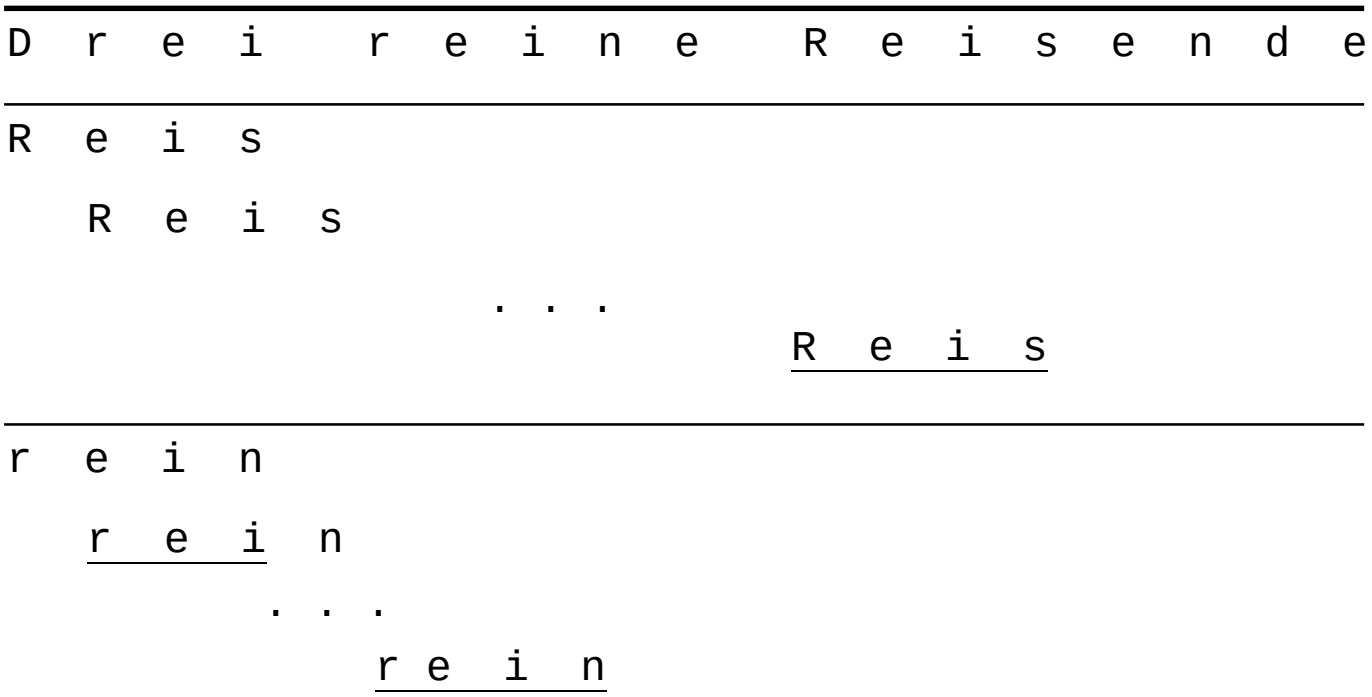
Suchen in Texten

- Sie wissen, wie in einem Text gesucht werden kann
- Sie wissen, wie in grossen Textsammlungen gesucht wird
- Sie wissen, was unscharfe/fuzzy Suche ist
- Sie können mit Regex in Java umgehen

Exakte Textsuche im Hauptspeicher

Finden eines Teilstrings in einem String



- int indexOf(String str, String pattern)
  - liefert die Position, an der das Muster beginnt
  - -1 falls das Muster nicht vorkommt

Brute-Force Suche in Java

```
static int indexOf(String str, String pattern) {
  for (int i = 0; i < str.length() - pattern.length() + 1; i++) {
    int k;
    for (k = 0;
         k < pattern.length() && str.charAt(i+k) == pattern.charAt(k);
         k++);
    if (k == pattern.length()) return i;
    }
  return -1;
}
```

- Muster wird an die Position i gesetzt
- Es wird mit dem String verglichen bis
  - Ende des Musters erreicht -> Erfolg
  - Nichtübereinstimmung
- Worst-Case Aufwand ist **O(n*m)**

Beschleunigte Stringsuche: KMP

- Problem: der String-Zeiger (i+k) wird immer wieder zurück gesetzt
- Grundidee von Knuth, Morris & Pratt: wenn ich bis zum "i" im Suchmuster gekommen bin, dann kann ich direkt weitergehen, wenn im Suchmuster dies das erste Vorkommen ist:
- im Muster "r e i n" kommt "i" nur einmal vor.
- Es wird ein Automat so konstruiert, dass im Text selber nur vorwärts gelesen wird.
- Worst-Case Aufwand **O(m + n)**

Brute-Force vs. KMP

- In der **praktischen Anwendung** ist Brute-Force **doppelt so schnell** wie KMP
  - KMP wird nicht mehr verwendet und deshalb hier auch nicht im Detail behandelt

Kann bei Bedarf in jedem ADS Lehrbuch oder Unterlagen der Kollegen nachgelesen werden

