

Chapter 2

Text Processing and Analysis

Sergej Sizov

Information Retrieval

Summer term 2008

Outline

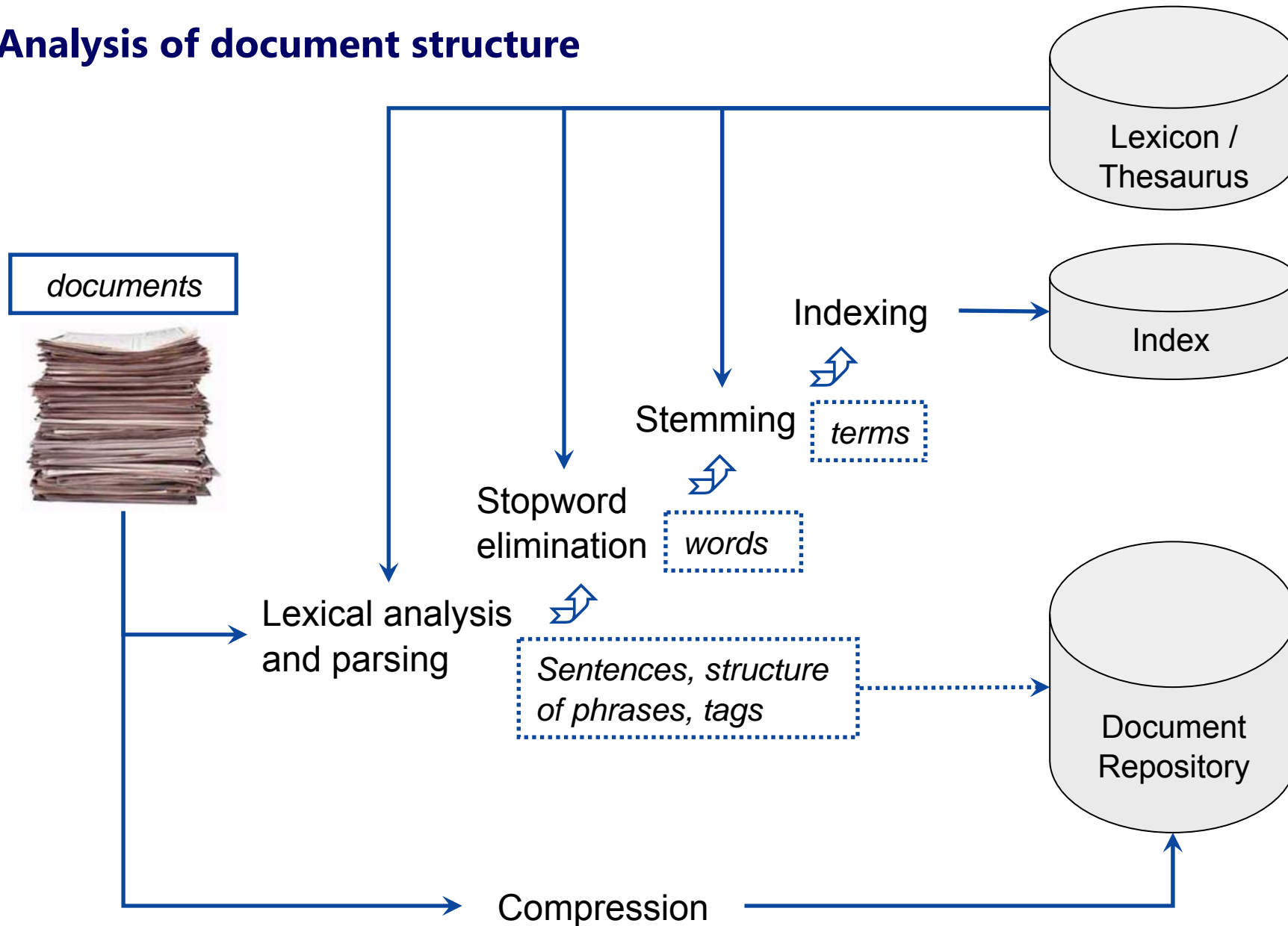
2.1 Text Processing

2.2 Vector Space Model

2.3 Text Indexing and Query Execution

2.1 Text Processing

Analysis of document structure



Stopword Elimination

Lookups in stopword lists

(potentially using domain-specific dictionary - lexikon/thesaurus)

e.g. „definition“ or „theorem“ for math documents

Common language-specific stopwords (prepositions, conjunctions, pronouns, „overloaded“ verbs etc. – several hundreds of stopwords):

a, also, an, and, as, at, be, but, by,
can, could, do, for, from, go,
have, he, her, here, his, how,
I, if, in, into, it, its,
my, of, on, or, our, say, she,
that, the, their, there, therefore, they,
this, these, those, through, to, until,
we, what, when, where, which, while, who, with, would,
you, your,

Morphologic Reduction (Lemmatization)

- ◆ Grammatical base form:

Nominative for nouns, infinitive for verbs, plural to singular, passive to active, etc.

Examples:

- „students“ to „student“, „going“ to „go“

Kontext dependent and phrase dependent

- „went“ to „go“,
- „have been“ to „be“

- ◆ Linguistical base form

Tracking of flexion (e.g. declination), composition, substantization, etc.

Examples:

- „nonfood“ to „food“
- „founds“ to „find“
- „Schweinkram“, „Schweinshaxe“ und „Schweinebraten“ to „Schwein“ etc.

Stemming

Ideas:

- use of dictionaries
- recognition through analysis of the linguistical strukture
- affix elimination: removal of prefixes and suffixes using (heuristic) rules

Example:

stresses → stress, stressing → stress, symbols → symbol

using rules sses → ss, ing → e, s → e, etc.

Note: the usefulness of stemming in IR is not undisputable

Example:

Bill is operating a company.

On his computer he runs the ... operating system.

Thesaurus

For each **concept** (word sense) we store:

- the set of synonyms or instances (*words*)
- the set of generalizations and specializations (hypernyms, hyponyms)
- „part-of“ and „contains“ relationships (meronyms, holonyms)
- concept-example relationships (e.g. fairytale and cinderella)
- the set of antonyms

For each **word** we store:

- the set of associated *concepts* (e.g. with some statistics)
- (for disambiguation of polysems or homonyms)

Example: WordNet, <http://www.cogsci.princeton.edu/~wn>

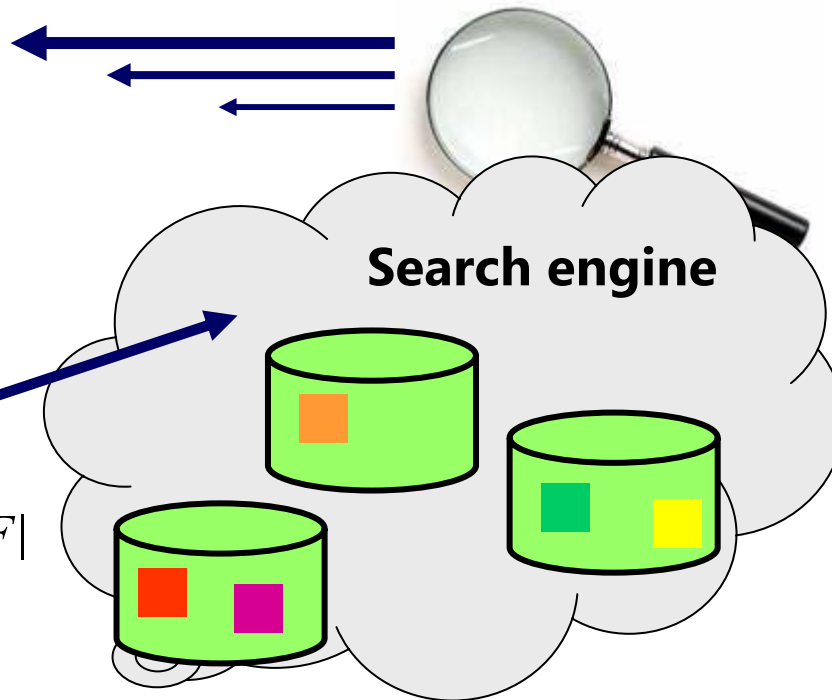
2.2 Vector Space Model

Basic Principles:

- Feature Space: words in documents are reduced to terms.
- Document model: each document is represented as vector $\in [0,1]^{|\mathcal{F}|}$ whereby d_{ij} is the weight of the j -th term in d_i .
- Queries: queries are vectors $q_i \in [0,1]^{|\mathcal{F}|}$
- Relevance: relevance of results is based on similarity function for vector space $[0,1]^{|\mathcal{F}|}$
- Indexing: for each term there is a list of Doc-IDs (e.g. URLs) with associated weights, implemented as „inverted file“ (search tree or hash table)
- Query execution: query is decomposed into several index-lookups for particular query terms in order to determine the ranked list of candidates

Vector Space Model Relevance Ranking

Ranking by descending relevance



Similarity metric:

$$\text{sim}(d_i, q) := \frac{\sum_{j=1}^{|F|} d_{ij} q_j}{\sqrt{\sum_{j=1}^{|F|} d_{ij}^2 \sum_{j=1}^{|F|} q_j^2}}$$

Query $q \in [0,1]^{|F|}$
(Set of weighted features)

Documents are **feature vectors** $d_i \in [0,1]^{|F|}$

e.g., using:

$$d_{ij} := w_{ij} / \sqrt{\sum_k w_{ik}^2}$$

$$w_{ij} := \frac{\text{freq}(f_j, d_i)}{\max_k \text{freq}(f_k, d_i)} \log \frac{\# \text{docs}}{\# \text{docs with } f_i}$$

tf*idf formula

Term Weighting

We consider following characteristics for N documents and M terms:

- ◆ tf_{ij} : term frequency - frequency of term t_i in document d_j
- ◆ df_i : document frequency - number of documents that contain t_i
- ◆ idf_i : inverse document frequency = N / df_i
- ◆ cf_i : corpus frequency – frequency of t_i in the corpus (e.g. separate counting of title terms, body terms, etc.)

Basic idea:

- ◆ The weight w_{ij} of term t_i in document d_j should increase monotonically with tf_{ij} and idf_i

First idea:

- ◆ use some tf-idf combination, e.g. $w_{ij} = f_{ij} * idf_i$ (**tf-idf formula**)
- ◆ w_{ij} can be normalized:
$$d_{ij} = \frac{w_{ij}}{\sqrt{\sum_k w_{kj}^2}}$$

Variations of Term Weighting

Empirical results show that *tf* and *idf* values usually must be dampened or normalized

- ◆ normalized *tf* values

$$tf_{ij} = \frac{tf_{ij}}{\max_k tf_{kj}}$$

- ◆ *tf* weighting mit dampening

$$tf_{ij} = 1 + \log tf_{ij}$$

- ◆ *idf* weighting mit dampening

$$idf_i = \log \frac{N}{df_i}$$

- ◆ common combination:
(*tf***idf* formula)

$$w_{ij} = \frac{tf_{ij}}{\max_k tf_{kj}} \log \frac{N}{df_i}$$

$$d_{ij} = \frac{w_{ij}}{\sqrt{\sum_k w_{kj}^2}}$$

Term Weighting in Queries

Depending of query interface and user category, simple or advanced term weightings may be used

- ◆ simple weighting: $w_{ij} \in \{0, 1\}$
- ◆ advanced weighting: $w_{ij} = \left(0.5 + \frac{0.5 \cdot tf_{ij}}{\max_k tf_{ij}}\right) \cdot \log \frac{N}{df_i}$
- ◆ term ranking: $w_{ij} = \frac{1}{k}$
(when conjunctive query q contains k terms and t_i is in k^{th} position)

Digression: Principles of Probabilistic Retrieval

[Robertson and Sparck Jones 1976]

Goal:

Ranking based on $\text{sim}(\text{doc } d, \text{ query } q) =$
 $P[R|d] = P [\text{ doc } d \text{ is relevant for query } q \mid d \text{ has term vector } X_1, \dots, X_m]$

Assumptions:

- Document relevance does not depend of other documents
- Relevant and irrelevant documents differ in their terms.
- **Binary Independence Retrieval (BIR) Model:**
 - Probabilities for term occurrence are **pairwise independent** for different terms.
 - **Term weights are binary:** $X_i \in \{0,1\}$.
- For terms that do not occur in query q the probabilities for such a term occurring are the same for relevant and irrelevant documents.

