.168.1.3	7kB
192.168.1.2	2kB
192.168.1.6	1kB
192.168.1.7	1kB

ClientIP	Bytes
192.168.1.1	19kB
192.168.1.4	15kB
192.168.1.3	12kB
192.168.1.5	5kB
192.168.1.7	2kB

Status der Anfrageverarbeitung:

$min_k = 28$

Kandidaten:

Id	worstscore	bestscore
192.168.1.3	17	45
192.168.1.1	28	40
192.168.1.4	12	40

NRA - Anfrageverarbeitung Beispiel

ClientIP	Bytes
192.168.1.3	17kB
192.168.1.4	12kB
192.168.1.2	11kB
192.168.1.5	4kB
192.168.1.6	2kB

ClientIP	Bytes
192.168.1.1	9kB
192.168.1.3	7kB
192.168.1.2	2kB
192.168.1.6	1kB
192.168.1.7	1kB

ClientIP	Bytes
192.168.1.1	19kB
192.168.1.4	15kB
192.168.1.3	12kB
192.168.1.5	5kB
192.168.1.7	2kB

Status der Anfrageverarbeitung:

$min_k = 28$

Kandidaten:

Id	worstscore	bestscore
192.168.1.3	24	43
192.168.1.1	28	40
192.168.1.4	12	38

NRA - Anfrageverarbeitung Beispiel

ClientIP	Bytes
192.168.1.3	17kB
192.168.1.4	12kB
192.168.1.2	11kB
192.168.1.5	4kB
192.168.1.6	2kB

ClientIP	Bytes
192.168.1.1	9kB
192.168.1.3	7kB
192.168.1.2	2kB
192.168.1.6	1kB
192.168.1.7	1kB

ClientIP	Bytes
192.168.1.1	19kB
192.168.1.4	15kB
192.168.1.3	12kB
192.168.1.5	5kB
192.168.1.7	2kB

Status der Anfrageverarbeitung:

$min_k = 28$

Kandidaten:

Id	worstscore	bestscore
192.168.1.3	24	39
192.168.1.1	28	40
192.168.1.4	27	34

NRA - Anfrageverarbeitung Beispiel

ClientIP	Bytes
192.168.1.3	17kB
192.168.1.4	12kB
192.168.1.2	11kB
192.168.1.5	4kB
192.168.1.6	2kB

ClientIP	Bytes
192.168.1.1	9kB
192.168.1.3	7kB
192.168.1.2	2kB
192.168.1.6	1kB
192.168.1.7	1kB

ClientIP	Bytes
192.168.1.1	19kB
192.168.1.4	15kB
192.168.1.3	12kB
192.168.1.5	5kB
192.168.1.7	2kB

Status der Anfrageverarbeitung:

$min_k = 28$

Kandidaten:

Id	worstscore	bestscore
192.168.1.3	24	39
192.168.1.1	28	39
192.168.1.4	27	34
192.168.1.2	11	33

NRA - Anfrageverarbeitung Beispiel

ClientIP	Bytes
192.168.1.3	17kB
192.168.1.4	12kB
192.168.1.2	11kB
192.168.1.5	4kB
192.168.1.6	2kB

ClientIP	Bytes
192.168.1.1	9kB
192.168.1.3	7kB
192.168.1.2	2kB
192.168.1.6	1kB
192.168.1.7	1kB

ClientIP	Bytes
192.168.1.1	19kB
192.168.1.4	15kB
192.168.1.3	12kB
192.168.1.5	5kB
192.168.1.7	2kB

Status der Anfrageverarbeitung:

$min_k = 28$

Kandidaten:

Id	worstscore	bestscore
192.168.1.3	24	39
192.168.1.1	28	39
192.168.1.4	27	29
192.168.1.2	13	28

NRA - Anfrageverarbeitung Beispiel

ClientIP	Bytes
192.168.1.3	17kB
192.168.1.4	12kB
192.168.1.2	11kB
192.168.1.5	4kB
192.168.1.6	2kB

ClientIP	Bytes
192.168.1.1	9kB
192.168.1.3	7kB
192.168.1.2	2kB
192.168.1.6	1kB
192.168.1.7	1kB

ClientIP	Bytes
192.168.1.1	19kB
192.168.1.4	15kB
192.168.1.3	12kB
192.168.1.5	5kB
192.168.1.7	2kB

Status der Anfrageverarbeitung:

$min_k = 36$

Kandidaten:

Id	worstscore	bestscore
192.168.1.3	36	36
192.168.1.1	28	39
192.168.1.4	27	29
192.168.1.2	13	25

NRA - Anfrageverarbeitung Beispiel

ClientIP	Bytes
192.168.1.3	17kB
192.168.1.4	12kB
192.168.1.2	11kB
192.168.1.5	4kB
192.168.1.6	2kB

