

Informationssysteme

Information Retrieval

- Einführung
- Retrievalmodelle
- Zipf-Verteilung
- Implementierung

Information Retrieval: Beispiele

Bibliothekskatalog

- Daten: Buchtitel, Autoren, Schlagworte, . . .
- Anfrage: „Suche *Database*-Bücher von *Ullman*.“

Volltextarchiv (wie Technical Reports, Online-Manuals)

- Daten: vollständiger Text jedes Technical Reports (eventuell nicht fehlerfrei aus Postscript-File rekonstruiert)
- Anfrage: „Finde einen *Übersichtsbericht* zum Thema *Performance Tuning in relationalen Datenbanken*.“

Internet Search Engine

- Daten: WWW-Seiten
- Anfrage: „Finde die *Homepage* von *Jürgen Kalinski*.“

Hochschulbibliothekszentrum des Landes NW, Köln

Besonderheit: Einem Dokument können mehrere Titelformen zugeordnet werden (z.B. Titel der Originalausgabe eines Buches und Titel der deutschen Übersetzung).

<u>Relation</u>	Tupel	Attribut	Bedeutung
Kandidatenschlüssel			
<u>Keyword</u>	1,7 Mio.		
(ID)		ID	Stichwort-ID
(Word)		Word	Titelstichwort
		Frequency	Vorkommens- häufigkeit
<u>Title</u>	7,3 Mio.		
(TID)		TID	Titel-ID
		Title	Titelform
		DID	Dokument-ID
		Title	Titelform
<u>Document</u>	5,9 Mio.		
<u>PersonName</u>	1,8 Mio.		
<u>Person</u>	1,4 Mio.		
<u>CorporateName</u>	1,3 Mio.		
<u>CorporateBody</u>	0,6 Mio.		

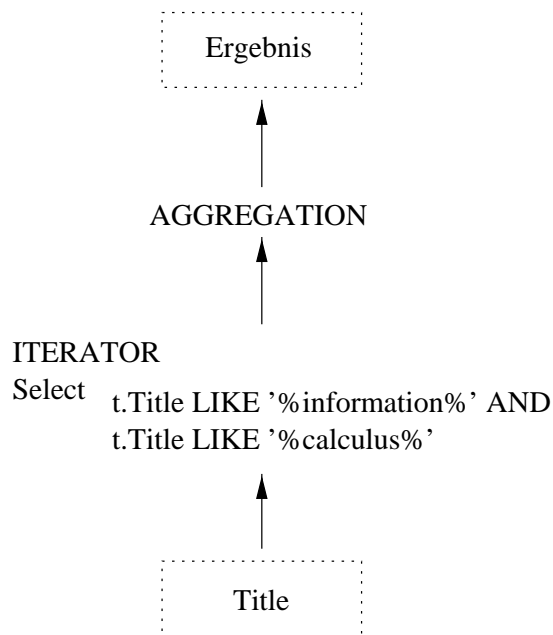
Anfrage:

„Bei wievielen Dokumenten kommen beide Stichwörter *information* und *calculus* in einer Titelform vor?“

„Naive“ Formulierung:

```
SELECT    COUNT(DISTINCT(t.DID))
FROM      Title t
WHERE     t.Title LIKE '%information%' AND
         t.Title LIKE '%calculus%'
```

Beachte: Es gibt keine Indexunterstützung bei LIKE-Selektionen! Die Anfrage kann also nur über einen sequentiellen Durchlauf durch die Relation *Title* ausgewertet werden.



Lösung:

Für das Retrieval wird jeder Titel explizit mit der Menge in ihm vorkommender Wörter assoziiert:

▷ Relation *Occurs*

Jedes Auftreten eines Stichworts mit ID *wid* in einem Titel *tid* zu Dokument *did* wird hier durch ein Tupel (*wid, tid, did*) repräsentiert.

<u>Relation</u>	Tupel	
Kandidatenschlüssel	Attribut	Bedeutung
<u><i>Occurs</i></u>	48 Mio.	
<i>(WID, TID)</i>	<i>WID</i>	Stichwort-ID
	<i>TID</i>	Titel-ID
	<i>DID</i>	Dokument-ID

Bei Änderungen in *Titel* muss die Relation *Occurs* von Seiten des Anwendungsprogramms oder mit Hilfe von Triggern entsprechend aktualisiert werden.

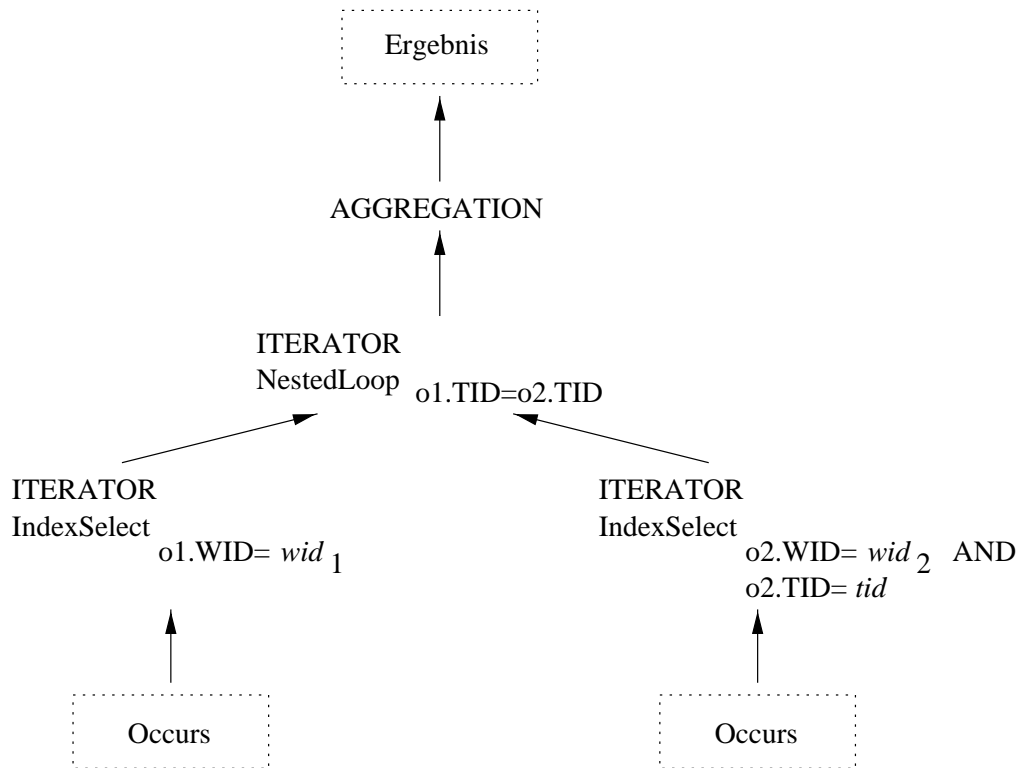
NB: Die *DID* wurde eingezogen, um die Anfrageauswertung zu optimieren.
(Normalform?)

Join-Formulierung:

Seien wid_1 und wid_2 die IDs der Stichwörter *information* und *calculus*.