Probabilistic Retrieval with Term Independence

Ranking Proportional to Relevance Odds

$$\text{sim}(d, q) = O(R | d) = \frac{P[R | d]}{P[\neg R | d]}$$

odds for relevance
(ratio of relevant documents)

$$= \frac{P[d | R] \times P[R]}{P[d | \neg R] \times P[\neg R]}$$

Bayes' theorem

$$\sim \frac{P[d | R]}{P[d | \neg R]} = \prod_i \frac{P[X_i | R]}{P[X_i | \neg R]}$$

independence or
linked dependence

$$\text{sim}(d, q)' = \log \prod_{i \in q} \frac{P[X_i | R]}{P[X_i | \neg R]}$$

$X_i = 1$ if d includes
 i -th term, 0 otherwise

$$= \sum_{i \in q} \log P[X_i | R] - \log P[X_i | \neg R]$$

Probabilistic Retrieval (2)

$$\text{sim}(d, q)' = \log \prod_{i \in q} \frac{P[X_i | R]}{P[X_i | \neg R]} = \sum_{i \in q} \log P[X_i | R] - \log P[X_i | \neg R]$$

$$= \sum_{i \in q} \log (p_i^{X_i} (1 - p_i)^{1 - X_i}) - \log (q_i^{X_i} (1 - q_i)^{1 - X_i}) \quad (\text{binary features})$$

with estimators $p_i = P[X_i = 1 | R]$ and $q_i = P[X_i = 1 | \neg R]$

$$= \sum_{i \in q} \log \left(\frac{p_i^{X_i} (1 - p_i)}{(1 - p_i)^{X_i}} \right) - \log \left(\frac{q_i^{X_i} (1 - q_i)}{(1 - q_i)^{X_i}} \right)$$

$$= \sum_{i \in q} X_i \log \frac{p_i}{1 - p_i} + \sum_{i \in q} X_i \log \frac{1 - q_i}{q_i} + \sum_{i \in q} \log \frac{1 - p_i}{1 - q_i}$$

$$\sim \sum_{i \in q} X_i \log \frac{p_i}{1 - p_i} + \sum_{i \in q} X_i \log \frac{1 - q_i}{q_i} = \text{sim}(d, q)''$$

Probabilistic Retrieval (3)

Robertson / Sparck Jones Formula

Estimate p_i and q_i based on training sample (query q on small sample of corpus) or based on intellectual assessment of first round's result (*relevance feedback*):

Let N be #docs in sample,
 R be # relevant docs in sample
 n_i #docs in sample that contain term i ,
 r_i # relevant docs in sample that contain term i

$$\Rightarrow \text{Estimate: } p_i = \frac{r_i}{R} \quad q_i = \frac{n_i - r_i}{N - R}$$

or: $p_i = \frac{r_i + 0.5}{R + 1} \quad q_i = \frac{n_i - r_i + 0.5}{N - R + 1}$ (Lidstone smoothing with $\lambda=0.5$)

$$\Rightarrow \text{sim}(d, q)'' = \sum_i X_i \log \frac{r_i + 0.5}{R - r_i + 0.5} + \sum_i X_i \log \frac{N - n_i - R + r_i + 0.5}{n_i - r_i + 0.5}$$

$$\Rightarrow \text{Weight of term } i \text{ in doc } d: \log \frac{(r_i + 0.5) (N - n_i - R + r_i + 0.5)}{(R - r_i + 0.5) (n_i - r_i + 0.5)}$$

Probabilistic Retrieval: tf*idf Formula

Assumptions (without training sample or relevance feedback):

- p_i is the same for all i .
- Most documents are irrelevant.
- Each individual term i is infrequent.

This implies:

- $\sum_i X_i \log \frac{p_i}{1-p_i} = c \sum_i X_i$ with constant c
- $q_i = P[X_i = 1 | \neg R] \approx \frac{df_i}{N}$
- $\frac{1-q_i}{q_i} = \frac{N-df_i}{df_i} \approx \frac{N}{df_i}$

$$\begin{aligned} \Rightarrow \text{sim}(d, q) &= \sum_i X_i \log \frac{p_i}{1-p_i} + \sum_i X_i \log \frac{1-q_i}{q_i} \\ &\approx c \sum_i X_i + \sum_i X_i \text{idf}_i \end{aligned}$$

Scalar product over the product of tf and dampend idf values for query terms

2.3 String Matching

Basic idea:

Given: Text d : array[1.. n] of char
Pattern p : array[1.. m] of char

Wanted:

all „shifts“ s_1, \dots, s_k ($0 \leq s_i \leq n-m$),
such that $d[s_i+1..s_i+m] = p[1..m]$

Naive solution:

for $s:=0$ to $n-m$ do	worst case complexity
if $p[1..m] = d[s+1..s+m]$ then	(# character comparisons):
print „pattern occurs with shift“ s	$\Theta((n-m+1)m)$

For correctness and complexity analysis see:

T. Cormen, C. Leiserson, R. Rivest: Introduction to Algorithms,
MIT Press, 1990, Chapter 34

Knuth-Morris-Pratt Algorithm (2)

ComputePrefixFunction (p):

```
 $\pi[1] := 0; k := 0;$   
for q:=2 to m do  
  while k>0 and  $\pi[k+1] \neq \pi[q]$  do k:=  $\pi[k]$  od;  
  if  $\pi[k+1] = \pi[q]$  then k:= k+1 fi;  
   $\pi[q]:=k;$   
od; return p
```

KMPsearch (d, p):

```
 $\pi :=$  ComputePrefixFunction (p); q:= 0;  
for i:=1 to n do  
  while q>0 and  $p[q+1] \neq d[i]$  do q:=  $\pi[q]$  od;  
  if  $p[q+1] = d[i]$  then q:= q+1 fi;  
  if q = m then print „pattern occurs with shift“ i-m; q:=  $\pi[q]$  fi;  
od;
```

Worst-case complexity

$\Theta(n+m)$

KMP Algorithm: Example

ComputePrefixFunction ($p[1..7] = a b a b a c a$):

$\pi[1] := 0$

$\pi[2] := 0$

$\pi[3] := 1$

$\pi[4] := 2$

$\pi[5] := 3$

$\pi[6] := 0$

$\pi[7] := 1$

KMPsearch $d[1..15] = b a c b a b a b a a b c b a b$
 $p[1..7] = a b a b a c a$

Boyer-Moore algorithm (2)

BMsearch (d, p):

```
λ[] := ComputeLastOccurrenceFunction (p, m, Σ);      worst-case complexity
σ[] := ComputeGoodSuffixFunction (p, m); s:= 0;      O((n-m+1)m + | Σ | )
while s <= n-m do                                    avg complexity
  j := m;                                           is much better
  while j>0 and p[j] = d[s+j] do j := j-1 od;
  if j = 0 then print „pattern occurs at shift“ s; s := s + σ[0]
    else s := s + max (σ[j], j - λ[d[s+j]] ) fi;
od;
```

ComputeLastOccurrenceFunction (p, m, Σ): $\Sigma \rightarrow \{0, \dots, m\}$

$\lambda[a]$:= Index des rechtesten Vorkommens von Zeichen a in p
bzw. 0, wenn a in p nicht vorkommt

ComputeGoodSuffixFunction (p, m): $\{1, \dots, m\} \rightarrow \{0, \dots, m-1\}$

$\sigma[j]$:= m - max {k: 0 <= k < m and
p[j+1..m] is a suffix of p[1..k] or vice versa}

Boyer-Moore-Horspool algorithm

Simplified, very fast BM modification

Uses only the LastOccurrenceFunction,

applies this function when $d[s+m]=p[m]$ (also) to $p[m]$

unoptimizedBMHsearch (d, p):

```
λ := ComputeLastOccurrenceFunction (p, m, Σ);
```

```
while s ≤ n-m do
```

```
  j := m;
```

```
  while j > 0 and p[j] = d[s+j] do j := j-1 od;
```

```
  if j = 0 then print „pattern occurs at shift“ s fi;
```

```
  t := s + m - λ[p[m]] );
```

```
  if j > 0 then s := max (t, s + j - λ[d[s+j]]) else s := t fi;
```

```
od;
```

Optimized implementations use various coding tricks to avoid repeated access to λ in a loop.

2.4 String Similarity

Hamming-Distanz von Strings $s_1, s_2 \in \Sigma^*$ mit $|s_1|=|s_2|$:

Anzahl verschiedener Zeichen (Kardinalität von $\{i: s_{1i} \neq s_{2i}\}$)

Levenshtein-Distanz (Editierdistanz) von Strings $s_1, s_2 \in \Sigma^*$:

minimale Anzahl der Editieroperationen auf s_1

(Ersetzen, Löschen oder Einfügen eines Zeichens),

um s_1 in s_2 zu ändern

Für

$\text{edit}(i, j)$: Levenshtein-Distanz von $s_1[1..i]$ und $s_2[1..j]$

gilt: $\text{edit}(0, 0) = 0$, $\text{edit}(i, 0) = i$, $\text{edit}(0, j) = j$

$\text{edit}(i, j) = \min \{ \text{edit}(i-1, j) + 1,$

$\text{edit}(i, j-1) + 1,$

$\text{edit}(i-1, j-1) + \text{diff}(i, j) \}$

mit $\text{diff}(i, j) = 1$ falls $s_{1i} \neq s_{2j}$, 0 sonst

→ Berechnung durch dynamische Programmierung

String similarity : Damerau-Levenshtein

Damerau-Levenshtein-Distanz von Strings $s_1, s_2 \in \Sigma^*$:

minimale Anzahl an Ersetzungs-, Einfüge-, Lösch- oder Transpositionsoperationen (Vertauschen benachbarter Zeichen), um s_1 in s_2 zu ändern

Für $\text{edit}(i, j)$: Damerau-Levenshtein-Distanz von $s_1[1..i]$ und $s_2[1..j]$

gilt: $\text{edit}(0, 0) = 0$, $\text{edit}(i, 0) = i$, $\text{edit}(0, j) = j$

$$\text{edit}(i, j) = \min \{ \text{edit}(i-1, j) + 1, \\ \text{edit}(i, j-1) + 1, \\ \text{edit}(i-1, j-1) + \text{diff}(i, j), \\ \text{edit}(i-2, j-2) + \text{diff}(i-1, j) + \text{diff}(i, j-1) + 1 \}$$

mit $\text{diff}(i, j) = 1$ falls $s_1[i] \neq s_2[j]$, 0 sonst

String Similarity: N-Grams

Bestimme für String s die Menge seiner N-Gramme:

$$G(s) = \{\text{Substrings von } s \text{ mit der Länge } N\}$$

(häufig werden Trigramme betrachtet mit $N=3$)

Ähnlichkeit von Strings s_1 und s_2 :

$$|G(s_1)| + |G(s_2)| - 2|G(s_1) \cap G(s_2)|$$

Beispiel:

$$G(\text{rodney}) = \{\text{rod, odn, dne, ney}\}$$

$$G(\text{rhodnee}) = \{\text{rho, hod, odn, dne, nee}\}$$

$$\text{Ähnlichkeit}(\text{rodney}, \text{rhodnee}) = 4 + 5 - 2 \cdot 2 = 5$$

Phonetic Similarity (1)

