

Inhaltsverzeichnis

1	Effiziente Algorithmen	2
2	Große Datenbestände indexieren („Information Retrieval“)	3
2.1	Einleitung	3
2.2	Beispielsystem FESAD	6
2.3	Grobarchitektur von IR-Systemen	7
2.4	Das Wörterbuch und seine Implementierung	9
2.4.1	Wörterbuchfunktionalität:	10
2.5	Tippfehlertoleranz	23
2.6	IR-Anfragen abarbeiten	28
2.7	Udi Manbers Technik <i>Glimpse</i>	29

Kapitel 2

Große Datenbestände indexieren („Information Retrieval“)

2.1 Einleitung

Begriffliches:

Informationssysteme (heute unspezifischer Oberbegriff)

Datenbanken (*database systems*)

z.B. relationale Datenbank, objektorientierte Datenbank

Dokumentationssysteme (*information retrieval systems*, „IR“)

z.B. Literaturrecherche, Dokumentation, Archivierung

Suchmaschinen (meist WWW-bezogen)

Work-Flow-Systeme z.B. „R3“ von SAP

Management-Informationssysteme

Literatur-Klassiker:

G. Salton: *Automatic Information Organisation and Retrieval*;
McGraw-Hill, 1968.

Neuste Ausgabe:

G. Salton, M. J. McGill: *Introduction to Modern Information Retrieval*;
McGraw-Hill, 1983.

G. Salton: *Automatic Text Processing: The Transformation, Analysis and Retrieval of Information by Computer*; Addison-Wesley, 1989.

Alte Philosophie:

Sauber aufbereitete Sammlung von Dokumenten mit bestimmten formalen Eigenschaften, z.B. nach formalen Regeln gestaltete Dokumentbeschreibungen, „Felder“ mit bestimmten Merkmalen.

(z.B. Literatur-Abstracts, genaue textuelle Dokumentationen von Fernsehproduktionen (FESAD wird noch ausführlicher diskutiert), Kunstwerke-Dokumentation, Literatur-Zitate im BibTeX:

```
@article{Gettys90,
  author = {Jim Gettys and Phil Karlton and Scott McGregor},
  title = {The {X} Window System, Version 11},
  journal = {Software Practice and Experience},
  volume = {20},
  number = {S2},
  year = {1990},
  abstract = {A technical overview of the X11 functionality.
    This is an update of the X10 TOG paper by Schleifler & Gettys.}
}
```

Vorteile: Berechenbare und verlässliche Anfrageergebnisse, weil der semantische Gehalt der meisten Einträge bekannt und vom Suchprogramm verwertbar ist. (z.B. „Titel von Kunstwerken“: Kann nur mit großem Sachverstand automatisch aus beschreibendem Text herausgelesen werden, hohe Fehlerrate.)

Nachteile: Daten müssen von Hand aufbereitet werden; das ist sehr arbeitsintensiv und kann nur von größeren Organisationen geleistet werden.

Menschheit hat keine Chance, in der Fülle der Informationen/Daten bzw. des Wissens in Handarbeit noch alles nachzuführen. Auch das Wissenssystem muss durch Teilautomatisierung unterstützt werden.

Neue Philosophie:

Suchmaschinen des Internet, alles lesen, was irgendwie Text ist und mit Benutzeranfragen assoziiert werden kann.

Also: Bereitstellung der Daten ist nicht mehr an irgendwelche Formate, Aufbereitungen, Sprachen oder Regeln gebunden. Die Daten werden so genommen, wie sie im Speicher liegen. Datenbereitstellung und Retrieval-Funktionen sind völlig voneinander entkoppelt.

Supersuchmaschinen der Zukunft: Alle Dateien im Internet durchstöbern, die öffentlich zugänglich sind, auch Verzeichnisse /pub/ . . . , sodann semantisches Netz zwischen allen Begriffen erstellen, die vorkommen, das „Semantik-Problem“ lösen, d.h. der Suchalgorithmus kann zumindest nach und nach verstehen, was ein Text aussagt und wie er zu klassifizieren ist – z.B. könnte eine solche Anfrage lauten:

„Wer hat sich in letzter Zeit in welche Publikationsorganen gegen eine Weiterentwicklung von UNIX ausgesprochen? Und mit welchen Argumenten?“

Beispiel zu heutigem Stand:

Suchanfrage bspw. „CAD CAE Robotic Kinematik“

Semantik:

möglichst „4 aus 4“ Begriffen innerhalb 1000 Byte Kontext,
sonst „3 aus 4“ ...,
sonst „2 aus 4“ ...

Nachweis einer Text-Fundstelle:

1. Name der Datei (voller Pfad-Name),
2. Position (in Anzahl Bytes) vom Anfang der Datei aus gerechnet,
3. Fundstelle mit Hervorhebung der Stichwörter und etwas Umgebungstext

Retrieval-Zeiten

mit oder ohne Vorverarbeitung?

Komplexitätsituationen:

- 10 MByte Datenbestand,
- 100 MByte Datenbestand,
- 1 GigaByte Datenbestand,
- 10, 1000, ... GigaByte,
- Internet (regional, international)

2.2 Beispielsystem FESAD

Fakten über FESAD

- Fernseharchiv-Dokumentationssystem
- Entwickelt von SWF, SDR, SR, BR und NDR
- Wird seit 1983 entwickelt und ist seit 1985 im Einsatz
- Mittlerweile ARD-weite Anwendung, (ZDF benutzt ähnliches System)
- Baut auf STAIRS auf¹
- Datenbankgegenstand: Film- und Videomaterial in den Archiven der Fernsehanstalten
- Textbasierte Datenbank, Dokumentationsdaten
- Zugriffszeit < 30 Sekunden (über Netz)
- Personalaufwand: beim SWF 29 Personen in der Dokumentations- und Archivierungsabteilung
- großer Datenumfang (allein SWF 65000 Datensätze / Dokumente)
- Zugriff auf Film- und Videomaterial nur über Boten, die Bänder aus Archiv holen

Unterschied zwischen „Dokumentation“ und „Datenbank“

- **Anfrage:**
„Welche Redakteure haben Beiträge zu Wissenschaftssendungen **und** zu Nachrichtensendungen geleistet?“

¹Storage and Information Retrieval System (IBM)

- **Beantwortung** erfordert Verknüpfung von Relationen
 \Rightarrow *Relationale Datenbank*

{ Typ = Wissenschaft; Redakteure = Foos; ... }

Mengenbildung:

{ $x \mid x = \text{Redakteur} \wedge \exists \text{Sendung}_1 \dots \wedge \exists \text{Sendung}_2 \dots$ }

2.3 Grobarchitektur von IR-Systemen

Komponenten:

- „Index“ (Wörterbuch mit Verweisen auf Fundstellen ..., Thesaurus-Probleme, Wortstämme, Handhabung von Flexionen,² Füllwörter)
- Datenmengengerüst, Elementardaten (Dateibäume, Partitionierungen, ...)
- Software-Module:
 1. Indizierung
 2. IR-Recherche
 3. Verwaltung allgemein / Hand-Dokumentation

Mengengerüst und Elementardaten

Elementardaten sind

- Dateien
- Records innerhalb von Dateien
- Quelltext von WWW-Seiten
- Grafik
- Bild (Pixel-Bilder)
- ...

Wenn man allen Daten gerecht werden will, so muss man in Richtung Multimedia denken:

- Record (komplexe Daten)

²Deklination, Konjugation, ...

