# Chapter 11: Text Indexing and Matching

*There were 5 Exabytes of information created between the dawn of civilization through 2003, but that much information is now created every 2 days.* **-- Eric Schmidt**

*There is nothing that cannot be found through some search engine.* **-- Eric Schmidt**

*The best place to hide a dead body is page 2 of Google search results.* **-- anonymous**

*An engineer is someone who can do for a dime what any fool can do for a dollar.* **-- anonymous**

# Outline

**11.1 Search Engine Architecture**

**11.2 Dictionary and Inverted Lists**

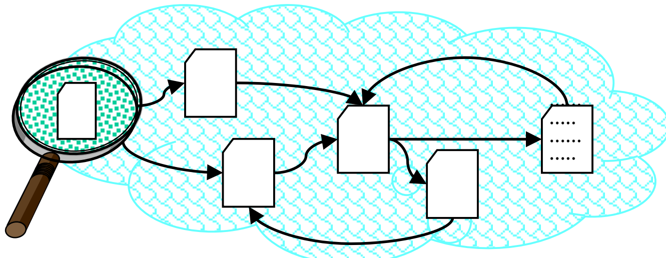
**11.3 Index Compression**

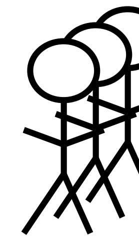
**11.4 Similarity Search**

mostly following Büttcher/Clarke/Cormack Chapters 2,3,**4**,**6**
(alternatively: Manning/Raghavan/Schütze Chapters **3**,**4**,**5**,6)

*11.2 mostly BCC Ch.4, 11.3 mostly BCC Ch.6, 11.4 mostly MRS Ch.3*

# 11.1 Search Engine Architecture





**crawl** → **extract & clean** → **index** → **search** → **rank** → **present**

**crawl**: strategies for crawl schedule and priority queue for crawl frontier

**extract & clean**: handle dynamic pages, detect duplicates, detect spam

**index**: build and analyze Web graph, index all tokens or word stems

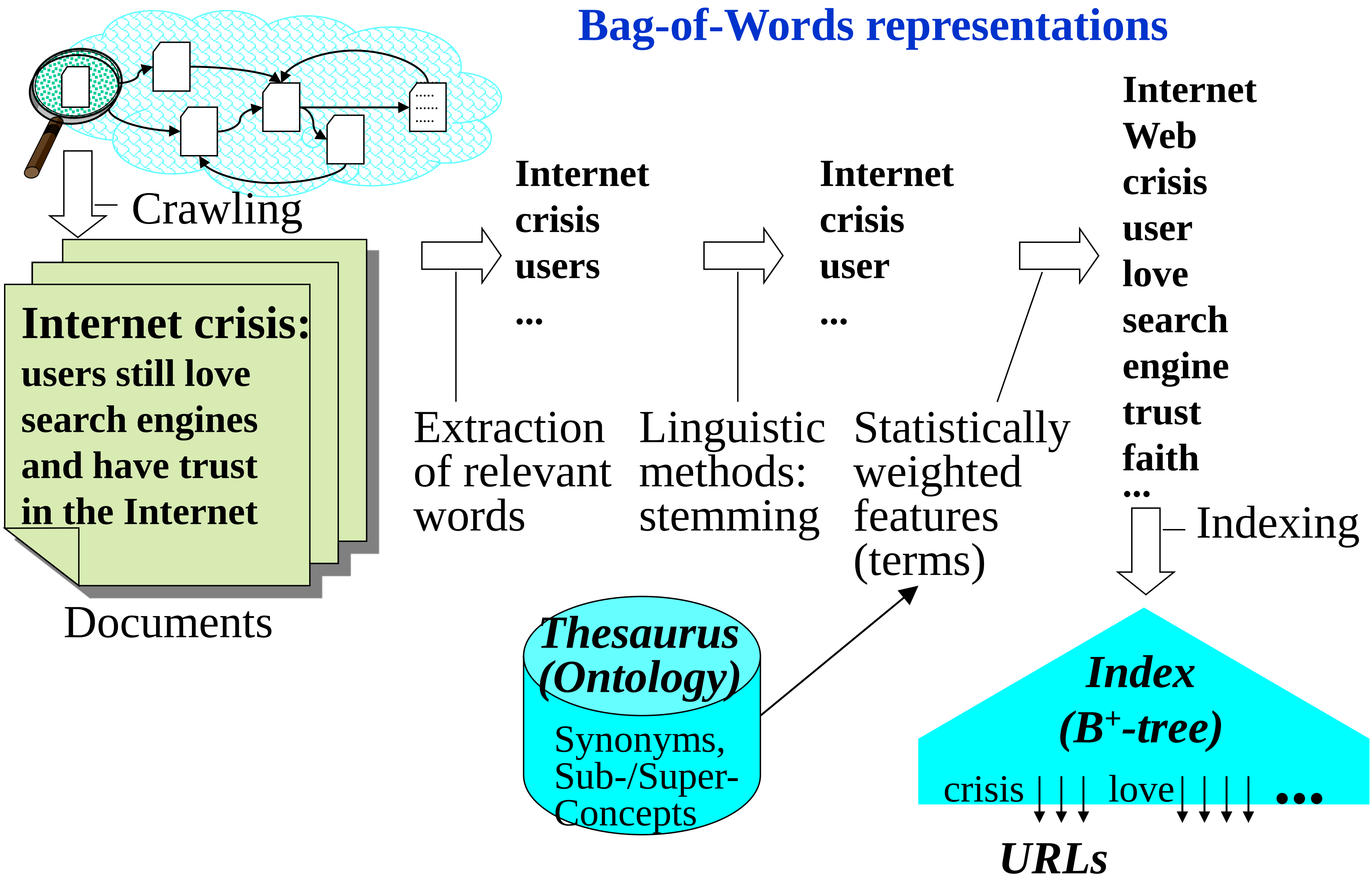
**search**: fast top-k queries, query logging, auto-completion

**rank**: scoring function over many data and context criteria

**present**: GUI, user guidance, personalization

server farm with 100 000‘s of computers,
distributed/replicated data in high-performance file system,
massive parallelism for query processing

# Content Gathering and Indexing



# Crawling

- **Traverse Web:** fetch page by http,
  parse retrieved html content for href links
- **Crawl frontier:** maintain priority queue
- **Crawl strategy:** breadth-first for broad coverage,
  depth-first for site capturing,
  clever prioritization
- **Link extraction:** handle dynamic pages (Javascript …)

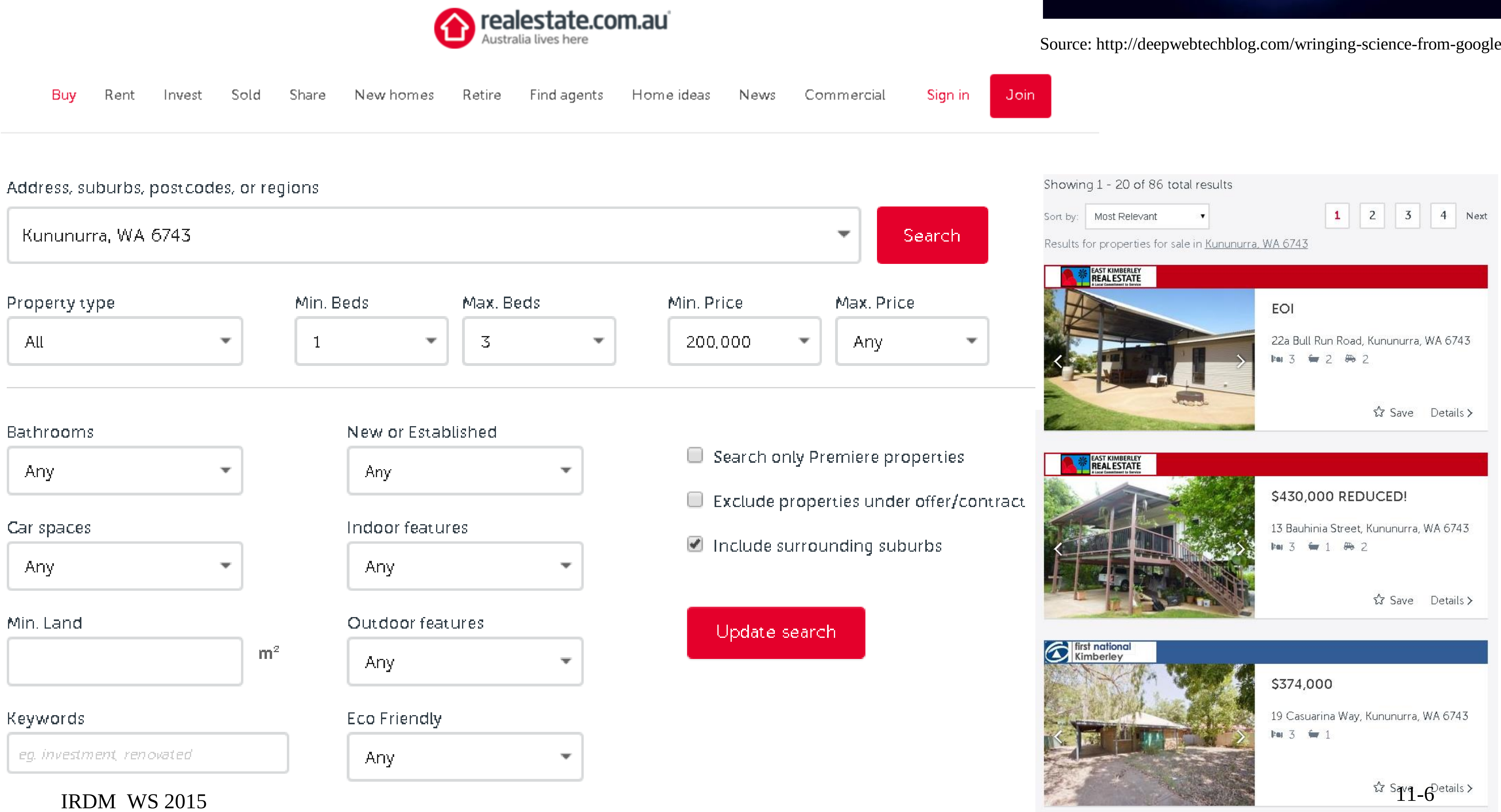
***Deep Web Crawling:*** generate form-filling queries