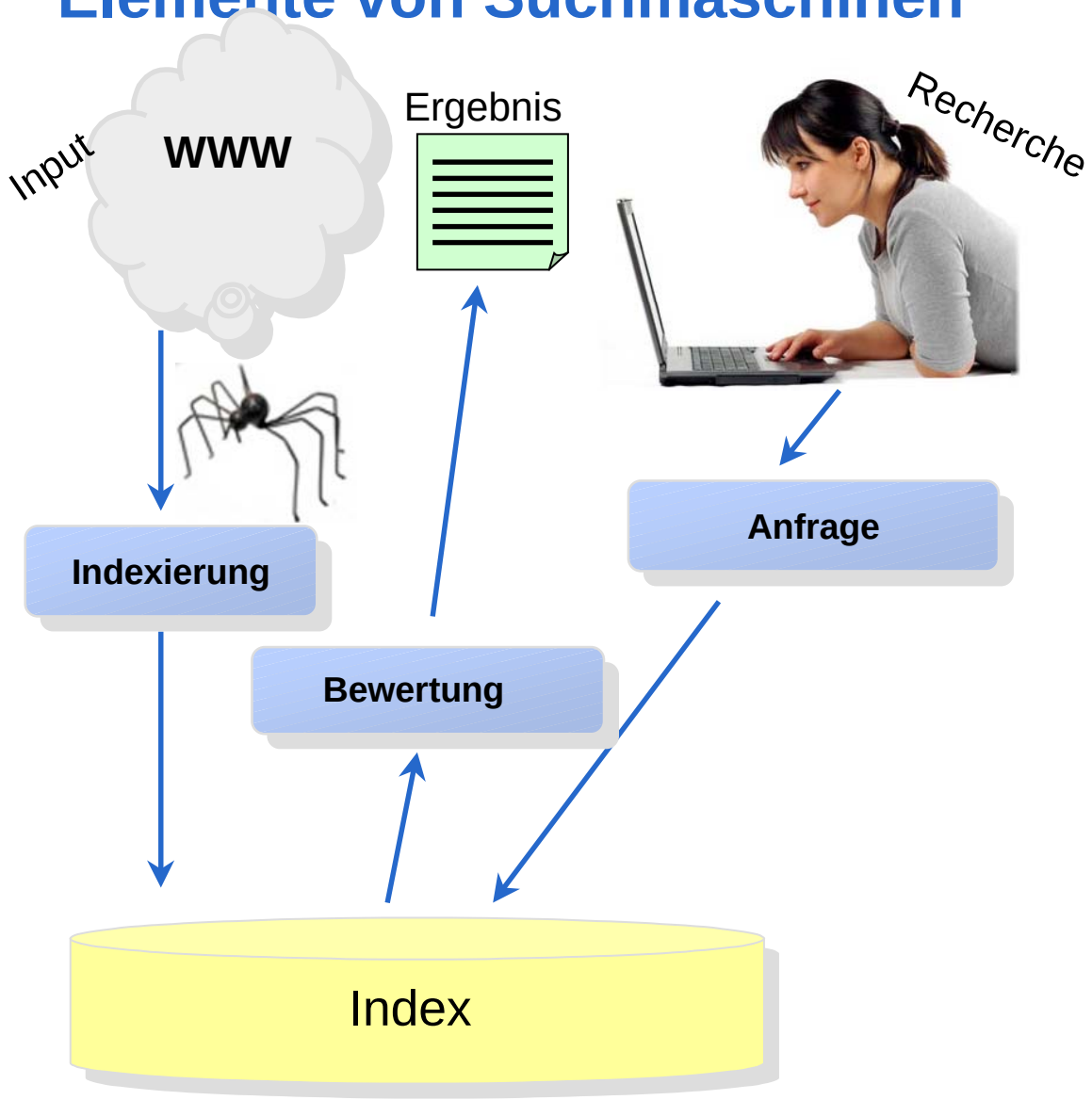
- Probleme von KMP:
  - Worst-Case Fall tritt in der Praxis selten auf
  - Er wurde für eine andere Speicherarchitektur entwickelt: Speicherbänder, wenig Hauptspeicher
  - Suchmuster muss vorher analysiert (compiliert) werden
    - *initialer Aufwand, der sich für kurze Suchmuster/Texte i.d.R. nicht auszahlt*
  - Suche wird als Automat implementiert
    - *der Speicher wird an alternierenden Stellen zugegriffen*
- Vorteile von Brute-Force
  - Speicher wird mehrheitlich linear zugegriffen -> Memory Cache Mechanismus moderner Prozessoren kann greifen
  - Wird z.T. von spezial Maschineninstruktion unterstützt, z.B. CMPS bei Intel Prozessoren

Einfacher Brute-Force Ansatz kann auch die beste Lösung sein

Einstein: Everything should be made as simple as possible, but no simpler.

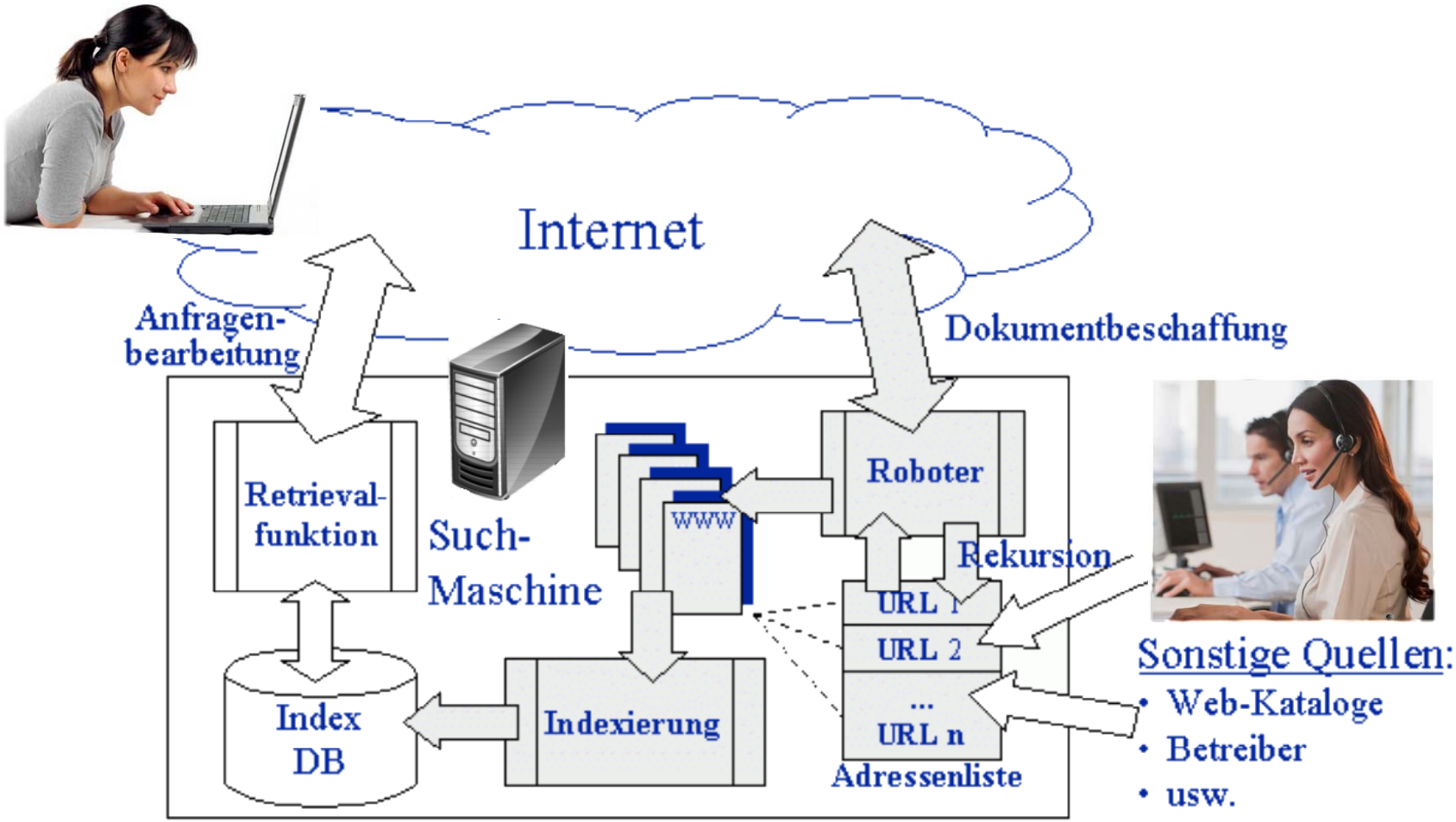
Suche in grossen Textsammlungen und Suchmaschinen

Elemente von Suchmaschinen



- **Web Roboter/Spider/ Crawler**
  - Durchlaufen regelmässig das Web nach neuen Informationen
- **Indexierung**
  - Aufbereitung von Dokumenten
  - Speicherung im Index / in der Datenbank der Suchmaschine
  - Dateisystem, das für die Suche geeignet ist
- **Retrievalsystem**
  - Suche im Index
  - Sortierung nach Relevanz
    - *Wo kommen die Suchbegriffe vor?*
    - *Wie oft kommen die Begriffe vor?*
    - *In welcher Reihenfolge?*
    - *Wie lang ist der Text?*
    - *Wie viele Links verweisen auf das Dokument?*