ClientIP	Bytes
192.168.1.1	9kB
192.168.1.3	7kB
192.168.1.2	2kB
192.168.1.6	1kB
192.168.1.7	1kB

ClientIP	Bytes
192.168.1.1	19kB
192.168.1.4	15kB
192.168.1.3	12kB
192.168.1.5	5kB
192.168.1.7	2kB

Status der Anfrageverarbeitung:

$min_k = 36$

Kandidaten:

Id	worstscore	bestscore
192.168.1.3	36	36
192.168.1.1	28	32
192.168.1.4	27	29
192.168.1.2	13	25
192.168.1.5	4	18

NRA - Bestscore und Worstscore (2)

- Anfrage als Menge von Indexlisten $Q = \{L_j\}$ und Parameter k
- Den zuletzt durch sequenziellen Zugriff gelesenen Wert einer Indexliste L_i wird mit $high_i$ bezeichnet
- Und sei jedem Objekt o_j ein boolesches Array E zugewiesen, mit $E[i] = true$ falls Score von o_j in Liste L_i bekannt ist, $E[i] = false$
- Dann können wir schreiben

$$\text{worstscore}(o_i) = \sum_{j \in Q \wedge E[j]} score(j, o_i)$$

$$\text{bestscore}(o_i) = \text{worstscore}(o_i) + \sum_{j \in Q \wedge \neg E[j]} high_j$$

NRA - Algorithmus Idee und Korrektheit

- Halte alle Kandidaten-Objekte im Hauptspeicher.
- **Kandidaten sind diejenigen Objekte, die noch eine Chance haben in das Endergebnis zu gelangen**
- Welche Objekte können ignoriert werden?

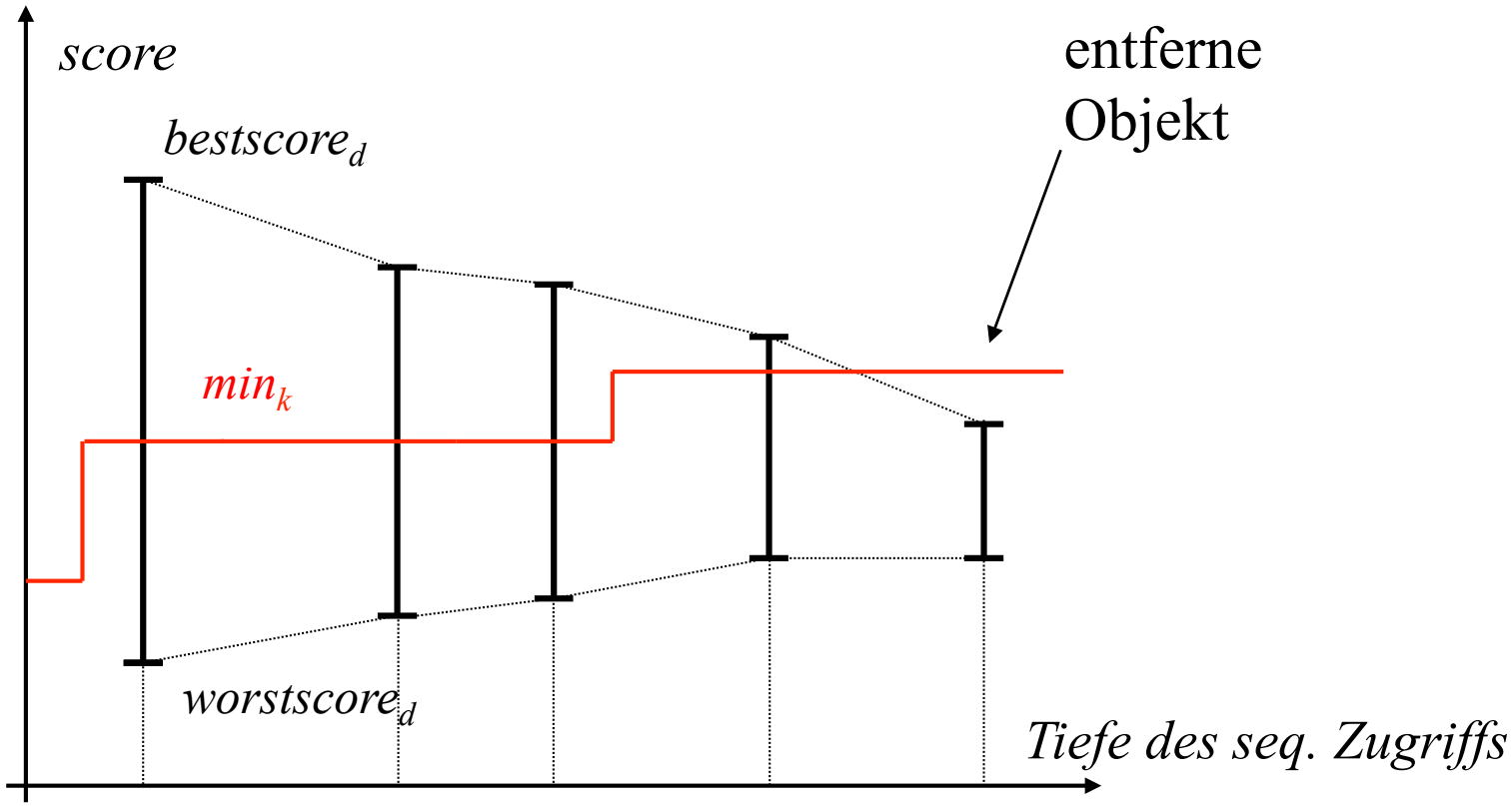
Eliminieren von nutzlosen Objekten (Pruning)