Soundex-Code:

Abbildung von Wörtern (speziell: Nachnamen) auf 4-Zeichen-Codes,
so daß ähnlich ausgesprochene Wörter denselben Code haben

- Erstes Codezeichen = erstes Zeichen des Wortes
- Codezeichen zwei bis vier (a, e, i, o, u, y, h, w werden ignoriert):

b, p, f, v → 1 c, s, g, j, k, q, x, z → 2

d, t → 3 l → 4

m, n → 5 r → 6

- Aufeinanderfolgende identische Codes werden zusammengefasst
(es sei denn, sie sind durch h getrennt)

Beispiele:

Powers → P620 , Perez → P620

Penny → P500, Penee → P500

Tymczak → T522, Tanshik → T522

Phonetic Similarity: Editex

Idea: Editierdistanz mit Berücksichtigung phonetischer Codes

Für $\text{editex}(i, j)$: Editex-Distanz von $s1[1..i]$ und $s2[1..j]$

gilt: $\text{editex}(0, 0) = 0$,

$\text{editex}(i, 0) = \text{editex}(i-1, 0) + d(s1[i-1], s1[i])$,

$\text{editex}(0, j) = \text{editex}(0, j-1) + d(s2[j-1], s2[j])$,

$\text{editex}(i, j) = \min \{ \text{editex}(i-1, j) + d(s1[i-1], s1[i]),$
 $\text{editex}(i, j-1) + d(s2[j-1], s2[j]),$
 $\text{edit}(i-1, j-1) + \text{diffcode}(i, j) \}$

mit $\text{diffcode}(i, j) = 0$ falls $s1i = s2j$,

1 falls $\text{group}(s1i) = \text{group}(s2j)$, 2 sonst

und $d(X, Y) = 1$ falls $X \neq Y$ und X ist h oder w,

$\text{diffcode}(X, Y)$ sonst

mit group :

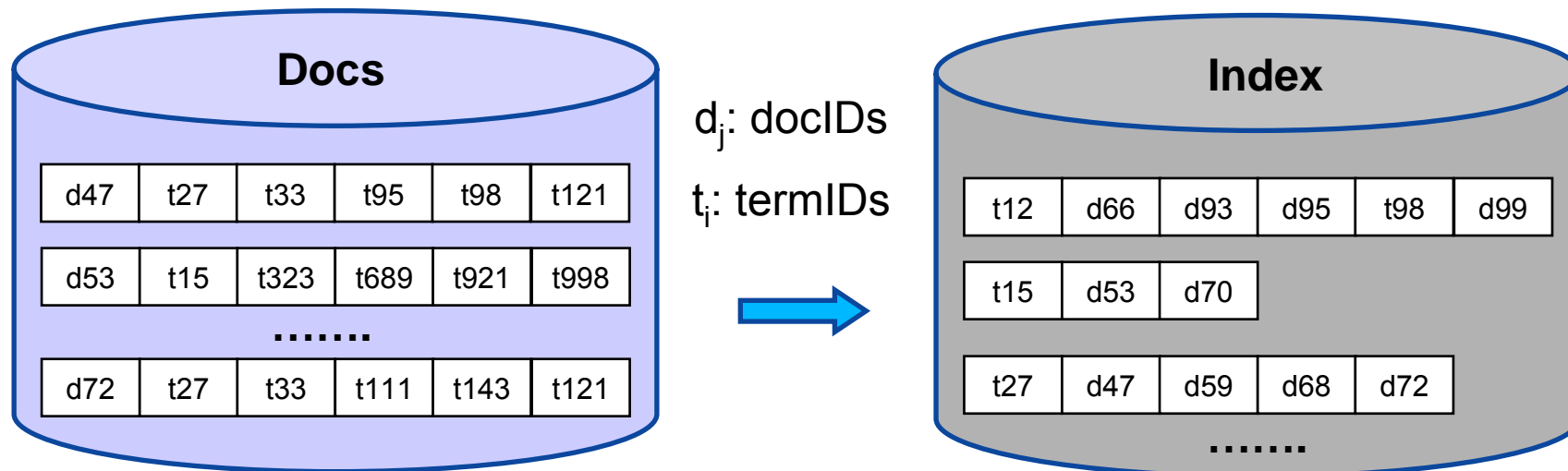
$\{a\ e\ i\ o\ u\ y\}$, $\{b\ p\}$, $\{c\ k\ q\}$, $\{d\ t\}$, $\{l\ r\}$,

$\{m\ n\}$, $\{g\ j\}$, $\{f\ p\ v\}$, $\{s\ x\ z\}$, $\{c\ s\ z\}$

2.5: Text Indexing and Query Execution

Konzeptionell:

invertierte Dateien (invertierte Listen) mit binärer Suche
nach Suchschlüsseln (Felder von Records, Strings in Texten)



Problem:

Speicherungsorganisation in Plattenblöcken (pagination) und effiziente Implementierung der (binären) Suche für

Exact-Match-Suche: search (key) returns ids

Bereichssuche: search (lowkey, highkey) returns ids

Präfixstringsuche: search (prefix) returns ids

bei dynamischen Updates

Binäre Suchbäume (für Hauptspeicher)

Schlüssel (mit den ihnen zugeordneten Daten)
bilden die Knoten eines binären Baums
mit der Invariante:

für jeden Knoten t mit Schlüssel $t.key$ und alle Knoten l im linken Teilbaum von t , $t.left$, und alle Knoten r im rechten Teilbaum von t gilt: $l.key \leq t.key \leq r.key$

Suchen eines Schlüssels k :

Traversieren des Pfades von der Wurzel bis zu k bzw. einem Blatt

Einfügen eines Schlüssels k :

Suchen von k und Anfügen eines neuen Blatts

Löschen eines Schlüssel k :

Ersetzen von k durch das „rechtteste“ Blatt links von k

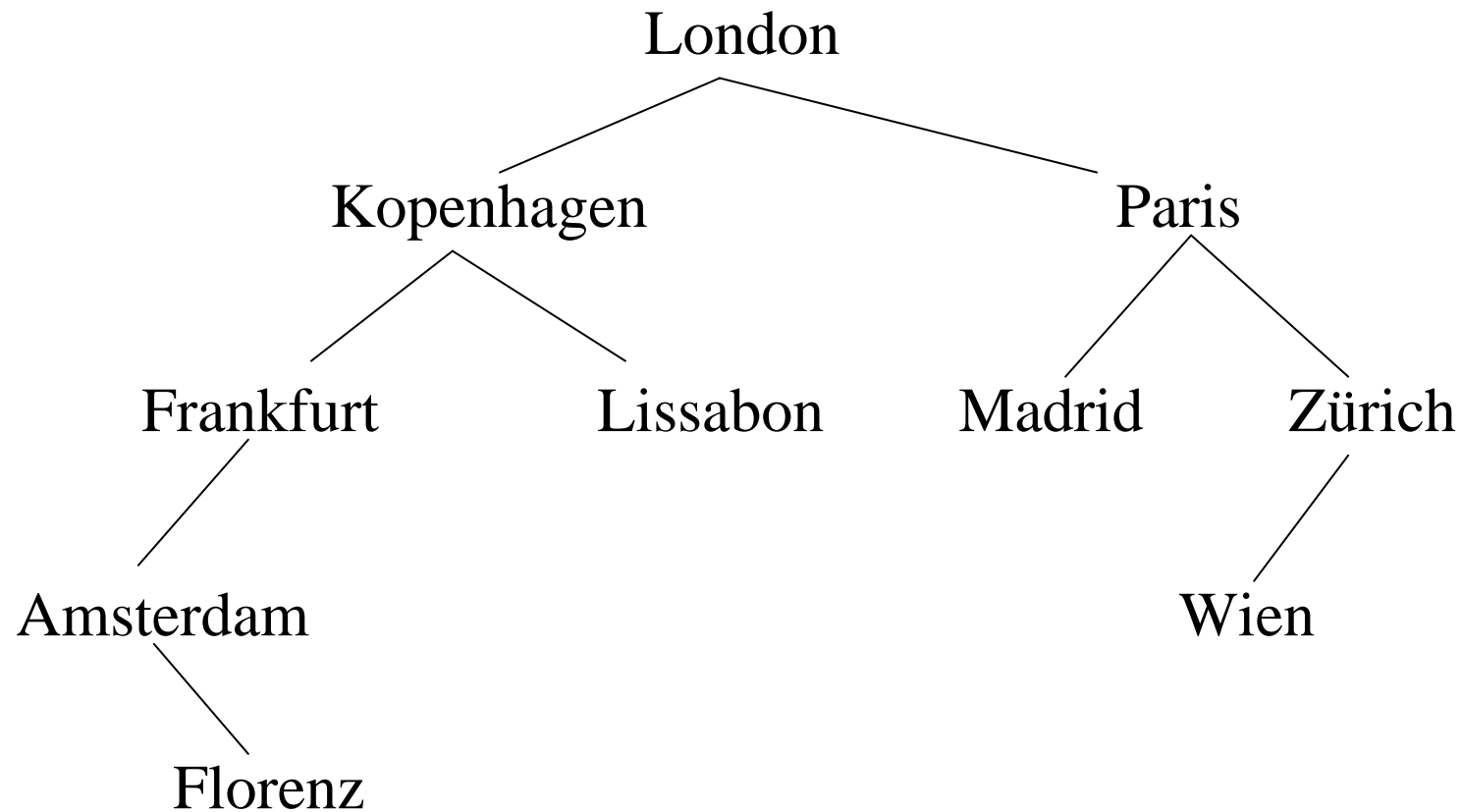
Worst-Case-Suchzeit für n Schlüssel: $O(n)$