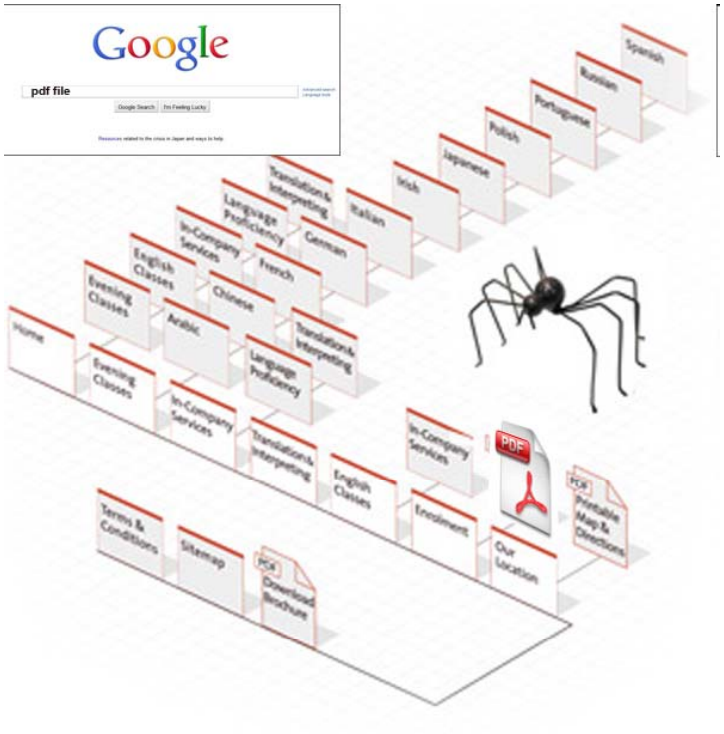
Wie funktioniert Google?



Quelle: Bekavac 2001

Index und Indexierung

- **Index** - Datenstruktur für die unstrukturierten Daten, die so aufgebaut ist, dass sie rasch durchsucht werden kann. (Tabellenform)
- **Indexierung** - Prozess der Erstellung eines Indexes.
- **Bsp**.: Stichwortverzeichnis im Buch
  - Schnellstmögliche Lokalisierung von Seiten, die bestimmte Themen behandeln.
- Google's Index berücksichtigt jeden erreichbaren Text im Internet



Indexierung – invertiertes Dateisystem

- Um einen schnellen Zugriff auf Textdokumente über enthaltene Stichwörter zu ermöglichen, müssen die Texte zu einem invertierten Dateisystem aufbereitet werden.

- Ein invertiertes Dateisystem besteht aus
  - Den **direkten Dateien**. Das sind die Textdokumente.
  - Dem **Index**
    - *Das ist eine alphabetische Liste der in den direkten Dateien enthaltenen **Stichwörter**.*
    - *Jede Indexeintragung verweist wiederum auf eine invertierte Datei.*

- Den **invertierten Dateien**. Sie sind die Verbindungen zwischen dem Index und den direkten Dateien.
  - Sie verweisen auf alle Textdokumente, in denen das bestimmte Stichwort vorkommt.
  - Ausserdem enthalten sie Informationen, die für ein Rankingverfahren

Glöggler, Kap. 3.2 und 3.3.3

Die direkte Datei

- Verbleiben auf dem Server

Doc1

Er konnte nie über etwas lachen, wie kann ein **Mensch** so tief verflachen!

http://www.morgenstern/texte/text1.html

Doc2

Wozu, so fragt man sich, Reichtum, Wohlstand, Macht, wenn alles dies die Menschen nur verflacht?

http://www.morgenstern/texte/text2.html

Die alphabetische Liste (Ausschnitt)

- **Ein Wort wird nur einmal in den Index aufgenommen**

Wird von Suchmaschine gespeichert

numerisch sortierter Index		alphabetisch sortierter Index	
Wortnr.	Stichwort	Stichwort	Wortnr.
9	ein	konnte	2
10	**mensch**	lachen	6
11	so	macht	20
12	tief	man	16
13	verflachen	**mensch**	10
14	wozu	menschen	25
15	fragt	nie	3
16	man	nur	26
17	sich	reichtum	18
18	reichtum	sich	17

Index – invertierte Datei

Nr. Stichwort	DocID / url	Position im Text	tag	Frequenz
...	...	...		...
10	http://www.morgenstern/texte/text1.html	10	<b>	1
11	http://www.morgenstern/texte/text1.html http://www.morgenstern/texte/text2.html	11 2	<p> <p>	1 1
...	...	...		...
27	http://www.morgenstern/texte/text2.html	15	<p>	1

Für Bewertung

Google's Map Reduce

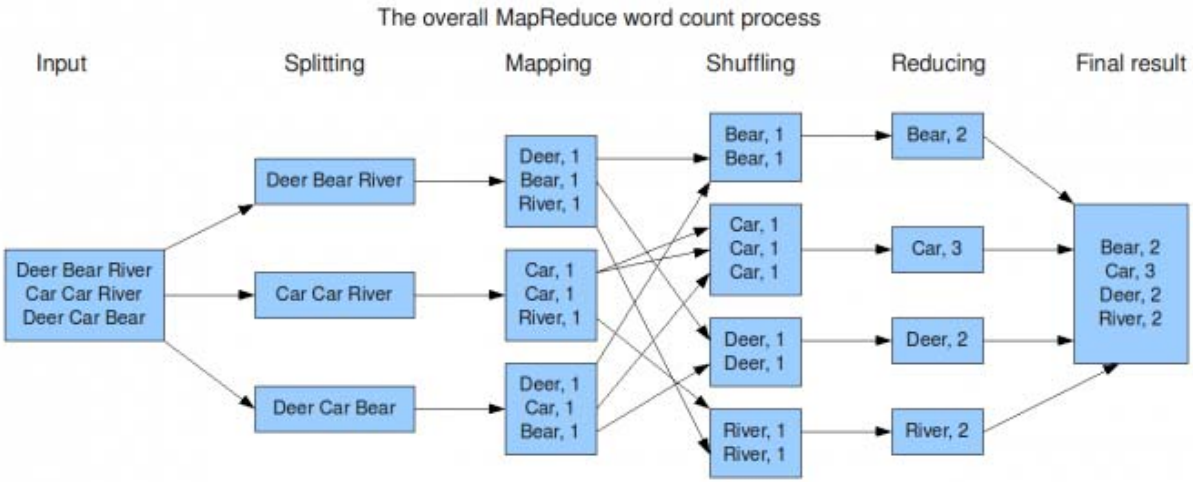
https://www.youtube.com/watch?v=CPjSvanPl7s

https://hub.packtpub.com/how-google-mapreduce-works-big-data-projects/

Struktur und Schritte von Map Reduce

- MapReduce als generelles Ausführungskonzept
  - Task werden in viele kleinere Subtasks aufgeteilt (Split)
    - *die durch billige PCs bearbeitet werden können*
  - Map: PC erledigt seinen Task: erzeugt (key, value) Paare
  - Shuffle: Paare werden gruppiert und an Reducer weitergeleitet
  - Reduce: Teilresultate werden kombiniert

