MI-IR-M: Information Retrieval – Gliederung

1. Suchmaschinen und Information Retrieval
2. Architektur von Suchmaschinen
3. Evaluierung von Suchmaschinen
4. Retrieval-Modelle
5. Ranking mit Indexstrukturen
6. Textverarbeitung
7. Anfragen / Benutzerschnittstellen / Interaktion
8. Crawling und Texterfassung
9. Suchmaschinenoptimierung, Werbung, ...
10. Bilder und vertikale Suchlösungen

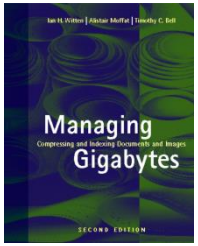
Grundlage der Vorlesung:

Croft, W Bruce; Metzler, Donald; Strohman, Trevor: Search Engines: Information Retrieval in Practice. International edition. Boston: Addison-Wesley / Pearson, 2009.

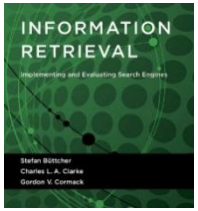
Gliederung

I. Einordnung
II. Ein Modell für das Ranking (als Beispiel)
III. Invertierte Indexstrukturen
IV. Kompression
V. Hilfsstrukturen
VI. Indexerstellung
VII. Anfrageverarbeitung

Weiterführende Literatur:



Ian H. Witten, Alistair Moffat, & Timothy C. Bell: Managing Gigabytes: Compressing and Indexing Documents and Images. Morgan Kaufmann Publishers, Inc., Second edition; 1999; 1-55860-570-3



Stefan Büttcher, Charles L. A. Clarke, and Gordon V. Cormack. Information Retrieval: Implementing and Evaluating Search Engines. MIT Press, 2010

I. Einordnung: Indexe

- Indexe sind Datenstrukturen, die entwickelt wurden, um die Suche zu beschleunigen
  - Textsuche hat spezielle Anforderungen, die zu speziellen Datenstrukturen führen

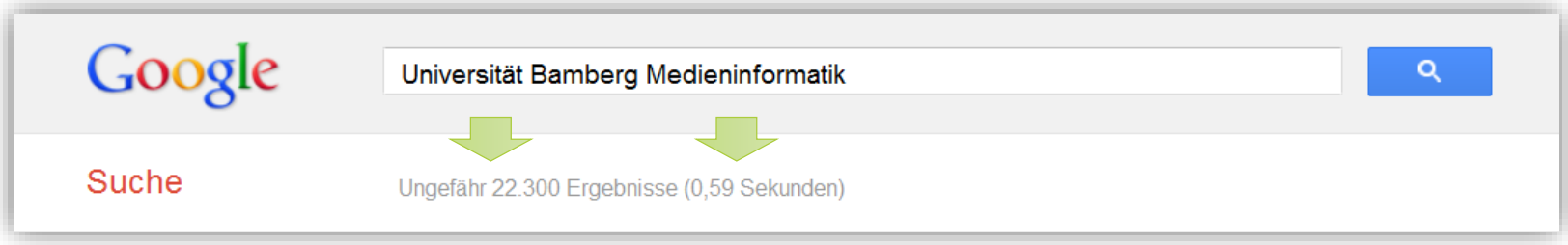
- Die gebräuchlichste Datenstruktur ist der invertierte Index
  - Auch „invertierte Liste“ oder „inverted file“ genannt
  - Allgemeiner Name für eine Klasse von Strukturen
  - „invertiert“, da hier
    - nicht wie normal Dokumenten die in ihnen vorkommenden Wörter
    - sondern Wörtern die Dokumente, in denen sie vorkommen, zugeordnet werden,
  - ähnlich wie bei einer Konkordanz

D_1 → „Haus“, „Wald“, „Italien“ …

„Haus“ → D_1, D_7, D_9 ...

Indexe und Ranking

- Indexe sind auf Unterstützung der Suche ausgelegt
  - Schnellere Antwortzeiten & unterstützen von Updates

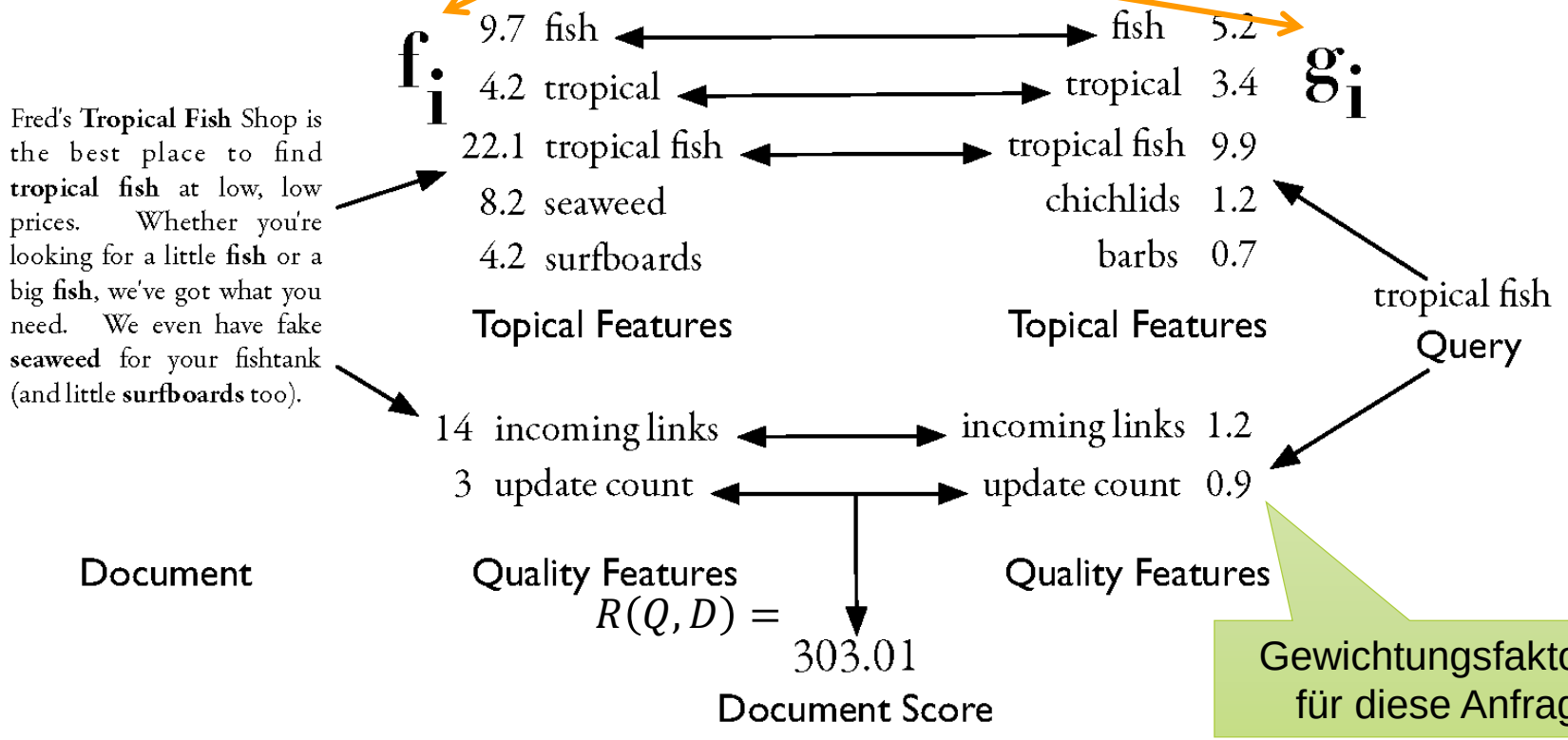


- Textsuchmaschinen setzen spezielle Form der Suche ein: Ranking
  - Dokumente in sortierter Folge auf Basis einer Kennzahl (Score R), die aus Dokument und Anfrage berechnet wird $R: \text{Anfrage} \times \text{Dokument} \rightarrow \mathbb{R}$
  - Bestimmende Faktoren, wie in Kapitel 4 betrachtet:
    - Repräsentation der Dokumente
    - Repräsentation der Anfrage
    - der Rankingalgorithmus (das Retrievalmodell)

II. Ein Modell für das Ranking (als Beispiel)

$$R(Q,D) = \sum_i g_i(Q) \cdot f_i(D)$$

f_i is a document feature function
g_i is a query feature function



III. Invertierte Indexstrukturen

- Jeder Indexterm ist mit einer invertierten Liste verknüpft
  - Enthält Listeneinträge
    - für Dokumente oder
    - Wortvorkommen in Dokumenten und
    - andere Information
  - Jeder Listeneintrag wird Posting genannt
  - Der Teil eines Postings, der auf ein bestimmtes Dokument oder einen (Speicher-)Ort verweist, wird Pointer genannt
  - Jedes Dokument in der Kollektion bekommt eine eindeutige Dokumentnummer (ID)
  - Die Listen werden häufig nach Dokumentnummer sortiert