***Focused Crawling:*** interleave with classifier

# Deep Web Crawling

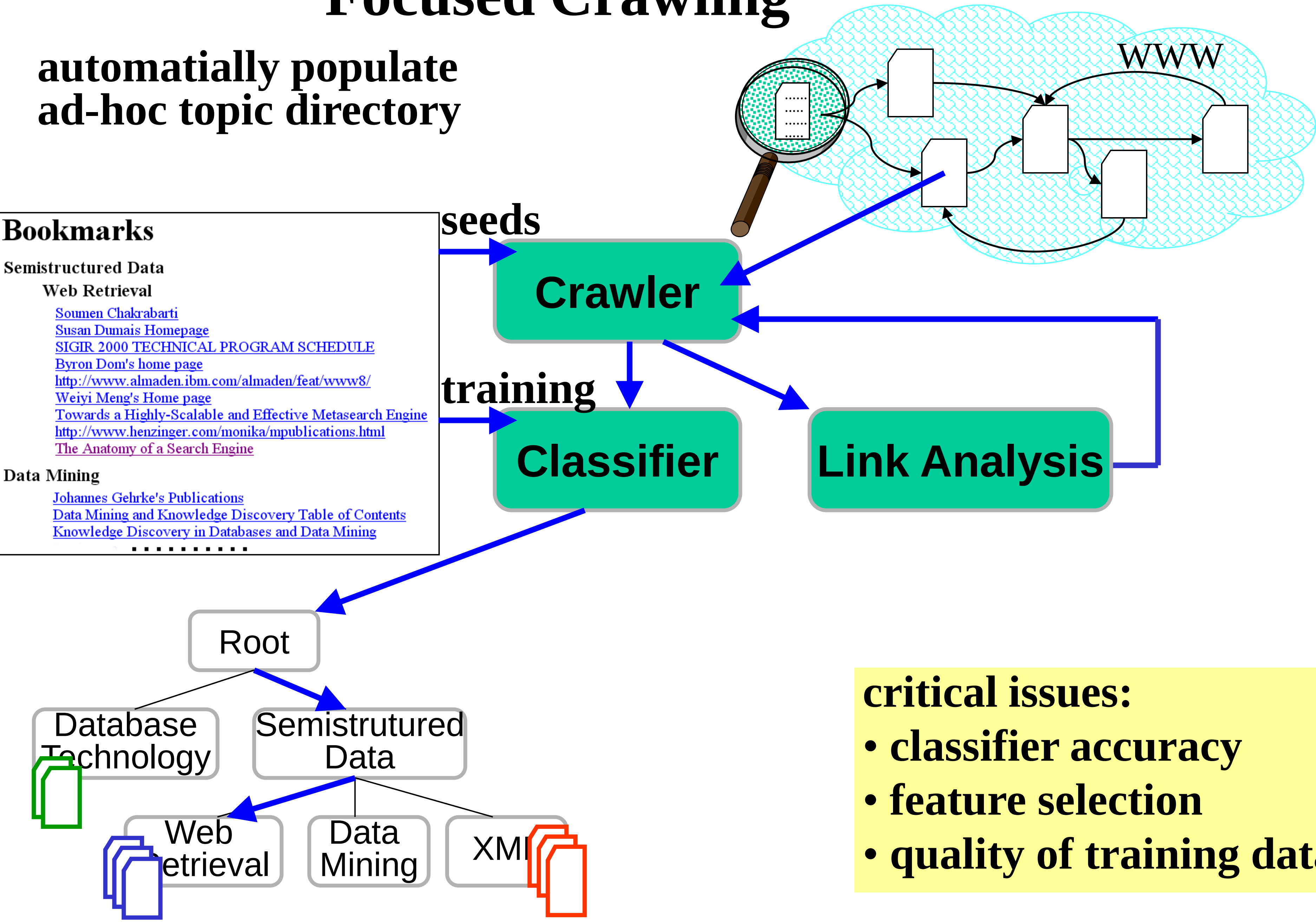
**Deep Web** (aka. **Hidden Web**):
DB/CMS content items without URLs
→ generate (valid) values for query form fields
in order to bring items to surface


Source: http://deepwebtechblog.com/wringing-science-from-google



# Focused Crawling

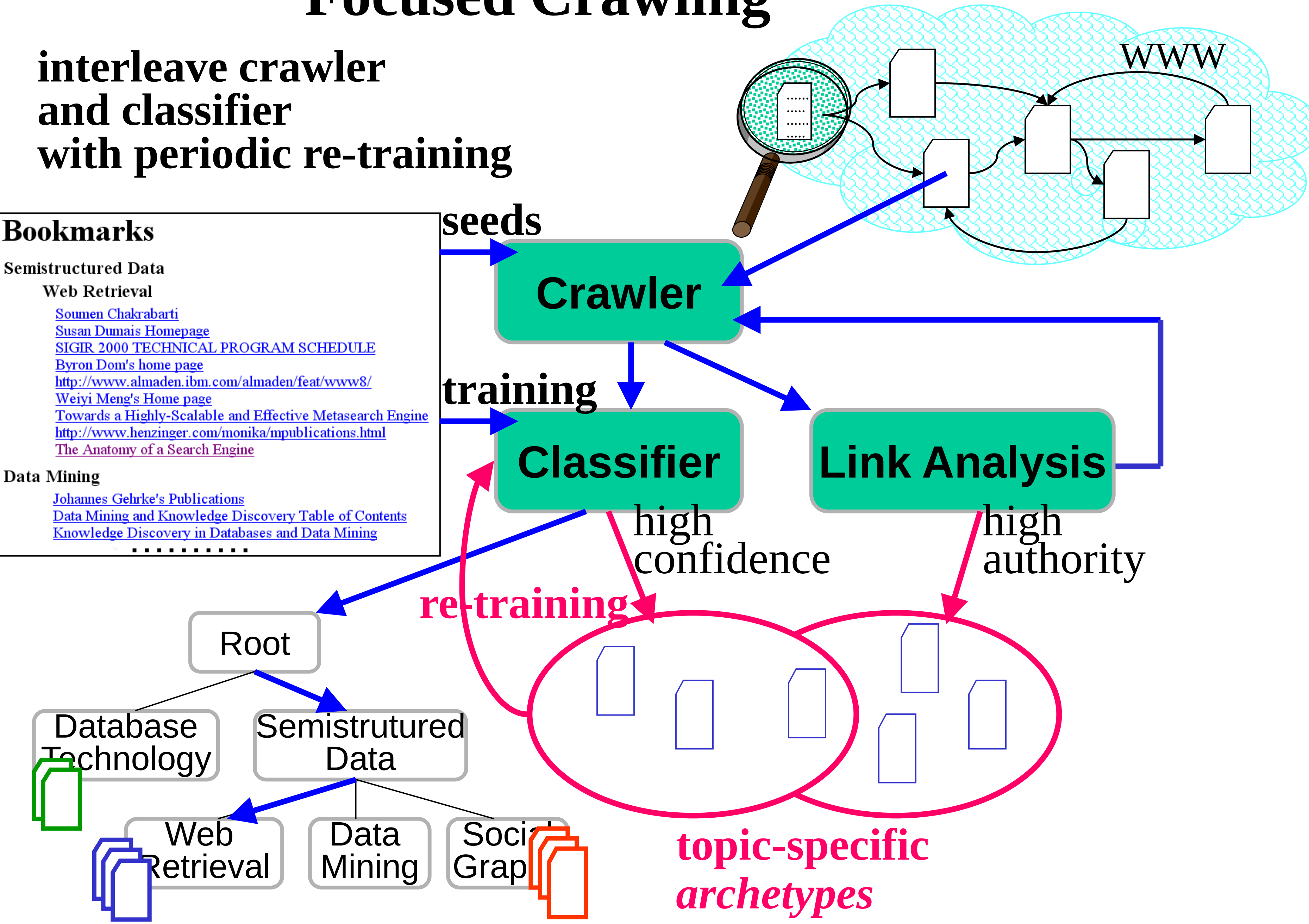
**automatially populate ad-hoc topic directory**




**critical issues:**
- **classifier accuracy**
- **feature selection**
- **quality of training data**

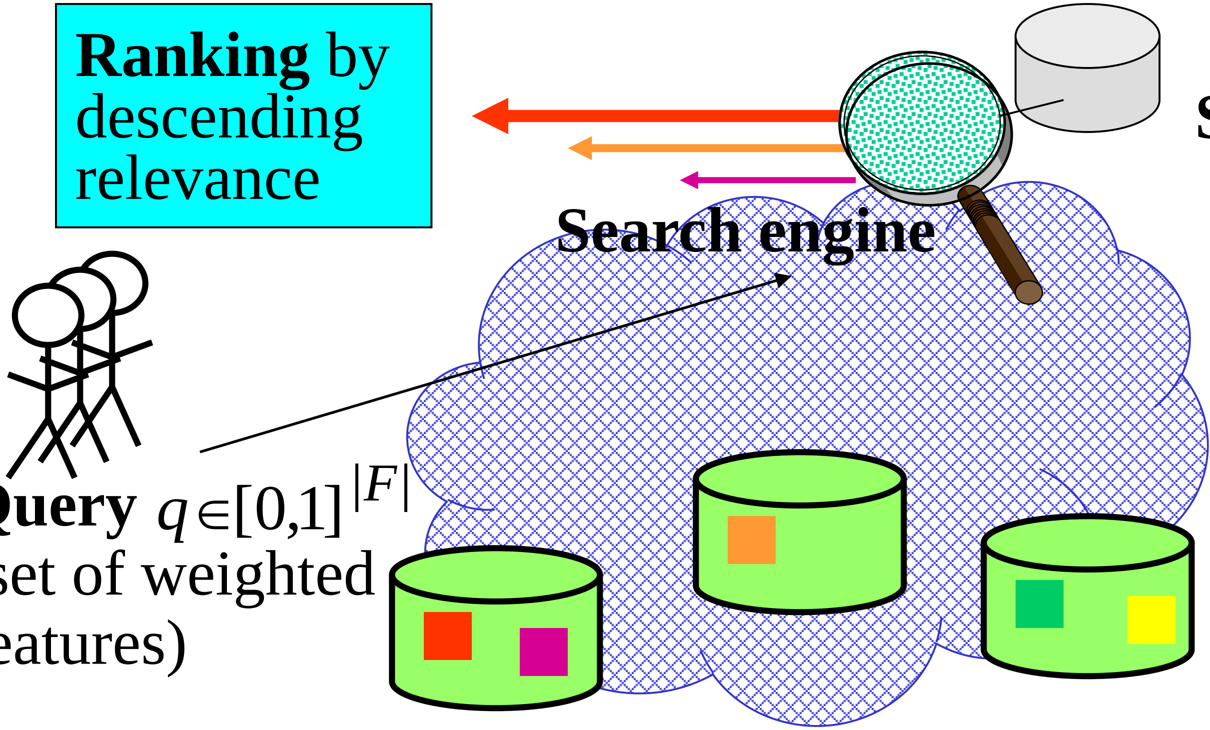
# Focused Crawling

**interleave crawler and classifier with periodic re-training**



# Vector Space Model for Content Relevance Ranking

**Ranking** by descending relevance




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

**Similarity metric:**

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

Documents are **feature vectors (bags of words)**

$d_i \in [0,1]^{|F|}$

e.g. weights by tf*idf model

Features are **terms** (words and other tokens)
or term-zone pairs (term in title/heading/caption/…)
can be stemmed/lemmatized (e.g. to unify singular and plural)
can also be multi-word phrases (e.g. bigrams)

# Vector Space Model: tf*idf Scores

tf ($d_i$, $t_j$) = term frequency of term $t_j$ in doc $d_i$
df ($t_j$) = document frequency of $t_j$ = #docs with $t_j$
idf ($t_j$) = N / df($t_j$) with corpus size (total #docs) N
dl ($d_i$) = doc length of $d_i$ (avgdl: avg. doc length over all N docs)

**tf*idf score** for single-term query (**index weight**):

$$d_{ij} = (1 + \ln(1 + \ln(tf(d_i, t_j)))) \cdot \ln \frac{1+N}{df(t_j)}$$ for tf($d_i$,$t_j$)>0, 0 else

plus optional length normalization

dampening & normalization

**cosine similarity** for ranking
(cosine of angle between q and d vectors
when vectors are L2-normalized):

$$sim(q, d_i) = \sum_j q_j \cdot d_{ij} = \sum_{j \in q \cap d_i} q_j \cdot d_{ij}$$ where $j \in q \cap d_i$ if $q_j \neq 0 \wedge d_{ij} \neq 0$

sparse scalar product

# (Many) tf*idf Variants: Pivoted tf*idf Scores

tf ($d_i$, $t_j$) = term frequency of term $t_j$ in doc $d_i$
df ($t_j$) = document frequency of $t_j$ = #docs with $t_j$
idf ($t_j$) = N / df($t_j$) with corpus size (total #docs) N
dl ($d_i$) = doc length of $d_i$ (avgdl: avg. doc length over all N docs)

**tf*idf score** for single-term query (**index weight**):

$$d_{ij} = (1 + \ln(1 + \ln(tf(d_i, t_j)))) \cdot \ln \frac{1+N}{df(t_j)}$$ for tf($d_i$,$t_j$)>0, 0 else