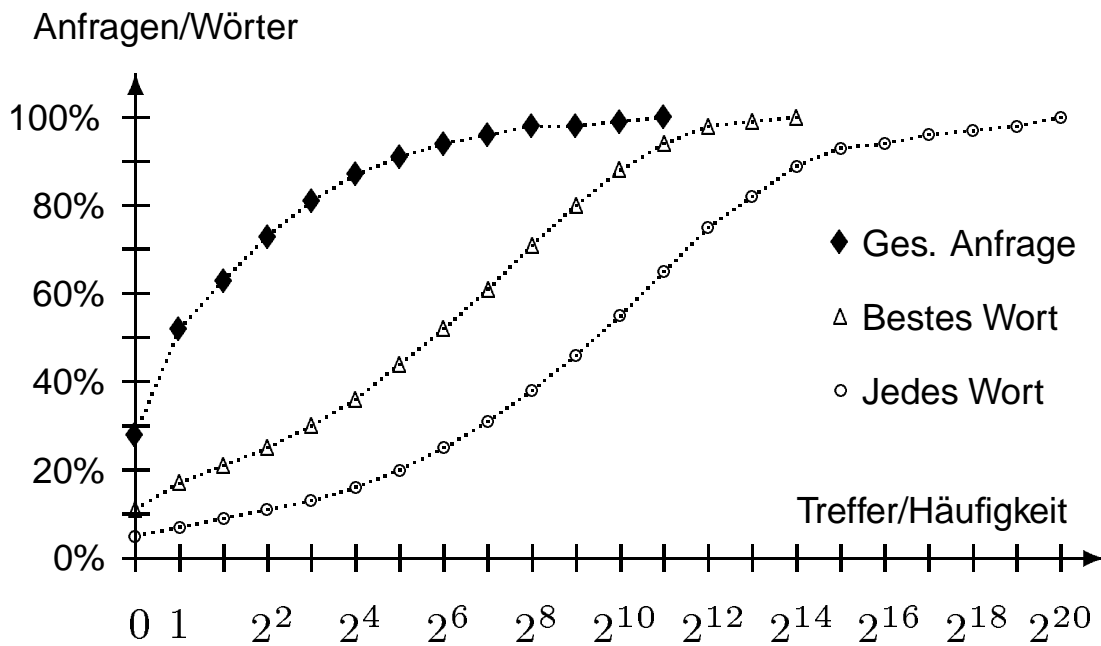
```
SELECT  COUNT(DISTINCT(o1.DID))
FROM    Occurs o1, Occurs o2
WHERE   o1.TID = o2.TID AND
        o1.WID =  $wid_1$  AND o2.WID =  $wid_2$ 
```

Annahme: Der Primärschlüssel (WID, TID) von *Occurs* wird durch einen Index unterstützt.



Verbesserung der Join-Formulierung:

Anordnung der Wörter in der Form, daß der Häufigkeitswert *Frequency* sinkt, je weiter außen ein Wort beim Nested Loop Joins anzutreffen ist.



Durchschnittswerte

Ergebnisgröße	89
<i>Frequency</i> -Wert eines Anfrageworts	36.614
minimaler <i>Frequency</i> -Wert pro Anfrage	642

Problem der Join-Formulierung:

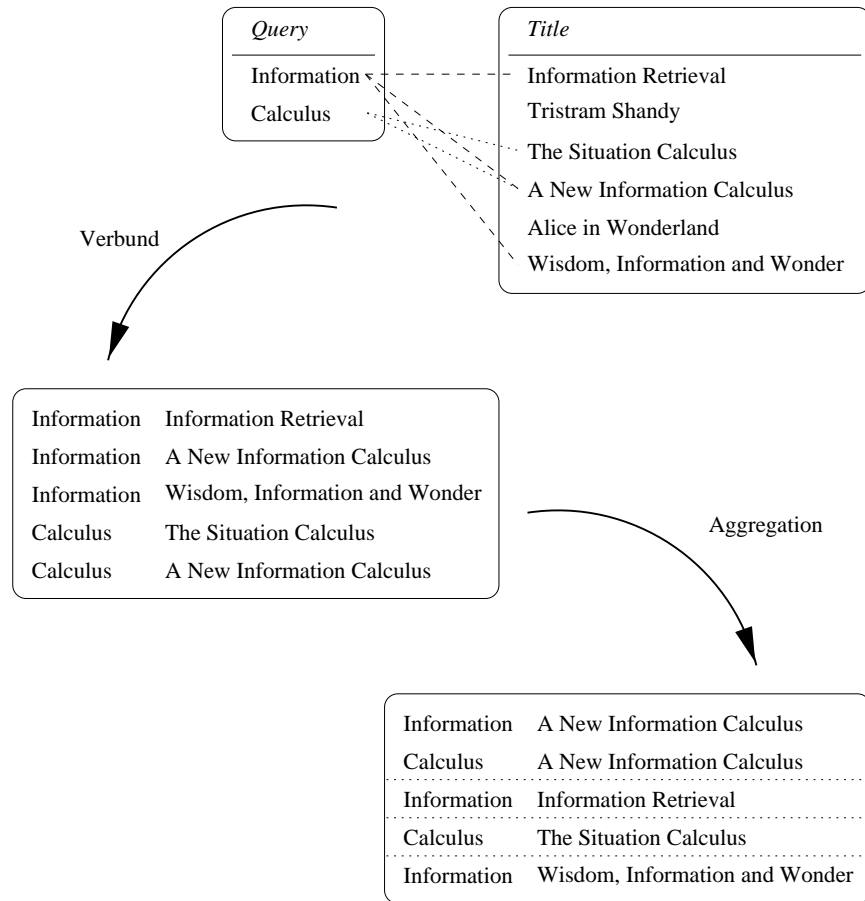
Komplexität der Optimierung, falls die Anfrage aus vielen Stichwörtern besteht. Jedes Wort bedingt eine Verbundoperation.

Eine-Anfrage-Formulierung:

In einem ersten Schritt werden die IDs der k Anfragewörter in die Relation *Query* eingefügt. Dann kann mit (im wesentlichen) ein und derselben Anfrage gearbeitet werden.

```
SELECT    COUNT(DISTINCT(DID))
FROM      Occurs
WHERE     WID=wid1 AND TID IN
          (SELECT  DISTINCT(t.TID)
           FROM    Query s, Occurs t
           WHERE   s.WID=t.WID
           GROUP BY t.TID
           HAVING  COUNT(t.WID)=k)
```

Prinzip der Eine-Anfrage-Formulierung:



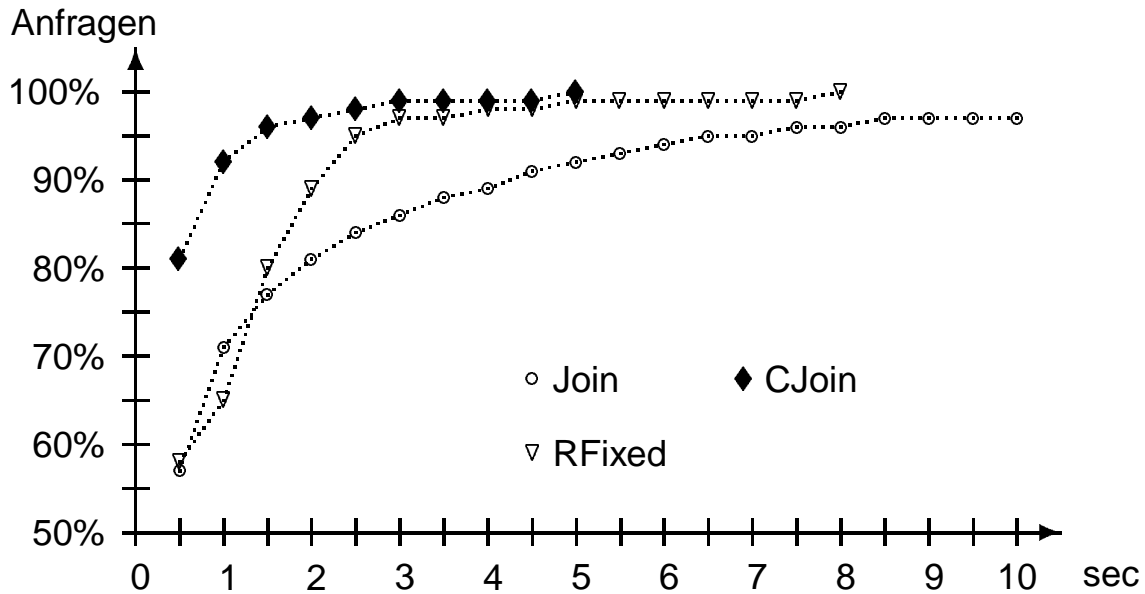
Laufzeit:

$$\left(\sum_{i=1}^k f_i \right) \cdot \log \left(\sum_{i=1}^k f_i \right),$$

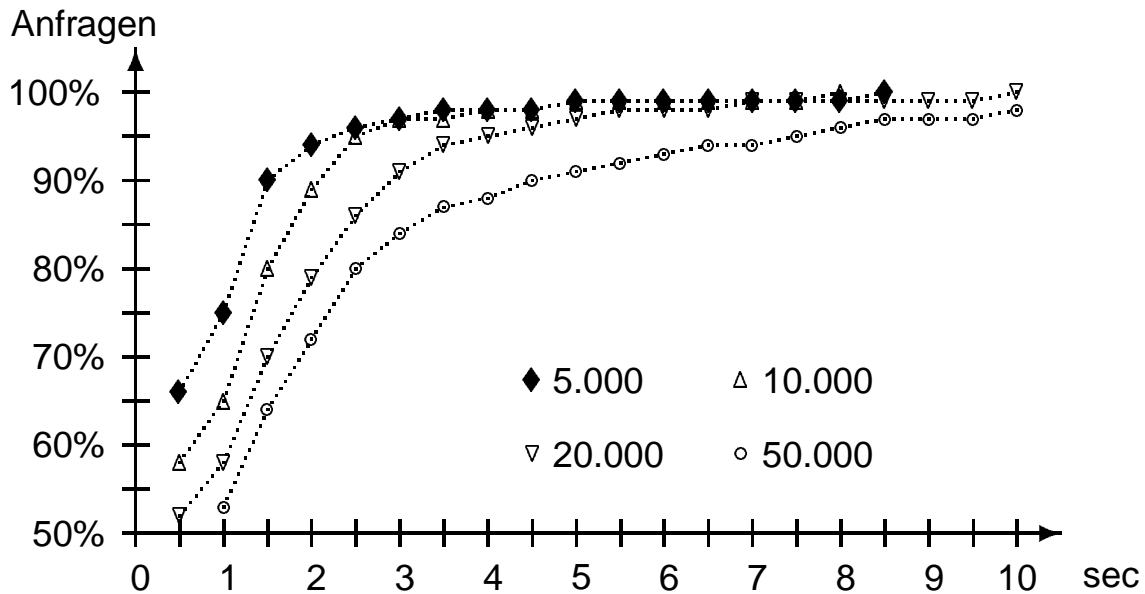
wobei f_i der *Frequency*-Wert des i -ten Anfrageworts ist.

- ▷ Ignorieren der „Stop-Wörter“,
bei denen f_i einen vorgegebenen Schwellwert überschreitet.

Laufzeitverhalten verschiedener Anfrageformulierungen



Eine-Anfrage-Formulierung bei unterschiedlichen Schwellwerten



Zusammenfassung:

1. Erheblicher zusätzlicher Speicherbedarf durch Relation *Occurs*
2. Berücksichtigung der extrem unausgeglichenen Häufigkeitswerte der Anfragewörter bei der Anfrageauswertung

Wie wir sehen werde, spiegelt die Häufigkeitsverteilung der Anfragewörter ein generelles Merkmal von Textsammlungen wieder. Hierdurch werden Speicherstrukturen und Auswertungsstrategien maßgeblich beeinflusst.

- Einführung
- Retrievalmodelle
- Zipf-Verteilung
- Implementierung

Datenmodell:

Sammlung von (mathematischen) Konzepten und Operatoren, mit denen die statischen/dynamischen Eigenschaften der Anwendungswelt erfaßt werden

- Objekte, Attribute, Beziehungen
- Operatoren
- Integritätsbedingungen

Retrievalmodell:

- Repräsentation von Dokumentinhalten/Dokumenten
- Repräsentation von Informationsbedürfnissen/Anfragen
- Matching-Funktion

Beispiele:

- Exact-Match Retrieval, Boolesches Retrieval
- Best-Match Retrieval, Vektorraum-Retrieval

Boolesches Retrieval:

- Dokument
 - ▷ Menge von Stichwörtern
 - Anfrage
 - ▷ aussagenlogische (Boolesche) Verknüpfungen von Wörtern
- Beispiel:

$$(\textit{information} \vee \textit{data}) \wedge \neg \textit{mathematical}$$

- Matching:
 - Ein Dokument erfüllt eine Konjunktion (Disjunktion), falls es jedes (mindestens ein) Teiglied erfüllt.
 - Es erfüllt ein Stichwort, falls es dieses Wort enthält.

Boolesches Retrieval:

Sei $D = \{d_1, \dots, d_m\}$ die Menge aller Dokumente und
 $T = \{t_1, \dots, t_n\}$ die Menge aller Stichwörter (Terme).

Dokumente als Wortmengen werden durch ihre charakteristische Funktion beschrieben, d.h. $w_{d,t} = 1$ genau dann, wenn t in d vorkommt.

Die Menge aller Anfragen Q ist die kleinste Obermenge von T , so daß

$$q_1, \dots, q_k \in Q \Rightarrow (q_1 \wedge \dots \wedge q_k) \in Q$$

$$q_1, \dots, q_k \in Q \Rightarrow (q_1 \vee \dots \vee q_k) \in Q$$

$$q \in Q \Rightarrow (\neg q) \in Q$$

Die *Retrievalfunktion* wertet jedes Dokument bezüglich einer Anfrage aus und liefert als Ergebnis den Wert 0 oder 1.

Definition:

Retrievalfunktion $eval_{\text{Bool}} : D \times Q \rightarrow \{0, 1\}$:

$$eval_{\text{Bool}}(d, t) = w_{d,t} \quad \text{für alle Terme } t \in T$$

$$eval_{\text{Bool}}(d, q_1 \wedge \dots \wedge q_k) = \begin{cases} 1 & , \text{ falls } eval_{\text{Bool}}(d, q_i) = 1 \text{ für alle } 1 \leq i \leq k \\ 0 & , \text{ sonst} \end{cases}$$

$$eval_{\text{Bool}}(d, q_1 \vee \dots \vee q_k) = \begin{cases} 0 & , \text{ falls } eval_{\text{Bool}}(d, q_i) = 0 \text{ für alle } 1 \leq i \leq k \\ 1 & , \text{ sonst} \end{cases}$$

$$eval_{\text{Bool}}(d, \neg q) = \begin{cases} 1 & , \text{ falls } eval_{\text{Bool}}(d, q) = 0 \\ 0 & , \text{ sonst} \end{cases}$$

Das *Retrievalergebnis* besteht aus der Menge aller qualifizierenden Dokumente

$$\{ d \mid d \in D, eval_{\text{Bool}}(d, q) = 1 \}.$$

Boolesches Retrieval

Vorteile:

- Möglichkeit zur genauen Charakterisierung der Ergebnismenge
- existierende effiziente Implementierungen
(„klassisches“ System: IBM STAIRS)

Nachteile:

- Größe der Ergebnismenge schwierig zu kontrollieren
- keine Ordnung der Antwortmenge in mehr oder weniger relevante Dokumente
- keine Unterscheidung zwischen wichtigen und weniger wichtigen Anfragetermen

Vektorraum-Retrieval

Wie zuvor:

$$D = \{d_1, \dots, d_m\} \quad \text{und} \quad T = \{t_1, \dots, t_n\}$$

- Dokument \triangleright n -dimensionaler Vektor

$$\vec{d} = \begin{pmatrix} w_{d,t_1} \\ \vdots \\ w_{d,t_n} \end{pmatrix}$$

Dabei ist w_{d,t_i} die „Relevanz“ von d für t_i .