Beispielkollektion

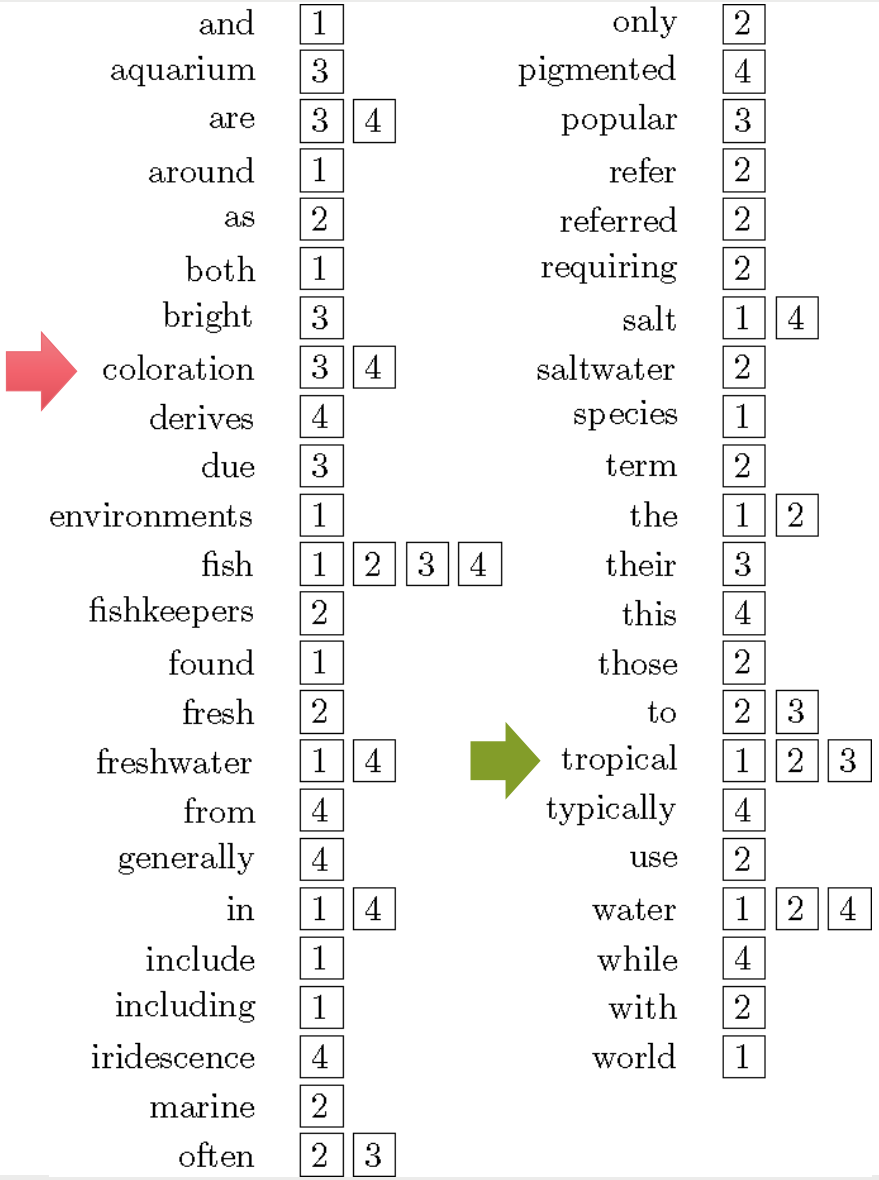
- Vier Sätze aus dem Wikipedia-Eintrag für „tropical fish“

S_1	Tropical fish include fish found in tropical environments around the world, including both freshwater and salt water species.
S_2	Fishkeepers often use the term tropical fish to refer only those requiring fresh water, with saltwater tropical fish referred to as marine fish.
S_3	Tropical fish are popular aquarium fish, due to their often bright coloration.
S_4	In freshwater fish, this coloration typically derives from iridescence, while salt water fish are generally pigmented.

Beispiel: einfach

Einfacher invertierter Index: 46 Listen

S_1	Tropical fish include fish found in tropical environments around the world, including both freshwater and salt water species.
S_2	Fishkeepers often use the term tropical fish to refer only those requiring fresh water, with saltwater tropical fish referred to as marine fish.
S_3	Tropical fish are popular aquarium fish, due to their often bright coloration.
S_4	In freshwater fish, this coloration typically derives from iridescence, while salt water fish are generally pigmented.



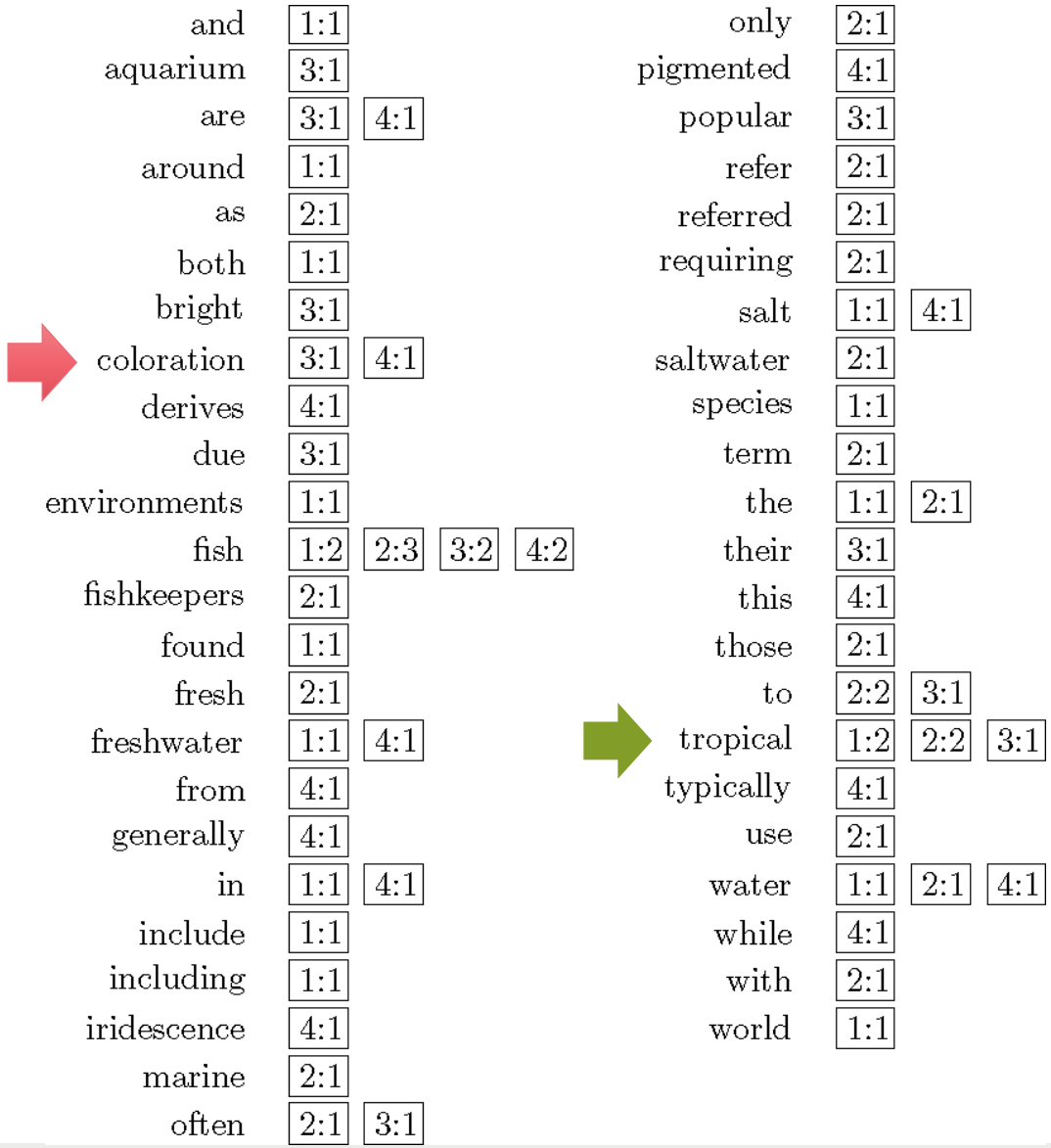
Beispiel: mit Häufigkeiten

Invertierter Index mit Häufigkeiten

DocID:Häufigkeit

(ermöglicht „bessere“ Ranking-Algorithmen)

S_1	Tropical fish include fish found in tropical environments around the world, including both freshwater and salt water species.
S_2	Fishkeepers often use the term tropical fish to refer only those requiring fresh water, with saltwater tropical fish referred to as marine fish.
S_3	Tropical fish are popular aquarium fish, due to their often bright coloration.
S_4	In freshwater fish, this coloration typically derives from iridescence, while salt water fish are generally pigmented.



Beispiel: mit Positionen

Invertierter Index mit Positionen

DocID:Position

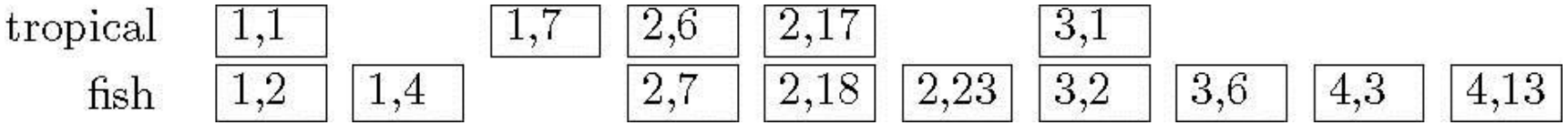
(ermöglicht Nachbarschaftssuche, Phrasensuche)

S_1	Tropical fish include fish found in tropical environments around the world, including both freshwater and salt water species.
S_2	Fishkeepers often use the term tropical fish to refer only those requiring fresh water, with saltwater tropical fish referred to as marine fish.
S_3	Tropical fish are popular aquarium fish, due to their often bright coloration.
S_4	In freshwater fish, this coloration typically derives from iridescence, while salt water fish are generally pigmented.

and	1,15					
aquarium	3,5					
are	3,3	4,14				
around	1,9					
as	2,21					
both	1,13					
bright	3,11					
coloration	3,12	4,5				
derives	4,7					
due	3,7					
environments	1,8					
fish	1,2	1,4	2,7	2,18	2,23	
			3,2	3,6	4,3	
			4,13			
fishkeepers	2,1					
found	1,5					
fresh	2,13					
freshwater	1,14	4,2				
from	4,8					
generally	4,15					
in	1,6	4,1				
include	1,3					
including	1,12					
iridescence	4,9					
marine	2,22					
often	2,2	3,10				
only	2,10					
pigmented	4,16					
popular	3,4					
refer	2,9					
referred	2,19					
requiring	2,12					
salt	1,16	4,11				
saltwater	2,16					
species	1,18					
term	2,5					
the	1,10	2,4				
their	3,9					
this	4,4					
those	2,11					
to	2,8	2,20	3,8			
tropical	1,1	1,7	2,6	2,17	3,1	
typically	4,6					
use	2,3					
water	1,17	2,14	4,12			
while	4,10					
with	2,15					
world	1,11					

Nachbarschaftssuche (proximity match)

- Phrasen oder Wörter in einem Fenster vergleichen
- Wortpositionen in invertierten Listen machen diese Art von Suchfeatures effizient
- Beispiel: paralleles Durchlaufen der Listen
  - „tropical fish“
  - „find tropical within 5 words of fish”