**pivoted tf*idf score:**

$$d_{ij} = \frac{1 + ln(1 + ln(tf(d_i, t_j)))}{(1-s) + s\,\frac{dl(d_i)}{avgdl}} \cdot ln \frac{1+N}{df(t_j)}$$

avoids undue favoring of long docs

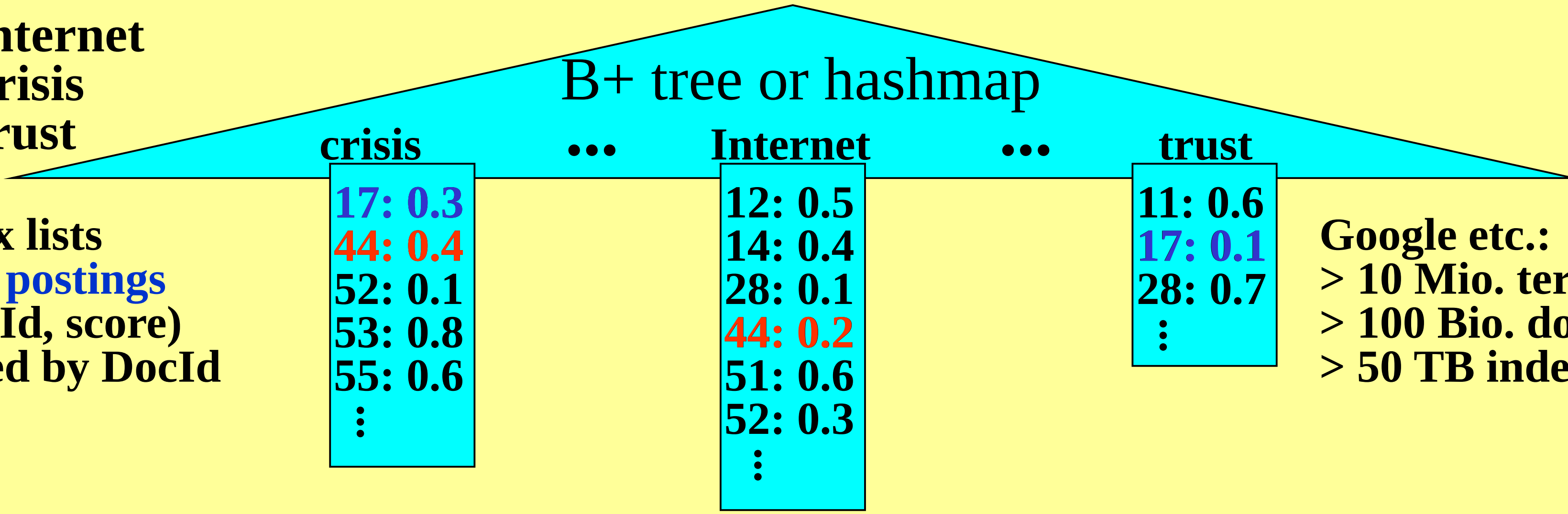
also uses scalar product for score aggregation

tf*idf scoring often works very well, but it has many ad-hoc tuning issues
→ Chapter 13: more principled ranking models

# 11.2 Indexing with Inverted Lists

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

**q: Internet**
**crisis**
**trust**




**index lists with postings (DocId, score) sorted by DocId**

**Google etc.:**
**> 10 Mio. terms**
**> 100 Bio. docs**
**> 50 TB index**

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

- index-list entries in **DocId order** for fast Boolean operations
- many techniques for excellent **compression** of index lists
- additional **position index** needed for phrases, proximity, etc.
  (or other precomputed data structures)

# Dictionary

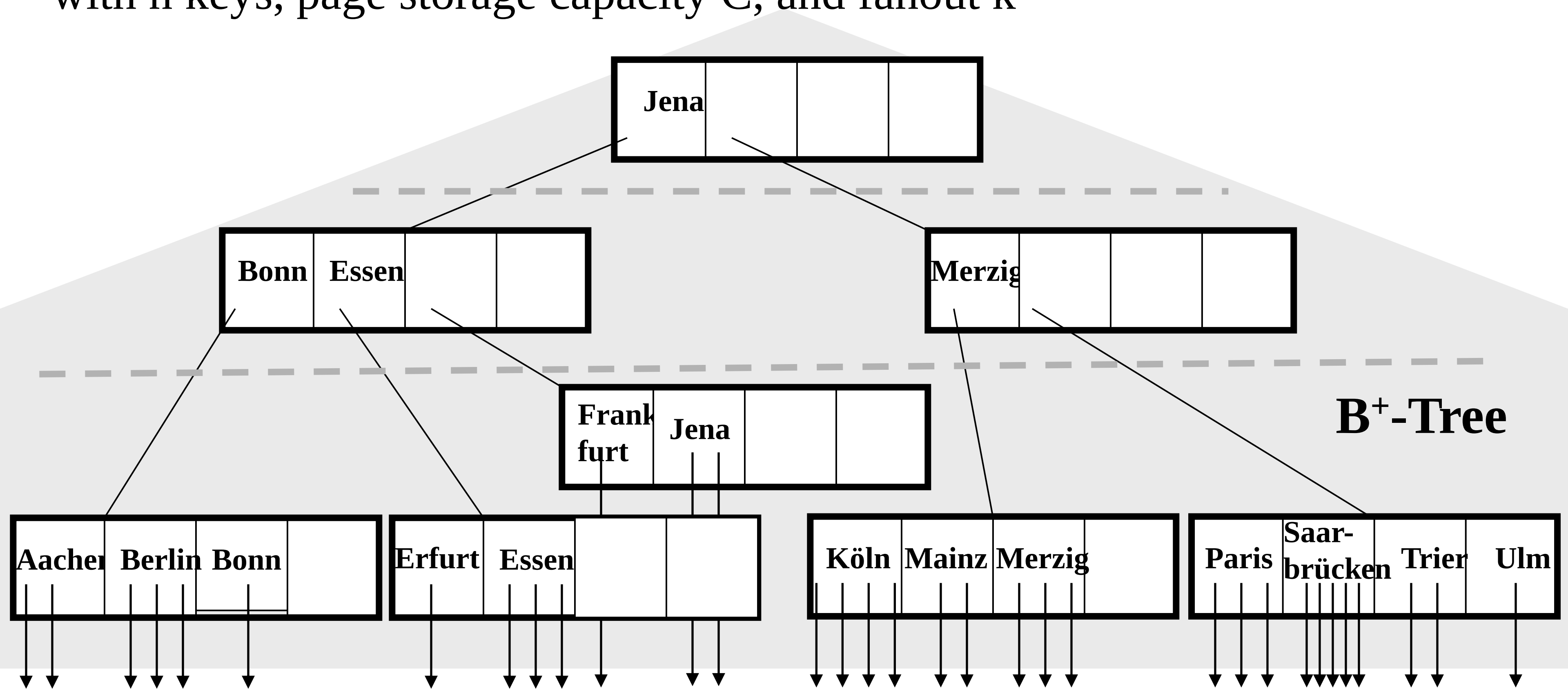
- Dictionary maintains information about terms:
  - mapping terms to unique term identifiers (e.g. *crisis* → 3141359)
  - location of corresponding posting list on disk or in memory
  - statistics such as document frequency and collection frequency

- Operations supported by the dictionary:
  - Lookups by term
  - range searches for prefix and suffix queries (e.g. *net**, **net*)
  - substring matching for wildcard queries (e.g. *cris*s*)
  - Lookups by term identifier

- Typical implementations:
  - B+ trees, hash tables, tries (digital trees), suffix arrays

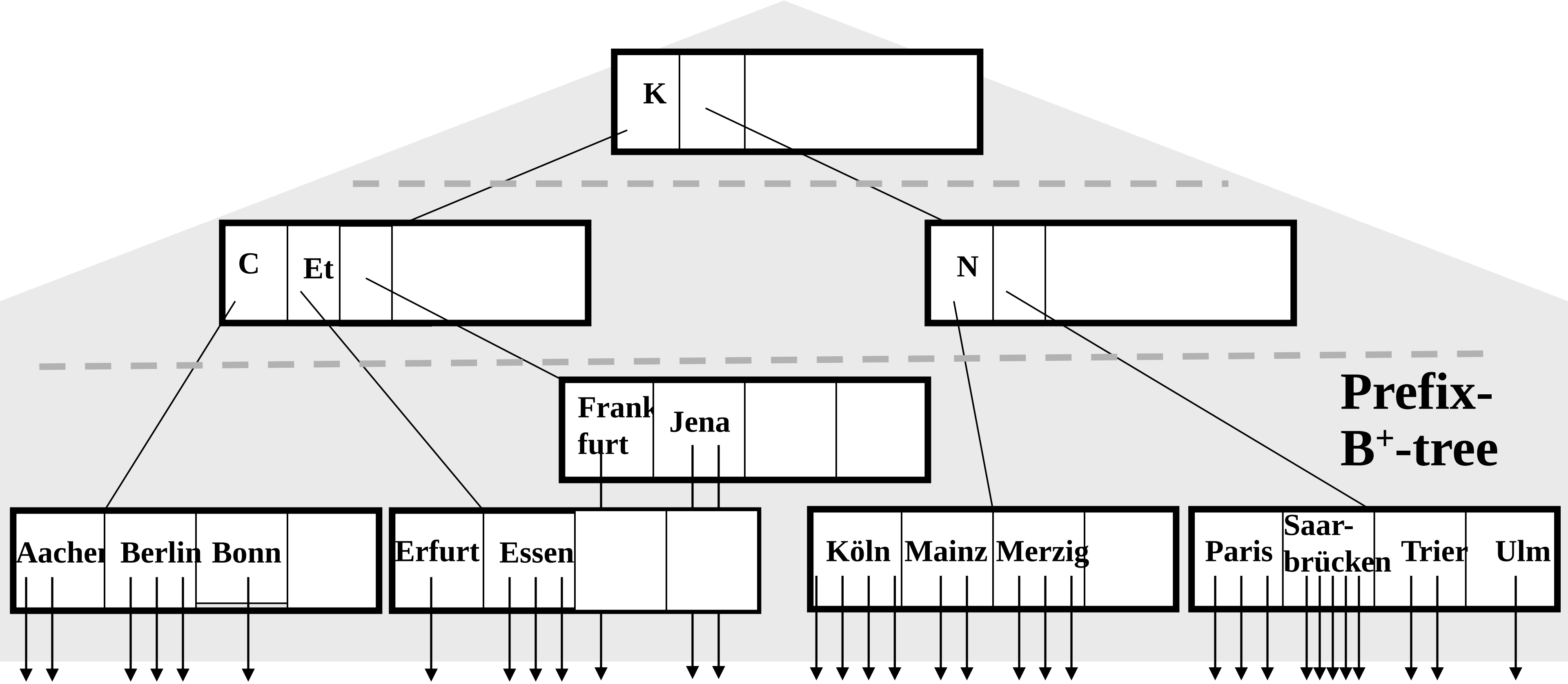
# $B^+$ Tree

- Paginated hollow multiway search tree with high fanout ($\Rightarrow$ low depth)
- Node contents: (child pointer, key) pairs as routers in inner nodes
  key with id list or record data in leaf nodes
- Perfectly balanced: all leaves have identical distance to root
- Search and update efficiency: $O(\log_k n/C)$ page accesses (disk I/Os) with n keys, page storage capacity C, and fanout k



# Prefix B$^+$ Tree for Keys of Type String

Keys in inner nodes are mere **Routers** for search space partitioning.
Rather than $x_i$ = max{s: s is a key in subtree $t_i$} a shorter router
$y_i$ with $s_i \le y_i < x_{i+1}$ for all $s_i$ in $t_i$ and all $s_{i+1}$ in $t_{i+1}$
is sufficient, for example, yi = shortest string with the above property.
→ even higher fanout, possibly lower depth of the tree