- Anfrage \triangleright n -dimensionaler Vektor \vec{q}

$$\vec{q} = \begin{pmatrix} w_{q,t_1} \\ \vdots \\ w_{q,t_n} \end{pmatrix}$$

Dabei ist w_{q,t_i} die „Relevanz“ von t_i für q .

- Matching \triangleright „Ähnlichkeit“ zwischen \vec{d} und \vec{q}

Mögliche Retrievalfunktionen

Skalarprodukt:

$$eval_{\text{Skalar}}(\vec{d}, \vec{q}) = \vec{d} \cdot \vec{q} = \sum_{t \in T} w_{d,t} w_{q,t}$$

Kosinusmaß:

$$eval_{\text{cos}}(\vec{d}, \vec{q}) = \frac{\vec{d} \cdot \vec{q}}{|\vec{d}| \cdot |\vec{q}|} = \frac{\sum_{t \in T} w_{d,t} w_{q,t}}{\sqrt{\sum_{t \in T} w_{d,t}^2} \sqrt{\sum_{t \in T} w_{q,t}^2}}$$

Retrievalergebnis: Dokumente sortiert nach Ähnlichkeit

Bemerkung:

Für binäre Gewichte ($w_{d,t}, w_{q,t} \in \{0, 1\}$) liefert $eval_{\text{Skalar}}$ die Anzahl an Anfragetermen, die im Dokument vorkommen („Coordination Level Match“).

Coordination Level Match in SQL

Variante der Eine-Anfrage-Formulierung

```
SELECT  t.TID, COUNT(t.TID)
FROM    Query s, Occurs t
WHERE   s.WID=t.WID
GROUP BY t.TID
ORDER BY 2 DESC
```

Mögliche Gewichtungen

Typischer Aufbau: TF * IDF

- TF (term frequency):
häufiges Vorkommen von t in $d \rightsquigarrow$ hoher TF-Wert
- IDF (inverse document frequency):
häufiges Vorkommen von t in vielen Dokumenten der Datenbank \rightsquigarrow
niedriger IDF-Wert

$f_{d,t}$ Anzahl der Vorkommen von t in d

Notation: f_d Anzahl an Dokumenten

f_t Anzahl an Dokumenten, die t enthalten

TF-Maße:

$f_{d,t}$ als TF-Maß bevorzugt große Dokumente in inadäquater Weise.

Besser:

$$c + (1 - c) \frac{f_{d,t}}{\max \{f_{d,t_i} \mid t_i \in T\}} \quad (0 \leq c \leq 1)$$

$c = 1$ binäre Gewichte, kein Einfluß der Termhäufigkeit

$c = 0$ maximaler Einfluß der Termhäufigkeit, normiert

IDF-Maße:

Üblich ist $\log \frac{f_d}{f_t}$.

Vorteile des Vektorraum-Retrievals:

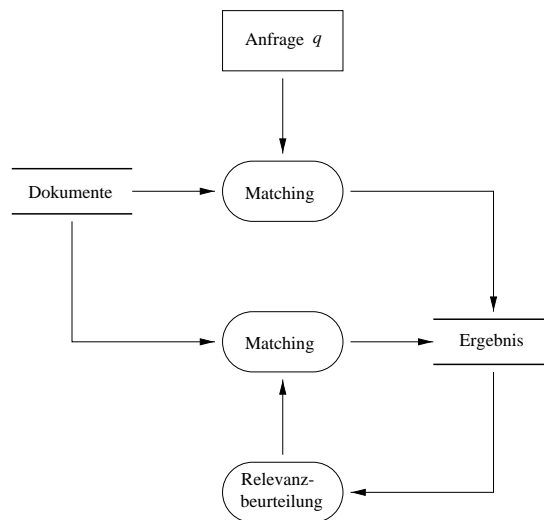
- einfach, anschaulich, benutzerfreundlich
- erprobt (SMART-Projekt von Salton et al. seit 1961)
- Unterstützung von Relevance Feedback

Nachteile des Vektorraum-Retrievals:

- stark heuristisch geprägt

Relevance Feedback:

1. Bearbeitung der Anfrage \vec{q}
2. Benutzer markiert in einer Teilmenge des Retrievalergebnisses relevante Dokumente D_R und nicht relevante Dokumente $D_{\bar{R}}$
3. Das Retrievalsystem kombiniert \vec{q} und Feedback-Information zu einer modifizierten Anfrage \vec{q}'



Intuition: \vec{q}' wird in die Richtung der relevanten Dokumente und von den nicht relevanten Dokumenten weg bewegt.

$$\vec{q}' = \alpha \cdot \vec{q} + \frac{\beta}{|D_R|} \sum_{\vec{d} \in D_R} \vec{d} - \frac{\gamma}{|D_{\bar{R}}|} \sum_{\vec{d} \in D_{\bar{R}}} \vec{d}$$

aus: C.J. van Rijsbergen, "Information Retrieval", 1979

	Data Retrieval	Information Retrieval
Matching	Exact match	Partial match, best match
Inference	Deduction	Induction
Model	Deterministic	Probabilistic
Items wanted	Matching	Relevant
Query language	Artificial	Natural

- Einführung
- Retrievalmodelle
- Zipf-Verteilung
- Implementierung

G.K. Zipf, 1949:

Seien t_1, \dots, t_n die Worte eines natürlichsprachlichen Textes sortiert nach abnehmenden Häufigkeitswerten f_1, \dots, f_n .

Dann ist $i \cdot f_i$ konstant.

Mit anderen Worten: Die Vorkommenswahrscheinlichkeit p_i von t_i ist umgekehrt proportional zu i :

$$p_1 = \frac{c}{1}, \quad p_2 = \frac{c}{2}, \quad \dots, \quad p_n = \frac{c}{n},$$

wobei

$$c = \frac{1}{H_n} \quad \text{und} \quad H_n = \sum_{k=1}^n \frac{1}{k} \quad (\text{Harmonische Reihe}).$$

Zur Erinnerung:

Mit $\gamma = 0.57721566 \dots$ (Eulersche Konstante) gilt:

$$H_n \approx \ln(n) + \gamma + \frac{1}{2n}.$$

Zipf-Verteilung:

Für eine Dokumentsammlung mit n Stichwörtern und m Wortvorkommen erhalten wir gemäß der Zipf-Verteilung

$$f_i = \frac{c}{i} \cdot m \approx \frac{m}{i \cdot (\ln(n) + \gamma + 1/2n)}.$$