- Sei min_k die Worstscore des Objekts, das momentan auf Rang k ist
- Dann brauchen alle Objekte o_i mit bestscore$(o_i) \leq min_k$ nicht mehr betrachtet zu werden

Beobachtung

- Die Worstscore steigt stetig, während die Bestscore stetig kleiner wird.
- Wieso geht die Bestscore nach unten? Weil die Score auf der Scan-Linie kleiner wird.

Verhalten von Worstscore vs. Bestscore mit der Zeit



- Das Intervall zwischen Worstscore und Bestscore wird immer kleiner.
- Sobald Bestscore kleiner oder gleich min_k kann Objekt "weggeworfen" werden.

NRA - Algorithmus

1. Lese sequenziell in parallel aus jeder der m Listen L_i.
   - Betrachte die Scores auf der aktuellen Scan-Linie, d.h. $high_i$
   - Für alle gesehenen Objekte, berechne worstscore und bestscore
   - Für noch nicht gesehene Objekte ist τ wieder die Aggregation der $high_i$ Werte, in diesem Fall also die bestscore eines virtuellen, nicht gesehenen Objekts.
   - Betrachte die momentan k besten Objekte anhand ihrer worstscore (bei Ties wird bestscore benutzt, bei gleicher bestscore willkürlich)
   - min_k sei worstscore des Objekts auf Rang k
2. Ein Objekt o_i heißt Kandidat falls bestscore$(o_i) > min_k$. Stoppe den Algorithmus falls (a) es mindestens k verschiedene Objekte gibt und (b) Kein Kandidat existiert, der nicht zu den besten k gehört.

Seite 25 in http://researcher.watson.ibm.com/researcher/files/us-fagin/jcss03.pdf

Variationen

Approximative Version des TA

- Erlaube dem Algorithmus zu terminieren, falls es mindestens k Objekte gibt, deren Score größer oder gleich τ/θ ist (τ ist die Score auf der Scan-Linie; $\theta > 1$)

Combined Algorithmus (CA)

- Abwägung zwischen wahlfreien und sequenziellen Zugriffen.

Top-K Anfragen in Verteilten Systeme

Jede Indexliste liegt auf einem anderen Server, z.b. HTTP Zugriffs-Logs:

www.server1.com:

ClientIP	Bytes
192.168.1.3	17kB
192.168.1.4	12kB
192.168.1.2	11kB
192.168.1.5	4kB
192.168.1.6	2kB

www.server2.com:

ClientIP	Bytes
192.168.1.1	9kB
192.168.1.3	7kB
192.168.1.2	2kB
192.168.1.6	1kB
192.168.1.7	1kB

www.server3.com:

ClientIP	Bytes
192.168.1.1	19kB
192.168.1.4	15kB
192.168.1.3	12kB
192.168.1.5	5kB
192.168.1.7	2kB