bei geeigneten Rebalancierungsalgorithmen

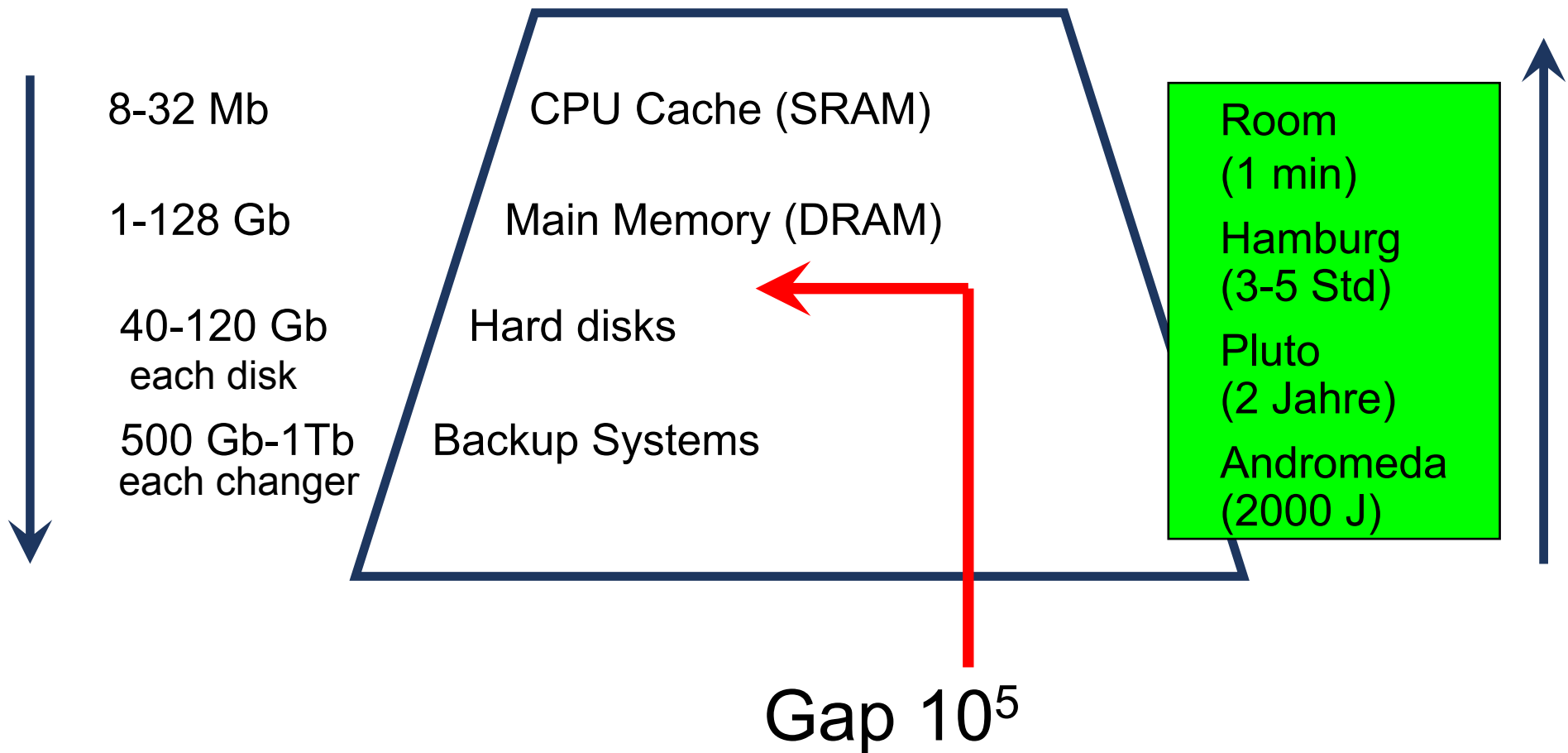
(AVL-Bäume, Rot-Schwarz-Bäume, usw.): $O(\log n)$

Beispiel für einen binären Suchbaum

London, Paris, Madrid, Kopenhagen, Lissabon, Zürich, Frankfurt, Wien, Amsterdam, Florenz

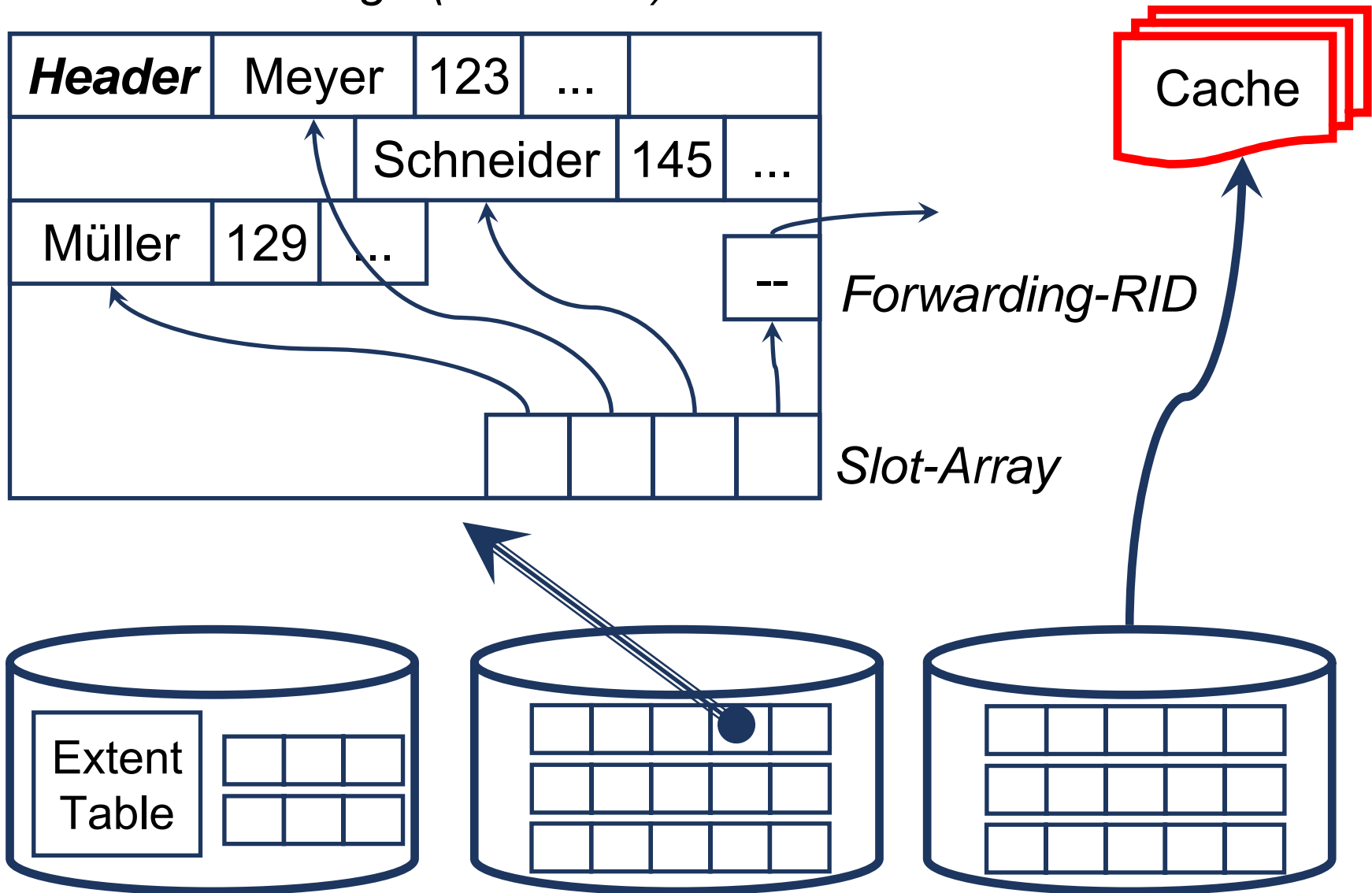


Properties of Media Types



Access: Physical Data Organization

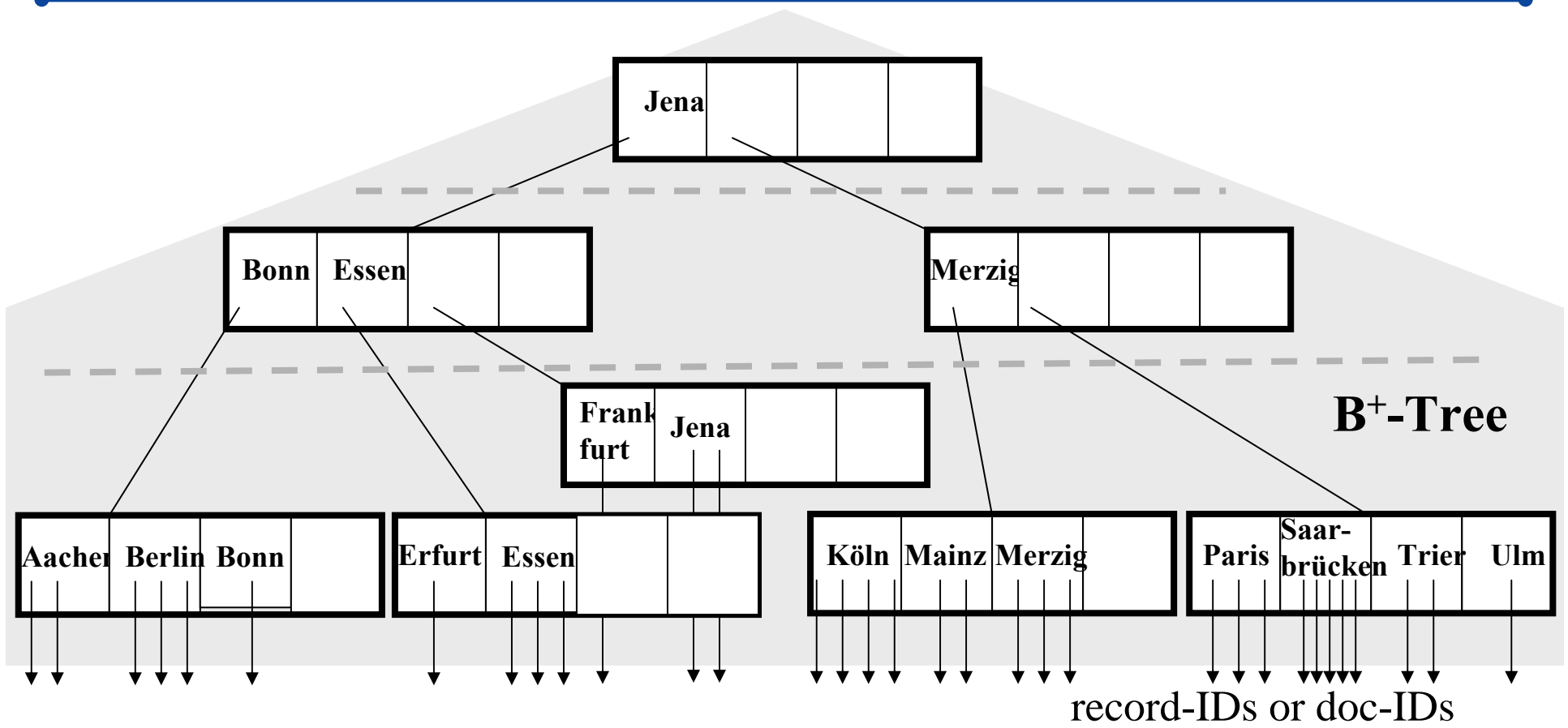
Database Page (32-64 Kb)



B⁺ Trees: Seitenstrukturierte Mehrwegbäume

- Hohler Mehrwegebaum mit hohem Fanout (\Rightarrow kleiner Tiefe)
- Knoten = Seite auf Platte
- Knoteninhalt:
 - (Sohnzeiger, Schlüssel)-Paare in inneren Knoten
 - Schlüssel (mit weiteren Daten) in Blättern
- perfekt balanciert: alle Blätter haben dieselbe Distanz zur Wurzel
- Sucheeffizienz $O(\log_k n/C)$ Seitenzugriffe (Platten-I/Os)
bei n Schlüsseln, Seitenkapazität C und Fanout k
pro Baumniveau: bestimme kleinsten Schlüssel $\geq q$ und
suche weiter im Teilbaum links von q
- Kosten einer Einfüge- oder Löschoperation $O(\log_k n/C)$
- mittlere Speicherplatzauslastung bei zufälligem Einfügen: $\ln 2 \approx 0.69$

B+ Trees: Example



B⁺ Tree: Definition

Ein Mehrwegbaum heißt B*-Baum (eng.: B⁺ Tree) der Ordnung (m, m^*) , wenn gilt:

- Jeder Nichtblattknoten außer der Wurzel enthält mindestens $m \geq 1$ und höchstens $2m$ Schlüssel (Wegweiser).
- Ein Nichtblattknoten mit k Schlüssel x_1, \dots, x_k hat genau $k+1$ Söhne $t_1, \dots, t_{(k+1)}$, so daß
 - für alle Schlüssel s im Teilbaum t_i ($2 \leq i \leq k$) gilt $x_{(i-1)} < s \leq x_i$ und
 - für alle Schlüssel s im Teilbaum t_1 gilt $s \leq x_1$ und
 - für alle Schlüssel im Teilbaum $t_{(k+1)}$ gilt $x_k < s$.
- Alle Blätter haben dasselbe Niveau (Distanz von der Wurzel)
- Jedes Blatt enthält mindestens $m^* \geq 1$ und höchstens $2m^*$ Schlüssel.