Verteilung von Worthäufigkeiten

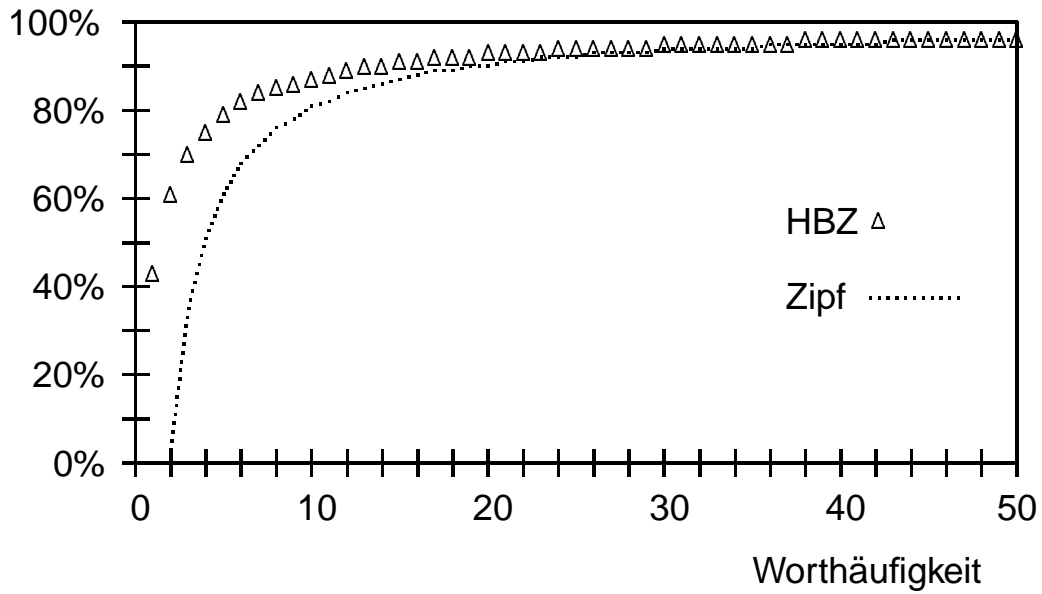
Hochschulbibliothekszentrum des Landes NW, Köln

8.706.720 Buchtitel

Stichwörter 1.670.861

Wortvorkommen 48.388.635

Stichwörter



Verteilung von Worthäufigkeiten

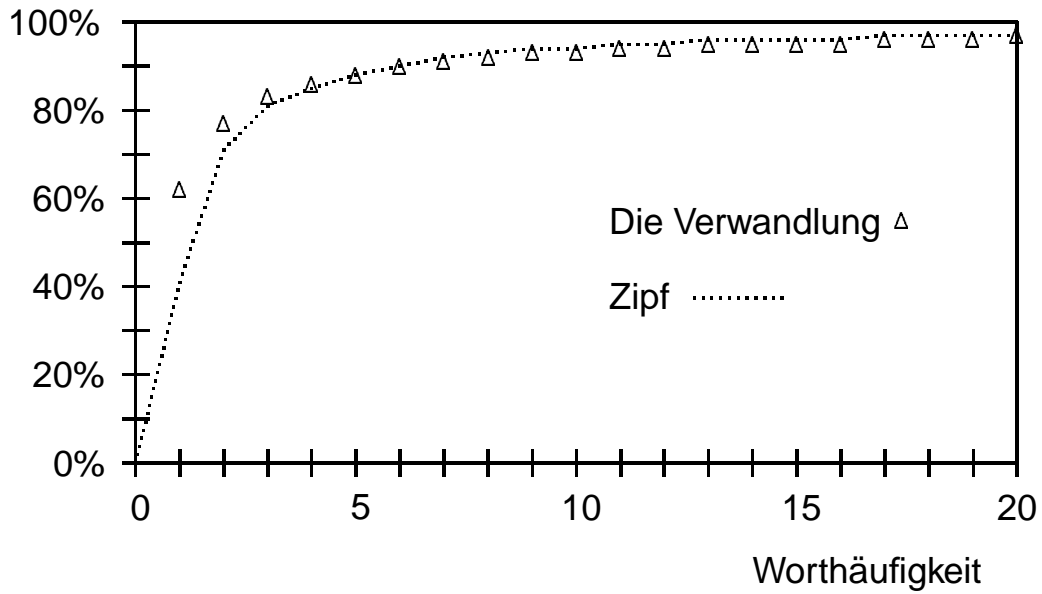
Franz Kafka (1883-1924)

Die Verwandlung

Stichwörter 3.735

Wortvorkommen 19.256

Stichwörter



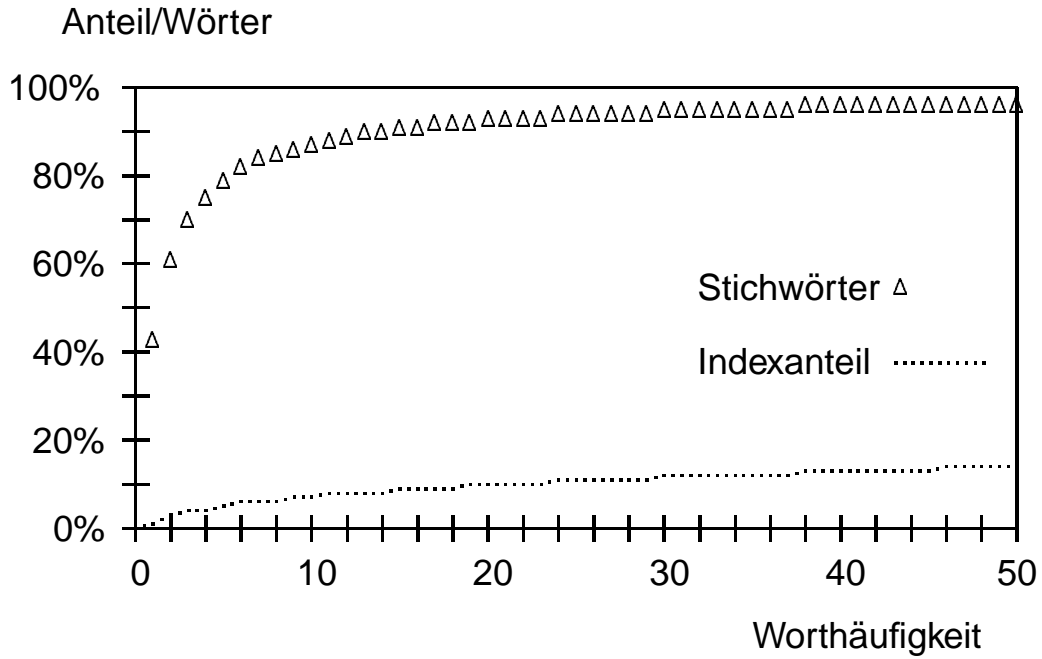
Auswirkungen der Zipf-Verteilung:

Anteil der Dokumente, die mit den k am häufigsten vorkommenden Stichwörtern assoziiert sind („Indexanteil der k häufigsten Wörter“):

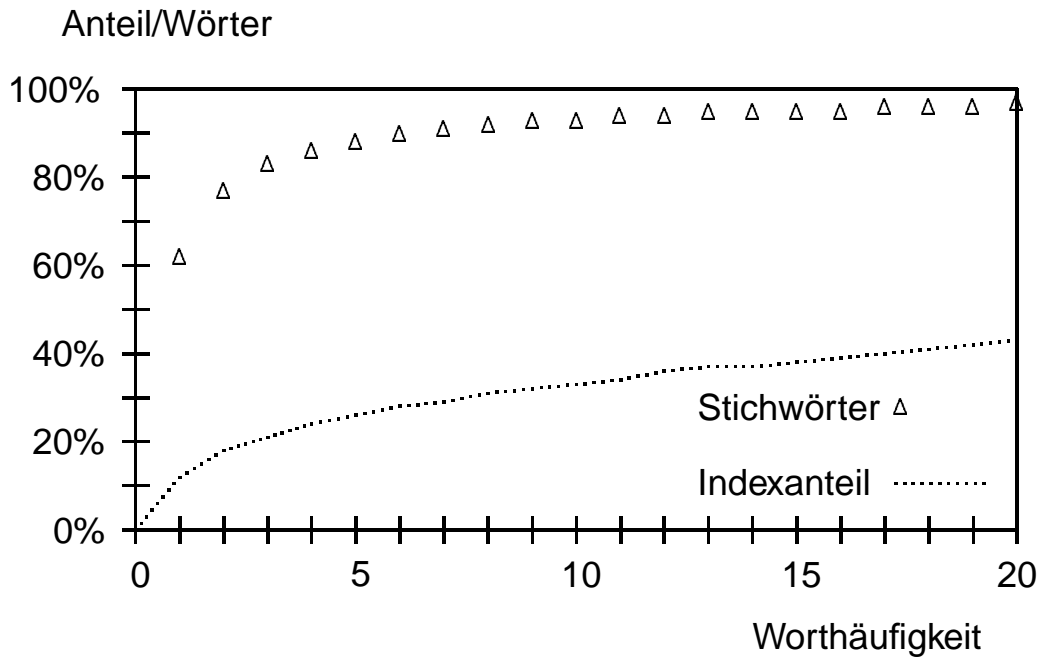
$$\frac{\sum_{i=1}^k f_i}{\sum_{i=1}^n f_i} = \frac{H_k}{H_n}.$$