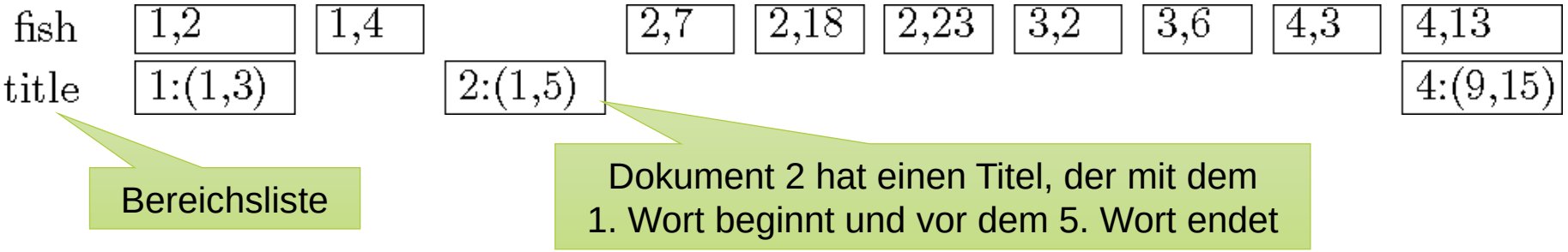
Felder und Extents

- Dokumentstruktur ist nützlich bei der Suche
  - Feldeinschränkungen
    - z. B. bei Mails: Datum, von:, etc.
  - Einige Felder sind wichtiger
    - z. B. Titel

- Alternative Umsetzungsmöglichkeiten:
  1. Getrennte invertierte Listen für jeden Feldtyp führen
     - eine Liste für „*Haus* im Titel“ eine für „*Haus* in normalem Text“
  2. Informationen über die Felder zu Postings hinzufügen
     - „Haus“ in D_{13} an Position 27, bold ist true, title ist false
  3. Verwendung von Bereichslisten (*extent lists*)

Bereichslisten

- Ein Bereich (extent) ist ein zusammenhängender Bereich eines Dokuments
  - Bereiche werden mit Wortpositionen repräsentiert
  - Invertierte Liste enthält alle Bereiche für einen gegebenen Feldtyp
  - z. B.



  - Anfrage „fish“ im Titel nun recht einfach zu bearbeiten
  - Wenn ein Element mehrfach im Dokument vorkommt, gibt es hierfür mehrere Einträge in der Bereichsliste

Weitere Aspekte

- Vorberechnete Feature-Werte in invertierter Liste
  - z. B. Liste für „fish“ $[(1: 3{,}6), (3: 2{,}2)]$, wobei $3{,}6$ der Gesamtwert einer Eigenschaft für Dokument 1 ist
  - Erlaubt die Verlagerung aufwändiger Berechnungen von der Query-time zur Indexing-time
  - Erhöht die Geschwindigkeit, reduziert jedoch die Flexibilität zur Query-time (Keine Wortpositionen mehr $\Rightarrow$ Was ist mit Anfragen nach Phrasen?)
- Weiterer Schritt: Nach Feature-Werten sortierte Listen
  - Retrieval-Engine kann sich auf den oberen Teil jeder invertierten Liste konzentrieren, wo die best-bewerteten Dokumente stehen

  $\Rightarrow$ Sehr effizient z. B. für Ein-Wort-Abfragen

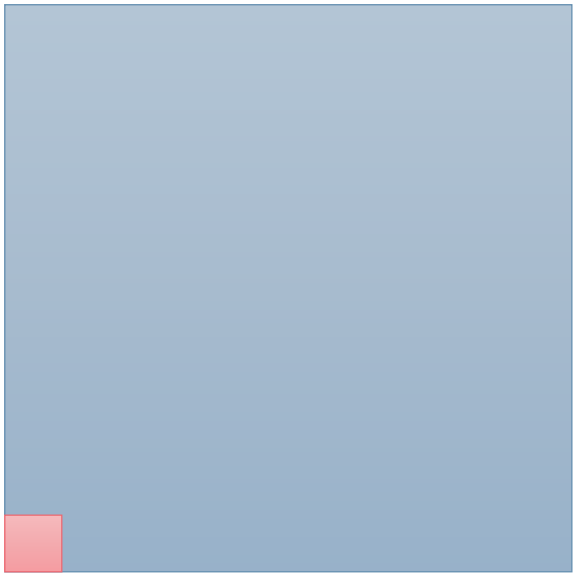
Warum funktionieren typische Web-Anfragen auf invertierten Listen so gut?

- Problem eigentlich:
  - Termvektoren bilden einen Raum sehr hoher Dimensionalität
  - Anfragen (Bereichsanfragen und k-Nearest-Neighbor-Anfragen) sind nur für niedrige Dimensionen effizient!
  - Bekannt als „Fluch der hohen Dimensionalität“
- Bsp.: 1.000 gleichverteilte Punkte; wir suchen die 10 „benachbarten“
  - Annahme: Anfragepunkt liegt im Nullpunkt (Koordinatenursprung)
  - In 1D müssen wir im Durchschnitt $\frac{10}{1000} = 0{,}01$ des Datenraumes betrachten

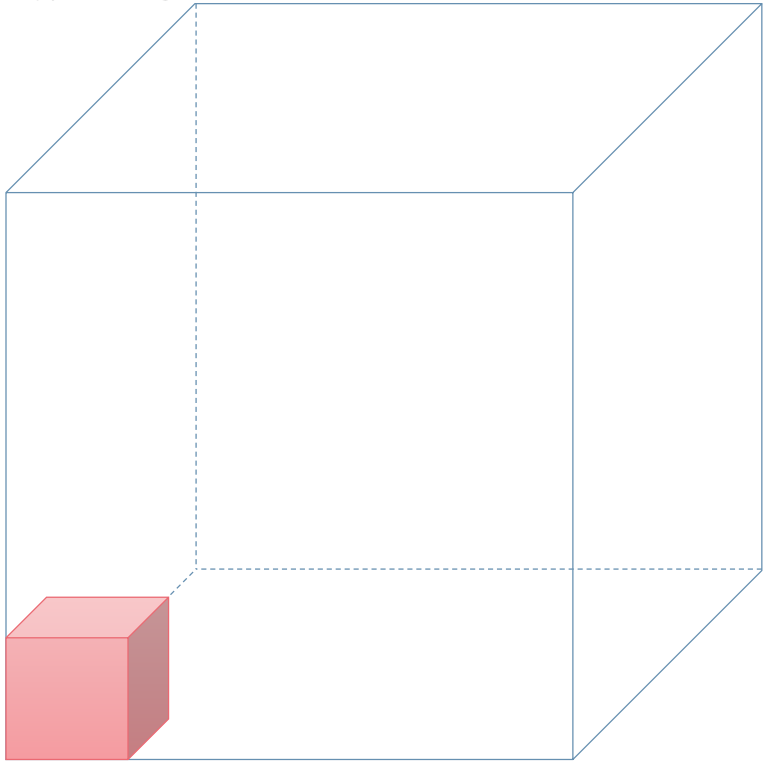
1% des Datenraumes

$d = 1$

$d = 2$



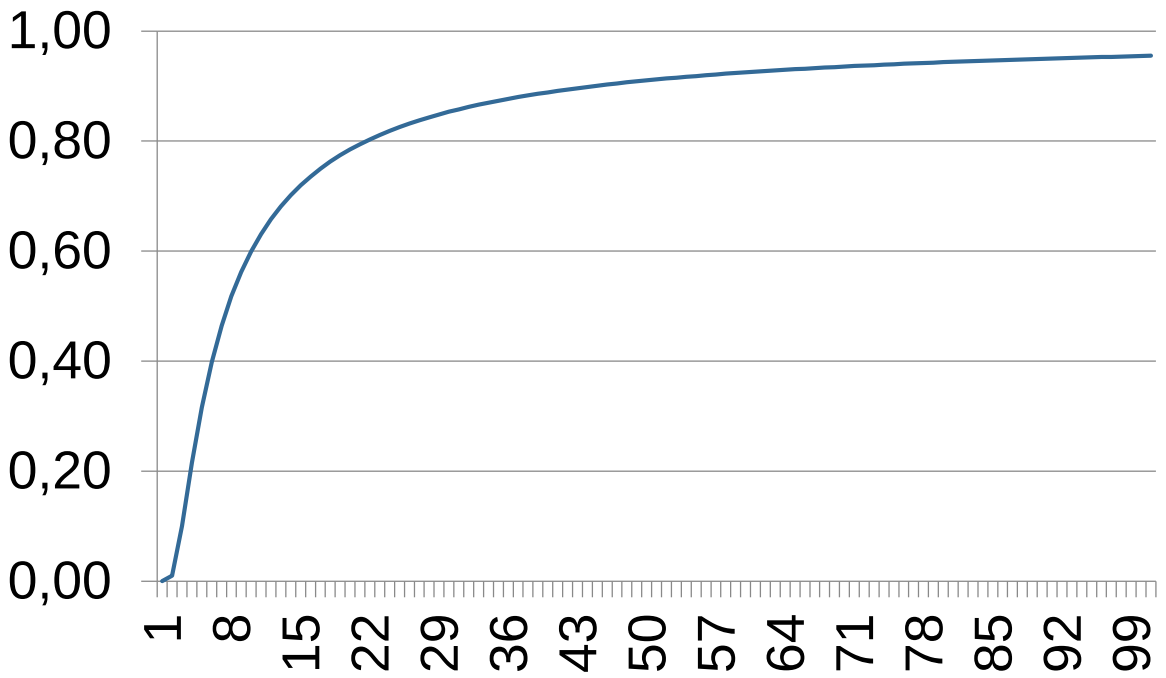
$d = 3$



Keine Selektivität bei hohem d

- Erforderliche „Kantenlänge“ in Abhängigkeit von d
- $l = 0{,}01^{1/d} = \sqrt[d]{0{,}01}$

d	l
1	0,01
2	0,10
3	0,22
4	0,32
5	0,40
6	0,46
7	0,52
8	0,56
9	0,60
10	0,63
11	0,66
12	0,68
13	0,70
14	0,72
15	0,74
16	0,75
17	0,76
18	0,77
19	0,78
20	0,79



Warum funktionieren dann typische Web-Anfragen auf invertierten Listen so gut?

- Invertierte Listen
  - Arbeiten bei der Anfragebearbeitung nicht im kompletten hochdimensionalen Raum!
  - Nutzen den Anfrageraum niedriger Dimensionalität!
  - Niedrige Dimensionalität weil alle Begriffe, die nicht in der Anfrage sind keine Rolle spielen! ($g_i(Q) = 0$ für diese Terme)

- Aber:
  - Was ist, wenn ich zu einem Dokument ähnliche Dokumente suche?