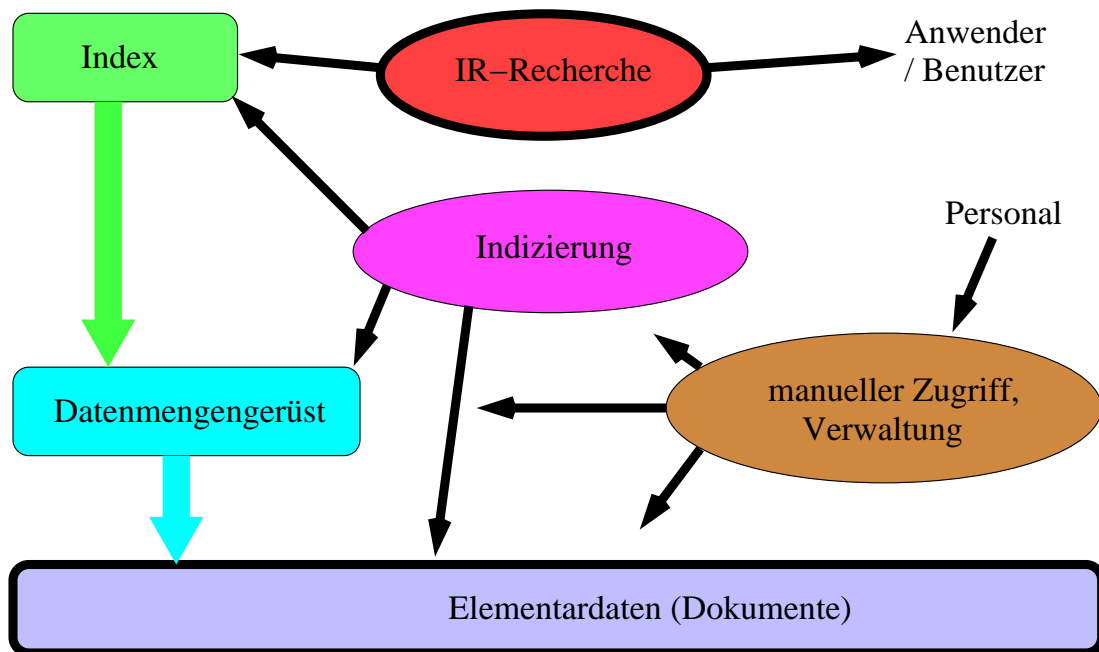


Abbildung 2.1: Architektur von IR-Systemen

- Text (gedruckte Sprache)
- Grafik (z.B. PS-Dateien, ...)
- Bild (Pixel-Bilder, JPEG, ...)
- Video (MPEG, ...)
- Ton (Geräusche, Musik, Sprache)
- ...

Da es aber z.B. sinnlos ist, die Byte-Sequenz eines Pixel-Bildes als Text zu interpretieren, gilt folgendes:

Das IR-System muss den *Typ* der Elementardaten erkennen können und die dem Dokument angemessene *Interpretation* der Merkmale vornehmen.

(Wir behandeln hier nur ASCII-Texte, aber: Grafik, Bild, Video, Ton müssen zukünftig ebenfalls klassifiziert werden können)

Mengengerüst := Lineare Liste der *Dokumente*,
Dokument : „Datei“ oder „Teil einer Datei“ oder „Menge von Dateien“ = **Elementardaten**

Ein Dokument hat

- eine eindeutige „ID“ = laufende Nummer (1, 2, ..., maxNr)
- ein Indizierungsdatum (Abgleich mit Änderungen ...)
- eine Größe (Länge in Bytes)
- evtl. einen Typ (Text, Quellcode, Bild, ...)
- einen „full path name“, der den Zugriff auf das Dokument erlaubt,
- eine Byte-Adressierungs-Ordnung: 0. Byte, 1. Byte, 2. Byte, ...
Jedes Byte des gesamten Elementardaten-Bestandes kann adressiert werden durch ein Tupel

[Dokument-Nr. , Byte-Adr.]

Die Größe und die maximale Anzahl der vorgesehenen Dokumente hat entscheidende Auswirkungen auf **Speicherbedarf**, **Indizierungsgeschwindigkeit** und **Antwortzeiten**.
Von größtem Einfluß sind vor allem Genauigkeit und Vollständigkeit der Adressierung.

2.4 Das Wörterbuch und seine Implementierung

Im Wörterbuch sind alle für Anfragen relevanten Begriffe gespeichert, die in den Elementardaten vorkommen.

2 Strategien:

1. **Thesarus**: aufbereitetes Fachwörterbuch mit Synonymen, mehrsprachigen Synonymen etc.
2. Alles, was an Wörtern vorkommt, evtl. mit Wortstämmen operierend (Flexion) ...

2.4.1 Wörterbuchfunktionalität:

- Wörterbuchzugriffe während Indizierungslauf:
Füge Wort w zusammen mit Adresse A in Wörterbuch ein.
(Ist w schon drin, so ergänze nur die neue Adresse)
- Nach dem Indizierungslauf:
Wörterbuch als Datei sichern.
- Vor der IR-Recherche:
Wörterbuch aus Datei in Speicher einlesen.
- Während der IR-Recherche:
Lese zu Wort w die Liste $(A_1, A_2, \dots, A_{k_w})$ der gespeicherten Adressen aus.

Wichtig: Alphabetische Ordnung der Wörter ist nicht erforderlich, also ist die Hash-Technik anwendbar.

Eine Syntax für die „Wörter“ muss festgelegt werden, d.h. gehört bspw. „-“ oder »Leerzeichen« oder Ziffer+Buchstabe dazu?

Jedoch: Ob Adressen ab-/aufsteigend geordnet sind, ergibt sich beim Indizieren automatisch.

Invertierte Datei:

Wörterbuch *aller* vorkommenden Wörter sowie *alle* Adressen der Wörter-Vorkommen (Fundstellen).

Beispiel: Wenn »Leerzeichen« als ein einziges Zeichen nicht in Wörtern gespeichert wird, kann man die Text-Datei aus dem Wörterbuch rekonstruieren!

Datei:

»ALLES WIRD REKONSTRUIERT.«

1 5 10 15 20 25

Wörterbuch:

ALLES	1
WIRD	7
REKONSTRUIERT	12
.	35

Voll invertierte Dokumente sind günstig, um z.B. Suchanfragen nach

folgenden Begriffen zu beschleunigen:

„Goethes Faust“

„Im Anfang war das Wort“

„CAD <50> CAE“ (d.h. höchstens 50 Zeichen später)

Speicherkatastrophe:

Annahmen: 100.000 Dokumente (Texte)

1 Dokument \approx 100 kByte

\Rightarrow 10 Giga-Byte Elementardaten

(2000 Bibeln \approx 100 Meter Bücher)

beinhalten ca. 500.000 verschiedene Wörter

Das bedeutet:

500.000 Wörter mit je \approx 10 Byte = 5 MByte

Für je 20 Byte an Elementardaten: 1 Adresse im Wörterbuch:

Dokument-Nr. = 3 Byte

+ Byte-Adr. = 3 Byte

= 1 Adresse = 6 Byte

Anzahl der Adressen:

$10 \cdot 10^9 / 20 = 0,5 \cdot 10^9 =$ halbe Mrd.

Anzahl Bytes:

$(0,5 \cdot 10^9) \cdot 6 = 3 \cdot 10^9$ Byte = 3 GB allein für die Adressen!

Einen Ausweg können boolesche Tabellen als **Bit-Vektoren** weisen
(sind nicht voll äquivalent zu Adressen):

	w_1	w_2	w_3	...	w_n	(Liste aller Wörter)
Dok ₁	0	0	1	...	1	
Dok ₂	1	0	0	...	1	
Dok ₃	1	0	1	...	1	
⋮	⋮	⋮	⋮	⋮	⋮	

$100.000 \cdot 500.000 / 8 \cong 6.000.000.000$ Byte = 6 GB!

Leichter Hoffnungsschimmer:

Weil die Bits dünn gesät sind, kann man vielleicht massiv komprimieren!³

Index-Verfahren

Klassisches Information Retrieval (IR):

- volle und genaue Invertierung

Neue Verfahren:

- nur noch Dokumente adressieren (verlangsamt Recherche, siehe Bitvektoren-Technik)
- nur noch Partitionen adressieren (Daten z.B. in 1024 Partitionen zerlegen; noch langsamer, aber Bitvektoren werden handlicher ...)