Achtung: Implementierungen verwenden **variabel lange Schlüssel** und eine Knotenkapazität in Bytes statt Konstanten $2m$ und $2m^*$

Sonderfall $m=m^*=1$: **2-3-Bäume** als Hauptspeicherdatenstruktur

Pseudo code for B⁺ Tree Lookups

Suchen von Schlüssel s in B^{*}-Baum mit Wurzel t :

t habe k Schlüssel x_1, \dots, x_k und $k+1$ Söhne $t_1, \dots, t_{(k+1)}$
(letzteres sofern t kein Blatt ist)

Bestimme den kleinsten Schlüssel x_i , so daß $s \leq x_i$

if $s = x_i$ (für ein $i \leq k$) und t ist ein Blatt

then Schlüssel gefunden

else

if t ist kein Blatt then

if $s \leq x_i$ (für ein $i \leq k$)

then suche s im Teilbaum t_i

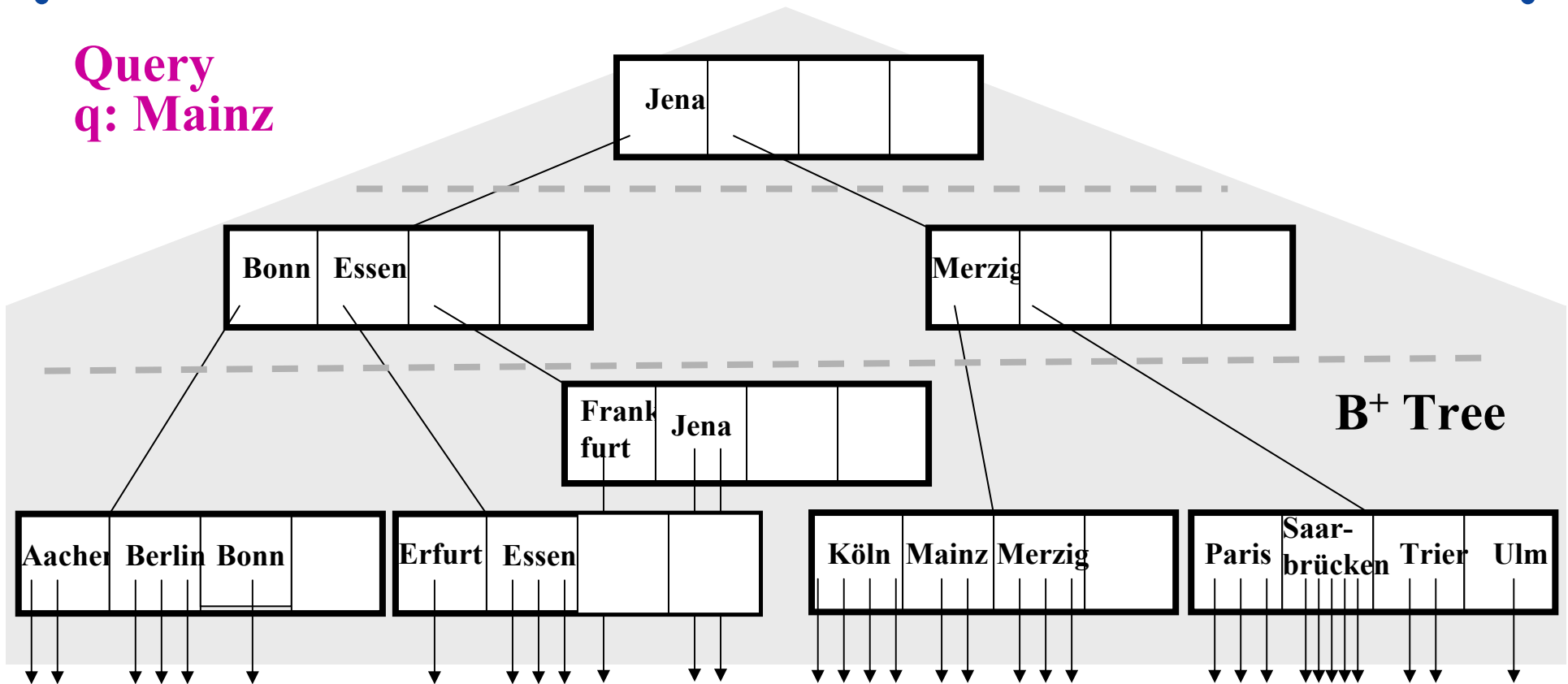
else suche s im Teilbaum $t_{(k+1)}$ fi

else Schlüssel s ist nicht vorhanden fi

fi

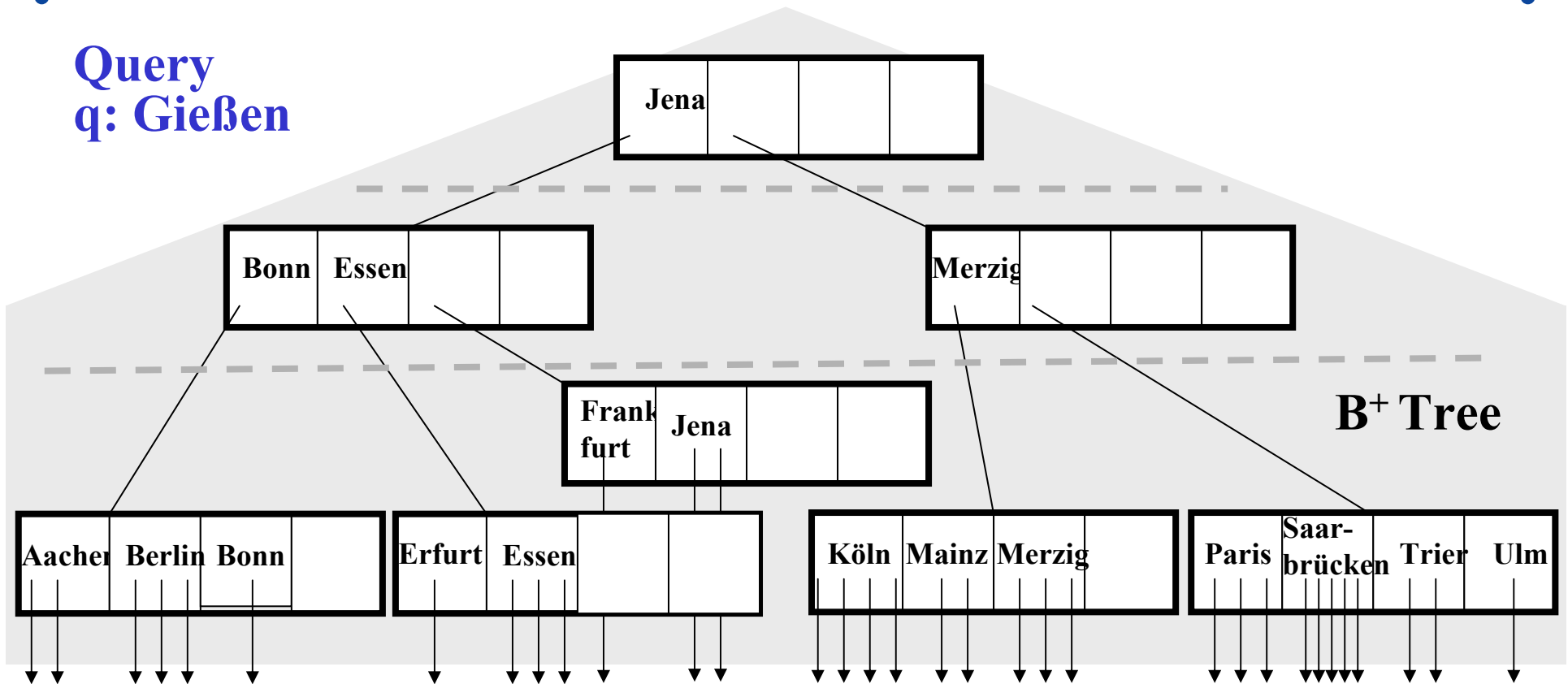
B+ Trees: Lookup (1)

Query
q: Mainz



B+ Trees: Lookup (2)

Query
q: Gießen



Pseudo code for inserting into B⁺ Tree (Grow&Post)

Suche nach einzufügendem Schlüssel e

if e ist noch nicht vorhanden then

Sei t das Blatt, bei dem die Suche erfolglos geendet hat

repeat

if t hat weniger als $2m^*$ bzw. $2m$ Schlüssel (d.h. ist nicht voll)

then füge e in t ein

else /* *Knoten-Split* */

Bestimme Median s der $2m^* + 1$ bzw. $2m+1$ Schlüssel inkl. e

Erzeuge Bruderknoten t' /* *Grow-Phase* */

if t ist Blattknoten then

Speichere Schlüssel $\leq s$ in t und Schlüssel $> s$ in t'

else Speichere Schlüssel $< s$ (mit Sohnzeigern) in t und

Schlüssel $> s$ (mit Sohnzeigern) in t' fi

if t ist Wurzel /* *Post-Phase* */

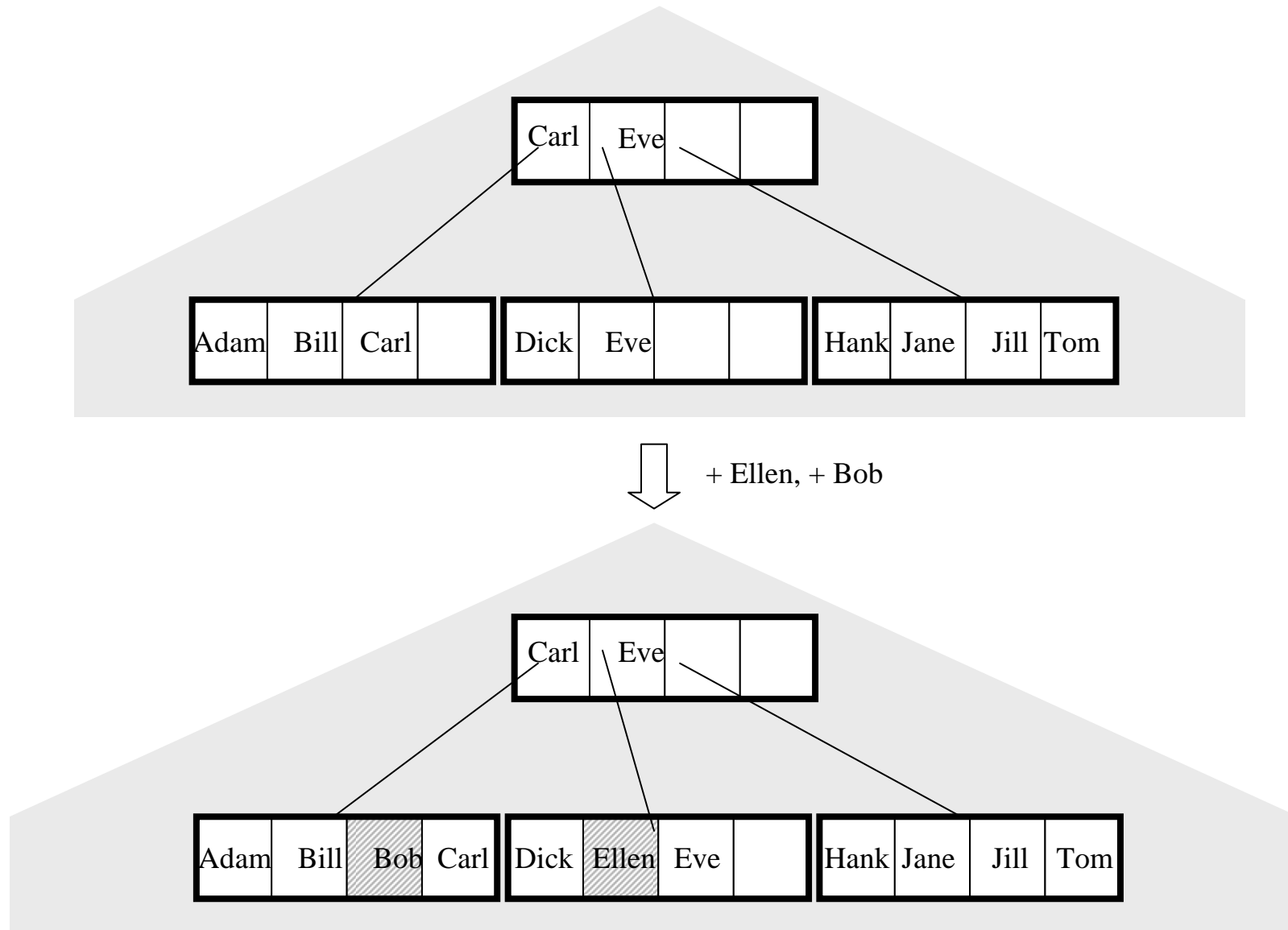
then Erzeuge neue Wurzel r mit Schlüssel s und Zeigern auf t und t'