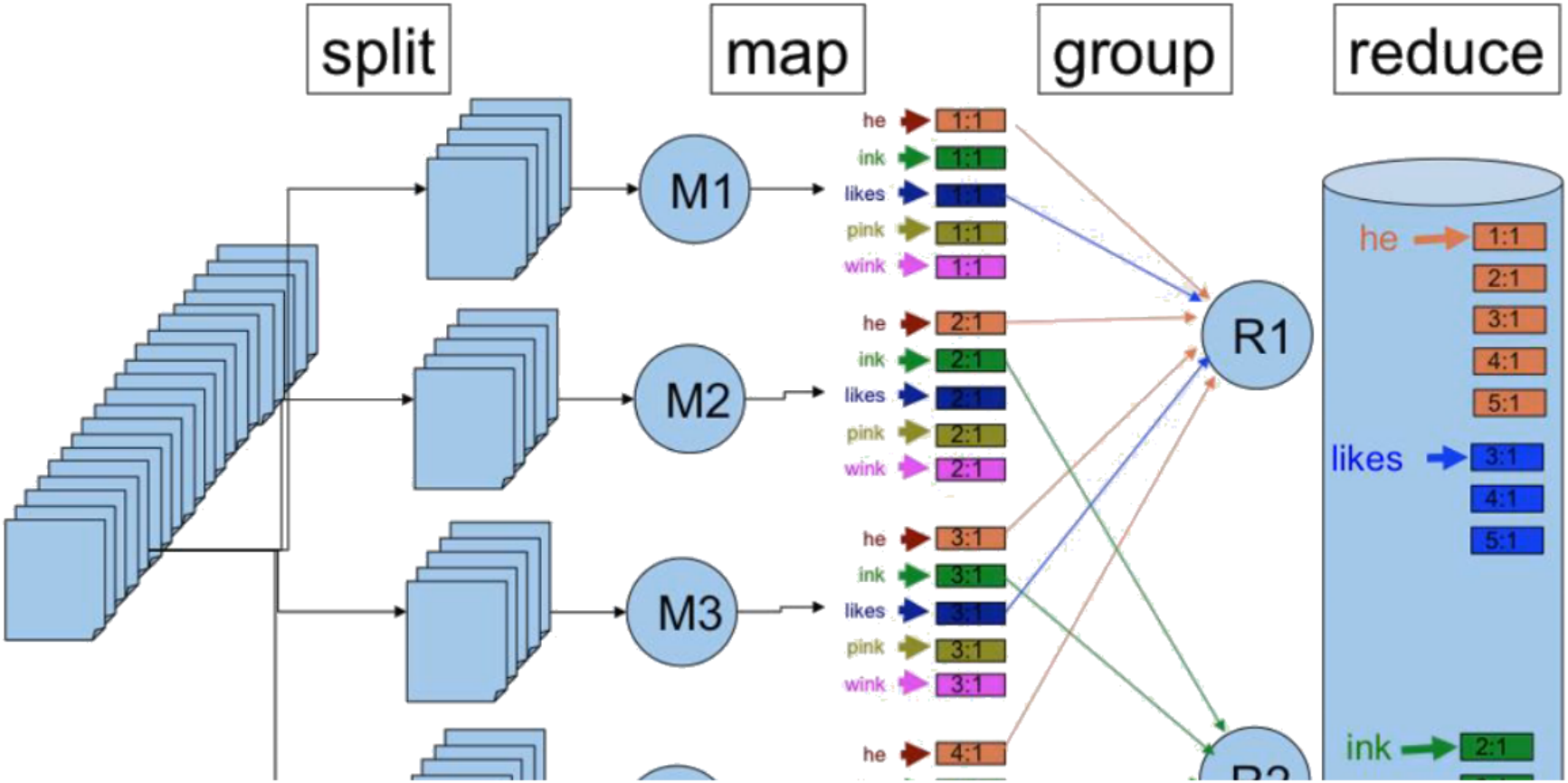
- Beispiel: bestimme Worthäufigkeit in Menge von Dokumenten



Parallele Indexerstellung mit Map Reduce

- Split in M Mengen von Dokumenten und leite an Mapper(rechner) weiter
- Mapper:
  - verarbeite zugewiesene Menge M_i und erstelle Key-Value Paare
    - *Key = Wort*
    - *Value = Dokument, Position, Häufigkeit,..*
- Shuffle/Group: Bestimme hash(key) module R
  - R = # Reducer (rechner)
  - leite an Reducer (rechner) weiter
    - *bestimmt durch Hash-Wert*
- Reducer:
  - hält vollständigen Index über ein oder wenige Worte
- Später: Suche nach Wort:
  - kann gleich auf derselben (verteilten) Struktur belassen/angewendet werden

… Parallele Indexerstellung mit Map Reduce



Parallel Streams in Java

https://www.baeldung.com/java-when-to-use-parallel-stream
https://www.geeksforgeeks.org/parallel-vs-sequential-stream-in-java/
http://www.angelikalanger.com/Articles/EffectiveJava/81.Java8.Parallel-Streams/81.Java8.Parallel-Streams.html

Streams Parallelisierung

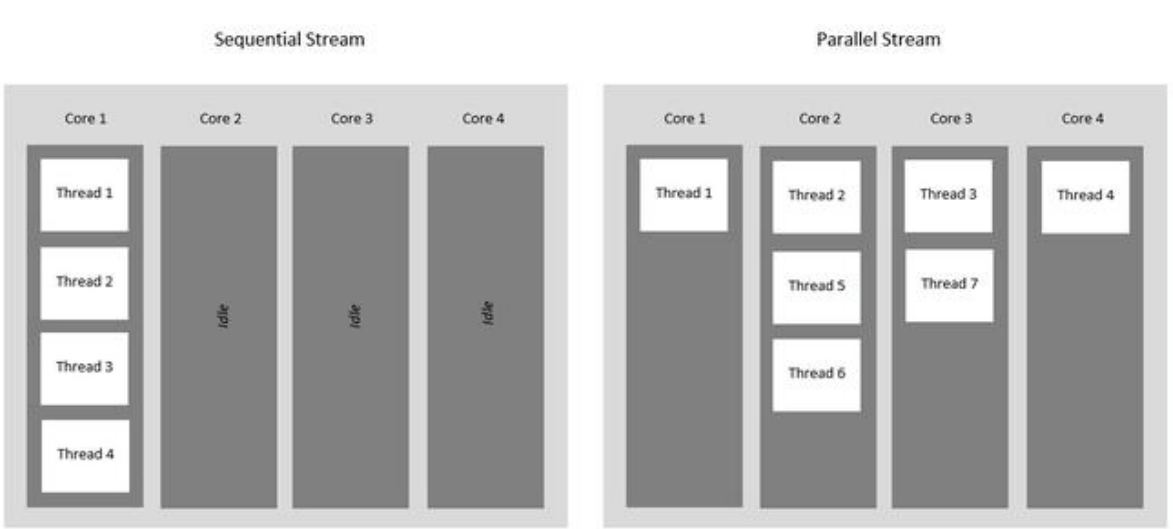
Bestimme grösste Zahl in Stream

- Sequentieller Stream

```
int max = Arrays.stream(ints).reduce(Integer.MIN_VALUE, Math::max);
```

- Paralleler Stream

```
int max = Arrays.stream(ints).parallel().reduce(Integer.MIN_VALUE, Math::max);
```