Da die Speicherung der Wörter heute quantitativ keine Probleme mehr verursacht und da wir mindestens auf unserem eigenem Rechner Rechenzeit zur Verfügung haben, spart man lieber am Speicherbedarf für die Adressierung.

STAIRS System (IBM)

Änderungen gegenüber unserem Ansatz:

- Volle Invertierung \Rightarrow Text könnte rekonstruiert werden.
- Jede Art von Anfrage kann *ohne* Zugriff auf Quelldaten erleichtert werden, nur ganze Dokumente ausgeben geht so nicht.
- Dokumente sind gegliedert, „Kontext“ ist klar, auch formatierte Felder wie z.B. `Author = Manber,Udi; Date = ...`

Vorschlag für die Organisation des Wörterbuchs

Sei A das Alphabet der in Wörtern zulässigen Zeichen.
Integer-äquivalente Ordnungsfunktion

$$Ord : A \rightarrow \{1, \dots, |A|\}$$

³vgl. Kapitel „Datenkompression“, später ...

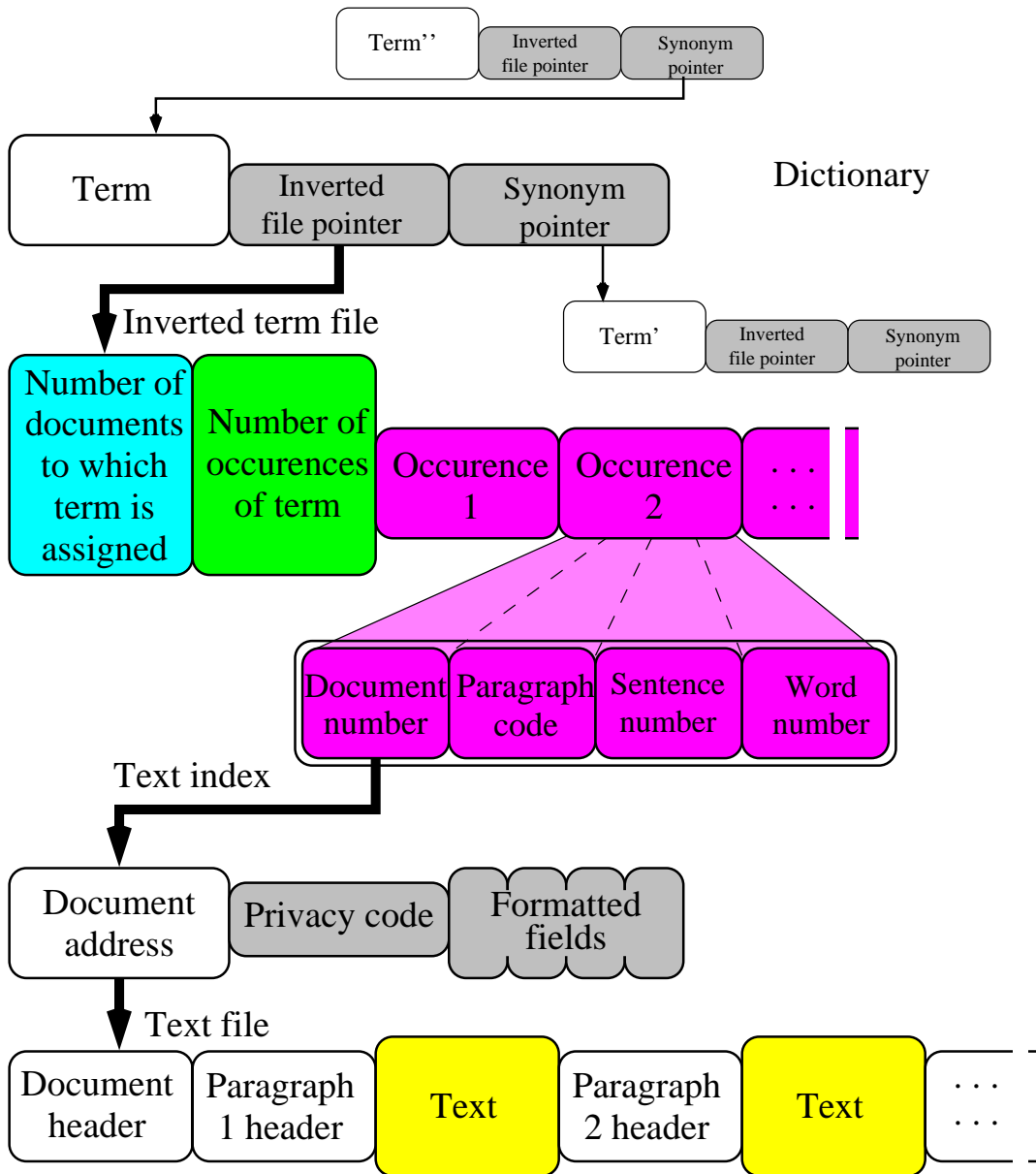


Abbildung 2.2: STAIRS File Organization

Hash-Funktion

Als Beispiel: „Schmitt-Hash“

Hashwert: $A^* \rightarrow \mathbb{N}$

Basis $B := |A| + 1$

Def.: Hashwert($a_1 a_2 \dots a_k$) := $\sum_{i=1}^k B^{(i-1) \bmod 4} \cdot Ord(a_i)$

Idee dahinter:

$$\begin{array}{rcccccl} & B^3 & B^2 & B^1 & B^0 & \\ & & & & & \text{Ziffern zur Basis } B, \\ & a_4 & a_3 & a_2 & a_1 & \text{eigentlich die } Ord(a_i) \\ + & a_8 & a_7 & a_6 & a_5 & \\ + & \dots & \dots & \dots & a_9 & \\ + & & a_k & \dots & \dots & \end{array}$$

$$(s_5) \quad s_4 \quad s_3 \quad s_2 \quad s_1 \quad (\text{Ist 4 ausreichend?})$$

Zahl muss noch mit 32-Bit-Integer locker darstellbar ein, auch für $k = |w| = 40$.

Wir wollen die Hash-Funktion für etwa 20.000 Fächer bzw. „Schlitze“ (slots) oder „Eimer“ (buckets) trimmen. Also:

$$H := (\text{größte Primzahl} < 20.000)$$

– statt 20.000 auch 100.000 wählbar –

H und B sollten teilerfremd sein \Rightarrow Wähle H prim, und man ist auf der sicheren Seite.

Definition:

Hash($a_1 a_2 \dots a_k$) := Hashwert($a_1 a_2 \dots a_k$) mod H

Was kann über die Qualität einer Hash-Funktion gesagt wer-

den?

Aufbau eines Wörterbuchs im Speicher

Wort w bei Zonenindex i (abhängig von Adresse) eintragen:

$\text{Insert}(w, i)$

Probleme:

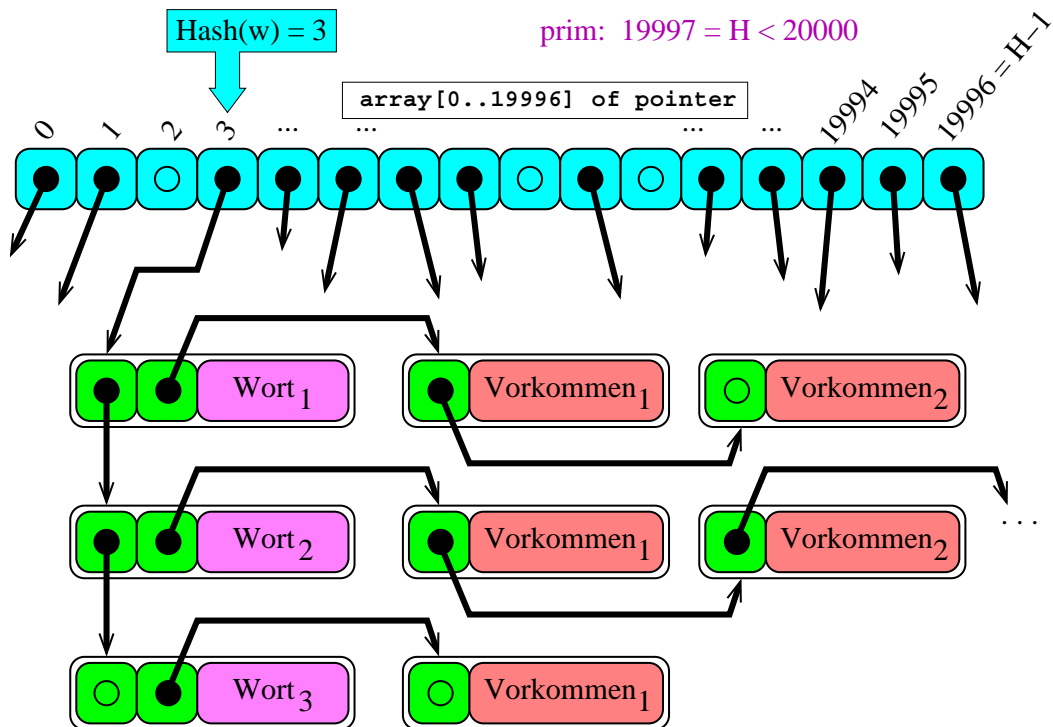
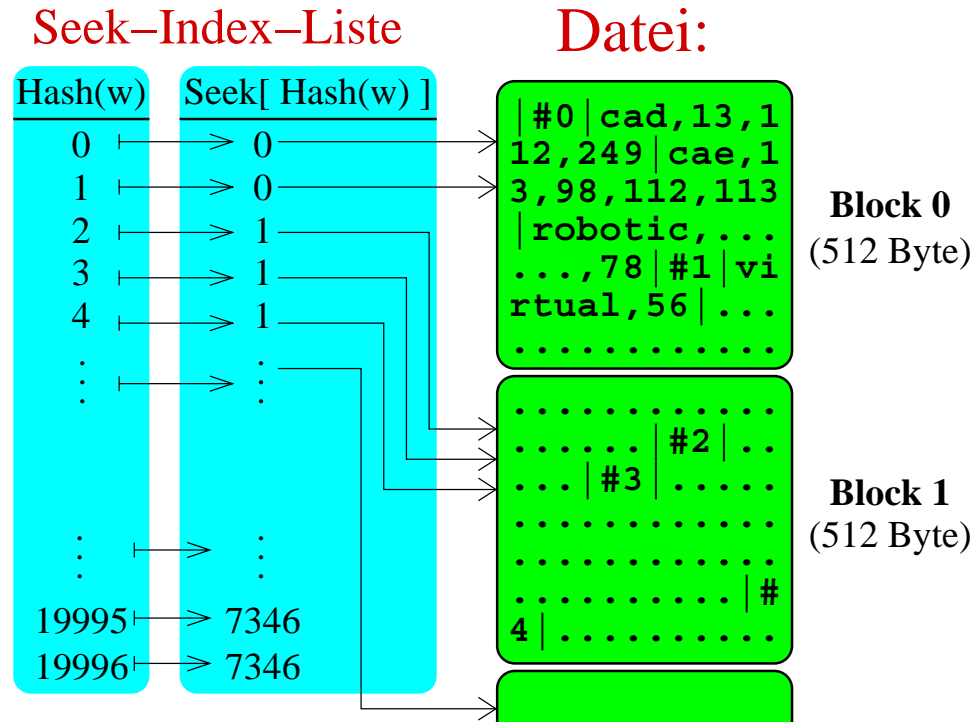


Abbildung 2.3: Aufbau des Wörterbuchs im Speicher

- Worte sind kurz oder lang.
Speicherverschnitt?
- Verfügbare Kapazität des Zentralspeichers?
- Wie legt man alles sequenziell in einer Datei ab?
Siehe z.B.: `java.io.RandomAccessFile`

Ablage des Wörterbuchs auf Datei

Lege große Byte-Datei an, in die mit `seek[...]` eingestochen werden kann: Speichere Wörter und Zonenindizes – lesbar! – ab, z.B. mit



Trennzeichen „|“.

Reihenfolge: $\text{Hash}(w) = 0, = 1, = 2, \dots, = H - 1$.

Seek-Index-Liste: ist relativ kompakt (ebenfalls als Datei zu speichern, muss beim Retrieval „geladen“ werden)

S_0
S_1
S_2
\vdots
S_i
\vdots
S_{19996}

$S_i = \text{seek}[i] =$ Seek-Index, an dessen Position alle Wörter w mit $\text{Hash}(w) = i$ beginnen (Wenn es keine solchen gibt, ist $S_i = 0$)

Wie vorgehen, wenn Zentralspeicher „hinten und vorne“ nicht reicht?

- Virtual Memory Management? → Seitenflattern, Tod.
- Da der Speicher zwar für die Wörter, aber nicht für die Adressen reicht, kann man wie folgt vorgehen:
 Baue Hash-Teillisten bis zu einer gewissen Größe auf und schreibe dann die Adressen im Block auf Datei. *Zum Schluß:* Alle Adressen sind auf Datei
 Baue jede Hash-Teilliste wieder vollständig auf (jetzt genug Platz vorhanden) und gebe alles kompakt auf Datei aus.

Zugriff auf das Wörterbuch beim Retrieval

1. Wort w in Kleinschreibung umwandeln.
2. $h := \text{Hash}(w)$
3. Beschaffe Seek-Index: $S := \text{seek}[h]$
 ($S = 0$ bedeutet: nichts da)
4. Positioniere mit Seek-Prozedur auf Wörterdatei.
5. Besorge $S' := \text{seek}[h + 1] =$ Ende des Suchbereichs.

6. In der Bytefolge $[S, \dots, S' - 1]$ suche nach Wort w .
 Wenn gefunden: Beschaffe Zonen-Indizes i_1, i_2, \dots , die nach w abgelegt sind.
 (Evtl. Zweiteilung Wörterbuch \leftrightarrow Zonen-Indizes-Datei)

Gute Eigenschaften:

- Funktioniert auch dann ziemlich schnell, wenn ≈ 200000 Wörter, da nur ca. 10 Wörter im Suchbereich.
- Kein Verschnitt! (Hash-Technik-Probleme verschwunden: verkettetes Hashen (siehe Goos-Buch), Kollisionsbehandlung etc.)
- Anlegen des Wörterbuchs sehr schnell.
(Geht es noch schneller?)
- Programmierung durchsichtig, einfach.
- Alle entstandenen Dateien haben Textformat, sind also gut lesbar. (Aber: Ist das wichtig? ...)

Schönheitsfehler? Adressen bzw. Zonen-Indizes könnte man kompakter speichern.

Können wir Geschwindigkeit für Eintrag im Wörterbuch abschätzen?

1)	10 Zeichen aus langem Char-Array als Bezeichner identifizieren:	20 μ s
2)	Wort w Extrahieren und in Kleinschreibung wandeln	20 μ s
3)	Hash-Code für w :	10 μ s
4)	Lineare Liste verfolgen, w suchen (5 Tests)	30 μ s
5)	Block-Index vergleichen (mittlere Zahl von Indizes)	50 μ s
6)	malloc (<i>memory allocation</i>), dann nur noch Pointer-Geschäfte (Eintragen, ...)	50 μ s
		180 μ s

$\Rightarrow 0,2$ ms

Nur Schritt 4) ist echt beeinflussbar: evtl. AVL-Baum statt lin. Liste

Bei Schritt 5) sind Bit-Vektoren evtl. erheblich schneller, z.B. bei 255 Zonen.

Hashen wird oft direkt in *Random-Access-Datei* empfohlen:
Aufbau des Wörterbuchs: 1 Zugriff mindestens 5-10 ms, sagen wir 5 ms.

Also: 5 ms mit 0,2 ms vergleichen: 25 mal schneller, wenn im Speicher! (Sehr wichtig für Indizierungslauf!)