else Betrachte Vater von t als neues t und s (mit Zeiger auf t') als e fi

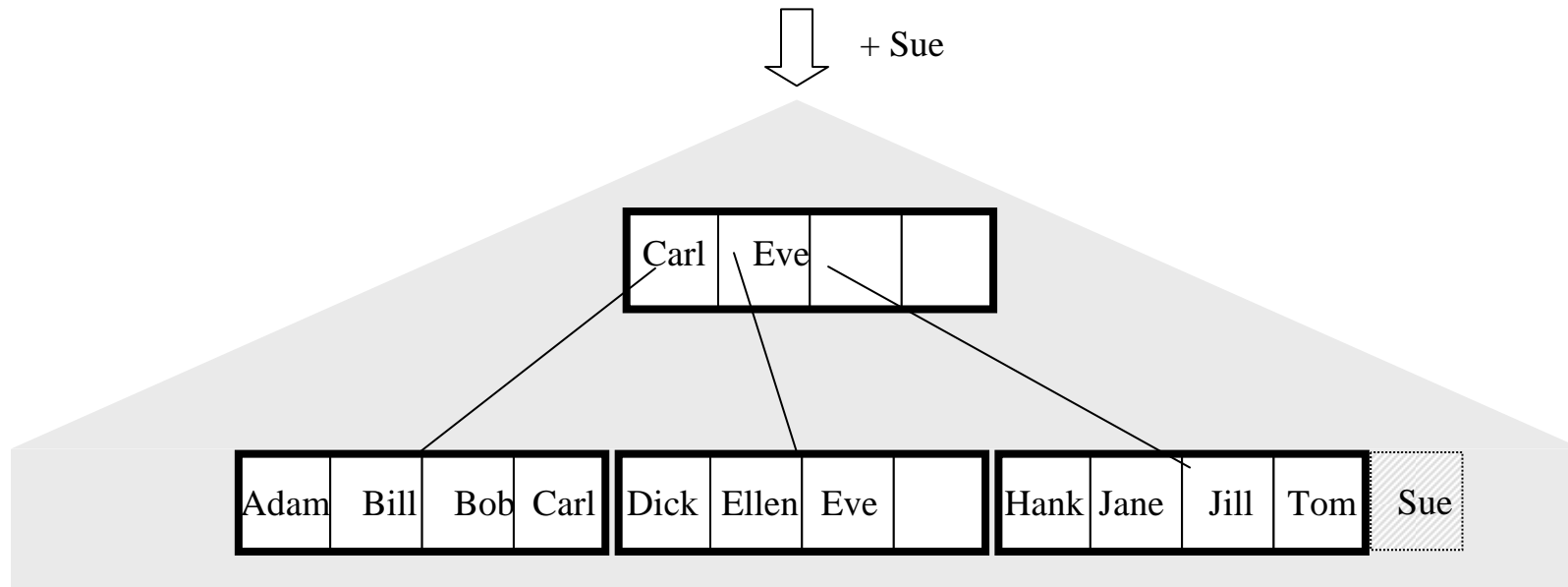
fi

until kein Knoten-Split mehr erfolgt

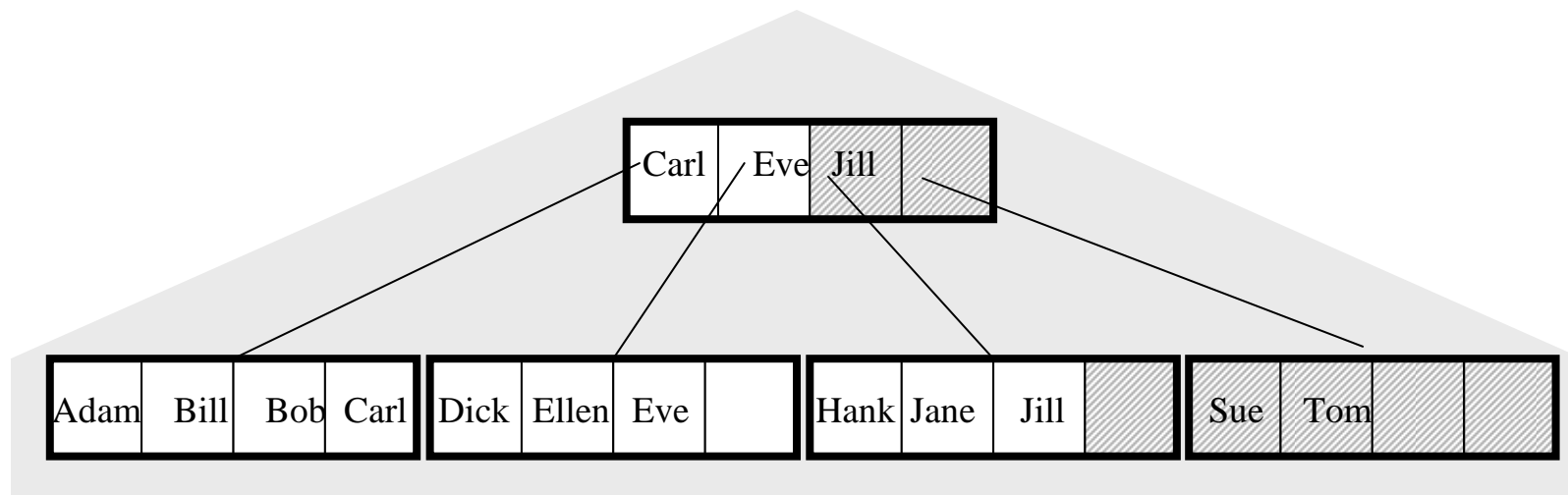
Example: Insert into B+ Tree



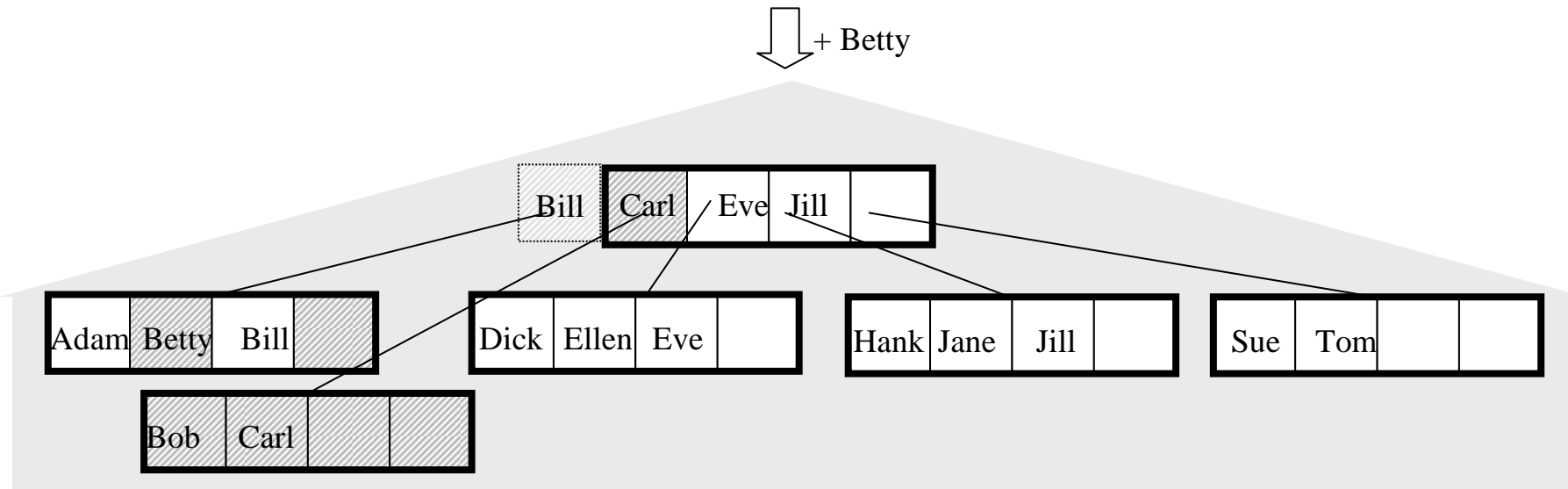
Example: Insert into B+ Tree with split



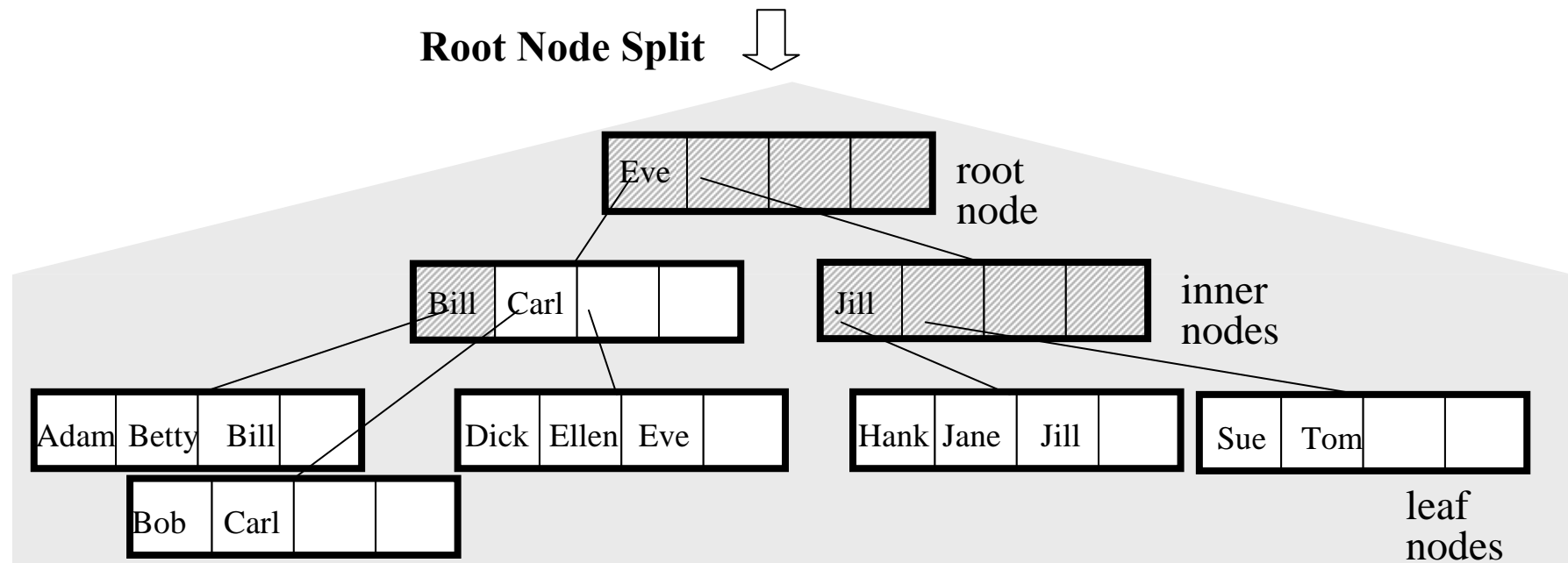
Leaf Node Split ↓



Example: Insert into B+ Tree with root split



Root Node Split



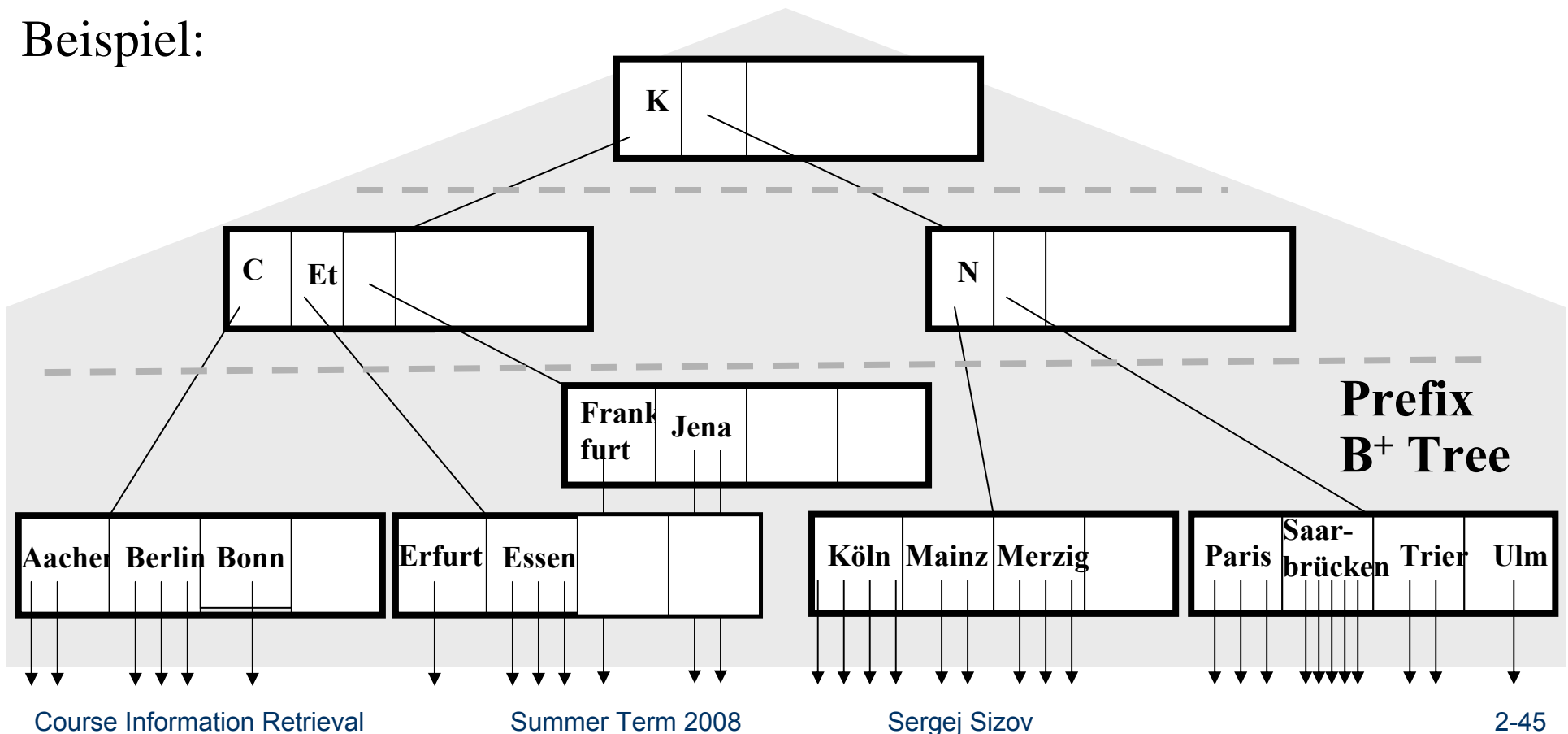
Prefix-Based B+ Trees for Strings

Schlüssel in inneren Knoten sind nur **Wegweiser (Router)** zur Partitionierung des Schlüsselraums.

Statt $x_i = \max\{s: s \text{ ist ein Schlüssel im Teilbaum } t_i\}$ genügt ein (kürzerer) Wegweiser x_i' mit $s_i \leq x_i' < x_{i+1}$ für alle s_i in t_i und alle s_{i+1} in t_{i+1} . Eine Wahl wäre $x_i' = \text{kürzester String mit der o.a. Eigenschaft.}$

→ höherer Fanout, potentiell kleinere Baumhöhe

Beispiel:



Simplified Query Processor

1. Remove stopwords from query and map all words onto terms (word stems)

2. Convert query q into normal form

```
[ (t_11 AND t_12 AND ... AND t_1k_1)
  OR ...
  (t_r1 AND t_r2 AND ... AND t_rk_r) ]
AND NOT t_r+1 AND NOT t_r+2 ...
```

3. For $i=1$ to r do

 find DocId-Lists for t_{i_1} bis $t_{i_{k_i}}$ and

 compute union V_i of these DocIds

 set $rank(d)$ of $d \in V_i$ auf $\text{sim}(d, q_i)$

Compute union V of DocId-Sets V_1, \dots, V_r

4. Sort all $d \in V$ in descending order of $rank(d)$

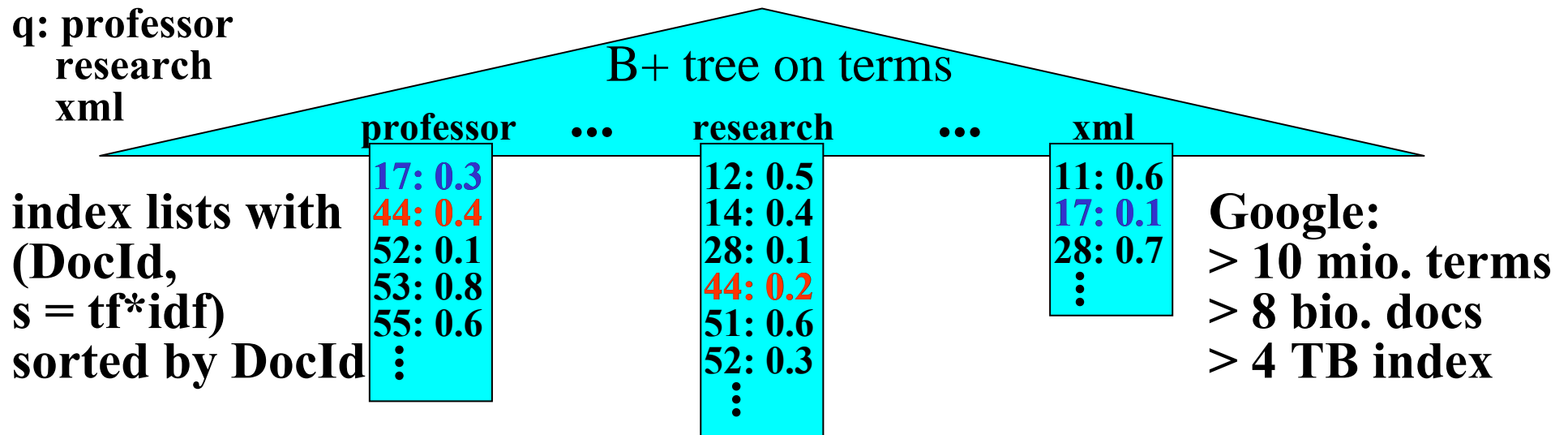