IV. Kompression

- Invertierte Listen sind sehr groß
  - z. B. 25-50% der Kollektion für TREC-Kollektionen, die die Indri (http://www.lemurproject.org/indri/) Suchmaschine verwenden
  - Anteil entsprechend höher, wenn n-Gramme indexiert werden

  $\Rightarrow$ Gigantische Datenmengen für die Listen bei Web-Suche

- Kompression von Indexen spart Festplatten- und/oder Arbeitsspeicher (erlaubt ggf. Daten im Arbeitsspeicher statt auf der Festplatte zu halten)
  - Normalerweise müssen Listen für die Benutzung dekomprimiert werden
  - Die besten Kompressionstechniken haben
    - gute Kompressionsraten und
    - sind einfach zu dekomprimieren

  Optimum abhängig von Speicherzugriffszeit und CPU-Leistung

- Verlustfreie Kompression – keine Informationen gehen verloren

Kompression

- Grundidee:
  - „Gebräuchliche“ Daten bekommen kurze Codewörter,
  - weniger „gebräuchliche“ Daten bekommen lange Codewörter

- Klassisch:
  - Huffman-Codierung
  - Arithmetische Codierung
  - Lauflängencodierung
  - …

} Einf. In die Medieninformatik

Wichtig für IR: Deltacodierung

- „Klassische Kompression“ arbeitet gut, wenn einige Werte häufig sind und andere selten!

⇒ Worthäufigkeitsdaten sind ein guter Kandidat für die Kompression:

  - viele kleine und wenig große Zahlen

- Dokumentnummern in einer invertierten Liste sind eher gleichverteilt und damit weniger „vorhersagbar“
  - Aber: Die Abstände zwischen den Dokumentnummern in einer geordneten Liste sind kleiner und leichter vorhersehbar

⇒ Deltacodierung

  - Kodiert die Abstände zwischen den Dokumentnummern (*d-gaps*)

Deltacodierung

- Invertierte Liste (Dokumentnummern ohne Häufigkeiten oder Positionen)

  `1, 5, 9, 18, 23, 24, 30, 44, 45, 48`

- Abstände zwischen benachbarten Nummern

  `1, 4, 4, 9, 5, 1, 6, 14, 1, 3`

- Abstände zwischen den Dokumentnummern für häufig vorkommende Wörter sind gut zu komprimieren, z. B.

  `1, 1, 2, 1, 5, 1, 4, 1, 1, 3, …`

- Abstände für selten auftretende Wörter sind groß, z. B.

  `109, 3766, 453, 1867, 992, …`

⇒ Lange Listen recht gut komprimierbar, kurze schlechter → OK

Bitbezogene Codierungen

- Beispiele: (hier nicht näher betrachtet)
  - Elias-γ Code
  - Elias-δ Code
  - Ziel: kompakte Codierung für kleine Zahlen
- Folge:
  - Übergänge / Wechsel zwischen codierten Zahlen können nach jeder Bitposition auftreten
  - Sehr kurze Codes
- Aber:
  - Codierung / Decodierung eher aufwändig
  - Problem für Prozessoren, die Bytes verarbeiten

Bytebezogene Kodierungen

- v-Byte ist eine populäre Bytecodierung
  - Ähnlich Unicode UTF-8
  - Kürzestes v-Byte Codewort ist 1 Byte
  - Zahlen haben 1 bis 4 Bytes mit dem höchstwertigen Bit

    $= 1$ im letzten Byte,

    $= 0$ sonst

v-Byte Kodierung

k	Anzahl der Bytes
$0 \leq k < 2^7$	1
$2^7 \leq k < 2^{14}$	2
$2^{14} \leq k < 2^{21}$	3
$2^{21} \leq k < 2^{28}$	4

k	Codierung binär	Hexadezimal
1	10000001	81
6	10000110	86
127	11111111	FF
128	00000001 10000000	01 80
130	00000001 10000010	01 82
20.000	00000001 00011100 10100000	01 1C A0

Kompressionsbeispiel

- Folgende invertierte Liste mit Positionen wird angenommen:
  - Aufbau der Einträge: (Dok.-ID, count, [positions])

    (1, 2, [1, 7]) (2, 3, [6, 17, 197]) (3, 1, [1])
  - Durch „count“ auch ohne Klammern decodierbar

    1, 2, 1, 7, 2, 3, 6, 17, 197, 3, 1, 1
- Deltacodierung der Dokumentnummern und Wortpositionen:

  (1, 2, [1, 6]) (1, 3, [6, 11, 180]) (1, 1, [1])

  bzw.: 1, 2, 1, 6, 1, 3, 6, 11, 180, 1, 1, 1
- Kompression mit v-Byte:

  81 82 81 86 81 82 86 8B 01 B4 81 81 81

„Hüpfen“ (Skipping)

- Suche schließt den Vergleich invertierter Listen verschiedener Längen ein: paralleles Durchlaufen
  - Kann sehr ineffizient sein (insbesondere bei der üblichen UND-Verknüpfung)
  - Beispiel: „Anabantoidei Futter“ (Häufigkeit 53.000 vs. 5.170.000)
    - viele „unnütze“ Einträge in der Liste für Futter

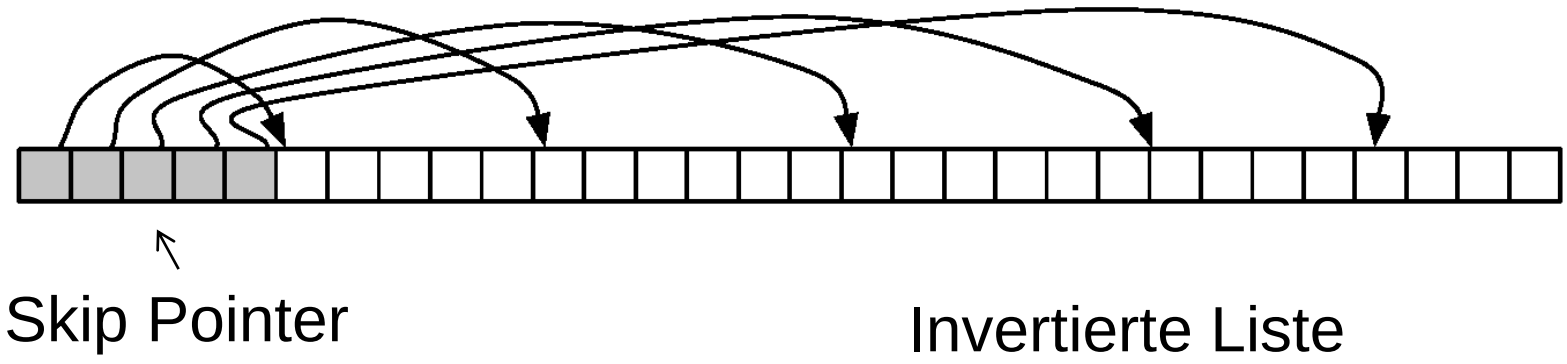
  ⇒ Statt alle Einträge beider Listen zu betrachten:
    - Durchlaufen der kürzeren Liste und
    - „Weiterhüpfen“ (Skipping) in der längeren Liste um Dokumentnummern zu überprüfen
  - Aber: Kompression erschwert dies
    - Variable Größe der Codierungen und nur d-gaps werden gespeichert

  ⇒ Skip Pointer als zusätzliche Datenstrukturen, die das Skipping ermöglichen

Bildquelle: http://en.wikipedia.org/wiki/Anabantoidei

Skip Pointer

- Ein Skip Pointer (d, p) enthält eine Dokumentnummer d und eine Byte- oder Bitposition p
  - Bedeutung: Es gibt ein Posting in der Liste, das
    - an (Byte-)Position p beginnt, und
    - das Posting davor war das von Dokument d
      (damit man die nachfolgenden d-Gaps interpretieren kann)



Skip Pointer

- Beispiel (zur Vereinfachung nur Dokumentnummern)
  - Invertierte Liste

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	...
5	11	17	21	26	34	36	37	45	48	51	52	57	80	89	91	94	101	104	119	...

  - d-Gaps

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	...
5	6	6	4	5	8	2	1	8	3	3	1	5	23	9	2	3	7	3	15	...

  - Skip Pointer (Zählung der Einträge in der Liste beginnt mit 0)

    (17, 3), (34, 6), (45, 9), (52,12), (89, 15), (101, 18)

- Suche Eintrag für 37!

In der Praxis ist die optimale Skip-Distanz oft ungefähr 100.

Ggf. mehrere Listen mit Skip-Pointern für unterschiedliche Distanzen nutzen.

V. Hilfsstrukturen

- Invertierte Listen normalerweise aus Effizienzgründen zusammen in einer einzigen (einigen wenigen) Datei(en) gespeichert
  - „invertierte Datei“
- Vokabular oder Lexikon
  - Enthält Nachschlagetabelle Indexterm → (Datei ×) Offsetwert, um die entsprechende Liste in der invertierten Datei zu finden
  - Typisch: eine Hash-Tabelle im Arbeitsspeicher;
    Alternative für umfangreichere Vokabulare: ein B-Baum
- Termstatistiken (z. B. in wie vielen Dokumenten kommt der Term vor) werden am Kopf der invertierten Listen gespeichert
- Kollektionstatistiken (z. B. wie viele Dokumente, Wörter, Terme … umfasst die Kollektion) werden in separaten Dateien gespeichert

VI. Indexerstellung

- Einfacher Indexierer im Hauptspeicher

```
procedure BUILDINDEX(D)
    I ← HashTable()                          ▶ D is a set of text documents
    n ← 0                                    ▶ Inverted list storage (Term → Array)
    for all documents d ∈ D do               ▶ Document numbering
        n ← n + 1
        T ← Parse(d)                         ▶ Parse document d into tokens
        Remove duplicates from T
        for all tokens t ∈ T do
            if I_t ∉ I then                  ▶ token t not yet in I
                I_t ← Array()                ▶ insert new array (resp. list) for t in I
            end if
            I_t.append(n)                    ▶ append at end ⇒ sorted by doc. ids
        end for
    end for
    return I
end procedure
```