10 GigaByte indizieren:

10 Byte in 1 ms

50 KByte in 1 s

$(10 \cdot 10^9)/(5 \cdot 10^4) = 2 \cdot 10^5$ s, also 2-3 Tage!!

(z.B. das Programm „ultrafind“)

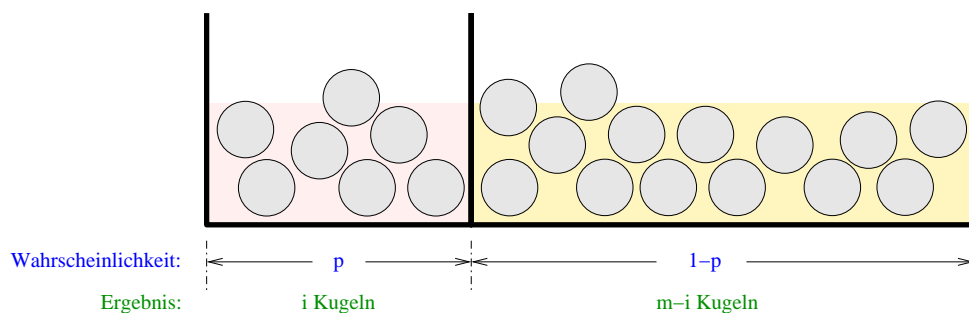
Wie effizient sind Hashen und Zellraster- Techniken?

Wenn man das Wörterbuch in der natürlichen Technik, also z.B. mit balancierten Binärbäumen z.B. a la AVL aufbauen würde, hätte man für die Operation $\text{Insert}(w)$

$$T_{max}(n) = O(\log n)$$

Zeitaufwand. Wie sieht es beim Hashen aus?

Stochastik-Experiment: m Elemente (Kugeln) zufällig einfügen.



Wie groß ist die Wahrscheinlichkeit, dass im p -Bereich genau i

Elemente auftreten? \rightarrow Binominalverteilung⁴

$$W_m(p, i) = \binom{m}{i} p^i (1-p)^{m-i}$$

Hash-Verfahren mit lineare Liste für Elemente, wie lange müssen wir suchen, um ein Element zu finden?

Einträge \Rightarrow Suchzeit:
 Einträge $i = 0 \Rightarrow$ Suchzeit: 1 (nicht 0!)
 Einträge $i = 1 \Rightarrow$ Suchzeit: 1
 Einträge $i = 2 \Rightarrow$ Suchzeit: 2
 Einträge $\vdots \Rightarrow$ Suchzeit: \vdots
 Einträge $i = k \Rightarrow$ Suchzeit: k

Erwartungswert für Suchzeit ist also

$$S_m(p) = 1 \cdot (1-p)^m + \sum_{i=1}^m i \cdot \binom{m}{i} \cdot p^i \cdot (1-p)^{m-i}$$

Es gilt $\binom{m}{i} \cdot i = \binom{m-1}{i-1} \cdot m$, was man mit $\binom{n}{k} := n!/((n-k)!k!)$ leicht verifiziert.

$$\begin{aligned} S_m(p) &= (1-p)^m + m \cdot \sum_{i=1}^m \binom{m-1}{i-1} \underbrace{p^i}_{p \cdot p^{i-1}} (1-p)^{m-i} \quad \text{substituiere } j := i - 1 \\ &= (1-p)^m + m \cdot \underbrace{p}_{p} \cdot \underbrace{\sum_{j=0}^{m-1} \binom{m-1}{j} p^j (1-p)^{m-j-1}}_{(p + (1-p))^{m-1}} \\ &= (1-p)^m + m \cdot p \cdot \underbrace{(p + (1-p))^{m-1}}_{=1} \end{aligned}$$

Also: $S_m(p) \leq 1 + m \cdot p$

Bsp.: m gleich wahrscheinliche Zellen, m Elemente
 $\Rightarrow S_m(\frac{1}{m}) \leq 1 + 1 = 2$

Bsp.: m gleich wahrscheinliche Zellen, $n = k \cdot m$ Elemente

⁴nach Jacob Bernoulli, vgl. Stochastik für Informatiker

$$\Rightarrow S_{k \cdot m}(\frac{1}{m}) \leq 1 + k \cdot m \cdot \frac{1}{m} = 1 + k$$

⇒ Der Erwartungswert verhält sich tatsächlich so, wie man es „erwarten“ würde.

Wir streben beim Hashen zwar Gleichverteilung an, können dies aber selten garantieren. Was ist bei Ungleichverteilung?

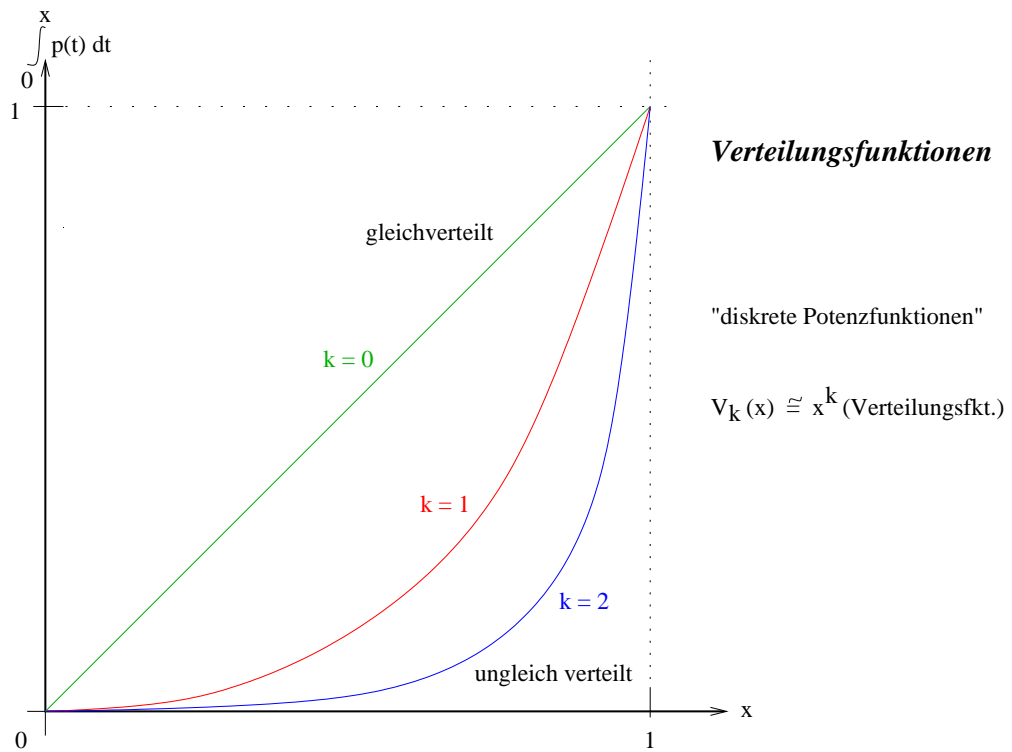


Abbildung 2.4:

Annahmen: m Fächer (fest)

p_i = Wahrscheinlichkeit, dass Fach i ausgewählt wird

$$\sum_{i=1}^m p_i = 1$$

1.Fall: $p_i = \frac{i}{\frac{1}{2}m(m-1)}$, erfüllt also obige Annahme.

$$\begin{aligned}
 E(p_1, p_2, \dots, p_m) &:= \sum_i p_i \cdot S_m(p_i) = \sum_i p_i(1 + m \cdot p_i) \\
 &= 1 + m \sum_i p_i^2 \\
 &= 1 + \frac{4 \cdot m}{m^2 \cdot (m-1)^2} \underbrace{\sum_{i=1}^m i^2}_{Poly_3(m) = \frac{1}{6}(2m+1)(m+1)m} \\
 &= 1 + c_1, \text{ also durch Konstante begrenzt, unabhängig von } m!
 \end{aligned}$$

Satz: Beim zufälligen Fächereinsortieren von m Elementen in m Fächer mit den Einfüllwahrscheinlichkeiten (Potenzverteilungen)

$$p_i := \frac{i^k}{\sum_{i=1}^m i^k} \quad (i = 1, 2, \dots, m)$$

ergibt sich genau wie bei Gleichverteilung eine von m unabhängige⁵ obere Schranke für den Erwartungswert $E(p_1, \dots, p_m)$ für die Anzahl der Elemente in einem Fach.