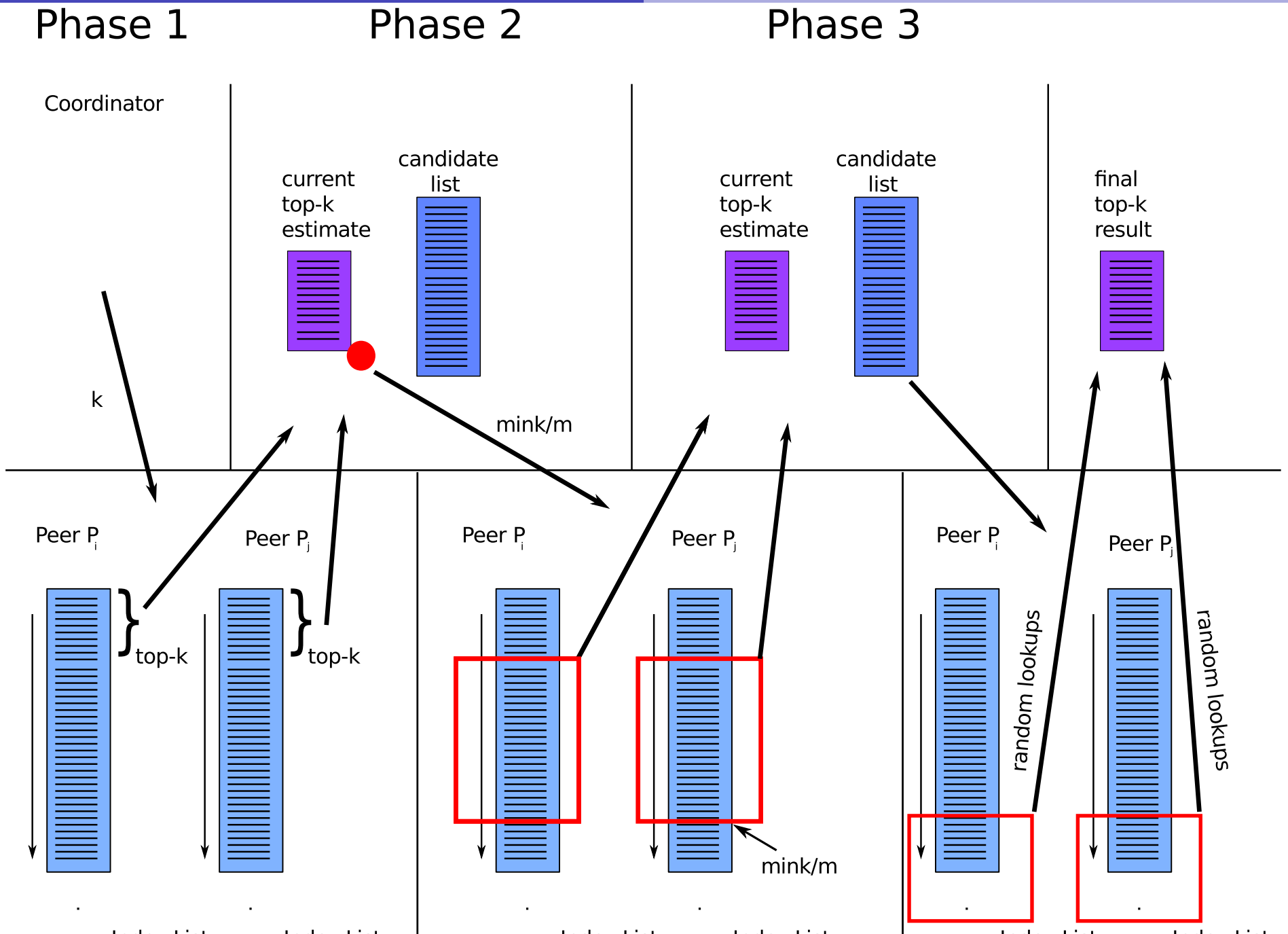
- **Beobachtung:** zuvor betrachtete Algorithmen haben sehr viele Zugriffe (auch wenn in Batches zugegriffen wird).
- In verteilten System möchte man dies nicht (wegen Roundtrips).
- Besser: Fixe Anzahl von Kommunikationsschritten (Phasen)

Three Phase Uniform Threshold Algorithm (TPUT)

Algorithmus für Top-k Anfragen in verteilten Systemen. Arbeitet in genau **drei Kommunikations-Schritten**.

1. Lese die ersten k Einträge der m Indexlisten L_i. Berechne aggregierte (partielle) Score der gesehenen Objekte.
2. Berechne Threshold min_k, der Wert des momentan k-besten Objekts. Lese von Indexliste nun alle Einträge mit $score > min_k/m$.
3. Lese fehlende Scores für alle Kandidaten-Objekte

Pei Cao, Zhe Wang: Efficient top-k query calculation in distributed networks. PODC 2004.


Phase 1
Phase 2
Phase 3
Coordinator
current top-k estimate
candidate list
current top-k estimate
candidate list
final top-k result
k
mink/m
Peer P_i
Peer P_j
top-k
top-k
Peer P_i
Peer P_j
mink/m
Peer P_i
Peer P_j
random lookups
random lookups
Index List
Index List
Index List
Index List
Index List
Index List

TPUT - Korrektheit

- **TPUT kann kein top-k Ergebnis verpassen**
- Annahme: Es wird doch ein Objekt, das ein top-k Ergebnis ist, verpasst. D.h. es wird nicht gesehen (nie!).
- Nicht gesehen bedeutet das Objekt hat eine Score von weniger als min_k/m in jeder Liste!
- D.h. aber, dass die gesamte (aggregierte) Score kleiner sein muss als min_k
- Es ist also doch nicht Teil des top-k Ergebnisses.