n	k	Assoziierter Dokumente
1.000.000	100	36%
	1.000	52%
	10.000	68%
	100.000	84%

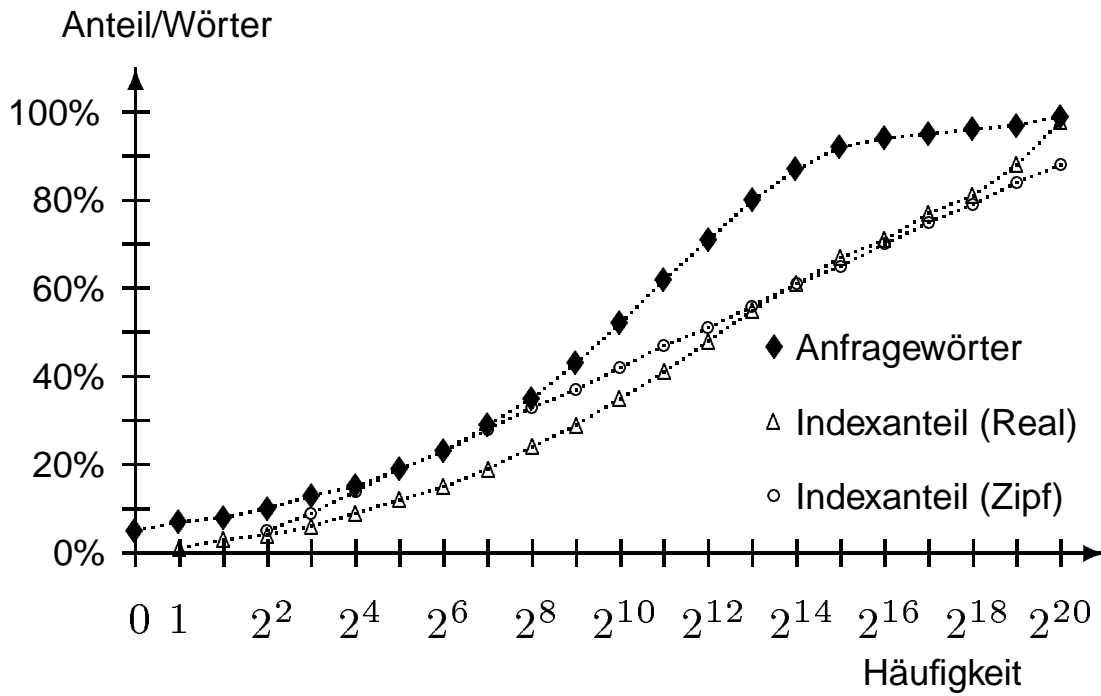
Hochschulbibliothekszentrum des Landes NW, Köln



Kafka: „Die Verwandlung“



Hochschulbibliothekszentrum des Landes NW, Köln



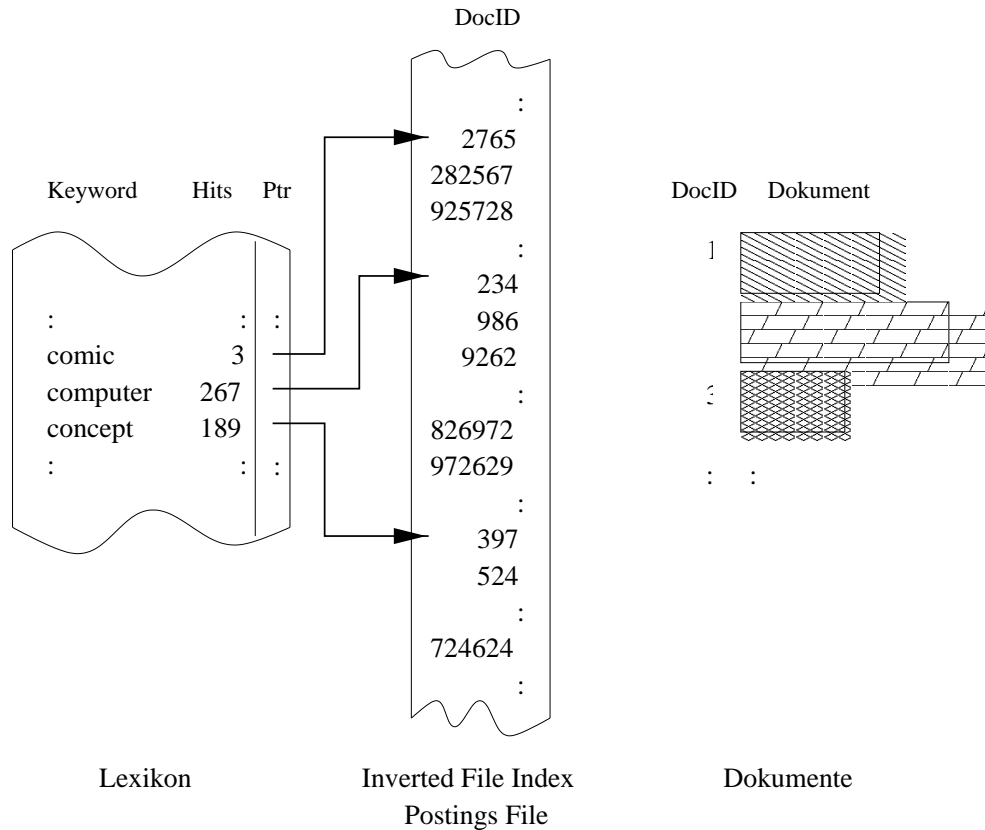
- Einführung
- Retrievalmodelle
- Zipf-Verteilung
- Implementierung

Implementierung von IR-Systemen:

- String Matching
Sequentielle Suche nur bei kleinen Dokumentsammlungen möglich.
- Lexikon für Stichwörter +
Invertierte Liste (Inverted File Index) für assoziierte Dokumente
- Signaturen

Invertierte Listen:

Ein *Inverted File Index (Postings File)* ordnet jedem Term eine (physikalisch möglichst zusammenhängende) Liste von Informationen über Vorkommen des Terms in den Textdokumenten zu.



Das Lexikon wird üblicherweise als sortierte Liste, B^+ -Baum, mittels Hashing o.ä. organisiert.

(\Rightarrow effiziente Termsuche)

Invertierte Listen:

Vorteil:

- Effizienz: Invertierte Listen sind die in den meisten kommerziellen Bibliotheks- und IR-Systemen verwendete Zugriffsmethode.
- Boolesches Retrieval mittels Mengenoperationen:

$t_1 \vee t_2$ Vereinigung der invertierten Listen

$t_1 \wedge t_2$ Schnitt der invertierten Listen

$t_1 \wedge \neg t_2$ Differenzbildung

Bei Sortierung der invertierten Listen: Realisierung in Linearzeit

Nachteile:

- Speicherbedarf: 10% bis 100% des Textdatenaufkommens oder mehr für Inverted File Index.
- Aufbau und Aktualisierung des Inverted File Index sind aufwendig.