5. Retrieve docs $d \in V$ in sorted order (possibly with heuristic termination criteria),

 eliminate docs d that contain one or more terms t_{r+1}, t_{r+2}, \dots

Top-k Query Processing with Scoring

Vector space model suggests *m*×*n* *term-document matrix*,
but data is sparse and queries are even very sparse
→ better use *inverted index lists* with terms as keys for B+ tree

q: professor
research
xml



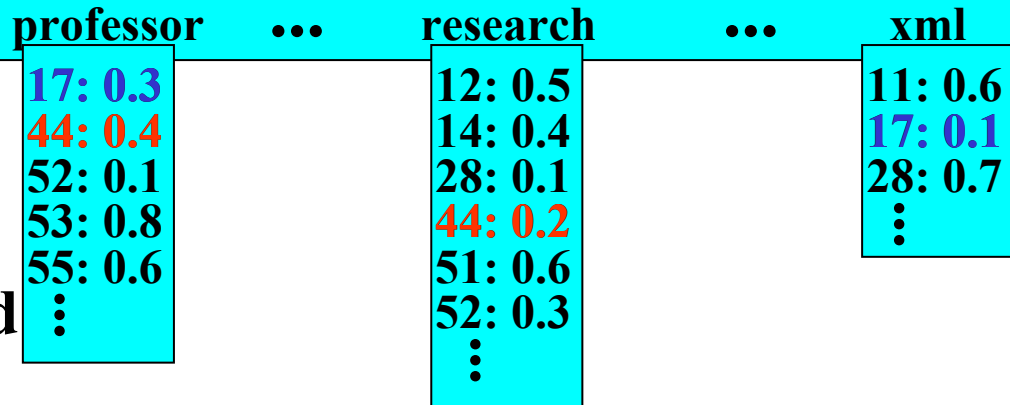
terms can be full words, word stems, word pairs,
word substrings, etc.
(whatever „dictionary terms“ we prefer for the application)

queries can be conjunctive or „andish“ (soft conjunction)

DBS-Style Top-k Query Processing

q: professor
research
xml

B+ tree on terms



index lists with
(DocId,
s = tf*idf)
sorted by DocId

Google:
> 10 mio. terms
> 8 bio. docs
> 4 TB index

Given: query $q = t_1 t_2 \dots t_z$ with z (conjunctive) keywords
similarity scoring function $\text{score}(q,d)$ for docs $d \in D$, e.g.: $\vec{q} \cdot \vec{d}$
with precomputed scores (index weights) $s_i(d)$ for which $q_i \neq 0$

Find: top k results w.r.t. $\text{score}(q,d) = \text{aggr}\{s_i(d)\}$ (e.g.: $\sum_{i \in q} s_i(d)$)

Naive join&sort QP algorithm:

top-k (

$\sigma[\text{term}=t_1]$ (index)		×		DocId
$\sigma[\text{term}=t_2]$ (index)		×		DocId
...		×		DocId
$\sigma[\text{term}=t_z]$ (index)		×		DocId

order by s desc)

Index List Processing by Merge Join

Keep $L(i)$ in **ascending order of doc ids**

Compress $L(i)$ by actually storing the gaps between successive doc ids
(or using some more sophisticated prefix-free code)