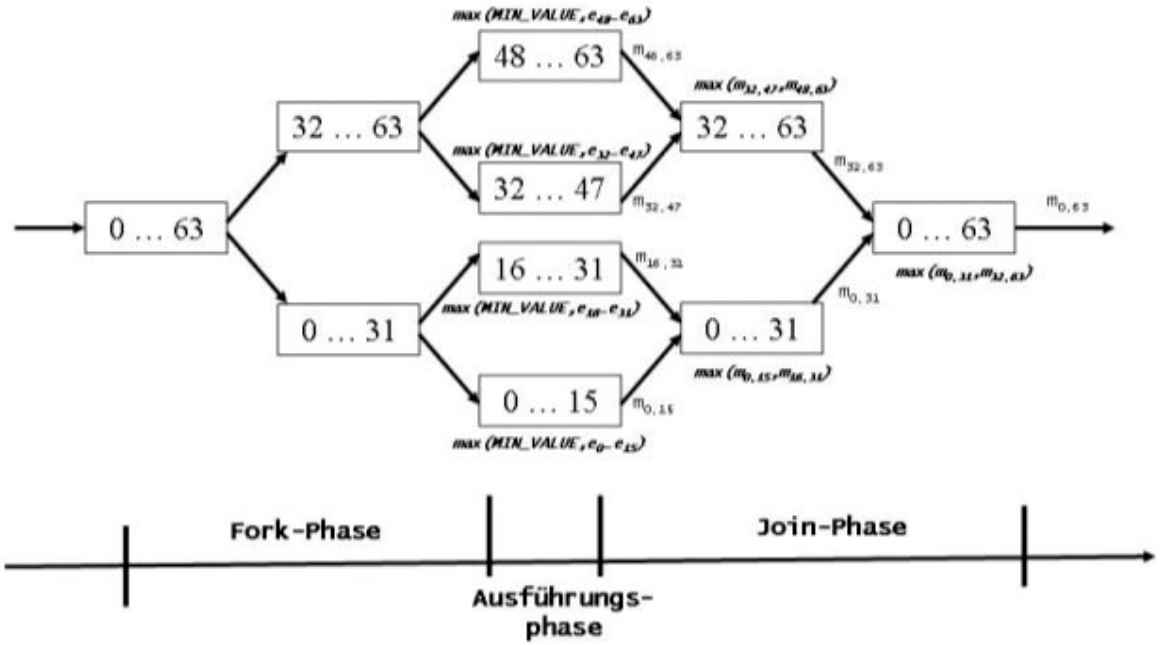


© geeks

Wie parallele Streams funktionieren

- Grundidee gleich wie Map Reduce à la Google
  - Zerlegung des Problems in Teilprobleme: Fork Phase (e.g. Map)
  - Lösen der Teilprobleme -> Teillösungen: Ausführungsphase (parallel)
  - Teillösungen zu Gesamtlösung vereinigen: Join Phase (e.g. Reduce)

- Bsp: Suche nach der grössten Zahl in einem Array [0..63]



Erzeuge parallele Streams

Folgende Varianten um Streams zu erzeugen

- Direkte Angabe der Werte

```
Stream<Integer> stream = Stream.of(1,2,3,4,5,6,7,8,9).parallel();
```

- Aus Array

```
Stream<Integer> stream = Stream.of(new Integer[]{1,2,3,4} ).parallel();
Arrays.stream(new Integer[]{1,2,9}.parallel();
```

- Aus Liste

```
Stream<Integer> stream = list.parallelStream();
```

- Generator Funktion: erzeugt Elemente mittels **Supplier**

```
Stream<Double> stream = Stream.generate(() -> Math.random()).parallel();
```

- Aus File

java.nio.file.*

```
Stream<String> stream = Files.lines(Paths.get(fileName)).parallel();
```

Verarbeite Streams (intermediate Operationen)

- Grundsätzlich **gleich wie für sequentielle Streams**
- Intermediäre Operationen werden in der Ausführungsphase abgearbeitet
  - Sie werden **parallel** ausgeführt
- Jedes parallele Teilproblem wird auf seinem Indexbereich so abgearbeitet wie ein sequentieller Stream auf dem gesamten Indexbereich.
- Bsp: suche den längstem String in Array

```
int max = Arrays.stream(strings).parallel ()
   .mapToInt(String::length)
   .reduce(Integer.MIN_VALUE, Math::max);
```

- Vorsicht: skip und limit wirken auf parallele Teilstreams

Konsumiere Strom - ungeordnet

- Die Verarbeitungreihenfolge der Elemente im parallelem Stream geht verloren
- Ungeordneter Consumer

```
public static void logStream(Stream<Integer> str) {
   str.forEach(System.out::println);
}
```

8 1 6 2 7 4 5 3

Konsumiere Strom - geordnet

- Falls Ordnung erhalten bleiben soll:

testet ob paralleler Stream

```
public static void logStream(Stream<Integer> str) {
  if (!str.isParallel())
     str.forEach(System.out::println);
  else
     str.forEachOrdered(System.out::println);
}
```

behält die ursprüngliche Reihenfolge

Konsumiere Strom - geordnet: Collect

- Collector erhält Reihenfolge der Quelle
  - unabhängig davon in welcher Reihenfolge die Threads ausgeführt werden

```
List<Integer> l = listOfIntegers
  .stream()
  .parallel()
  .collect(Collectors.toList());
```

[1, 2, 3, 4, 5, 6, 7, 8]

https://stackoverflow.com/questions/29709140/why-parallel-stream-get-collected-sequentially-in-java-8

Konsumiere Strom - ungeordnet: Reduce

- Reduce parallel, ungeordnet ausgeführt -> Fall OK

```
List<Integer> listOfNumbers = Arrays.asList(1, 2, 3, 4);
int sum = listOfNumbers.parallelStream().reduce(0, Integer::sum);
```

[10]

- Vorsicht bei Startwert != 0 - sequentiell

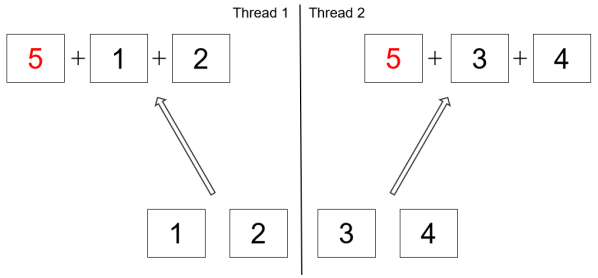
```
List<Integer> listOfNumbers = Arrays.asList(1, 2, 3, 4);
int sum = listOfNumbers.stream().reduce(5, Integer::sum);
```

[15]

- Bei Startwert != 0 - parallel

```
List<Integer> listOfNumbers = Arrays.asList(1, 2, 3, 4);
int sum = listOfNumbers.parallelStream().reduce(5, Integer::sum);
```

[!=15]



Performance

- Parallelisierung kann Performance steigern

  aber:

- Overhead durch Parallelisierung kann auch zu Verschlechterung der Performance führen

- Memory Lokalität kann wichtiger sein als parallele Ausführung

- Parallele Streams i.d.R. vorteilhaft wenn
  - Viele Daten
  - Viele Operationen pro Datenelement