Invertierte Listen:

Die in der invertierten Liste zu jedem Term abgelegten Informationen können mehr oder weniger detailliert sein und beeinflussen damit den Speicheraufwand wie auch die Anfragemöglichkeiten.

mittelfeiner Index Term \mapsto Liste von DocIDs

feiner Index Term \mapsto Liste von Tupeln mit DocID, Position des Terms innerhalb des Dokuments, Gewichtung, . . .

Vorteil: Anfragen mit "Entfernungsangaben", Vektorraum-Retrieval

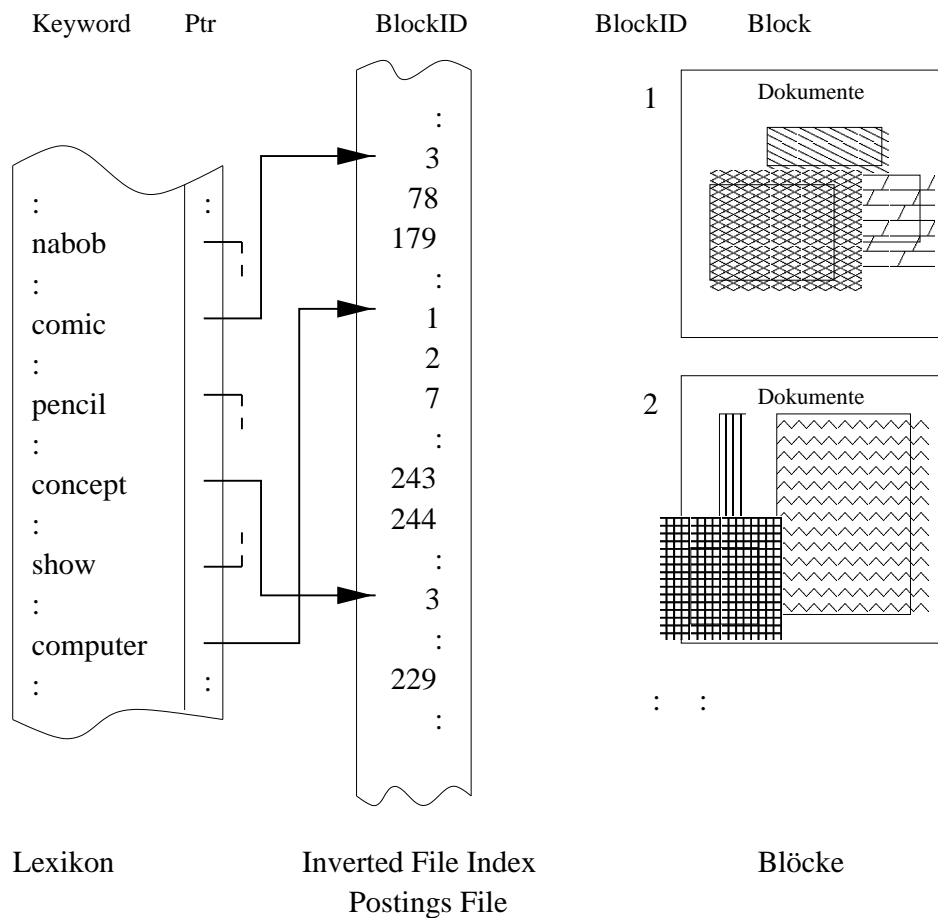
grober Index Term \mapsto Liste von Blöcken (siehe *Glimpse*)

Nachteil: False Matches

Glimpse (Manber/Wu, 1993):

Ansatz: zweistufige Vorgehensweise (two-level query approach),
Kombination von String Matching und Invertierten Listen

1. Dateien werden in maximal 256 *Blöcken* gruppiert.
2. Zu jedem Term wird eine grobe invertierte Liste von BlockIDs verwaltet. Eine BlockID b wird einem Term t zugeordnet, falls t in einer der Dateien von b vorkommt.



Glimpse (Manber/Wu, 1993):

Anfrage $t_1 \wedge t_2$:

1. Das Lexikon wird sequentiell mit einem String-Matching-Algorithmus (*agrep*) durchsucht. Die invertierten Listen von t_1 und t_2 werden geschnitten.
2. In jedem Block des Durchschnitts der invertierten Listen wird jedes Dokument mittels *agrep* durchlaufen, um das gemeinsame Vorkommen von t_1 und t_2 zu überprüfen.

Vorteil:

- Die Verwendung von *agrep* erlaubt approximative Suche.

Nachteil:

- False Matches durch groben Index.
Wenn t_1 und t_2 zwar innerhalb eines Blocks b , aber nicht innerhalb eines Dokuments dieses Blocks vorkommen, qualifiziert sich b beim ersten Auswertungsschritt, so daß alle Dateien des Blocks mit *agrep* erfolglos gelesen werden.

Komprimierung von Invertierten Listen:

1. Aufsteigend sortierte Folge von DocIDs zu einem Term

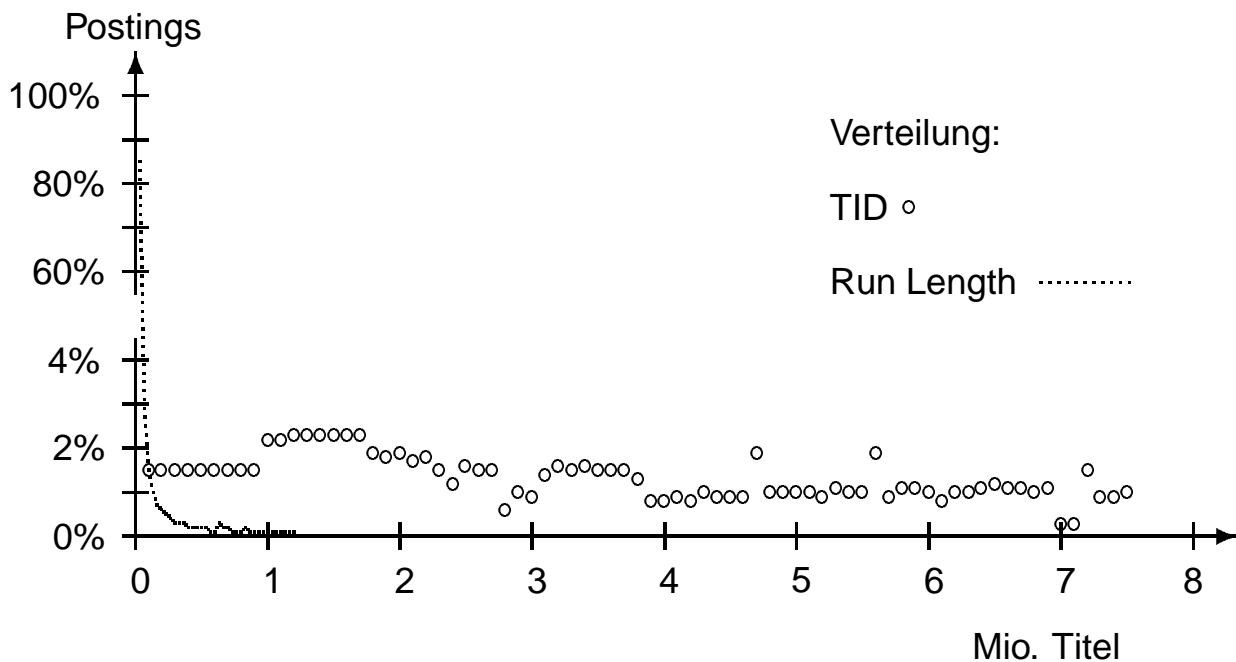
7	14	18	27	30	31
---	----	----	----	----	----

2. Abbildung auf „Laufängen“ (Run Lengths)

7	7	4	9	3	1
---	---	---	---	---	---

3. Komprimierte Speicherung der Laufängen-Kodierung (z.B. γ -Code)

11011	11011	11000	1110001	101	0
-------	-------	-------	---------	-----	---



Speicherbedarf für HBZ-Anwendung (48 Mio. Run Lengths)