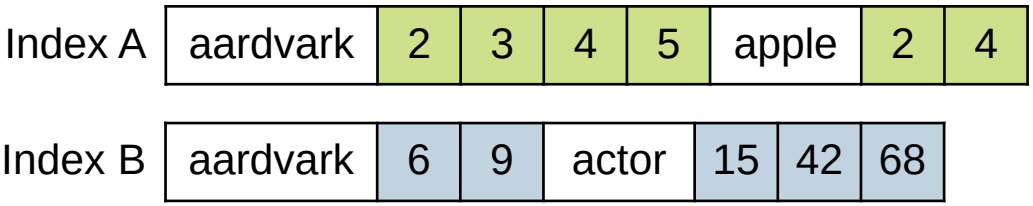
(Index-)Merging

Würde man alle Operationen immer direkt auf Dateien vom Hintergrundspeicher ausführen, wäre die Indexerstellung viel zu langsam

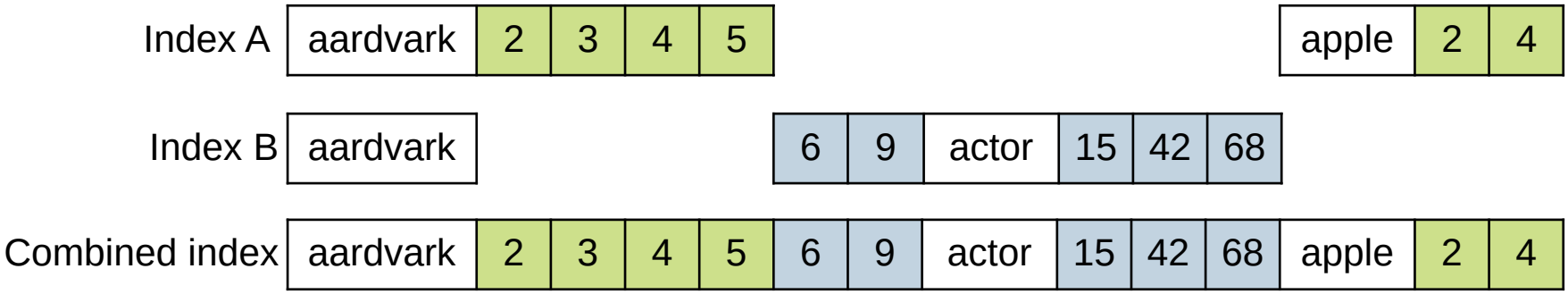
- Merging versucht dem Problem des limitierten (Haupt-)Speicherplatzes entgegen zu wirken
  - Die invertierte Liste wird erstellt, bis der Arbeitsspeicher voll ist
  - Dann wird dieser Teil des Index auf die Festplatte geschrieben und es wird ein neuer Index erzeugt
  - Am Ende dieses Prozesses ist die Festplatte mit vielen partiellen Indexen gefüllt, die dann wieder zusammengefügt („gemerged“) werden
- Partielle Listen müssen so entworfen sein, dass sie aus kleinen Teilen zusammengefügt werden können
  - z. B. durch Ablage in alphabetischer Folge

(Index-)Merging

- Erstellen der einzelnen Indexe



- Anschließend Merging:



Folge: Merging effizient in parallelem Durchlauf umsetzbar; $O(n)$

Verteilte Indexierung

- Verteilte Verarbeitung wird notwendig, wenn sehr große Datenmengen indexiert und analysiert werden müssen (wie bei Websuchmaschinen)
- Meist werden bevorzugt viele günstige Server (commodity PCs) verwendet, anstatt einige große teure Maschinen zu kaufen

- MapReduce ist ein Framework für verteilte Berechnungen, das auf Indexierungs- und Analyseaufgaben ausgelegt ist

Bildquelle: https://www.google.com/intl/de_ALL/about/datacenters/gallery/index.html#/ Abruf: 29.11.2018

Beispielszenario für Map Reduce

- Gegeben ist eine große Textdatei, die Daten über Kreditkartentransaktionen enthält
  - Jede Zeile der Datei enthält eine Kreditkartennummer und einen Geldbetrag (Umsatz)
  - Die (Gesamt-)Umsätze sollen für alle Kreditkartennummern ermittelt werden
- Eine Hashtabelle zu den Kartennummern könnte verwendet werden ⇒ Speicherprobleme
- Andererseits:
  - Häufigkeiten oder Summen sind in einer sortierten Datei einfach zu bestimmen
- Ähnlich bei einem verteilten Ansatz:
  - Sortierung und Anordnung sind entscheidend

MapReduce: Komponenten

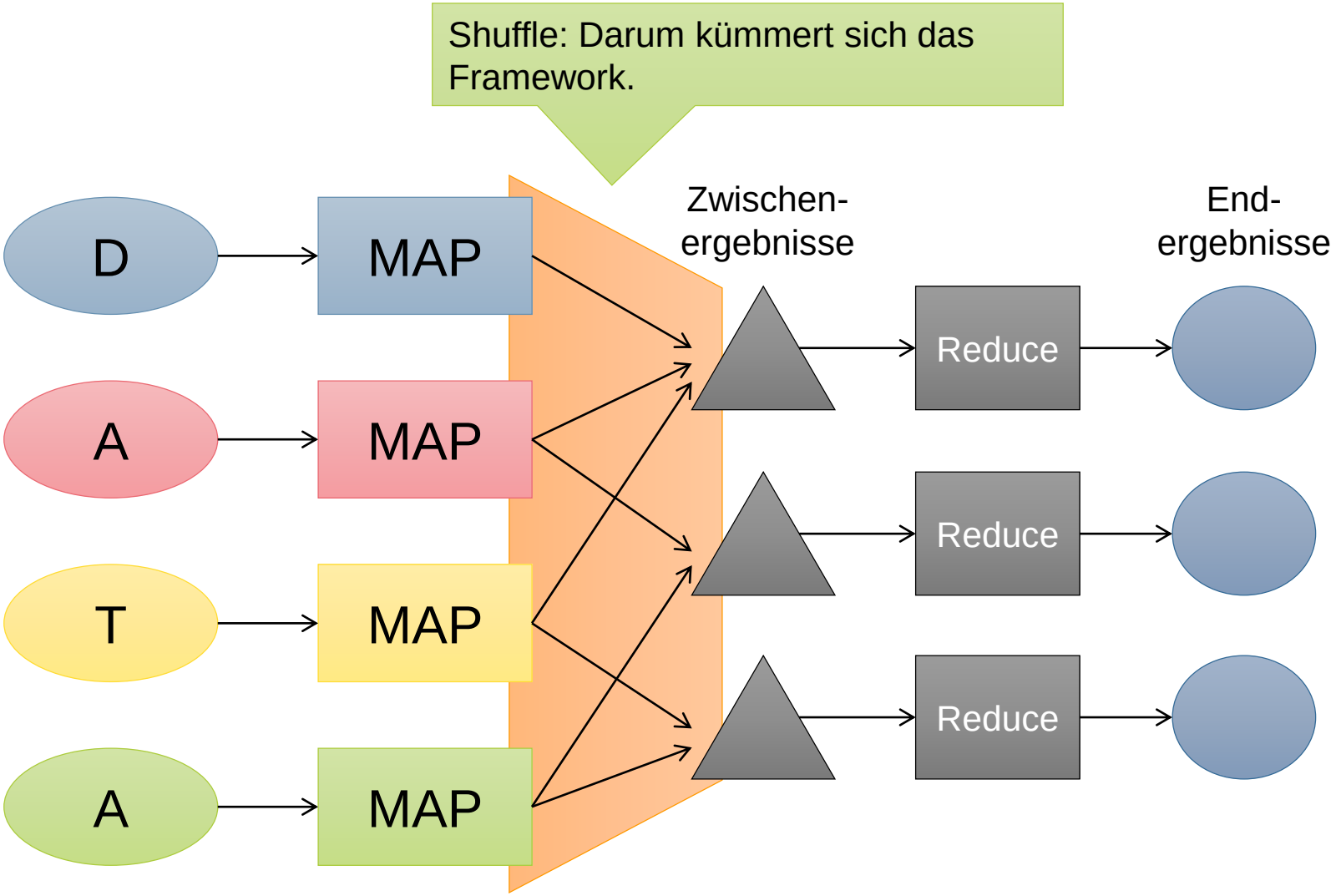
- Framework für die verteilte, nebenläufige Berechnung, das auf Datenanordnung und Verteilung ausgerichtet ist
- *Mapper*
  - Transformiert eine Liste von Items in eine andere Liste von Items mit der selben Mächtigkeit aber klarer (Schlüssel, Wert)-Struktur
- *Reducer*
  - Transformiert eine Liste von Items in ein einzelnes Item
  - (Definitionen sind nicht so streng, was die Anzahl an Outputs angeht)
- Auf einem Rechnercluster laufen viele Mapper- und Reducer-Tasks

http://labs.google.com/papers/mapreduce.html

MapReduce: Ablauf

- Grundprozess
  - Map-Phase, welche die Daten in Paare transformiert, jedes Paar besteht aus einem Schlüssel und einem Wert
  - Shuffle setzt eine Hash-Funktion ein, so dass alle Paare mit dem selben Schlüssel „nebeneinander“ und auf dem gleichen Rechner enden
  - Reduce-Phase verarbeitet die Daten satzweise, wobei alle Paare mit dem gleichen Schlüssel „gleichzeitig“ verarbeitet werden
- Idempotenz von Mapper und Reducer sorgt für Fehlertoleranz
  - ⇒ mehrfache Ausführung der Operationen auf den gleichen Eingabewerten führen trotzdem zu gleichen Ergebnissen

MapReduce



angelehnt an http://de.wikipedia.org/wiki/MapReduce

Beispiel (Kreditkartentransaktionen)

Verarbeite die Eingaben und bringe sie satzweise in die gewünschte Struktur.

```
procedure MapCreditCards(input)
    while not input.done() do
        record ← input.next()
        cardno ← record.cardno
        amount ← record.amount
        Emit (cardno, amount)
    end while
end procedure
```

Liste von Kartentransaktionen

Verrechne alle Values, die zu einem Key gehören.

```
procedure ReduceCreditCards(key, values)
    total ← 0
    cardno ← key
    while not values.done() do
        amount ← values.next()
        total ← total + amount
    end while
    Emit (cardno, total)
end procedure
```

Indexierungsbeispiel

Dokumente

Verarbeite Dokumente und erzeuge für jedes vorkommende Wort ein Posting, das per „Shuffle“ zu dem Rechner / der Task kommt, die für das Wort zuständig ist.

```
procedure MapDocumentsToPostings(input)
    while not input.done() do
        document ← input.next()
        id ← document.id
        position ← 0
        tokens ← Parse(document.text)
        for each word w in tokens do
            Emit(w, id:position)
            position ← position +1
        end for
    end while
end procedure
```