Unscharfe Suche

Unscharfe/Fuzzy Suche

- Nicht ein exakter Übereinstimmung sondern ungefähre gewünscht
- Ansätze:
  - Suche anhand nötiger Buchstabenänderungen -> **Levenshtein**
    - *Vorteil: theoretisch gut verstanden, Analogie/Verwandtschaft zu Hemming Distanz*
  - Suche anhand maximaler Anzahl gleicher Wortgruppen -> **Trigram**
    - *Vorteil: einfach, effizient und auch robust gegen Wortvertauschungen*
  - Suche ähnlich klingender Wörter -> **Phonetische Suche**
    - *Vorteil: "natürliche" Ähnlichkeit,*
  - Suche anhand einer Grammatik -> **RegEx**
    - *Vorteil: deterministisch, Muster durch eine Grammatik definiert*
  - Suche anhand trainiertem **ANN** (Neuronales Netzwerk, hier nicht behandelt)
    - *Vorteil: ANN "lernt" selber Übereinstimmungskriterien*

In der Praxis oft eine Kombination von Algorithmen

http://radar.zhaw.ch/adressbook/_Adressbook.aspx

Levenshtein Distanz

Levenshtein Distanz

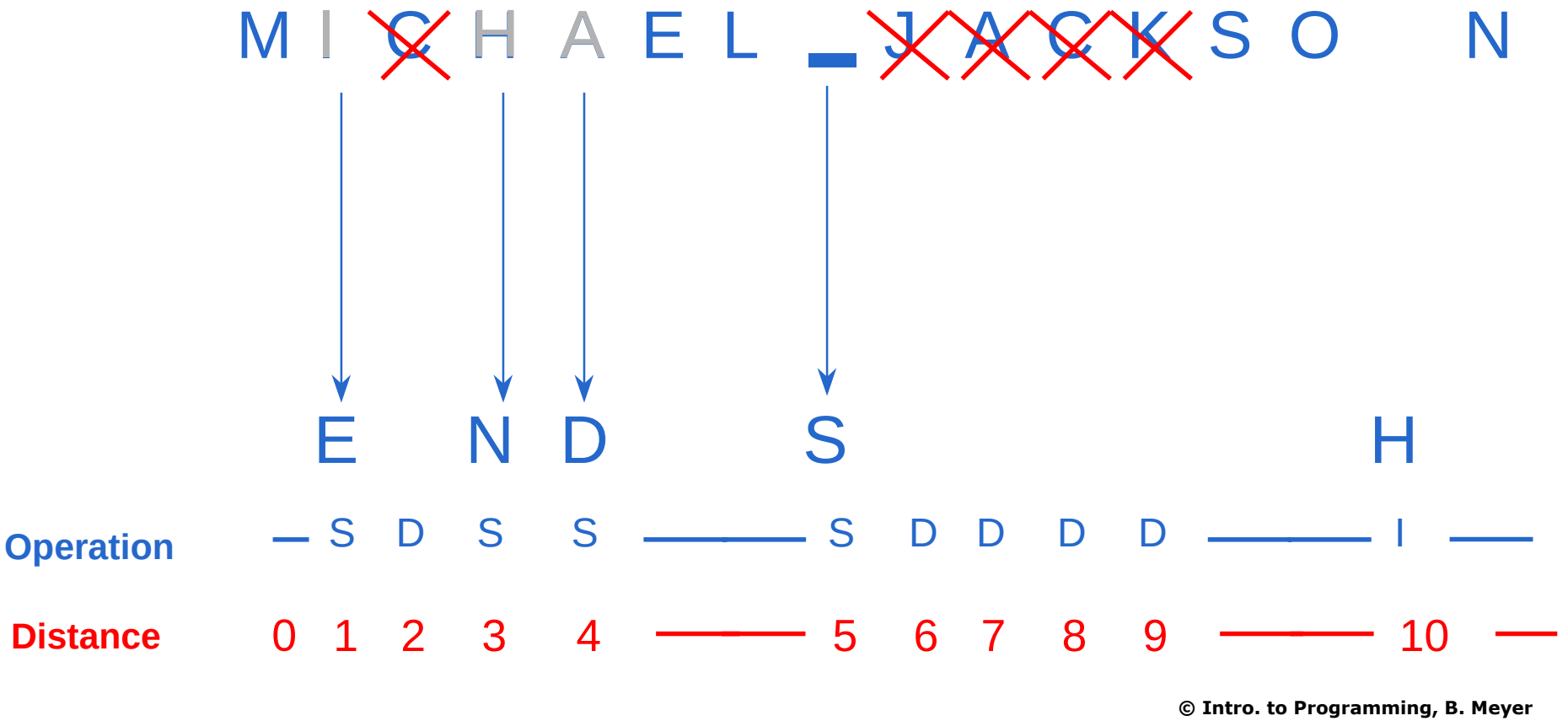
- Auch als "Edititierdistanz" bezeichnet
- Wieviele Editieroperationen sind minimal nötig um den einen String in den andern überzuführen
  - **Ersetzen**
  - **Einfügen**
  - **Löschung**

$$
\begin{aligned}
&m = |u| \\
&n = |v| \\
&D_{0,0} = 0 \\
&D_{i,0} = i, 1 \leq i \leq m \\
&D_{0,j} = j, 1 \leq j \leq n \\
&D_{i,j} = \min \begin{cases} D_{i-1,j-1} & +0 \text{ falls } u_i = v_j \\ D_{i-1,j-1} & +1 \text{ (Ersetzung)} \\ D_{i,j-1} & +1 \text{ (Einfügung)} \\ D_{i-1,j} & +1 \text{ (Löschung)} \end{cases} \\
&\quad 1 \leq i \leq m, 1 \leq j \leq n
\end{aligned}
$$