Voraussetzung: $m \gg 1$

Beweis: Wie gehabt, also $p_i = \frac{i^k}{\Theta(m^{k+1})}$

$$\begin{aligned}
 E(p_1, \dots, p_m) &:= \sum_{i=1}^m p_i \cdot S_m(p_i) \\
 &= \sum_{i=1}^m p_i(1 + mp_i) \\
 &= 1 + m \sum_{i=1}^m p_i^2 \\
 &= 1 + \frac{m}{\Theta(m^{2k+2})} \underbrace{\sum_{i=1}^m i^{2k}}_{\Theta(m^{2k+1})}
 \end{aligned}$$

⁵aber von k abhängige!

$$= 1 + \underbrace{\frac{m \cdot \Theta(m^{2k+2})}{\Theta(m^{2k+2})}}_{\text{für hinreichend große } m \text{ durch Konstante begrenzt}} \leq 1 + c_k$$

für hinreichend große m durch Konstante begrenzt

2.5 Tippfehlertoleranz

Wie kann man im Wörterbuch Tippfehler, Druckfehler und verschiedene Schreibweisen von Wörtern berücksichtigen?

Benutzerfreundlichkeit von Software? Maschine soll menschliche Irrtümer verzeihen, tolerieren, korrigieren, ...

Man kann es so machen:

1. Erstelle zusätzlich reines Wörterbuch ohne Zonen-Indizes:
 $w_1|w_2|\dots|w_m$ als langen Char-String, 1–5 MByte.
2. Für jedes Wort u_1, u_2, \dots, u_n der Suchanfrage:
 Suche in diesem Char-String alle w_i , für die $\text{Abstand}(u, w_i) \leq \alpha$,
 ordne diese w_i dem Suchbegriff u zusätzlich zu.
 Abstand z.B.: minimale Anzahl elementarer Editier-Operationen,
 die aus u w_i machen:
 1 Zeichen einfügen, löschen, ändern.

Ergebnis:

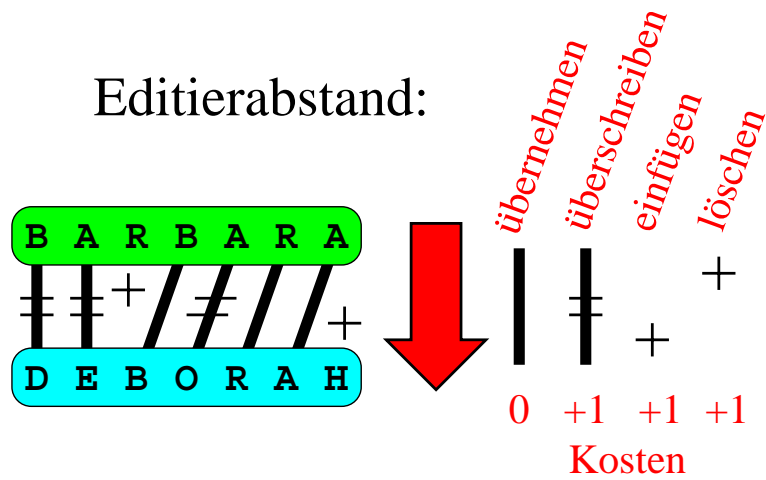
- Jedem u_i werden zusätzlich weitere ähnliche Worte zugeordnet
 \rightarrow Zonen-Indizes sind zu vereinigen!
- Anfrage assoziiert auch leicht abweichend geschriebene Wörter!
- Stellt auch fest, *ob ein Wort falsch geschrieben* wurde (im gesamten Datenbestand!)

Knifflig ist nur, ob man Schritt 2. auch schnell genug durchführen kann, z.B. in 1–5 Sekunden!

Editier-Abstand von Zeichenketten:

$$2 \cdot \text{„}\neq\text{“ und } 1 \cdot \text{„}\neq\text{“} \Rightarrow \text{Editier-Abstand} = 3$$

Editierabstand:



Methode: Kürzesten Weg durch Matrix suchen (Dynamisches Programmieren!)

		B A R B A R A						
	0	1	2	3	4	5	6	7
D	1	2	3	4	5	6	7	8
E	2	3	4	5	6	7	8	9
B	3	2	3	4	5	6	7	8
O	4	3	4	5	6	7	8	9
R	5	4	5	4	5	6	7	8
A	6	5	4	5	6	5	6	7
H	7	6	5	6	7	6	7	8

Aufwand dafr ist $\approx O(\max\{|w_1|, |w_2|\} \cdot \alpha)$, weil $\text{Abst}(w_1, w_2) \leq \alpha$

Da nur Werte um $\alpha = 1, 2$ interessieren, kann man von linearem Aufwand ausgehen.

Härtetest:

		A A A A A A					
	0	1	2	3	4	5	6
A	1	0	1	2	3	4	5
A	2	1	0	1	2	3	4
A	3	2	1	0	1	2	3
A	4	3	2	1	0	1	2

Ist der Editier-Abstand **eine Metrik**? Ja, denn

1. $\text{Abst}(w_1, w_2) \geq 0$ und $= 0$ genau dann, wenn $w_1 = w_2$
(pos. Definitheit)
2. $\text{Abst}(w_1, w_2) = \text{Abst}(w_2, w_1)$
(Symmetrie)
3. $\text{Abst}(w_1, w_2) + \text{Abst}(w_2, w_3) \geq \text{Abst}(w_1, w_3)$
(Dreiecksungleichung)

Satz:

Der Editier-Abstand zwischen Zeichenketten ist eine Metrik.

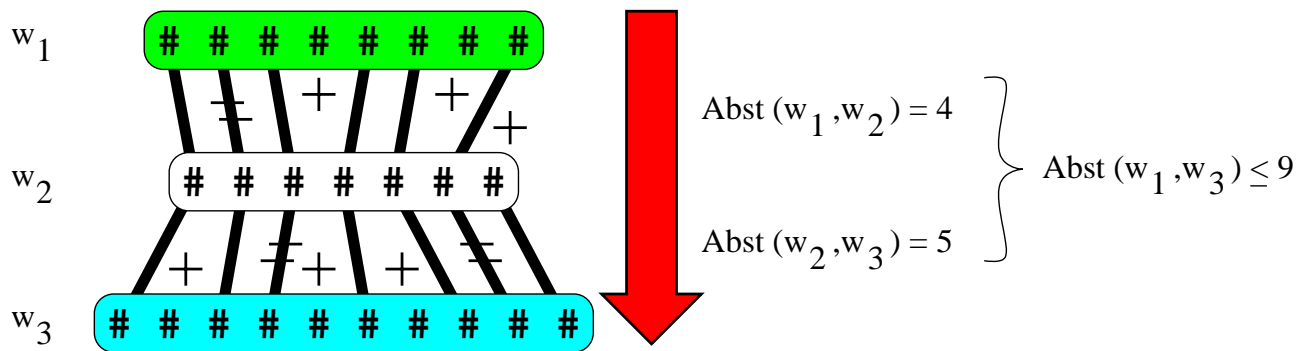
Beweis:

1. trivial.
2. auch trivial, weil keine Vorzugsrichtung festgelegt ist.
3. Beweis für Dreiecksungleichung:

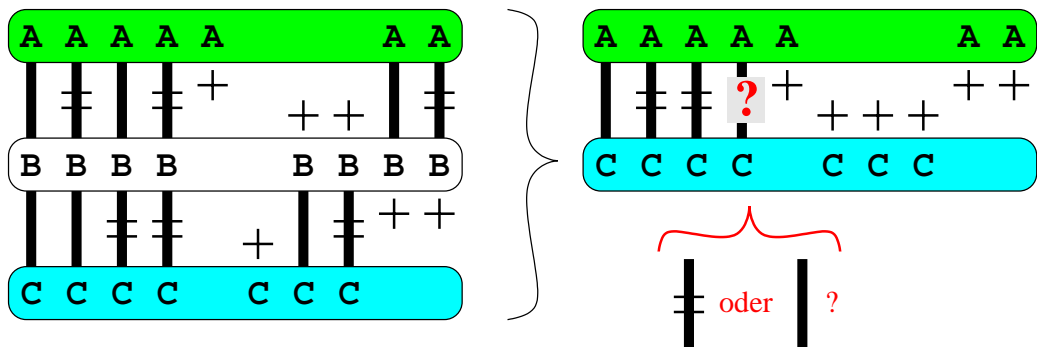
Abstand ergibt sich aus optimalen Zuordnungen.