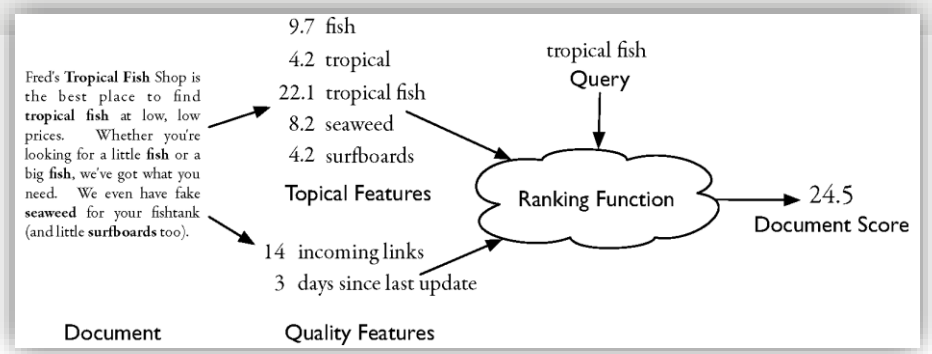
Hier wird die Liste für das Wort erzeugt.

```
procedure ReducePostingsToLists(key, values)
    word ← key
    WriteWord(word)                ▶ Create the list
    while not values.done() do
        EncodePosting(values.next())
    end while
end procedure
```

Notwendigkeit des Ergebnismerging

- Indexmerging ist eine gute Strategie, um mit Updates umzugehen, wenn diese in größeren Bündeln anfallen
- Für kleine Updates ist es sehr ineffizient
  - Statt dessen werden separate Indexe für neue Dokumente erzeugt und die Ergebnisse der Suchvorgänge auf dem Hauptindex und dem Deltaindex verschmolzen (in der Praxis existieren sogar noch mehr Ebenen)
  - dies kann im Arbeitsspeicher durchgeführt werden → schnell für Updates und Suche
- Gelöschte Dokumente können mit einer delete list berücksichtigt werden
  - Bei Modifikationen wird dann die alte Version des Dokuments auf die delete list gesetzt und die neue Version wird als neues Dokument zum Index hinzugefügt

VII. Anfrageverarbeitung

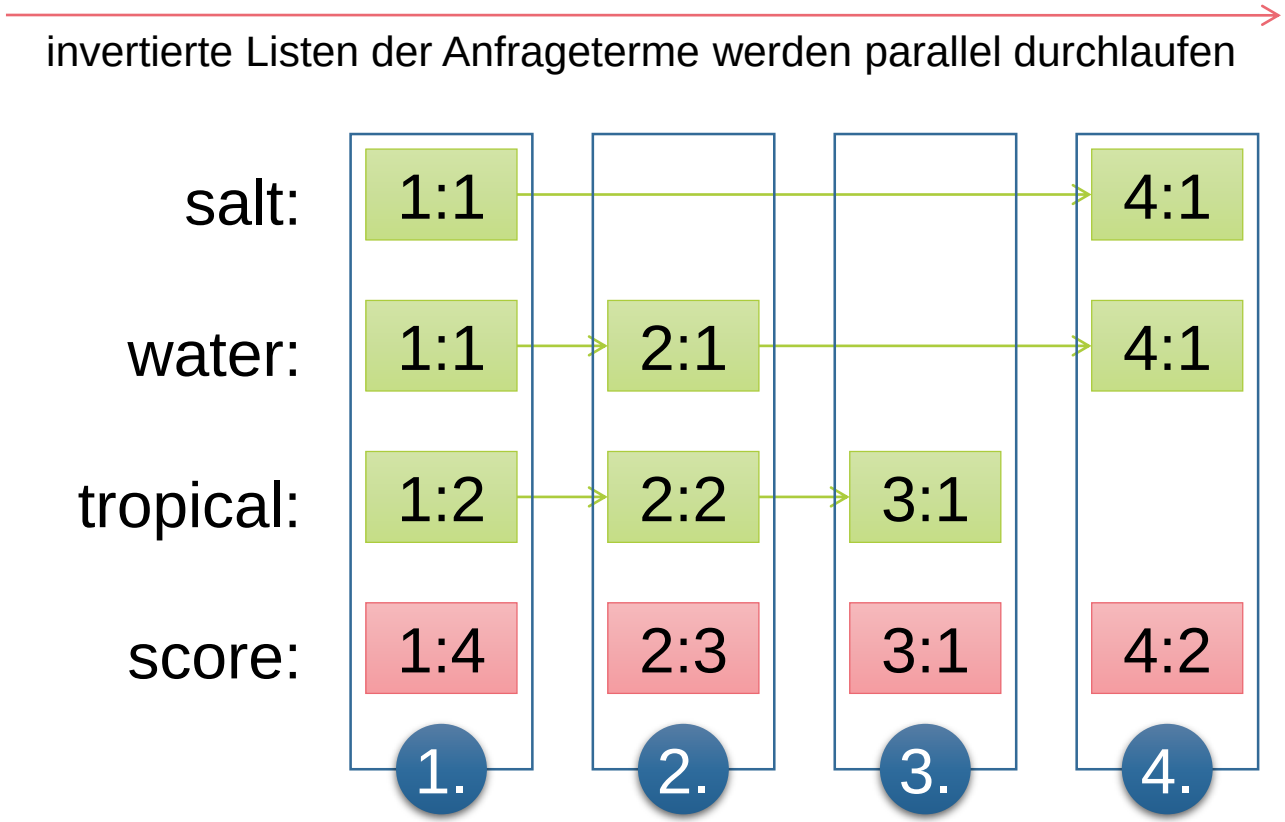


- Problem:
  - Kein boolesches Retrievel, sondern Berechnung eines Rankings
- Document-at-a-time
  - Berechnet vollständige Bewertungen (Scores) für Dokumente durch parallele Verarbeitung aller Termlisten, ein Dokument nach dem anderen
- Term-at-a-time
  - Akkumuliert (addiert) die Bewertungen für Dokumente, indem eine Termliste nach der anderen verarbeitet wird
- Für beide Ansätze existieren Optimierungstechniken

Document-At-A-Time

- Anfrage: „salt water tropical“

Hier:
einfaches IR-Modell:
Score = Summe der Vorkommens-häufigkeiten der Anfrageterme



Document-At-A-Time

Q: Anfrage
I: inv. Index
f: berechnet die Termgewichte für Dokumente
g: berechnet die Termgewichte für die Anfrage
k: Anzahl der gewünschten Ergebnisse

```
procedure DocumentAtATimeRetrieval(Q, I, f, g, k)
    L ← Array()
    R ← PriorityQueue(k)
    for all terms w_i ∈ Q do
        l_i ← InvertedList(w_i, I)
        L.add(l_i)
    end for
    for all documents d ∈ I do
        s_d = 0
        for all inverted lists l_i ∈ L do
            if l_i.getCurrentDocument() = d then
                s_d ← s_d + g_i(Q) · f_i(d)
            end if
            l_i.movePastDocument(d)
        end for
        R.add(s_d, d)
    end for
    return the top k results from R
end procedure
```

Alle Listen für Q aufsammeln.

Array für die invertierten Listen zu den Anfragetermen

Enthält immer die k bisher besten Ergebnisse

Für den Score.

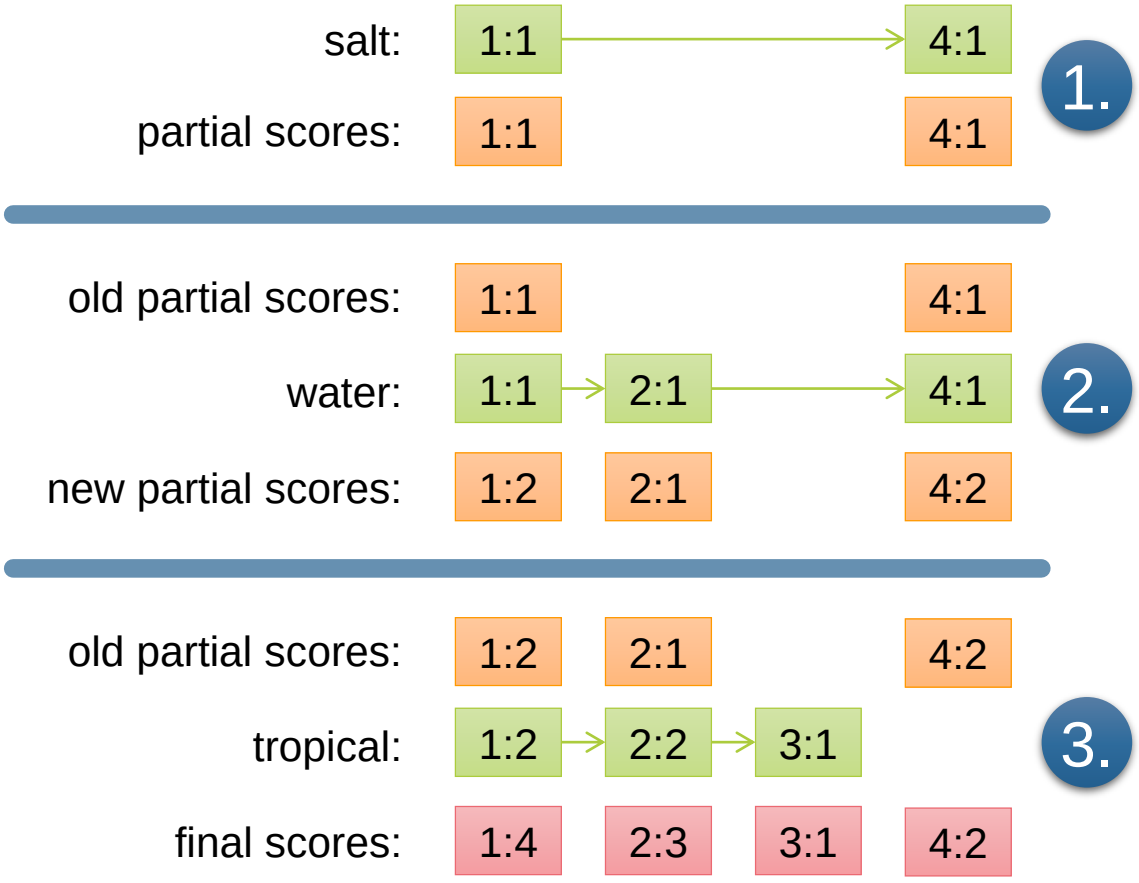
Alle Listen parallel durchlaufen.