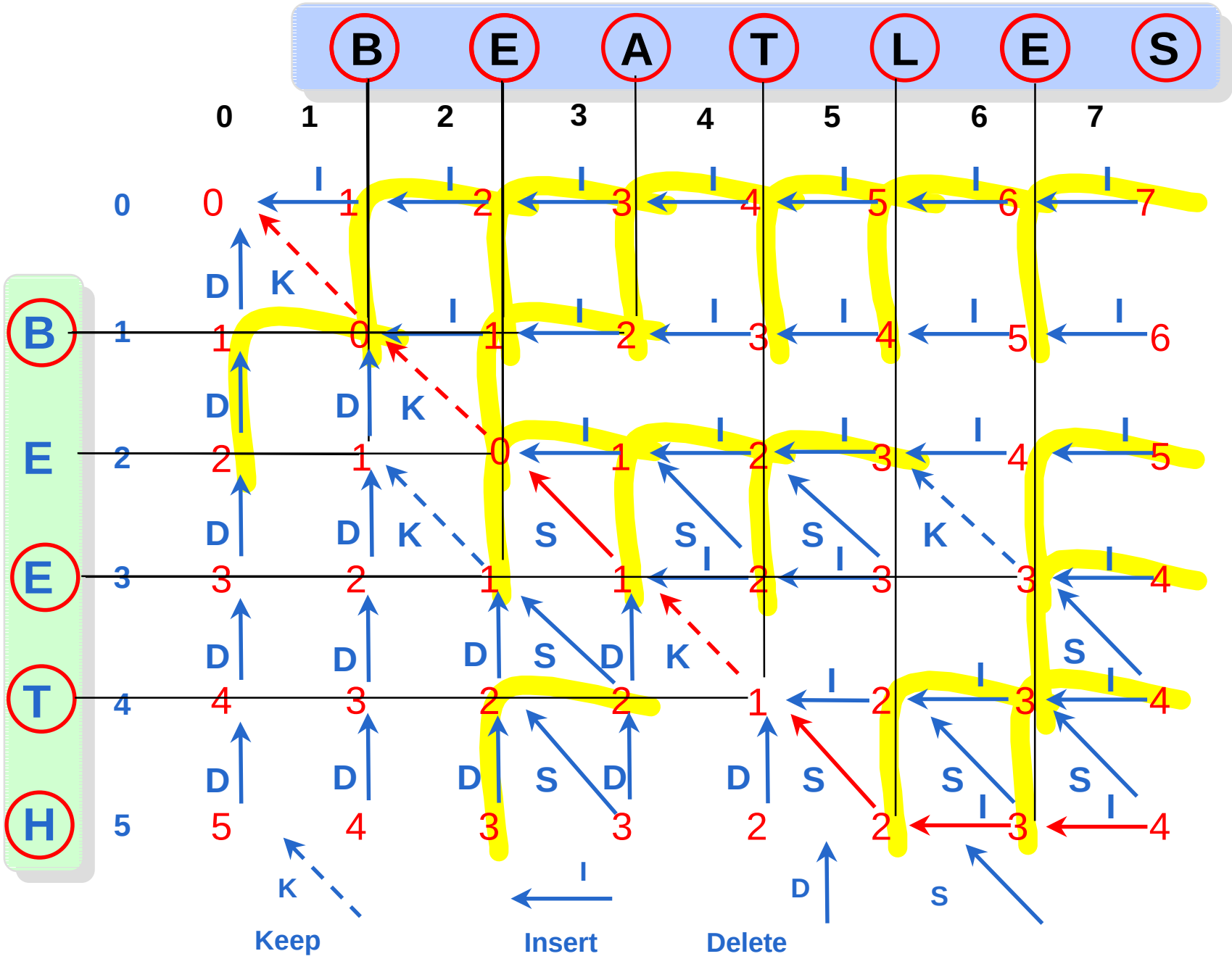
© Wikipedia

Beispiel

"Michael Jackson" **to** "Mendelssohn"



© Intro. to Programming, B. Meyer


B E A T L E S
0 1 2 3 4 5 6 7
B
E
E
T
H
Keep
Insert
Delete
Substitute

Java Implementation

- Ein 2-dim Array "distance" um die minimale Distanz zwischenzuspeichern

```
public class LevenshteinDistance {
    private static int minimum(int a, int b, int c) {
      return Math.min(Math.min(a, b), c);
    }

    public static int computeLevenshteinDistance(String str1,String str2) {
      int[][] distance = new int[str1.length() + 1][str2.length() + 1];

      for (int i = 0; i <= str1.length(); i++) distance[i][0] = i;
      for (int j = 1; j <= str2.length(); j++) distance[0][j] = j;

      for (int i = 1; i <= str1.length(); i++)
         for (int j = 1; j <= str2.length(); j++) {
            int minEd = (str1.charAt(i - 1) == str2.charAt(j - 1)) ? 0 : 1;
            distance[i][j] = minimum(
                   distance[i - 1][j] + 1,
                   distance[i][j - 1] + 1,
                   distance[i - 1][j - 1]+ minEd);
         }
      return distance[str1.length()][str2.length()];
    }
}
```

Min von 3 Werten

all insert

all delete

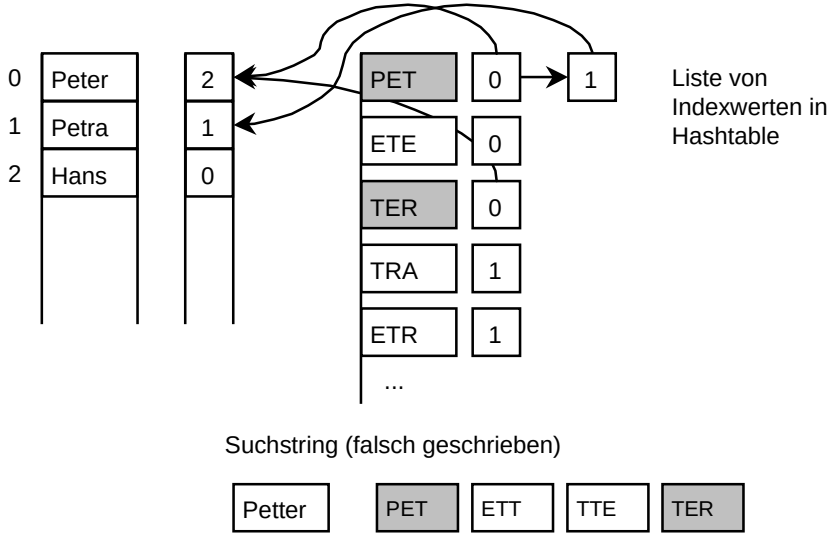
equal

substitute

Trigram Suche

Trigram Suche

- Fehlertolerante Suche (auch für Wortverdreher, z.B Vor- Nachname)
- Effizient für grosse Datenbestände
- Index -> Wort in 3-Buchstaben Gruppen unterteilt ,
  - z.B. sind das bei „Peter“, die 3-er Gruppen „PET“, “ETE“, “TER“.
  - Diese 3-er Gruppen werden für alle vorkommenden Worte gebildet und in Hashtabelle gespeichert
- Das zu suchende Wort wird ebenfalls in 3-er Gruppen zerlegt
  - das gesuchte Wort mit am meisten Übereinstimmungen wird genommen



Phonetische Suche

Phonetische Suche

- z.B. Soundex Phrasen nach ihrem Klang in englischer/deutscher Sprache.
- Wort besteht aus seinem ersten Buchstaben, gefolgt von drei Ziffern
  - Kurze Worte: mit 0 auffüllen; 0 werden für den Vergleich ignoriert
  - Die Vokale A, E, I, O und U und die Konsonanten H, W und Y sind ausser beim ersten Zeichen zu ignorieren (in D auch ä,ö,ü).
  - Ziffern sind Konsonanten nach folgender Tabelle

E

Ziffer	Repräsentierte Buchstaben
1	B, F, P, V
2	C, G, J, K, Q, S, X, Z
3	D, T
4	L
5	M, N
6	R

D

Ziffer	Repräsentierte Buchstaben
0	a, e, i, o, u, ä, ö, ü, y, j, H
1	b, p, f, v, w
2	c, g, k, q, x, s, z, ß
3	d, t
4	l
5	m, n
6	r
7	ch

- Beispiele

Britney → BRTN → **B635**	bewährten → BRTN → **B635**
Spears → SPRS → **S162**	Superzicke → SPRZCK → S16222 → **S162**

Suchen nach Mustern

Reguläre Ausdrücke: Regex

- Zum Suchen von **definierten Mustern** in Texten d.h. Strings
  - ein bestimmter String: "ZHAW"
  - Muster kann "unscharf" definiert sein: z.B. IT13a, IT12a, IT12c
  - (Teil-)Muster kann sich wiederholen: 170.12.34.12
- Die meisten heutigen Programmiersprachen unterstützen die Suche nach Muster in Form von reguläre Ausdrücke (Regular Expressions oder kurz Regex)
- Regex ist unabhängig von Java definiert
- Java Klassenbibliothek definiert im Package java.util.regex
- die Klassen **Pattern** und **Matcher**