# Posting Lists and Payload

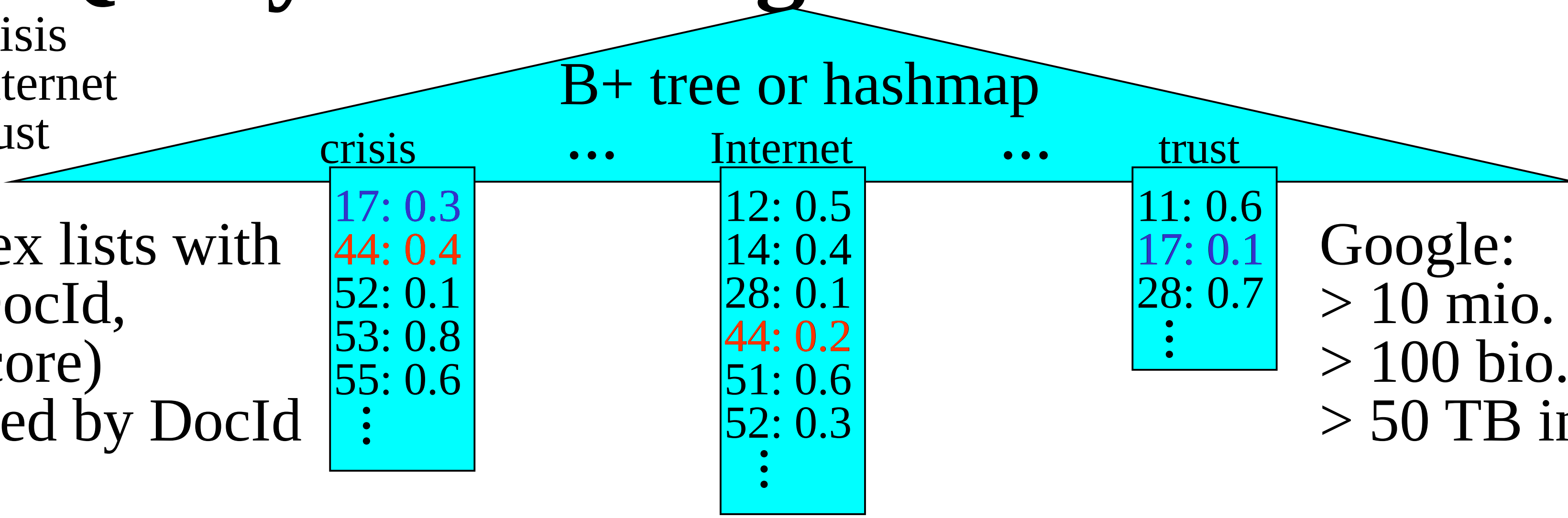
- Inverted index keeps a **posting list** for each term
  with the following **payload** for each posting:
  - **document identifier** (e.g. $d_{123}$, $d_{234}$, …)
  - **term frequency** (e.g. $tf(crisis, d_{123}) = 2$, $tf(crisis, d_{234}) = 4$)
  - **score impact** (e.g. $tf(crisis, d_{123}) * idf(crisis) = 3.75$)
  - **offsets**: positions at which the term occurs in document
- Posting lists can be **sorted by doc id** or **sorted by score impact**
- Posting lists are compressed for space and time efficiency

posting list for *crisis* $\boxed{d_{123}, 2, [4, 14]}$ $\boxed{d_{234}, 4, [47]}$ $\boxed{d_{266}, 3, [1, 9, 20]}$

$$\underbrace{d_{123}, \underbrace{2, [4, 14]}_{\text{payload: tf, offsets}}}_{\text{posting}}$$

# Query Processing on Inverted Lists

q: crisis
Internet
trust




index lists with (DocId, score) sorted by DocId

Google:
> 10 mio. terms
> 100 bio. docs
> 50 TB index

Given: query $q = t_1\ t_2 \dots t_z$ with z (conjunctive) keywords
similarity scoring function 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. score(q,d) $= aggr\{s_i(d)\}$ (e.g.: $\Sigma_{i \in q}\ s_i(d)$)

Merge Algorithm:

- merge lists for $t_1\ t_2 \dots t_z$
- compute score for each document
- keep top-k results with highest scores (in priority queue or after sort by score)

# Index List Processing by Merge Join

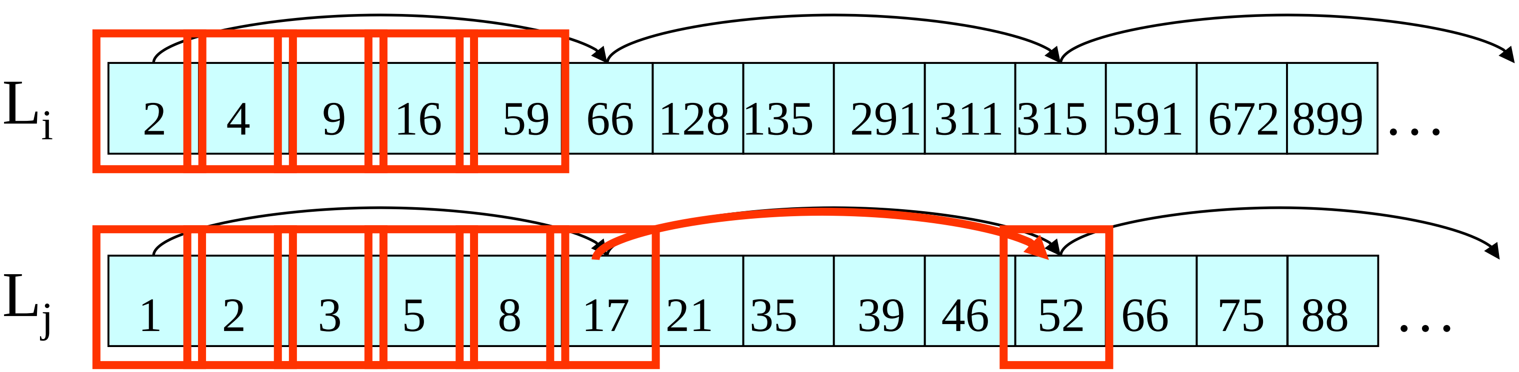
Keep L(i) in **ascending order of doc ids**

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



# Different Query Types

**conjunctive** queries:
all words in q = q1 … qk required

**disjunctive** („andish") queries:
subset of q words qualifies,
more of q yields higher score

**mixed-mode** queries and **negations**:
q = q1 q2 q3 +q4 +q5 –q6

**Keyword queries**:
all by list processing
on inverted indexes

incl. variant:
- scan & merge
  only subset of qi lists
- lookup long
  or negated qi lists

**phrase** queries and **proximity** queries:
q = "q1 q2 q3" q4 q5 …

fuzzy queries: **similarity search**
e.g. with tolerance to spelling variants

see 11.4

# Forward Index

Forward index maintains information about documents

- compact representation of content:
  **sequence of term identifiers** and document length

Forward index can be used for various tasks incl.:

- **result-snippet generation** (i.e., show context of query terms)
- computation of **proximity scores** for advanced ranking
  (e.g. width of smallest window that contains all query terms)

$d_{123}$: *the giants played a fantastic season. it is not clear …*

$d_{123}$ **dl**:428 **content**:< 1, 222, 127, 3, 897, 233, 0, 12, 6, 7, … >

# Index Construction and Updates

## Index construction:

- extract (docId, termId, score) triples from docs
  - can be partitioned & parallelized
  - scores need idf (estimates)
- sort triples by termId (primary) and docId (secondary)
  - disk-based merge sort (build runs, write to temp, merge runs)
  - can be partitioned & parallelized
- load index from sorted file(s), using large batches for disk I/O

## Index updating:

- collect batches of updates in separate files
- sort these files and merge them with index lists

# Disk-Based Merge-Sort

1) ***Form runs*** of records, i.e., sorted subsets of the input data:
   - load M consecutive blocks into memory
   - sort them (using Quicksort or Heapsort)
   - write them to temporary disk space

   repeat these steps for all blocks of data

2) ***Merge runs*** (into longer runs):
   - load M blocks from M different runs into memory
   - merge the records from these blocks in sort order
   - write output blocks to temporary disk space
     and load more blocks from runs as needed

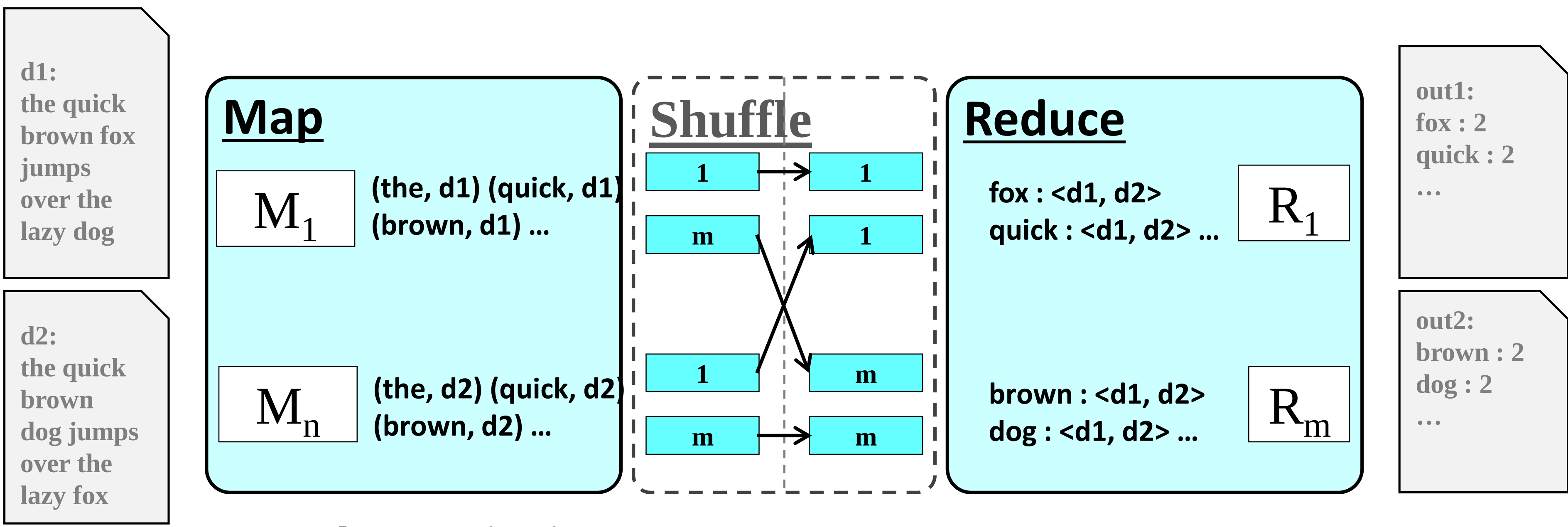
3) ***Iterate*** merge phase
   until only one output run remains

# Map-Reduce Parallelism for Web-Scale Data

[J. Dean et al. 2004, Hadoop, etc.]

Automated Scalable 2-Phase Parallelism (bulk synchronous)

- **map** function: (hash-) partition inputs onto m compute nodes
  local computation, emit (key,value) tuples
- **implicit shuffle**: re-group (key,value) data
- **reduce** function: aggregate (key,value) sets




**Example: counting items**
**(words, phrases, URLs, IP addresses, IP paths, etc.)**
**in Web corpus or traffic/usage log**