Beweis-Idee: Verkette zwei solche Zuordnungen, also:

Konstruiere aus $\bullet_1 \longleftrightarrow \bullet_2$ und $\bullet_2 \longleftrightarrow \bullet_3$
eine Zuordnung $\bullet_1 \longleftrightarrow \bullet_3$.



Wie wird konstruiert?



$$\Rightarrow \text{Abst}(a, b) + \text{Abst}(b, c) = \underbrace{\widetilde{\text{Abst}}(a, c)}_{\text{konstruiert}} \geq \underbrace{\text{Abst}(a, c)}_{\text{optimal}}$$

Abschätzungen

500.000 Wörter mit je 10 Zeichen im Wörterbuch
 \Rightarrow 5 MByte Wortinformation.

Übungsaufgabe:

Man kann zu einer gegebenen Menge $M := \{w_1, w_2, \dots, w_k\}$ von Anfragewörtern eine reguläre Sprache $L \subset X^*$ angeben, die alle Wörter w' enthält, die z.B. von den Wörtern in M einen Editier-Abstand ≤ 3 haben.

\Rightarrow Es existiert ein sehr schneller (konstruierbarer) Algorithmus, der zu M alle Wörter im Wörterbuch findet, die einen Editier-Abstand

≤ 3 haben.

Wie schnell? 2–3 Befehle pro Zeichen.

Warum ist die Toleranz gegenüber kleinen Abweichungen so interessant?

- Benutzer vertippt sich
- Wörter haben verschiedene Schreibweisen (z.B. Foto/Photo, Citrus, Frisör, ...)
- Texte, die mit OCR-Scanning gewonnen wurden, haben hohe Rate an Lesefehlern („hohe Rate“: 1%, ..., 0,1% aller Zeichen falsch erkannt) \Rightarrow Buchlesemaschine ...
- Alte vs. neue Rechtschreibung

Die Metrik-Eigenschaft macht das Abstandsprinzip auch für die Mustererkennung interessant.

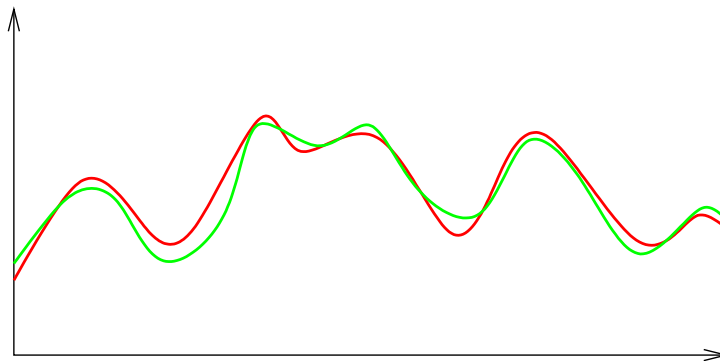


Abbildung 2.5:

Literatur:

P.A.V Hall, G.R. Dowling: *Approximate String Matching*. ACM Computing Surveys 12 (1980), Seiten 381-402.

2.6 IR-Anfragen abarbeiten

1. **Boolesche Verknüpfungen** von Suchbegriffen (komfortable Variante, miteinander kombinierbar):

Zum Beispiel $(w_1 \wedge w_2 \wedge \neg w_3) \vee (w_1 \wedge w_4)$ soll vorkommen

Abarbeitung (volle Invertierung):

$I(w_j)$ = Menge der Adressen, an denen w_j im Datenbestand vorkommt
Adresse: [Dokument-Nr., Byte-Adresse]

Zunächst: Relevante Dokumentnummer, dann evtl. alle Kontexte (ca. 1000–2000 Zeichen) finden, in denen die Boolesche Verknüpfung erfüllt ist.

Kontext ist nur bei großen Dokumenten erforderlich. Strikte Auswertung:

- $w_1 \wedge w_2$: $I(w_1) \cap I(w_2)$ (evtl. kontextabhängig)
- $\wedge \neg w_3$: $\setminus I(w_3)$
- etc.

2. **Toleranzvergleich**

$$I'(w) := \bigcup_j I(w_j)$$

$\forall w_j$: $\text{Abst}(w, w_j) \leq \alpha$ (für ein abgefragtes α), bzw.

$\forall w_j$: w ist Teilwort von w_j

Ansonsten: Vorgehen wie gehabt.

3. **k -aus- n -Anfrage**

Gegeben sind n Suchbegriffe. Liefere die Textstellen so, dass möglichst n aus n Wörtern, dann $(n - 1)$ aus n Wörtern usw. vertreten sind („Weichere“ Variante der booleschen Anfragebearbeitung).

Verbesserung: Einige Wörter müssen bzw. dürfen nicht vorkommen:

Eingabe des Benutzers: $w_1 + w_2 - w_3 w_4 w_5$

Es kommen Textstellen mit

w_1	+	-	w_4	w_5	$(3 \in 3)$
w_1	+	-	w_4	-	$(2 \in 3)$
w_1	+	-	-	w_5	$(2 \in 3)$
-	+	-	w_4	w_5	$(2 \in 3)$
w_1	+	-	-	-	$(1 \in 3)$
-	+	-	w_4	-	$(1 \in 3)$
-	+	-	-	w_5	$(1 \in 3)$
-	+	-	-	-	$(0 \in 3)$

Abarbeitung? Textstellen nach Rang sortieren bzw. vormerken.

$$(I(w_2) \wedge \neg I(w_3)) \wedge (I(w_1) \vee I(w_4) \vee I(w_5))$$

Opt.
Opt.
Opt.

Strategien der Auswertung: beteiligte Adressmengen möglichst schnell dezimieren ...

Grob-Adressierung: Die Kontext-Kriterien können nicht mehr ohne Zugriff auf die Quelldaten geprüft werden, Details bei „Glimpse“.

2.7 Udi Manbers Technik *Glimpse*

Udi Manber: *Introduction to Algorithms - A Creative Approach*. Addison Wesley 1989.

Zielsetzung: PC, kleiner Index, schnelle Indizierung.

1. **Vergrößere die Adressen z.B. auf nur noch 256 verschiedene.**

Konsequenz: Im schlimmsten Fall müssen für ein Wort höchstens 256 Byte an Adressinformation gespeichert werden, jede Adresse höchstens einmal, meistens aber viel weniger.

Verwendet man Bitvektoren, so sind es 32 Byte⁶ Adressinformation pro Wort, unkomprimiert.

⁶Bitvektor: 32 Byte = 256 bit / (8 bit/Byte)

2. **Die einzelne Adresse steht für eine Partition = Zone von ca. 1/256-tel der Originaldatenmenge.** Wenn diese 1 GigaByte Umfang hat, sind es also ca. 4 MByte.
3. **Mit Bitvektor-Technik können UND, ODER etc. durchsichtig und effizient (maschinennah) realisiert werden.** Innerhalb einer potentiellen Zielpartition muss allerdings durch schnelle Textdurchmusterung sichergestellt werden, dass der geforderte Kontext von z.B. 2000 Zeichen gegeben ist.