Definition des regulären Ausdrucks

School of Engineering

- Zuerst muss der reguläre Ausdruck vorbereitet werden
- Die Klasse **Pattern**
  - `Pattern pat = Pattern.compile("ZHAW");`
- Das Muster kann im einfachsten Fall ein Textzeichen-String sein
- Alle Zeichen sind erlaubt ausser: `([{\^-$|]})?*+.`
- Diese müssen mit \ vorangestellt geschrieben werden
  - Vorsicht in Java String Konstante muss "\\" für "\" geschrieben werden
- Beispiel
  - `Pattern pat = Pattern.compile("wie geht's \\?");`

Abfrage nach den gefundenen Stellen

- Ausgabe der gefundenen Stellen
- Die Klasse **`Matcher`**
  - `Matcher matcher = pat.matcher("Willkommen an der ZHAW");`
- Suche nächste Textstelle **`boolean find()`**
  - true falls gefunden
  - `matcher.find();`
- Gebe gefunden Teilstring zurück **`String group()`**
  - `matcher.group(); // ZHAW`
- gefundene Start und Endposition innerhalb String **`int start()`** und **`int end()`**
  - `matcher.start(); //18`
  - `matcher.end();  //22`

Vollständiges Beispiel

```
import java.util.regex.*;


…
Pattern pat = Pattern.compile("ZHAW");
Matcher matcher = pat.matcher("Willkommen an der ZHAW");
while (matcher.find()) {
   String group = matcher.group();
   int start = matcher.start();
   int end = matcher.end();
   // do something
}
```

Platzhalter

- Oftmals wird nach unscharfen Muster gesucht, z.B. alle IT Klassen
- Es sind Platzhalter Zeichen erlaubt, die *Zeichenmengen* matchen
  - z.B. "." für beliebiges Zeichen "\d" für Zahl, ^\d für *keine* Zahl

Platzhalter	Beispiel	Bedeutung	Menge der gültigen Literale
`.`	`a.b`	Ein beliebiges Zeichen	aab, acb, aZb, a[b, ...
`\d`	`\d\d`	Digit[0-9]	78, 10
`\D`	`\D`	kein Digit	a, b , c
`^`	`^\d`	Negation	a, b , c
`\s`	`\s`	Leerzeichen (Blank,etc)	blank, tab, cr,
`\S`	`\S`	kein Leerzeichen	

- Aufgabe: Geben Sie das Suchmuster für beliebige IT Klassen an: IT10a, IT08b, IT09c

Eigene Zeichenmengen

- Statt vordefinierte Zeichenmengen zu verwenden, können auch eigene definiert werden
- diese werden in "[" "]" geklammert

- Aufzählung der Zeichen in der Zeichenmenge
  - ein Zeichen aus der Menge
  - z.B. "a", "b" oder "c" : [abc]
- Bereiche
  - z.B. alle Kleinbuchstaben [a-z] alle Buchstaben [a-zA-Z]
- Negation: Alle Zeichen ausser
  - z.B. [^a]
- Aufgabe: Geben Sie das Suchmuster für beliebige IT Klassen an; es gäbe aber nur "a" bis "d"

Optional, Alternative und Wiederholung

- Optionale Teile: ?
  - wenn einzelner Buchstaben optional, z.B. ZHA?W -> ZHW oder ZHAW
- Alternative |
  - wenn ein A oder B -> ZH(A|B)W -> ZHAW oder ZHBW
  - "natürliche" Verwendung von Klammern

Wiederholungen

Auch 0 mal erlaubt

- Beliebig oft *
  - eine Folge von Ziffern \d* -> _, 2,23,323,423,..
- Mindestens einmal +
  - eine Folge von Ziffern aber mindest eine \d+ -> 3,34,234,…
- Bestimmte Anzahl mal {n}
  - eine Folge von drei Ziffern \d{3} -> 341,241,123 ..
- Mindestens, maximal Anzahl mal {n,m}
  - eine Folge von 1 bis 3 Ziffern \d{1,3} -> 1, 23, 124, ...

Zusammenfassung Metasymbole

- (Meta-)Sprache zur Beschreibung der Bildungsregeln von Sätzen
- Metasymbole: `([{\^-$|]})?*+.`

Metasymbol	Beispiel	Bedeutung	Menge der gültigen Literale
*	ax*b	0 oder mehrere x	ab, axb, axxb, axxxb, ...
+	ax+b	1 oder mehrere x	axb, axxb, axxxb, ...
?	ax?b	x optional	ab, axb
|	a|b	a oder b	a, b
()	x(a|b)x	Gruppierung	xax, xbx
.	a.b	Ein beliebiges Zeichen	aab, acb, aZb, a[b, ...
[]	[abc]x	1 Zeichen aus einer Menge	ax, bx, cx
[-]	[a-h]	Zeichenbereich	a,b,c, ..., h
\d	\d\d	Digit[0-9]	78, 10
\D	\D	kein Digit	a, b , c
^	^\d	Negation	a, b , c
\s	\s	Leerzeichen (Blank,etc)	blank, tab, cr,
\S	\S	kein Leerzeichen	

Übungsbeispiele

Regulärer Ausdruck	Gültigen Sätze
a?b+	
	ein, eine, einer
a(x|y)?b*	
	Binär-Zahlen
	Ganze Zahl

Weitere Methoden für Regex

- **Prüfe ob ganzer String einem Regex Muster entspricht**
- **`boolean matches(String regexp);`**
  - ```
    String text = "Hallo Welt";
    boolean passt;
    passt = text.matches("H.*W.*");
    passt = text.matches("H..o Wel?t");
    passt = text.matches("H[alo]* W[elt]+
    passt = text.matches("Hal+o Welt.+");
    ```
- **Aufgabe: Regex zum Prüfen ob ein String eine Int Zahl enthält**
- **Aufgabe: Regex zum Prüfen ob ein String eine IP Adresse enthält**
    ```

… Weitere Methoden für Regex

- **`String replaceAll(String regexp, String replaceStr);`**
- **`String replaceFirst(String regexp, String replaceStr);`**
  - Ersetzt im gegebenen String alle (bzw. den Ersten bei **`replaceFirst`**) Substrings, die **`regexp`** entsprechen, mit **`replaceStr`**
  - **`String new = text.replaceAll("l+", "LL");   // HaLLo WeLLt`**

- **`String[] split(String regexp);`**
  - Teilt den gegebenen String in mehrere Strings, **`regexp`** ist die Grenzmarke
  - Das Resultat ist ein Array mit Teilstrings
  - **`String data = "4, 5, 6 2,8,, 100, 18"`**
  - **`String[] teile = data.split("[ ,]+");    // Menge der Zeichen " " und","`**
    `// 4 5 6 2 8 100 18`
    `// teile[0] = "4", teile[1] = "5", ….`

Zusammenfassung

- Suche von Strings in Strings
- Suchmaschinen und Index
- Unscharfe Suche
- Suchen nach Mustern

Reguläre Ausdrücke – Lösung

Regulärer Ausdruck	Menge der gültigen Literale
`a?b+`	b, bb, bbb..., ab, abb, abbb...
eine?r?	`{ein, eine, einer}`
`a(x|y)?b*`	a, ax, ay, ab, axb, ayb, abb, axbb...
[_$a-zA-Z][_$a-zA-Z0-9]*	`Java-Bezeichner`
(0|-?[1-9][0-9]*)	`alle ganze Zahlen`