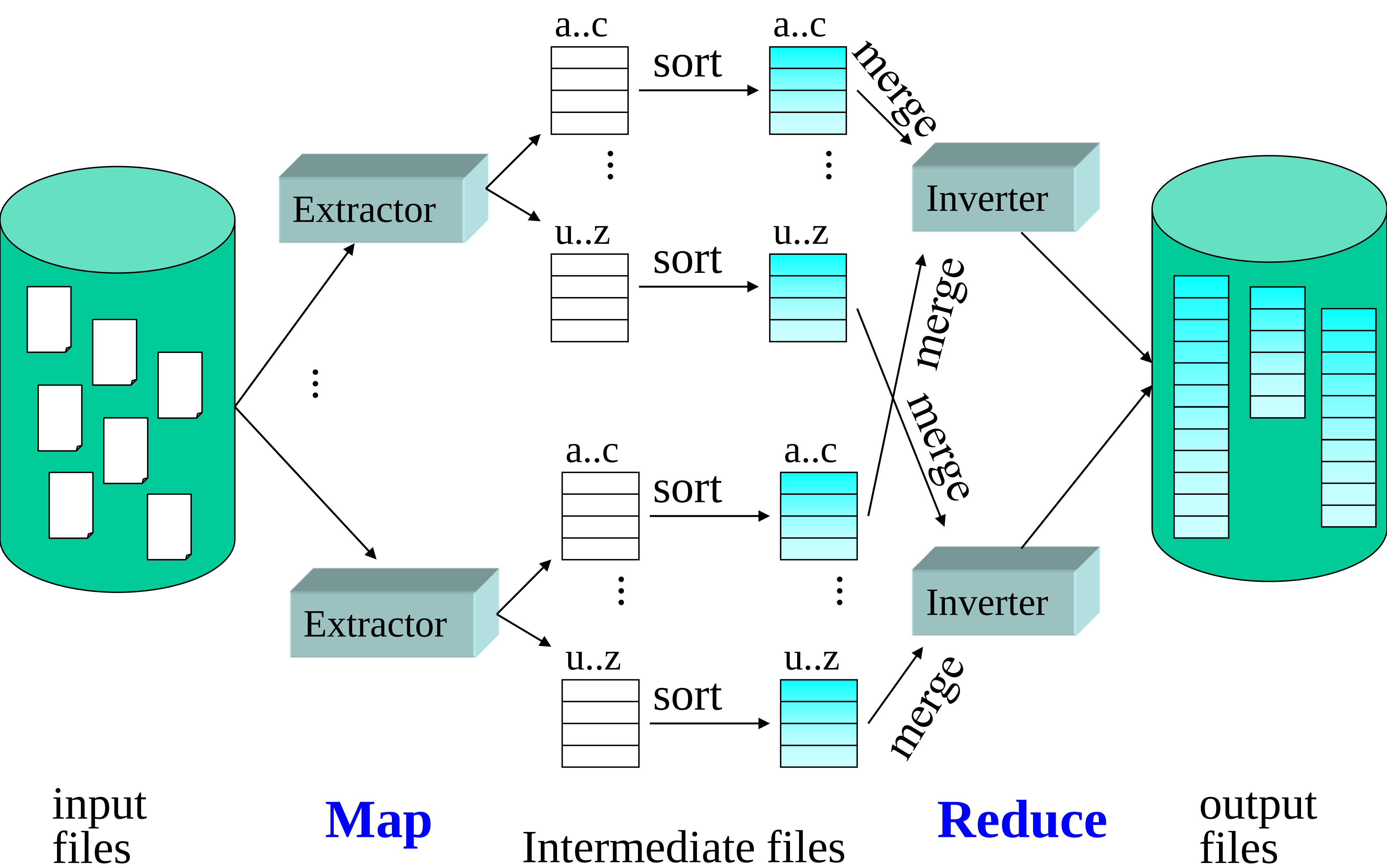
# Map-Reduce Parallelism

Programming paradigm and infrastructure
for scalable, highly parallel data analytics
- can run on 1000's of computers
- with built-in load balancing & fault-tolerance
  (automatic scheduling & restart of worker processes)

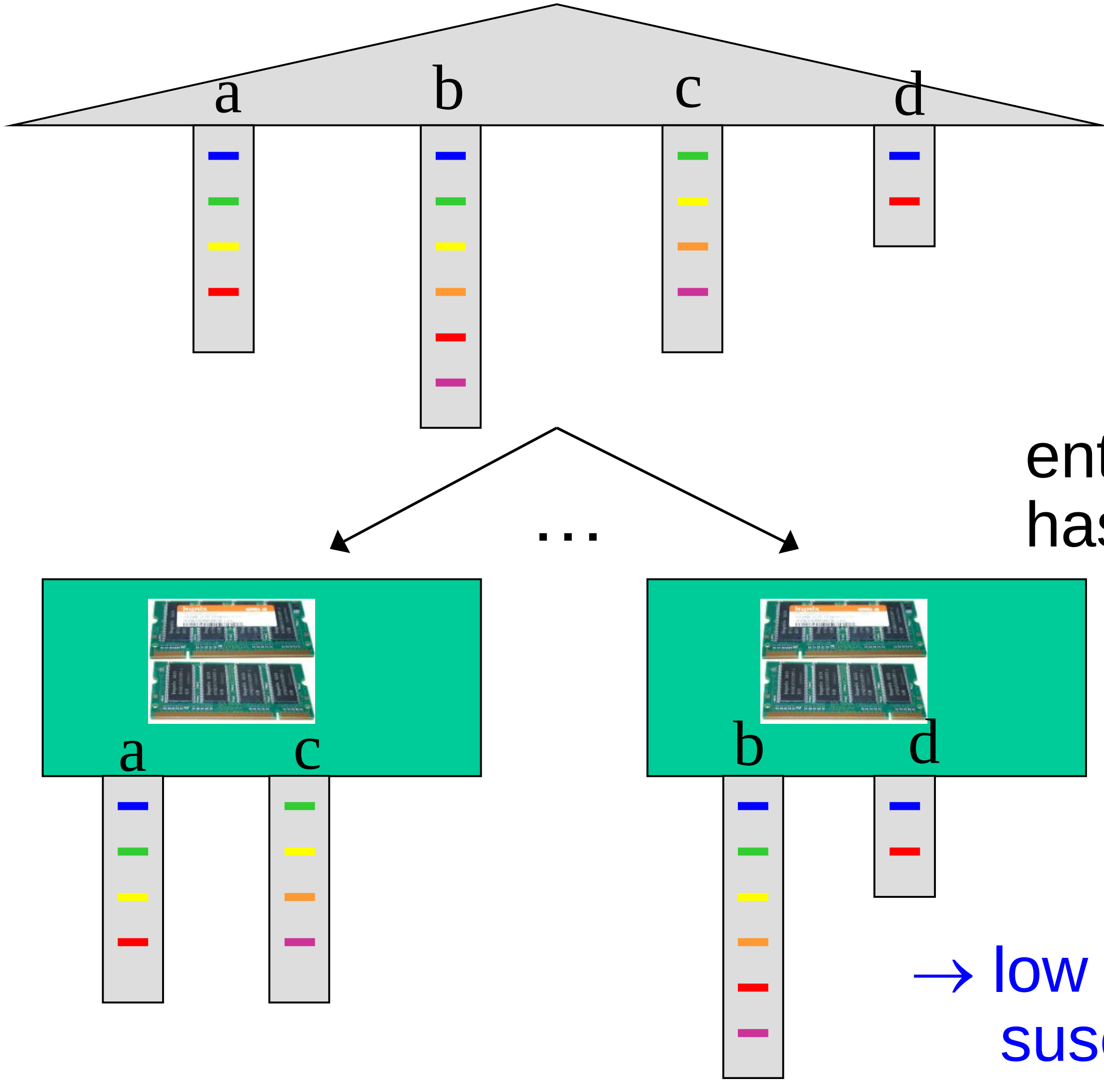
easy programming with key-value pairs:
Map function: $K \times V \rightarrow (L \times W)^*$
$(k1, v1) \mapsto (l1,w1), (l2,w2), \ldots$
Reduce function: $L \times W^* \rightarrow W^*$
$l1, (x1, x2, \ldots) \mapsto y1, y2, \ldots$

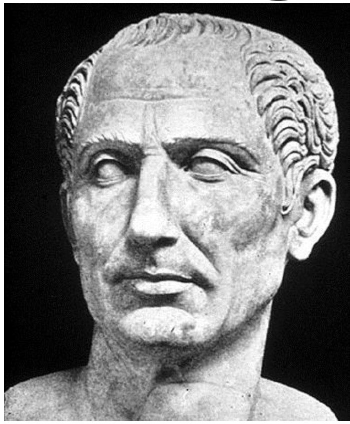
Examples:
- index building: K=docIds, V= contents, L=termIds, W=docIds
- click log analysis: K=logs, V=clicks, L=URLs, W=counts
- web graph reversal: K=docIds, V=(s,t) outlinks, L=t, W=(t,s) inlinks

# Map-Reduce Parallelism for Index Building



# Distributed Indexing: Term Partitioning



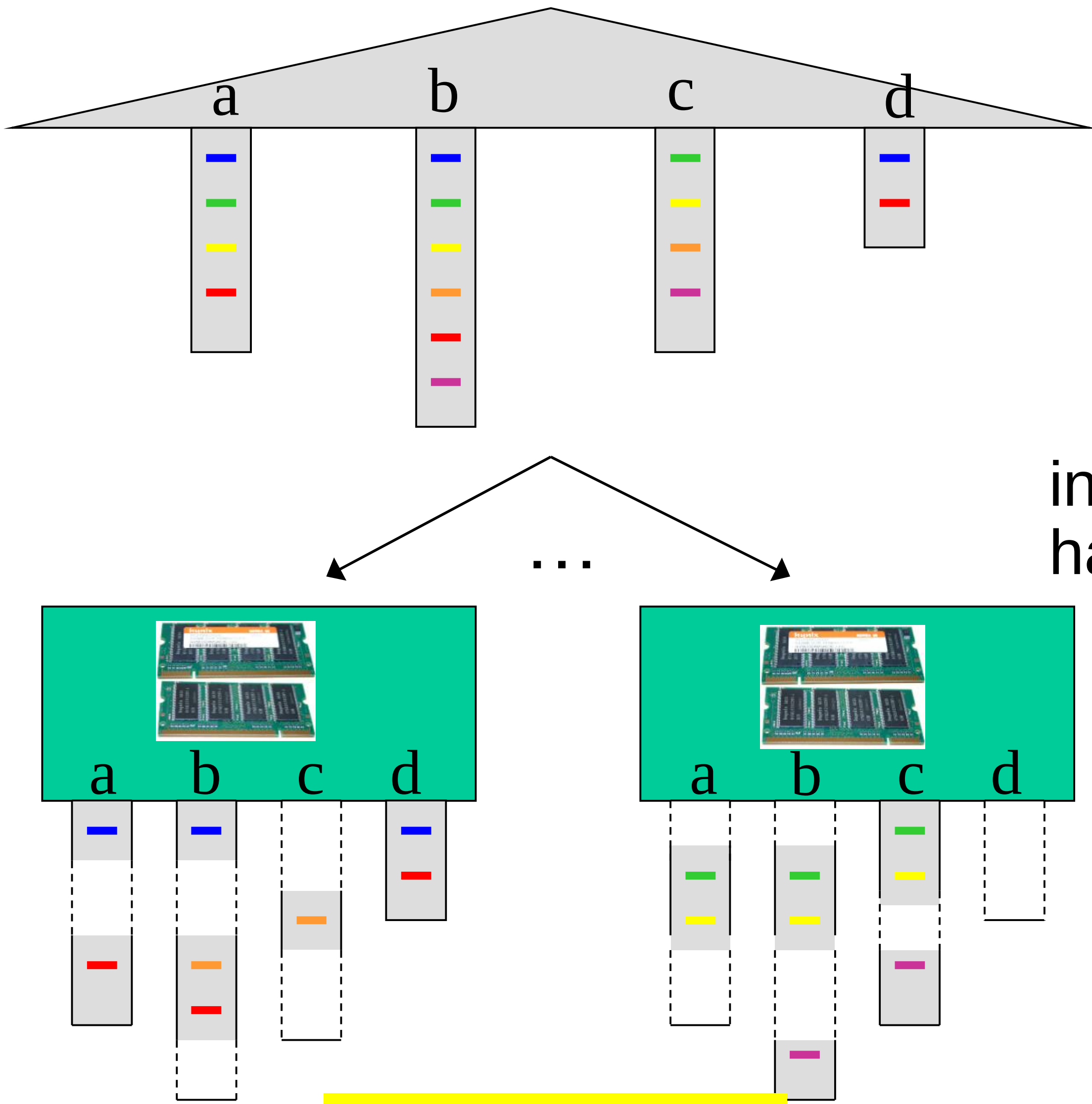



entire index lists are hashed onto nodes by TermId

queries are routed to nodes with relevant terms

→ low resource consumption, susceptible to imbalance (because of data or load skew), index maintenance non-trivial

# Distributed Indexing: Doc Partitioning




index-list entries are hashed onto nodes by DocId

each complete query is run on each node; results are merged

→ perfect load balance, embarrasingly scalable, easy maintenance

**Index Sharding**

# Dynamic Indexing

News, tweets, social media
require the index to be always fresh

- New postings are **incrementally inserted** into inverted lists
  - avoid insertion in middle of long list:

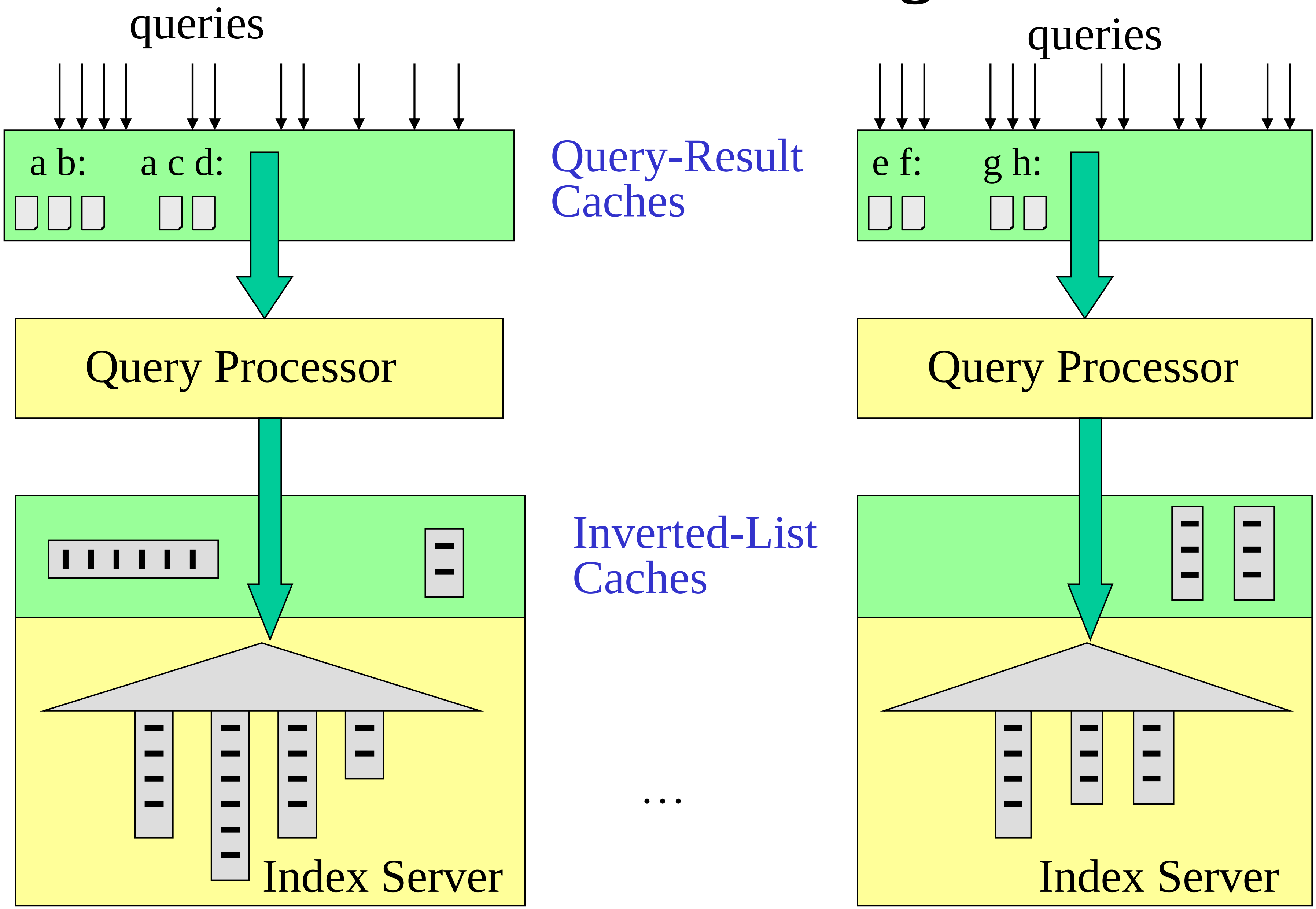
    **partition long lists**, insert in / append to partition,

    merge partitions lazily
- Index **updates in parallel to queries**
  - Light-weight locking needed to ensure **consistent reads**

    (and consistency of index with parallel updates)

More detail see e.g. Google Percolator (Peng/Dabek: OSDI 2010)

# Index Caching



# Caching Strategies

What is cached?

- index lists for individual terms
- entire query results
- postings for multi-term intersections

Where is an item cached?

- in RAM of responsible server-farm node
- in front-end accelerators or proxy servers
- as replicas in RAM of all (or many) servers

When are cached items dropped?

- estimate for each item: temperature = access-rate / size
- when space is needed, drop item with lowest temperature
  Landlord algorithm [Cao/Irani 1997, Young 1998], generalizes LRU-k [O'Neil 1993]
- prefetch item if its predicted temperature is higher than
  the temperature of the corresponding replacement victims

# 11.3 Index Compression

**Heap‘s law** (empirically observed and postulated):
size of the vocabulary (distinct terms) in a corpus

$$E[\,distinct\ \ terms\ \ in\ \ corpus\,] \approx \alpha \cdot n^{\beta}$$

with total number of term occurrences *n*, and constants $\alpha$, $\beta$ ($\beta < 1$),
classically $\alpha \approx 20$, $\beta \approx 0.5$

**Zipf‘s law** (empirically observed and postulated):
relative frequencies of terms in the corpus

$$\mathrm{P}[\mathrm{k}^{\mathrm{th}} \text{ most popular term has rel. freq. x}] \sim \left(\frac{1}{\mathrm{k}}\right)^{\theta}$$

with parameter $\theta$, classically set to 1

The two laws strongly suggest opportunities for compression

# Compression: Why?

- **reduced space** consumption on disk or in memory (and SSD and L3/L2 CPU caches)
- more **cache hits**, since more postings fit in cache
- 10x to 20x **faster query processing**, since decompressing may often be done as fast as **sequential scan**

# Basics from Information Theory

Let f(x) be the probability (or relative frequency) of the x-th symbol in some text d. The **entropy** of the text (or the underlying prob. distribution f) is:

$$H(d) = \sum_x f(x) \log_2 \frac{1}{f(x)}$$

H(d) is a lower bound for the bits per symbol needed with optimal coding.

For two prob. distributions f(x) and g(x) the **relative entropy (Kullback-Leibler divergence)** of f to g is

$$D(f \| g) := \sum_x f(x) \log_2 \frac{f(x)}{g(x)}$$

relative entropy measures (dis-)similarity of probability or frequency distributions

D is the average number of additional bits for coding events of f when using optimal code for g

**Jensen-Shannon divergence** of f(x) and g(x): $\frac{1}{2} D(f||g) + \frac{1}{2} D(g||f)$

**Cross entropy** of f(x) to g(x):

$$H(f,g) := H(f) + D(f \| g) = -\sum_x f(x) \log g(x)$$

# Compression

- Text is sequence of symbols (with specific frequencies)
- Symbols can be
  - letters or other characters from some alphabet Σ
  - strings of fixed length (e.g. trigrams)
  - or words, bits, syllables, phrases, etc.

***Limits of compression:***

Let $p_i$ be the probability (or relative frequency) of the i-th symbol in text d

Then the (empirical) ***entropy*** of the text: $H(d) = \sum_i p_i \log_2 \frac{1}{p_i}$

is a ***lower bound*** for the average number of bits per symbol in any compression (e.g. Huffman codes)

Note:

compression schemes such as *Ziv-Lempel* (used in zip)
are better because they consider context beyond single symbols;
with appropriately generalized notions of entropy
the lower-bound theorem does still hold

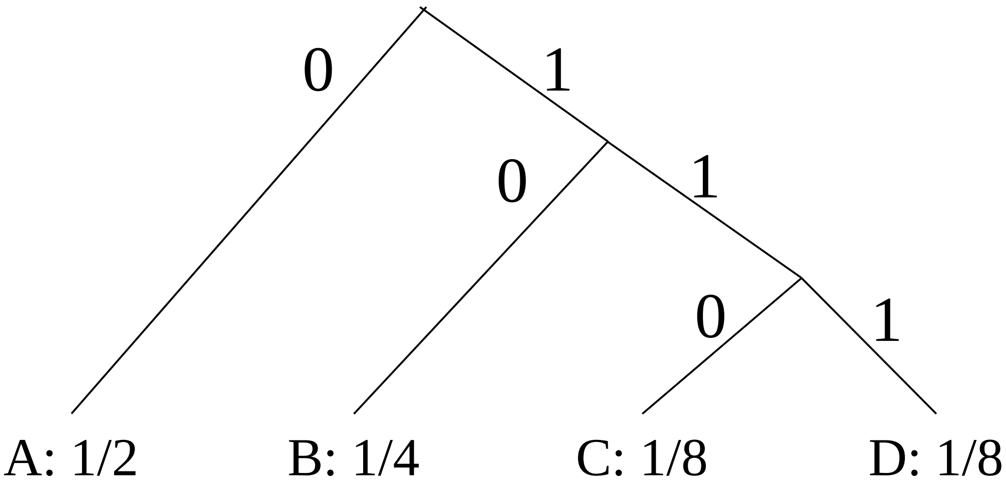
# Basic Compression: Huffman Coding

Text in alphabet $\Sigma = \{A, B, C, D\}$
$P[A] = 1/2$, $P[B] = 1/4$, $P[C] = 1/8$, $P[D] = 1/8$

$H(\Sigma) = 1/2*1 + 1/4*2 + 1/8*3 + 1/8*3 = 7/4$

Optimal (prefix-free) code from **Huffman tree**:

$A \to 0$
$B \to 10$
$C \to 110$
$D \to 111$




Avg. code length: 0.5*1 + 0.25*2 +2* 0.125*3 = 1.75 bits

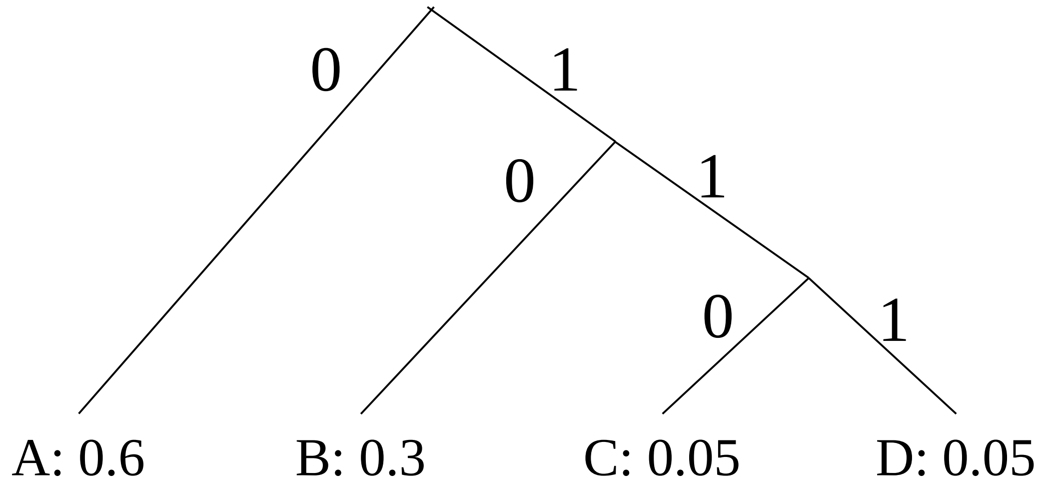
# Basic Compression: Huffman Coding

Text in alphabet $\Sigma = \{A, B, C, D\}$
$P[A] = 0.6$, $P[B] = 0.3$, $P[C] = 0.05$, $P[D] = 0.05$

$$H(\Sigma) = 0.6*\log\frac{10}{6} + 0.3*\log\frac{10}{3} + 0.05*\log 20 + 0.05*\log 20 \approx 1.394$$

Optimal (prefix-free) code from **Huffman tree**:

$A \rightarrow 0$
$B \rightarrow 10$
$C \rightarrow 110$
$D \rightarrow 111$




Avg. code length: $0.6*1 + 0.3*2 + 0.05*3 + 0.05*3 = 1.5$ bits

# Algorithm for Computing a Huffman Code

```
n := |Σ|
priority queue Q := Σ sorted in ascending order by  p(s) for s∈Σ
for i:=1 to n-1 do
   z := MakeTreeNode( )
   z.left := ExtractMin(Q)
   z.right := ExtractMin(Q)
   p(z) := p(z.left) + p(z.right)
   Insert (Q, z)
od
return ExtractMin(Q)
```