Kodierung	Bits
23-Bit-Kodierung	1.112.938.605
γ -Code	734.148.759
δ -Code	595.356.411
optimaler Code (Shannon)	537.316.679

Laufzeitverhalten komprimierter vs. nicht-komprimierter Listen

- HBZ-Anwendung

durchschn. Anzahl Terme / Anfrage	2,5
durchschn. Anzahl Postings / Anfrageterm	36.615
durchschn. Anzahl Postings / Anfrage	91.396

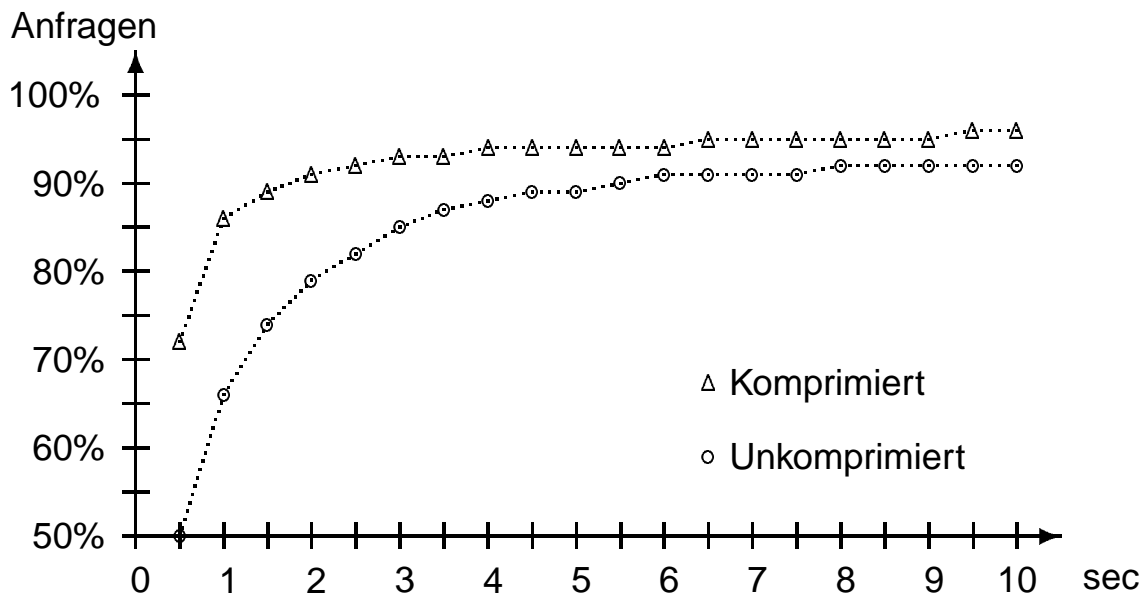
- Boolesches Retrieval, konjunktive Anfragen mittels Berechnung des Durchschnitts der invertierten Listen

Laufzeitverhalten komprimierter vs. nicht-komprimierter Listen

- Lauflängenkomprimierung mit δ -Code und Verwaltung der Bitsequenzen in Binary Large Objects der Datenbank
- Nicht komprimierte invertierte Listen mittels

```
SELECT  TID
FROM    Occurs
WHERE   WID= $wid_i$    ORDER BY TID
```

Explizite Sortierung wird durch Index (WID,TID) umgangen!



Best-Match Retrieval mit invertierten Listen:

Naiv

1. Berechnung der Vereinigung der invertierten Listen für alle $t \in q$
2. Lesen aller Repräsentationen zu allen Dokumenten der Vereinigung und Berechnung von $eval(d, q)$

▷ ineffizient: zu viele Lesevorgänge

On the fly

- Speichern der Worthäufigkeit f_t im Lexikon und der Within-Document Frequency $f_{d,t}$ bei den Postings der invertierten Liste zu t
- daraus Berechnung der Indexterm-Gewichte $w_{d,t}$ während des Lesens der invertierten Liste, z.B. als $w_{d,t} = f_{d,t} \cdot \log(f_d/f_t)$
- „inkrementelle“ Berechnung von $eval(d, q)$ beim Durchlaufen der invertierten Liste in einem *Akkumulator* A_d

Best-Match Retrieval mit invertierten Listen

$$eval_{\text{Skalar}}(d, q) = \sum_{t \in T} w_{d,t} \cdot w_{q,t}$$

Ranking

- (1) AccuSet \mathcal{A} (leere Menge aller Akkumulatoren)
- (2) **for all** $(t, w_{q,t}) \in q$
- (3) **for all** $(d, f_{d,t})$ in invertierter Liste zu t mit f_t
- (4) berechne $w_{d,t}$
- (5) **if** $A_d \notin \mathcal{A}$
- (6) $A_d = 0$ (neuer Akkumulator)
- (7) $\mathcal{A} = \mathcal{A} \cup \{A_d\}$
- (8) $A_d = A_d + w_{d,t} \cdot w_{q,t}$
- (9) **return** \mathcal{A}

Bemerkungen:

- Zuordnung $d \mapsto A_d$ über ein Feld der Länge m , Hashing, . . .
 - Problematisch: Speicherplatz für \mathcal{A}
 - Problematisch: $eval_{\text{Skalar}}(d, q)$ wird für alle Dokumente der invertierten Listen berechnet, obwohl der Benutzer eventuell nur an den k besten interessiert ist
- ▷ zusätzliche Heuristiken und Optimierungen

Best-Match Retrieval mit invertierten Listen

- Testdaten: 100.000 künstlich erzeugte Volltexte
- Anfragen: künstlich erzeugt mit 5 / 10 / 20 Wörtern

Operation	Laufzeit (sec)		
	5 Wörter	10 Wörter	20 Wörter
Best-Match Retrieval mit Eine-Anfrage-Realisierung	14,0	26,9	56,6
Lesen			
nicht komprimierter invertierter Listen	2,9	5,5	10,3
Best-Match Retrieval mit komprimierten invertierten Listen	5,6	9,9	17,9

Signaturen:

Idee:

1. Dokument ▷ Kombination von Hash-Werten seiner Wörter
(Dokument-Signatur)
2. Anfrage ▷ Kombination von Hash-Werten ihrer Wörter
(Anfrage-Signatur)
3. Matching ▷ sequentieller Vergleich der Signaturen aller
Dokumente mit der Signatur der Anfrage

Dokumente

A database-management system (DBMS) consists
of a collection of interrelated data and a set of programs
to access those data.
:

Stop dawdling and do something useful! Don't dawdle
away your time! I do not sympathize with your ambition
to go on the stage.
:

Signaturen

da ma sy co co in da se pr ac da
:

st da so us da aw ti sy am go st
:

Anfrage

database system

Signatur

da sy

Signaturen:

1. Jedes Wort wird auf eine Bitsequenz abgebildet. Die Signatur eines Dokuments ist das logische ODER der Bitsequenzen seiner Wörter. Die Signaturen aller Dokumente sind in einer Signaturdatei gespeichert.

2. Behandlung der Anfrage analog

Wort	Signatur
database	001 000 110 010
system	000 010 101 001
Anfragesignatur	001 010 111 011

3. Die Signaturdatei muß klein genug sein, daß sie sequentiell durchsucht werden kann. Falls in einer Dokumentsignatur alle 1-Bits der Anfragesignatur auf 1 gesetzt sind, qualifiziert sich das Dokument als Kandidat.
4. Jedes Kandidaten-Dokument muß eingelesen werden, um zu überprüfen, ob es tatsächlich die Anfragewörter enthält oder ein False Match ist.

Funktionales Modell: IR