Aktualisiere den Score des Dokuments d

Hilfsdatenstruktur R übernimmt die Sortierung.

Term-At-A-Time

Die Listen werden nacheinander betrachtet und die Scores zunächst nur partiell berechnet und dann schrittweise ergänzt.



Term-At-A-Time

procedure TERMATATIMERETRIEVAL (Q, I, f, g, k)
- $A \leftarrow$ HashTable()
- $L \leftarrow$ Array()
- $R \leftarrow$ PriorityQueue(k)
- **for all** terms $w_i \in Q$ **do**
  - $l_i \leftarrow$ InvertedList(w_i, I)
  - L.add(l_i)
- **end for**
- **for all** inverted lists $l_i \in L$ **do**
  - **while** l_i is not finished **do**
    - $d \leftarrow l_i$.getCurrentDocument()
    - $A_d \leftarrow A_d + g_i(Q) \cdot f_i(d)$
    - l_i.moveToNextDocument()
  - **end while**
- **end for**
- **for all** accumulators $A_d \in A$ **do**
  - $s_d \leftarrow A_d$
  - R.add(s_d, d)
- **end for**
- **return** the top k results from R

end procedure

Hier werden die partiellen Scores für die Dokumente verwaltet.

Annahme: Alle Werte für die Dokumente sind zunächst mit 0 initialisiert

Alle Listen nacheinander abarbeiten.

Die in A angesammelten Scores mit Hilfe von R sortieren.

Optimierungstechniken

- Term-At-A-Time
  - verbraucht mehr Speicher für die Akkumulation der Bewertungen,
  - greift aber effizienter auf die Festplatte zu

Unterschiede bei Kostenstruktur zwischen Hauptspeicher- und Arbeitsspeicher-basierten Ansätzen!

- Zwei Optimierungsklassen:
  1. Weniger Daten von den invertierten Listen lesen
     - bereits unter 5.4 „Kompression“ betrachtet
     - gut insbesondere für einfache Feature-Funktionen
  2. Berechnen der Bewertungen für weniger Dokumente
     - z. B. konjunktive Verarbeitung (conjunctive processing)
     - besser für komplexe Feature-Funktionen

Conjunctive Term-at-a-Time

- Jedes Dokument muss alle Anfrageterme enthalten, um zum Ergebnis der Anfrage zu gehören
- Arbeitet besonders gut, wenn ein Anfrageterm recht selten ist

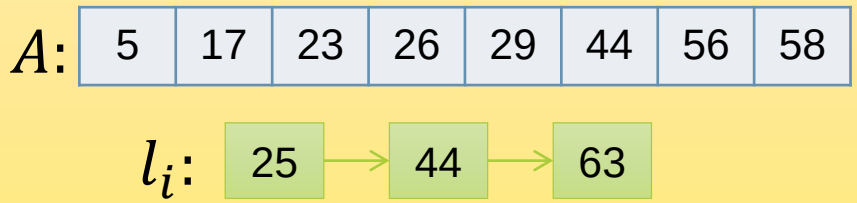
```
procedure TermAtATimeRetrieval (Q, I, f, g, k)
    A ← Map();  L ← Array();  R ← PriorityQueue(k)
    for all terms w_i ∈ Q do
        l_i ← InvertedList(w_i, I);  L.add(l_i)
    end for
    for all inverted lists l_i ∈ L do
        d_0 ← −1
        while l_i is not finished do
            if i = 0 then
                d ← l_i.getCurrentDocument()
                A_d ← g_i(Q) · f_i(d)
                l_i.moveToNextDocument()
            else
                d ← l_i.getCurrentDocument()
                d' ← A.getNextAccumulatorAtLeast(d)
                A.removeAccumulatorsBetween(d_0, d')
                if d = d' then
                    A_d ← A_d + g_i(Q) · f_i(d)
                    l_i.moveToNextDocument()
                else
                    l_i.skipForwardToDocument(d')
                end if
                d_0 ← d
            end if
        end while
    end for
    for all accumulators A_d ∈ A do
        s_d ← A_d;  R.add(s_d, d)
    end for
    return the top k results from R
end procedure
```

Die 1. betrachtete Liste; initialer Aufbau von A

Rücke bis zum kleinsten $d' \geq d$ vor

d war nicht in A also: Vorwärts springen

Beispiel (nur Dok-Ids):



Conjunctive Document-at-a-Time

- Schnelles, paralleles „Skippen“ durch die Listen

Bestimme die größte Dok.-ID unter allen nächsten Dok.-Ids in den Listen.

Versuche es mit Dokument d

Einer der Terme ist nicht in d; also: Versuch fehlgeschlagen

- hier nur das Prinzip, weitere Optimierungen sind möglich

```
procedure DocumentAtATimeRetrieval(Q, I, f, g, k)
    L ← Array();   R ← PriorityQueue(k)
    for all terms w_i ∈ Q do
        l_i ← InvertedList(w_i, I);   L.add(l_i)
    end for
    while all lists in L are not finished do
        d ← −1
        for all inverted Lists l_i ∈ L do
            if l_i.getCurrentDocument() > d then
                d ← l_i.getCurrentDocument()
            end if
        end for        ▶ d is the first element with the highest index
        s_d ← 0
        skip ← false
        for all inverted lists l_i ∈ L do
            l_i.skipForwardToDocument(d)
            if l_i.getCurrentDocument() = d then
                s_d ← s_d + g_i(Q) · f_i(d)
                l_i.movePastDocument(d)
            else
                skip ← true
            end if
        end for
        if not skip then R.add(s_d, d)
    end while
    return the top k results from R
end procedure
```

Schwellenwertmethoden

Eine weitere Optimierungstechnik:

- Schwellenwertmethoden verwenden
  1. die Anzahl der benötigten bestplatzierten Dokumente (k),
     - für die meisten Applikationen ist k klein
  2. und die bisher erzielten Zwischenergebnisse

  um die Anfrageverarbeitung zu optimieren

- Die Bewertung des aktuell k-besten Dokuments gibt den Schwellenwert τ' (*minimum score*) vor

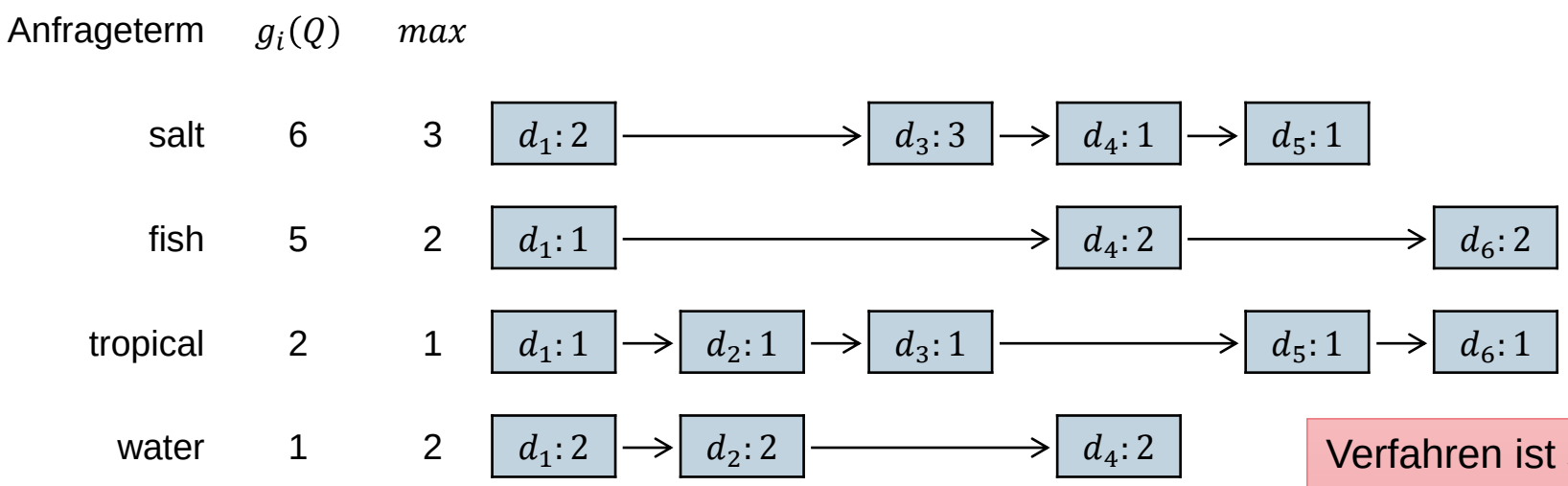
Schwellenwertmethode: Beispiel 1

- Term-at-a-Time (nicht conjunctive) mit unterschiedlichen $g_i(Q)$
- zunächst: Anfrageterme nach $g_i(Q)$ sortieren
- gesucht werden die „besten“ $k = 2$ Treffer

Anfrageterm	$g_i(Q)$	max	
salt	6	3	$d_1:2 \rightarrow d_3:3 \rightarrow d_4:1 \rightarrow d_5:1$
fish	5	2	$d_1:1 \rightarrow d_4:2 \rightarrow d_6:2$
tropical	2	1	$d_1:1 \rightarrow d_2:1 \rightarrow d_3:1 \rightarrow d_5:1 \rightarrow d_6:1$
water	1	2	$d_1:2 \rightarrow d_2:2 \rightarrow d_4:2$

Hier erspart uns die Schwellenwertmethode in vielen Fällen alle Listen anschauen zu müssen.

Schwellenwertmethode: Beispiel 1



Ergebnis nach der Liste „salt“:

1. $d_3:18$
2. $d_1:12$
3. $d_4:6$
4. ...

noch mögliches Gewicht:
$5 \cdot 2 + 2 \cdot 1 + 1 \cdot 2 = 14$
⇒ Top 2 noch nicht stabil!

Ergebnis nach der Liste „fish“:

1. $d_3:18$
2. $d_1:17$
3. $d_4:16$
4. ...

noch mögliches Gewicht:
$2 \cdot 1 + 1 \cdot 2 = 4$
⇒ Top 2 noch nicht stabil!