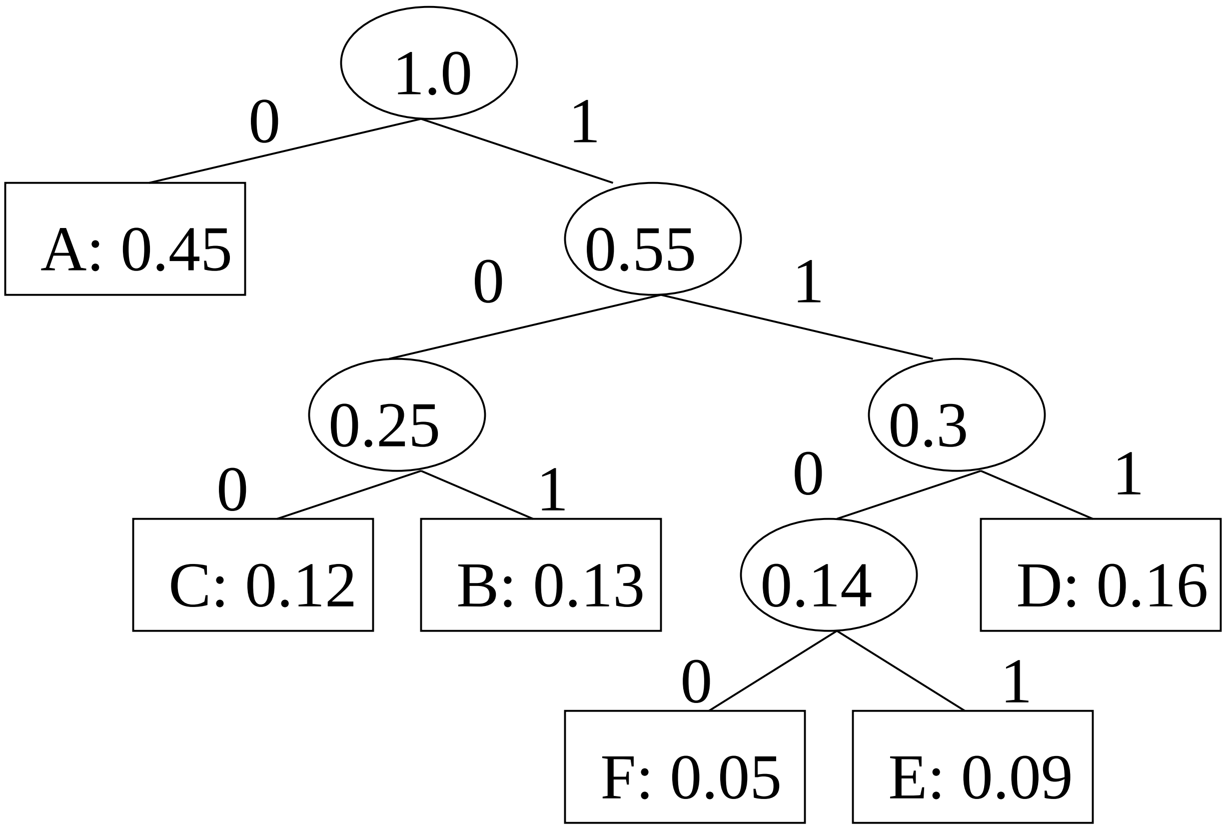
Theorem: The Huffman code constructed with this algorithm is an optimal prefix-free code.

Remark:
Huffmann codes need to scan a text twice for compression (or need other sources of text-independent symbol statistics)

# Example: Huffman Coding

Example:
$|\Sigma|=6$, $\Sigma=\{a,b,c,d,e,f\}$,
P[A]=0.45, P[B]=0.13, P[C]=0.12, P[D]=0.16, P[E]=0.09, P[F]=0.05

A $\rightarrow$ 0
B $\rightarrow$ 101
C $\rightarrow$ 100
D $\rightarrow$ 111
E $\rightarrow$ 1101
F $\rightarrow$ 1100



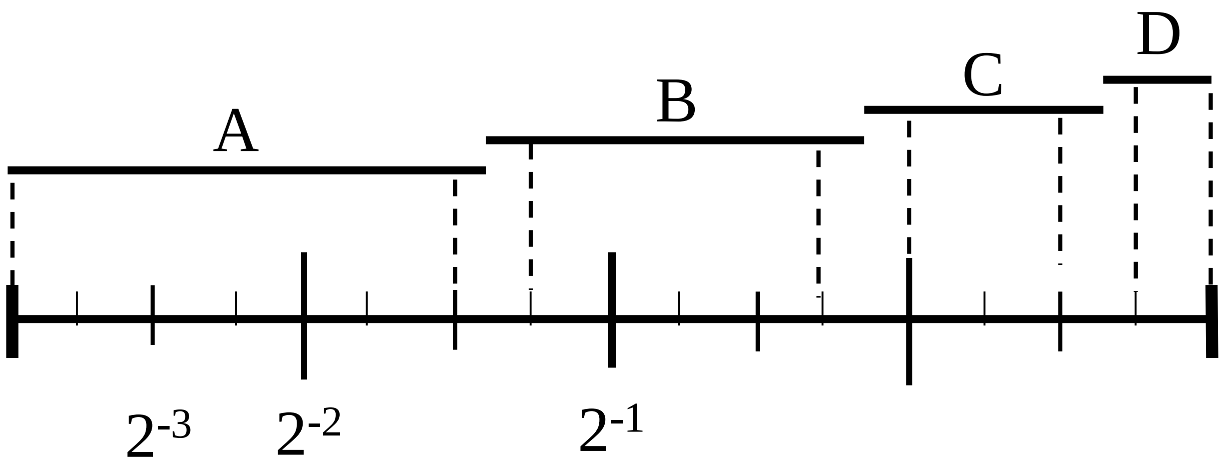
# Arithmetic Coding

Generalizes Huffman coding

Key idea: for alphabet $\Sigma$ and probabilities P[s] of symbols $s \in \Sigma$

- Map s to an interval of real numbers in [0,1] using the cdf values of the symbols and encode the interval boundaries
- Choose sums of negative powers of 2 as interval boundaries

Example: $\Sigma=\{A,B,C,D\}$ with P[A]=0.4, P[B]=0.3, P[C]=0.2, P[D]=0.1
$\rightarrow$ F(A)=0.4, F(B)=0.7, F(C)=0.9, F(D)=1.0




Encode symbol (or symbol sequence) by a binary interval contained in the symbol‘s interval

# General Text Compression: Ziv-Lempel

**LZ77 (Adaptive Dictionary)** and further variants:

- scan text & identify in a ***lookahead window*** the longest string that occurs repeatedly and is contained in a ***backward window***
- replace this string by a „pointer“ to its previous occurrence.

encode text into list of triples ***<back, count, new>*** where

- ***back*** is the backward distance to a prior occurrence of the string that starts at the current position,
- ***count*** is the length of this repeated string, and
- ***new*** **i**s the next symbol that follows the repeated string.

triples themselves can be further encoded (with variable length)

better variants use explicit dictionary with statistical analysis (need to scan text twice)
and/or clever permutation of input string → Burrows-Wheeler transform

# Example: Ziv-Lempel Compression

peter_piper_picked_a_peck_of_pickled_peppers

**<back, count, new>**

| | | |
|---|---|---|
| <0, 0, p> | for character 1: | p |
| <0, 0, e> | for character 2: | e |
| <0, 0, t> | for character 3: | t |
| <-2, 1, r> | for characters 4-5: | er |
| <0, 0, _> | for character 6: | _ |
| <-6, 1, i> | for characters 7-8: | pi |
| <-8, 2, r> | for characters 9-11: | per |
| <-6, 3, c> | for charaters 12-13: | _pic |
| <0, 0, k> | for character 16 | k |
| <-7,1,d> | for characters 17-18 | ed |

...

great for text compression, but not easy to use with index lists

# Index Compression

Posting lists with ordered doc ids have small gaps
→ **gap coding**: represent list by first id and sequence of gaps
gaps in long lists are small, gaps in short lists long
→ **variable bit length coding**
good for doc ids and offets in payload

Other lists may have many identical or consecutive values
→ **run-length coding**: represent list by first value and frequency of repeated or consecutive values

# Gap Compression: Gamma Coding

Encode **gaps** in inverted lists (successive doc ids), often small integers

**Unary coding**:
gap of size x encoded by:
x times 0 followed by one 1
(x+1 bits)

good for short gaps

**Binary coding:**
gap of size x encoded by
binary representation of number x
($\log_2$ x bits)

good for long gaps

**Elias's γ coding:**
*length*:= floor($\log_2$ x) in unary, followed by
*offset* := x – 2**(floor($\log_2$ x)) in binary
(1 + $\log_2$ x + $\log_2$ x bits)

→ generalization: Golomb code (optimal for geometr. distr. of x)
→ still need to pack variable-length codes into bytes or words

# Example for Gamma Coding

| x | length (unary) | offset (binary) |
|---|---|---|
| $1 = 2^0$ | 1 | 1 |
| $4 = 2^2$ | 001 | 100 |
| $17 = 2^4+2^0$ | 00001 | 10001 |
| $24=2^4+2^3$ | 00001 | 11000 |
| $63=2^5+\ldots$ | 000001 | 111111 |
| $64=2^6$ | 0000001 | 1000000 |

leading 1
can be omitted

Note 1: as there are no gaps of size x=0, one typically encodes x-1
Note 2: a variant called δ coding uses γ encoding for the length

# Byte or Word Alignment and Variable Byte Coding

Variable bit codes are typically aligned
to start on byte or word boundaries
→ some bits per byte or word may be unused (extra 0‘s “padded“)

**Variable byte coding** uses only 7 bits per byte,
the first (i.e. most significant) bit is a continuation flag
→ tells which consecutive bytes form one logical unit

Example: var-byte coding of gamma encoded numbers:

| | | | |
|---|---|---|---|
| 1 0000000 | 1 0100101 | 0 1000000 | 0 0011000 |

# Golomb Coding / Rice Coding

**Colomb coding** generalizes Gamma coding:

for tunable parameter M (modulus), split x into

- **quotient** $q = floor(x/M)$ – stored in unary code with q+1 bits
- **remainder** $r = x \bmod M$ – stored in binary code with $ceil(\log_2 r)$ bits

let $b = ceil(\log_2 M)$ → remainder needs either b or b-1 bits

can be further optimized to use b-1 bits for the smaller numbers:

If $r < 2^b - M$ then r is stored with b-1 bits

If $r \geq 2^b - M$ then $r+2^b-M$ is stored with b bits

**Rice coding** specializes Golomb coding to choice $M = 2^k$

→ processing of encoded numbers can exploit bit-level operations

# Example for Golomb Coding

***Golomb encoding (M=10, b=4): simple variant***

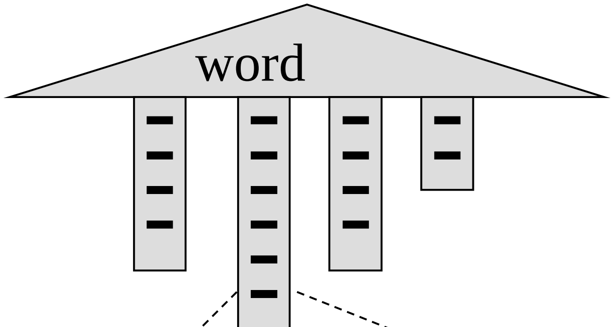
| *x* | *q* | *bits(q)* | *r* | *bits(r)* |
|---|---|---|---|---|
| *0* | *0* | *1* | *0* | *0000* |
| *33* | *3* | *0001* | *3* | *0011* |
| *57* | *5* | *000001* | *7* | *0111* |
| *99* | *9* | *0000000001* | *9* | *1001* |

***Golomb encoding (M=10, b=4) with additional optimization***

| *x* | *q* | *bits(q)* | *r* | *bits(r)* |
|---|---|---|---|---|
| *0* | *0* | *1* | *0* | *000* |
| *33* | *3* | *0001* | *3* | *011* |
| *57* | *5* | *000001* | *7* | *1101* |
| *99* | *9* | *0000000001* | *9* | *1111* |

# Practical Index Compression: Layout of Index Postings

[Jeff Dean (Google): WSDM'09]




word | skip table | block 1 | … | block N

one block
(with n postings):

| | |
|---|---|
| delta to last docId in block | header |
| #docs in block: n … | header |
| n-1 docId deltas: $\text{Rice}_k$ encoded | postings |
| n values tf: Gamma encoded | payload (of postings) |
| tf attributes: Huffman encoded | payload (of postings) |
| tf positions: Huffman encoded | payload (of postings) |

layout allows incremental decoding

# 11.4 Similarity Search

**Exact Matching:**

- given a string s and a longer string d,
  find (all) occurrences of s in d
  string can be a word or a multi-word phrase
- algorithms include Knuth-Morris-Pratt, Boyer-Moore, …

→ see Algorithms lecture

**Fuzzy Matching:**

- given a string s and a longer string d,
  find (all) approximate occurrences of s in d
  e.g. tolerating missing characters or words, typos, etc.