QP may start with those $L(i)$ lists that are short and have high idf

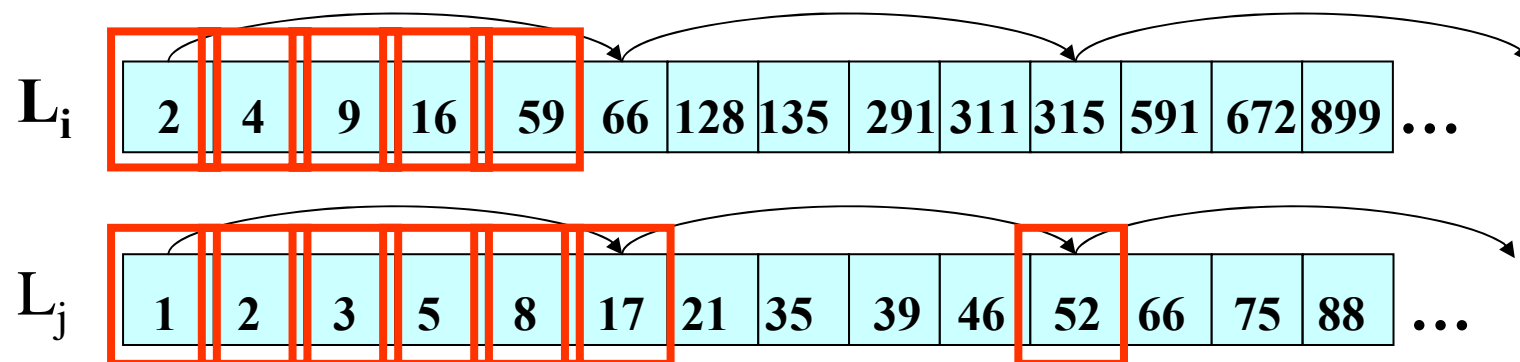
Candidate results need to be looked up in other lists $L(j)$

To avoid having to uncompress the entire list $L(j)$,

$L(j)$ is encoded into groups of entries

with a **skip pointer** at the start of each group

→ \sqrt{n} evenly spaced skip pointers for list of length n



Computational Model for Top-k Queries

Assume *local scores* s_i for query q , data item d , and dimension i , and *global scores* s of the form $s(q, d) = \text{aggr}\{s_i(q, d) / i = 1..m\}$ with a *monotonic* aggregation function $\text{aggr} : [0,1]^m \rightarrow [0,1]$

Examples: $s(q, d) = \sum_{i=1}^m s_i(q, d)$ $s(q, d) = \max\{s_i(q, d) / i = 1..m\}$

Find top-k data items with regard to global scores:

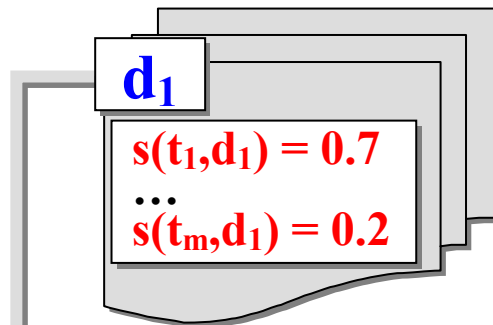
- process m *index lists* L_i with *sorted access (SA)* to entries $(d, s_i(q, d))$ in *ascending order of doc ids* or *descending order of $s_i(q, d)$*
- maintain for each candidate d a set $E(d)$ of evaluated dimensions and a *partial score „accumulator“*
- for candidate d with incomplete $E(d)$ consider looking up d in L_i for all $i \in R(d)$ by *random access (RA)*
- terminate index list scans when enough candidates have been seen
- if necessary sort final candidate list by global score

Efficient Top-k Search

[Buckley85, Güntzer/Balke/Kießling 00, Fagin01]

threshold algorithms: efficient & principled top-k query processing with monotonic score aggr.

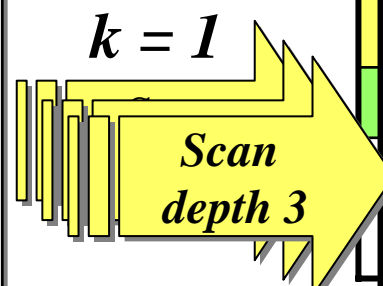
Data items: d_1, \dots, d_n



Query: $q = (t_1, t_2, t_3)$

Index lists						
t_1	d78 0.9	d23 0.8	d10 0.8	d1 0.7	d88 0.2	...
t_2	d64 0.8	d23 0.6	d10 0.6	d10 0.2	d78 0.1	...
t_3	d10 0.7	d78 0.5	d64 0.4	d99 0.2	d34 0.1	...

TA with sorted access only (NRA):
 can index lists; consider d at pos_i in L_i ;
 $E(d) := E(d) \cup \{i\}$; $high_i := s(t_i, d)$;
 $worstscore(d) := aggr\{s(t_v, d) \mid v \in E(d)\}$;
 $bestscore(d) := aggr\{worstscore(d), aggr\{high_v \mid v \notin E(d)\}\}$;
 if $worstscore(d) > min-k$ then add d to top-k
 $min-k := \min\{worstscore(d') \mid d' \in top-k\}$;
 else if $bestscore(d) > min-k$ then
 $cand := cand \cup \{d\}$; s
 $threshold := \max\{bestscore(d') \mid d' \in cand\}$;
 if $threshold \leq min-k$ then exit;



Rank	Doc	Worst-score	Best-score
1	d10	2.1	2.1
2	d78	1.4	2.0
3			1.8
4		1.2	2.0



keep $L(i)$ in descending order of scores

Threshold Algorithm (TA, Quick-Combine, MinPro)

(Fagin'01; Güntzer/Balke/Kießling; Nepal/Ramakrishna)

```
scan all lists  $L_i$  ( $i=1..m$ ) in parallel:
  consider  $d_j$  at position  $pos_i$  in  $L_i$ ;
   $high_i := s_i(d_j)$ ;
  if  $d_j \notin \text{top-k}$  then {
    look up  $s_v(d_j)$  in all lists  $L_v$  with  $v \neq i$ ; // random access
    compute  $s(d_j) := \text{aggr} \{s_v(d_j) \mid v=1..m\}$ ;
    if  $s(d_j) > \text{min score among top-k}$  then
      add  $d_j$  to top-k and remove min-score  $d$  from top-k; }
   $\text{threshold} := \text{aggr} \{high_v \mid v=1..m\}$ ;
  if  $\text{min score among top-k} \geq \text{threshold}$  then exit;
```

*but random accesses
are expensive !*

$m=3$
aggr: sum
 $k=2$

f: 0.5
b: 0.4
c: 0.35
a: 0.3
h: 0.1
d: 0.1

a: 0.55
b: 0.2
f: 0.2
g: 0.2
c: 0.1

h: 0.35
d: 0.35
b: 0.2
a: 0.1
c: 0.05
f: 0.05

top-k:
~~f: 0.75~~
a: 0.95
b: 0.8

No-Random-Access Algorithm

(NRA, Stream-Combine, TA-Sorted)

scan index lists in parallel:

consider d_j at position pos_i in L_i ;

$E(d_j) := E(d_j) \cup \{i\}$; $high_i := si(q, d_j)$;

$bestscore(d_j) := aggr\{x_1, \dots, x_m\}$

with $x_i := si(q, d_j)$ for $i \in E(d_j)$, $high_i$ for $i \notin E(d_j)$;

$worstscore(d_j) := aggr\{x_1, \dots, x_m\}$

with $x_i := si(q, d_j)$ for $i \in E(d_j)$, 0 for $i \notin E(d_j)$;

$top-k := k$ docs with largest $worstscore$;

$threshold := bestscore\{d \mid d \text{ not in } top-k\}$;

if $\min worstscore$ among $top-k \geq threshold$ then exit;

$m=3$
aggr: sum
 $k=2$

f: 0.5
b: 0.4
c: 0.35
a: 0.3
h: 0.1
d: 0.1

a: 0.55
b: 0.2
f: 0.2
g: 0.2
c: 0.1

h: 0.35
d: 0.35
b: 0.2
a: 0.1
c: 0.05
f: 0.05

top-k:

a: 0.95

b: 0.8

candidates:

f: $0.7 + ? \leq 0.7 + 0.1$

h: $0.35 + ? \leq 0.35 + 0.5$

c: $0.35 + ? \leq 0.35 + 0.3$

d: $0.35 + ? \leq 0.35 + 0.5$

g: $0.2 + ? \leq 0.2 + 0.4$

2-53

Optimality of TA

Definition:

For a class \mathcal{A} of algorithms and a class \mathcal{D} of datasets, let $\text{cost}(A, D)$ be the execution cost of $A \in \mathcal{A}$ on $D \in \mathcal{D}$.

Algorithm B is *instance optimal* over \mathcal{A} and \mathcal{D} if

for every $A \in \mathcal{A}$ on $D \in \mathcal{D}$: $\text{cost}(B, D) = O(\text{cost}(A, D))$,

that is: $\text{cost}(B, D) \leq c \cdot \text{cost}(A, D) + c'$

with optimality ratio (competitiveness) c .

Theorem:

- TA is instance optimal over all algorithms that are based on sorted and random access to (index) lists (no „wild guesses“).
TA has optimality ratio $m + m(m-1) C_{RA}/C_{SA}$
with random-access cost C_{RA} and sorted-access cost C_{SA}
- NRA is instance-optimal over all algorithms with SA only.

*if „wild guesses“ are allowed,
then no deterministic algorithm is instance-optimal*

Execution Cost of TA Family

Run-time cost is $O\left(n^{\frac{m-1}{m}} \cdot k^{\frac{1}{m}}\right)$ with arbitrarily high probability

(for independently distributed Li lists)

Memory cost is $O(k)$ for TA

and $O(n^{(m-1)/m})$ for NRA (priority queue of candidates)

Approximate Top-k Query Processing

Approximation TA:

A θ -approximation T' for top-k query q with $\theta > 1$ is a set T' of docs with:

- $|T'|=k$ and
- for each $d' \in T'$ and each $d'' \notin T'$: $\theta * \text{score}(q, d') \geq \text{score}(q, d'')$

Modified TA:

...

Stop when $\min_k \geq \text{aggr}(\text{high}_1, \dots, \text{high}_m) / \theta$

General heuristics:

- **disregard index lists with idf below threshold**
- **for index scans give priority to index lists that are short and have high idf**

Pruning with Combined Authority/Similarity Scoring

(Long/Suel 2003)

Focus on $\text{score}(q,d_j) = r(d_j) + s(q,d_j)$

with normalization $r(\cdot) \leq a$, $s(\cdot) \leq b$ (and often $a+b=1$)

Keep index lists sorted in **descending order of „static“ authority $r(d_j)$**

Conservative authority-based pruning:

$\text{high}(0) := \max\{r(\text{pos}(i)) \mid i=1..m\}$; $\text{high} := \text{high}(0) + b$;

$\text{high}(i) := r(\text{pos}(i)) + b$;

stop scanning i -th index list when $\text{high}(i) < \text{min score of top } k$

terminate algorithm when $\text{high} < \text{min score of top } k$

effective when total score of top- k results is dominated by r

First- k' heuristics:

scan all m index lists until $k' \geq k$ docs have been found

that appear in all lists;

the stopping condition is easy to check because of the sorting by r

DB Integration example: Oracle Text (Oracle interMedia)

Modern commercial DB systems offer...

Creating and maintaining various text indexes

Searching text (exact match, inexact match, ranked retrieval, etc.)

Semantic query expansion, topic recognition, summarization

Text classification and clustering

Various text format conversions

Email filtering and classification

Multi-lingual parsing, indexing, and cross-language search

example:

```
SELECT score(1), artikel_id
FROM wiki_articles
WHERE CONTAINS (text_body, 'Koblenz NEAR Carnival', 1) > 0
order by score(1) DESC
```

Text Retrieval: Limitations and Problems with Web Mining Apps

IR necessary but not sufficient for Web search !

- Doesn't capture authority
 - An article on BBC as good as a copy on john-doe-news.com
- Doesn't address web navigation
 - Query ibm seeks www.ibm.com
 - www.ibm.com may look less topical than a quarterly report