Ergebnis nach der Liste „tropical“:

1. $d_3:20$
2. $d_1:19$
3. $d_4:16$
4. ...

noch mögliches Gewicht:
$1 \cdot 2 = 2$
⇒ Top 2 nun stabil!

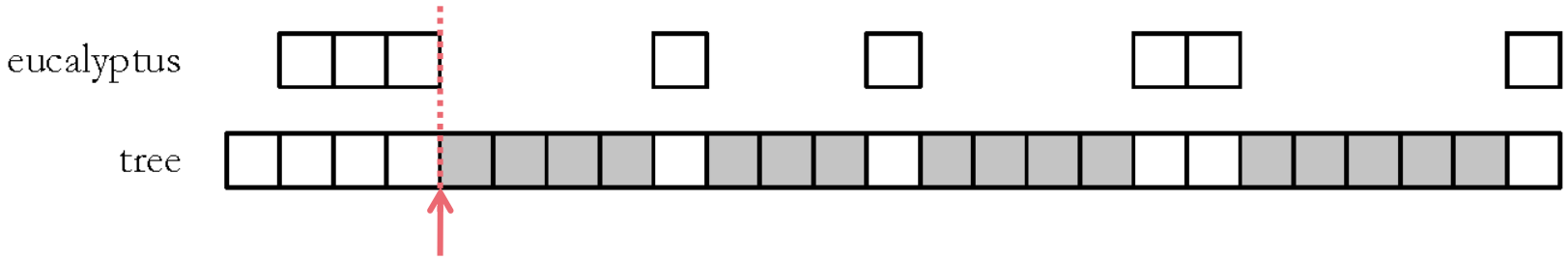
Verfahren ist so approximativ, weil es innerhalb der Top k noch Verschiebungen geben kann! [Wir tauschen Genauigkeit gegen Laufzeit!]

Schwellenwertmethode: Beispiel 2

- Hier: Eine Methode für Document-at-a-time (nicht conjunctive)
- Die MaxScore-Methode vergleicht die größte Bewertung, die ausstehende Dokumente bekommen könnten, mit τ *bzw.* τ'
  - τ ist der Score des k-besten Ergebnisses (aber leider erst nach Bearbeitung der Anfrage bekannt)
  - $\tau' \leq \tau$ ist eine untere Schranke für τ, z. B. der Score des aktuell bekannten k-besten Ergebnisses
  - Sichere Optimierung, da das Ranking ohne Optimierung genauso aussehen würde

Diese Schwellenwertmethode erlaubt uns mit Hilfe der Skip-Pointer zahlreiche Elemente in Listen zu überspringen.

Beispiel zu MaxScore



- Der Indexer berechnet μ_{tree} als den höchsten Beitrag, den tree zum Score leisten kann
  (z. B. indem er die höchste Vorkommenszahl von „tree“ in einem Dokument kennt)
  $\Rightarrow$ maximale Bewertung für jedes Dokument, das nur „tree“ enthält
- Angenommen $k = 3$ und
  - τ' sei die Bewertung des drittbesten Dokumentes, nachdem die ersten 4 Dokumente betrachtet wurden (roter Pfeil)
- Prüfung ergibt nun: $\tau' > \mu_{tree}$
  $\Rightarrow$ Alle grauen Positionen können ab hier unproblematisch übersprungen (geskippt) werden, weil nur noch Dokumente, die beide Anfrageterme enthalten, eine Chance haben, einen höheren Score als τ' zu erreichen

Andere Ansätze zur Optimierung

- Listensortierung:
  - Invertierte Listen werden nach einem Qualitätsmaß (z. B. PageRank) oder nach ihrer partiellen Bewertung geordnet
  - Listen werden einzeln oder parallel durchlaufen
  - Es wird stets berechnet, ob noch Elemente in die Top k kommen können
  - Ggf. können hier viele Elemente am Listenende ignoriert werden
  - Es gibt exakte (müssen die Listen länger durchlaufen) und approximative Varianten (können früher abbrechen)

Hier erlaubt uns die Schwellenwertmethode die Listen nicht bis zum Ende durchlaufen zu müssen.

- Radikal und ohne Schwellenwert: Hochfrequente Terme ignorieren
  - Wortlisten von häufig vorkommenden Termen werden bei der Term-at-a-time-Verarbeitung einfach ignoriert (analog zur Betrachtung als Stoppwort)

Strukturierte Anfragen

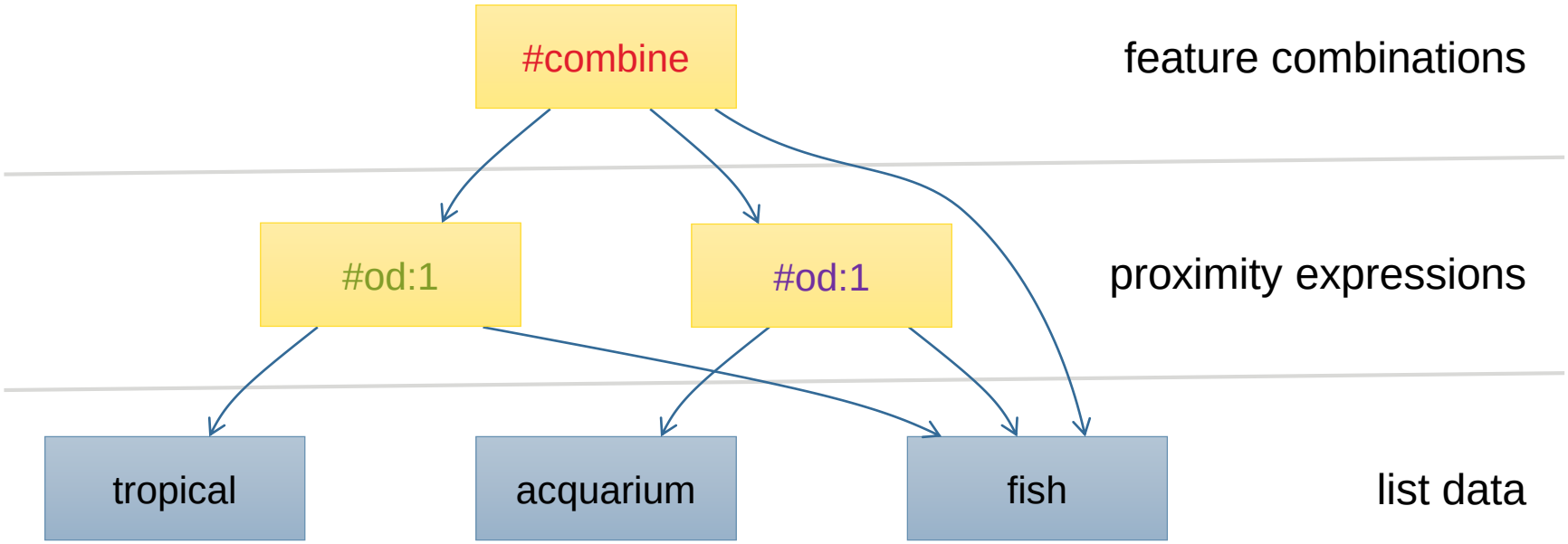
- Eine Anfragesprache (query language) kann die Spezifikation komplexer Features erleichtern
  - Ähnlich SQL für Datenbanksysteme
  - Anfrageübersetzer konvertieren die Benutzereingaben in die strukturierte Anfragerepräsentation
  - Die Galago-Anfragesprache ist ein Beispiel, das hier benutzt wird
  - Beispiel für eine Galago-Anfrage:

    #combine(#od:1(tropical fish) #od:1(aquarium fish) fish)

Begriffe müssen in dieser Reihenfolge nahe beieinander auftreten.

Auswertungsbaum für eine strukturierte Anfrage

#combine(#od:1(tropical fish) #od:1(aquarium fish) fish)



- Auswertung durch geschachtelte Iteratoren (nested loop; get-next)
- Die Iteratoren werden auf der jeweils übergeordneten Ebene wie „normale“ invertierte Listen bearbeitet

Verteilte Auswertung

- Grundprozess:
  - Alle Anfragen werden an eine „director machine“ gesendet
  - Diese sendet dann Nachrichten an viele Indexserver
  - Jeder Indexserver trägt seinen Anteil zur Anfrageverarbeitung bei
  - Die director machine organisiert die Ergebnisse und stellt sie dem Benutzer zur Verfügung

- Zwei Hauptansätze
  1. Dokumentverteilung
     - bei weitem am populärsten
  2. Termverteilung

Dokumentverteilung

- Jeder Indexserver agiert als Suchmaschine für einen kleinen Bruchteil der gesamten Kollektion
- Director machine sendet eine Kopie der Anfrage an jeden Indexserver, wovon jeder die k-besten Ergebnisse zurück gibt
- Die Ergebnisse werden vom director zu einer geordneten Liste verschmolzen

- Die Kollektionsstatistiken sollten für ein effektives Ranking allen (global) zur Verfügung stehen

Termverteilung

- Für das gesamte Rechnercluster wird ein einziger Index erstellt
- Jede invertierte Liste in diesem Index wird dann einem Indexserver zugewiesen

  ⇒ In den meisten Fällen sind die Daten für die Verarbeitung einer Anfrage nicht auf einem einzelnen Computer gespeichert
- Einer der Indexserver wird dazu bestimmt (Election), die Anfrage zu verarbeiten
  - Gewöhnlich der Indexserver, der die längste invertierte Liste hält
- Die anderen Indexserver senden ihre Informationen zu diesem Server
- Die endgültigen Ergebnisse werden zum director gesendet

Caching

- Anfrageverteilung verhält sich nach dem Zipf‘schen Gesetz
  - Ungefähr 50% der täglichen Anfragen sind einmalig, aber einige wenige sind sehr populär
- Caching kann die Effizienz stark verbessern:
  - Caching der Resultate populärer Anfragen
  - Caching wichtiger invertierter Listen
- Das Cachen invertierter Listen kann auch für einmalige Anfragen, die die entsprechenden Terme enthalten, helfen
- Natürlich muss der Cache aktualisiert werden, um (stark) veraltete Daten zu vermeiden

Ranking mit Indexstrukturen: Gliederung

I. Einordnung
II. Ein Modell für das Ranking (als Beispiel)
III. Invertierte Indexstrukturen
IV. Kompression
V. Hilfsstrukturen
VI. Indexerstellung
VII. Anfrageverarbeitung