Testen Sie Glimpse:

The Collection of Computer Science Bibliographies

<http://liinwww.ira.uka.de/bibliography/index.html>

This is a collection of scientific literature in computer science from various sources, covering most aspects of computer science. The bibliographies are updated monthly from their original locations that you'll always find the most recent versions here.

The collection currently contains more than 1,2 million references (mostly to journal articles, conference papers and technical reports), clustered in about 1400 bibliographies, and consists of 660 MBytes of BibTeX entries. More than 19000 references contain crossreferences to citing or cited publications. More than 150000 references contain URLs to an online version of the paper. There are more than 2000 links to other sites carrying bibliographic statistics.

Since the bibliographies are not just referenced by links, but actually mirrored and present as a local copy, they are searchable. This search facility uses Glimpse (by Udi manber, Sun Wu, and Burra Gopal).

U.Manber, Sun Wu: *Glimpse: A Tool to Search Through Entire File Systems*. TR 93-34 (Univ. of Arizona, Tuscon). Dept. of Computer Science (Als postscript-Datei verfügbar, URL verfolgen.)

Anfragezeiten: z.B. in 69 MByte (4300 Dateien) wurde

Usenix AND Winter 19 mal gefunden \Rightarrow 5 Sekunden CPU-Zeit.
Indizieren: 5 Minuten (DEC Workstation), 9 Minuten Realzeit (elapsed time)
Indexgröße: 1,9 MByte \approx 2,7% relativ, 205 Zonen
Search-Zeiten: 2–10 Sekunden,
Czechoslovakia (\pm 2 Tippfehler) \rightarrow 3,6 Sekunden
protein AND matching⁷ \rightarrow 7,7 Sekunden

Glimpse Strength

1. Very small index.
2. Approximate matching is supported.
3. Fast index construction.
4. No need to define document boundaries ahead of time. It can be done at query time.
5. Easy to customize to user preferences.
6. Easy to adapt to parallel computer (different blocks can be searched by different processors).
7. Easy to modify the index due to its small size. Therefore, dynamic texts can be supported.
8. No need to extract stems. Subword queries are supported automatically (even subwords that appear in the middle of words).
9. Queries with wildcards, classes of characters, and even regular expressions are supported.

Glimpse Weakness

1. Slower compared to inverted indexes for some queries. Not suitable for application where speed is the predominant concern.
2. Too slow, at this stage (but we're working on it), for very large texts (more than 500 MByte).
3. Boolean queries containing common words are slow.

⁷sehr häufig vorkommende Wörter

Wie soll Glimpse weiterentwickelt werden?

1. Komprimierte Texte durchsuchen

Interessant vor allem für natürlichsprachige Texte. Dekomprimieren vermeiden: Geeignete Kompressionsalgorithmen einsetzen. Suche evtl. schneller als im nicht komprimierten Text.

Simple Kompressionsschema:

$$\begin{array}{ccc} \langle \text{Byte}_1 \rangle & \langle \text{Byte}_2 \rangle & \langle \text{Byte}_3 \rangle \\ \in & \in & \in \\ 0, \dots, 127 & 0, \dots, 127 & 128, \dots, 255 \end{array}$$

- Die 127 häufigsten Wörter der Sprache : 1 Byte⁸
- Die 127² nächsthäufigen Wörter : 2 Byte⁹
- Die restlichen 1.000.000 Wörter mit 3 Byte¹⁰

Natürliche Sprache (z.B. Deutsch) \approx 1,4 bit Informationsgehalt pro Zeichen, theoretisch also auf ca. 1/5 komprimierbar!

2. Inkrementelles Indizieren

Hinzunehmen von weiteren Files in die Indizierung, ohne Index völlig neu zu erstellen

- kleinste Zone um neues File erweitern (oder weitere Zone anlegen)
- Index in Speicher lesen (Zeiger- Datenstruktur)
- Datei indizieren (alle Wörter mit evtl. neuem Zonen-Index ergänzen)
- Index auf Datei schreiben
- ⇒ Nachrichtendienste, tägliche oder stündliche Aktualisierung

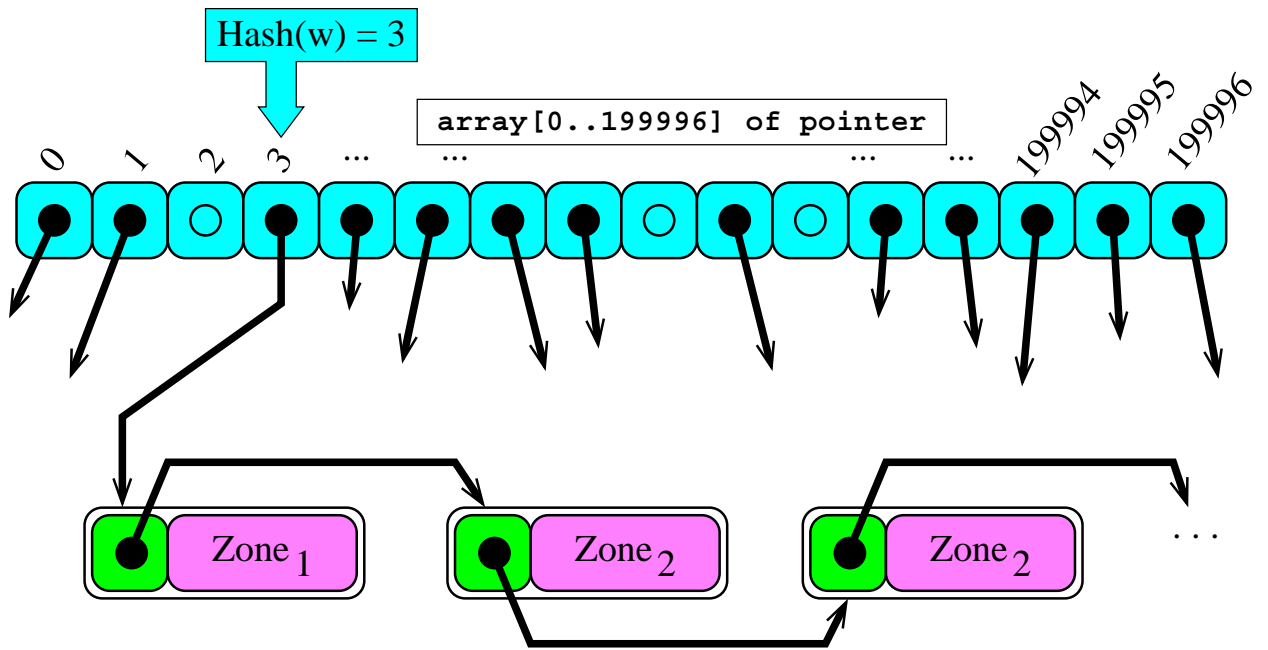
Assoziationsprinzip auf die Spitze treiben:

Definiere statt 20.000 Hash-Slots z.B. 200.000 Slots. Merke die Wörter mit $\text{Hash}(w) = i_0$ nicht mehr, sondern: Liste der *Zonen*, in denen diese Wörter vorkommen.

⁸höchstes Bit bei *Byte*₁ gesetzt

⁹höchstes Bit nicht bei *Byte*₁, aber bei *Byte*₂ gesetzt

¹⁰höchstes Bit nicht bei *Byte*₁ und *Byte*₂, aber bei *Byte*₃ gesetzt



Alle Wörter mit dem gleichen Hash-Index werden in einen Topf geworfen, also nicht mehr unterschieden.

Wörterbuch:
Abbildung $\text{Hash}(w) \mapsto \text{Zone}_i$

Wenn praktisch nur noch ein Wort in jedem Slot steckt, ist die Zonenliste aussagekräftig.
Da wir ohnehin die Quelltexte durchmustern müssen, ist es evtl. tolerierbar, wenn wir ein paar Zonen mehr anschauen müssen.
Attraktiv wird diese Technik, wenn man z.B. mit 5000 Zonen und wirkungsvoller Bitvektor-Kompression zur Darstellung der Zonenlisten arbeitet.