→ this lecture

# Similarity Search with Edit Distance

Idea:
tolerate mis-spellings and other variations of search terms and score matches based on edit distance

Examples:

1) query: Microsoft
   fuzzy match: Migrosaft
   score ~ edit distance 2
2) query: Microsoft
   fuzzy match: Microsiphon
   score ~ edit distance 3+5
3) query: Microsoft Corporation, Redmond, WA
   fuzzy match at token level: MS Corp., Readmond, USA

# Similarity Measures on Strings (1)

**Hamming distance** of strings s1, s2 $\in \Sigma^*$ with $|s1|=|s2|$:
number of different characters (cardinality of $\{i: s1_i \neq s2_i\}$)

**Levenshtein distance (edit distance)** of strings s1, s2 $\in \Sigma^*$:
minimal number of editing operations on s1
(replacement, deletion, insertion of a character)
to change s1 into s2

For edit (i, j): Levenshtein distance of s1[1..i] and s2[1..j] it holds:

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

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

with $\text{diff}(i, j) = 1$ if $s1_i \neq s2_j$, 0 otherwise

$\rightarrow$ efficient computation by **dynamic programming**

# Example for Levenshtein edit distance:
# *grate[1..i] → great[1..j]*

| | *g* | *r* | *e* | *a* | *t* |
|---|---|---|---|---|---|
| *g* | 0 | 1 | 2 | 3 | 4 |
| *r* | 1 | 0 | 1 | 2 | 3 |
| *a* | 2 | 1 | 1 | 1 | 2 |
| *t* | 3 | 2 | 2 | 2 | 1 |
| *e* | 4 | 3 | 2 | 3 | 2 |

**edit (s[1..i], t[1..j]) = min {**

$\downarrow$ **edit (s[1..i-1], t[1..j]) + 1,**

$\rightarrow$ **edit (s[1..i], t[1..j-1]) + 1,**

$\searrow$ **edit (s[1..i-1], t[1..j-1]) + diff (s[i], t[j] }**

# Similarity Measures on Strings (2)

**Damerau-Levenshtein distance** of strings s1, s2 $\in\Sigma^*$:
minimal number of replacement, insertion, deletion, or **transposition** operations (exchanging two adjacent characters) for changing s1 into s2

For edit (i, j): Damerau-Levenshtein distance of s1[1..i] and s2[1..j] :

$$edit(0, 0) = 0,\ edit(i, 0) = i,\ edit(0, j) = j$$

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

with $diff(i, j) = 1$ if $s1_i \neq s2_j$, 0 otherwise

# Similarity based on N-Grams

Determine for string s the set or bag of its N-Grams:

$G(s)$ = {substrings of s with length N}

(often trigrams are used, i.e. N=3)

**Distance of strings s1 and s2:**

$$|G(s1)| + |G(s2)| - 2|G(s1) \cap G(s2)|$$

<u>Example:</u>

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

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

distance (rodney, rhodnee) = 4 + 5 – 2*2 = 5

**Alternative similarity measures:**

**Jaccard coefficient:** $|G(s1) \cap G(s2)| \, / \, |G(s1) \cup G(s2)|$

**Dice coefficient:** $2 \, |G(s1) \cap G(s2)| \, / \, (|G(s1)| + |G(s2)|)$

# N-Gram Indexing for Similarity Search

<u>Theorem</u> (Jokinen and Ukkonen 1991):
for query string s and a target string t,
the Levenshtein edit distance is bounded by the
N-Gram bag-overlap:

$$edit(s,t) \le d \Rightarrow \left| Ngrams(s) \cap Ngrams(t) \right| \ge |s| - (N-1) - dN$$

$\rightarrow$ for similarity queries with edit-distance tolerance d,
perform query over inverted lists for N-grams,
using count for score aggregation

# Example for Jokinen/Ukkonen Theorem

**edit(s,t) $\leq$ d** $\Rightarrow$ **overlap(s,t) $\geq$ |s| – (N–1) – dN**

**overlap(s,t) < |s| – (N–1) – dN** $\Rightarrow$ **edit(s,t) > d**

s = ababababababa
|s|=11
N=2 $\rightarrow$ Ngrams(s) = {ab(5),ba(5)}
N=3 $\rightarrow$ Ngrams(s) = {aba(5), bab(4)}
N=4 $\rightarrow$ Ngrams(s) = {abab(4), baba(4)}

t1 = ababababab, |t1|=10
t2 = abacdefaba, |t2|=10
t3 = ababaaababa, |t3|=11
t4 = abababb, |t4|=7
t5 = ababaaabbbb, |t5|=11

**task: find all ti with edit(s,ti) $\leq$ 2**

$\rightarrow$ **prune all ti with edit(s,ti) > 2 = d**

$\rightarrow$ **overlapBound = |s| – (N–1) – dN**
**= 6 (for N=2)**

$\rightarrow$ **prune all ti with overlap(s,ti) < 6**

N=2:
Ngrams(t1) = {ab(5),ba(4)}
Ngrams(t2)
= {ab(2),ba(2),ac,cd,de,ef,fa}
Ngrams(t3) =
= {ab(4),ba(4),aa(2)}
Ngrams(t4) = {ab(3),ba(2),bb}
Ngrams(t5)
= {ab3),ba(2),aa(2)bb(3)}

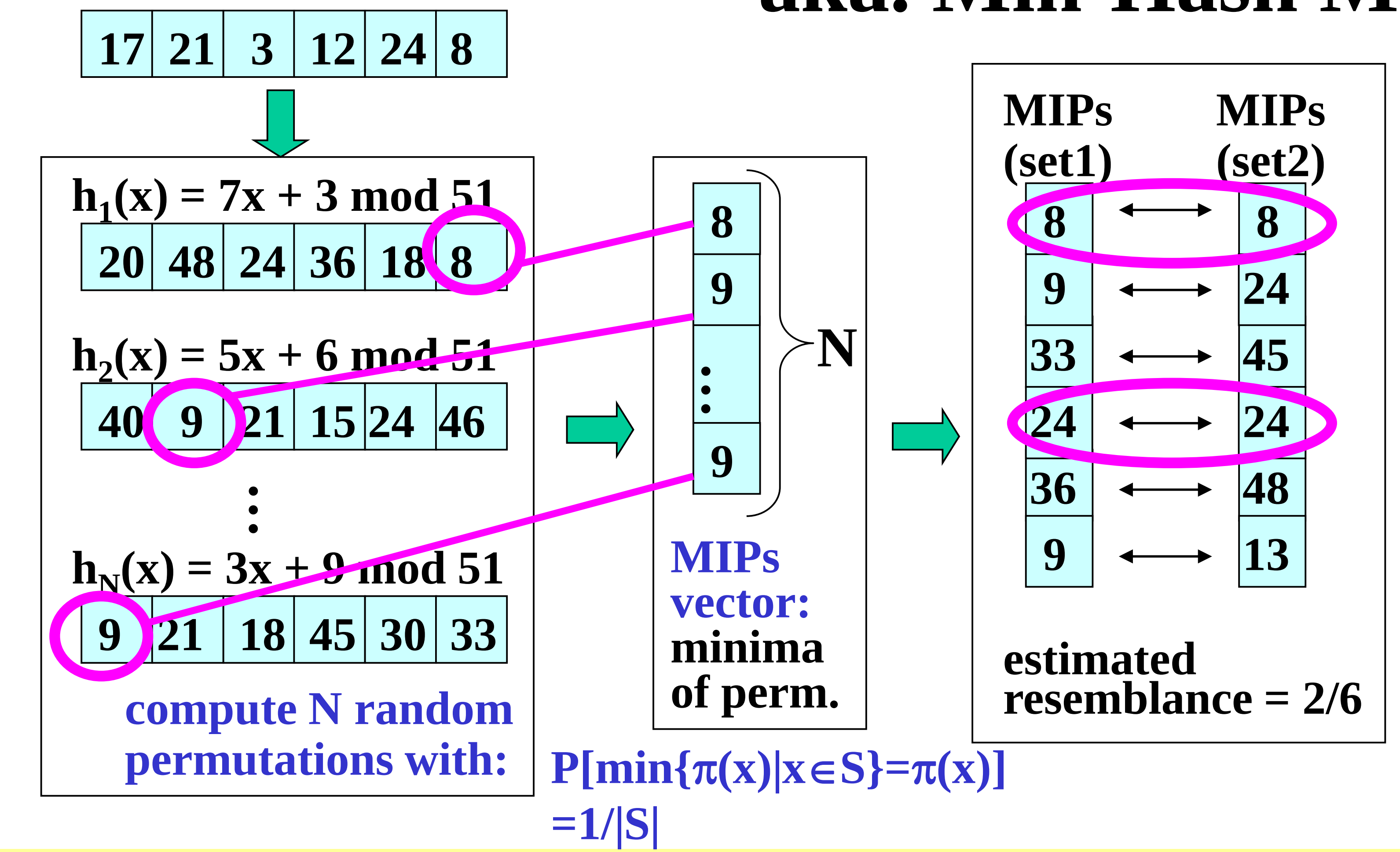
$\rightarrow$ **prune t2, t4, t5 because overlap(s,tj) < 6 for these tj**

# Similar Document Search

**Given a full document d: find similar documents (related pages)**

- **Construct representation of d:**
  **set/bag of terms, set of links,**
  **set of query terms that led to clicking d, etc.**
- **Define similarity measure:**
  **overlap, Dice coeff., Jaccard coeff., cosine, etc.**
- **Efficiently estimate similarity and design index:**
  **use approximations based on N-grams (shingles)**
  **and statistical estimators**
  **→ min-wise independent permutations / min-hash method:**
  **compute min(π(D)), min(π(D‘) for random permutations π**
  **of N-gram sets D and D‘ of docs d and d‘**
  **and test min(π(D)) = min(π(D‘))**

# Min-Wise Independent Permutations (MIPs) aka. Min-Hash Method




MIPs are unbiased estimator of resemblance:

$$P\,[\min \{h(x) \mid x \in A\} = \min \{h(y) \mid y \in B\}] = |A \cap B| \,/\, |A \cup B|$$

MIPs can be viewed as repeated sampling of x, y from A, B

# Duplicate Elimination [Broder et al. 1997]

**duplicates on the Web may be slightly perturbed**
**crawler & indexing interested in identifying near-duplicates**

**Approach:**

- **represent each document d as set (or sequence) of shingles (N-grams over tokens)**
- **encode shingles by hash fingerprints (e.g., using SHA-1), yielding set of numbers S(d) $\subseteq$ [1..n] with, e.g., n=$2^{64}$**
- **compare two docs d, d' that are suspected to be duplicates by**
  - **resemblance:** $\frac{|S(d) \cap S(d')|}{|S(d) \cup S(d')|}$ Jaccard coefficient
  - **containment:** $\frac{|S(d) \cap S(d')|}{|S(d)|}$
- **drop $d'$ if resemblance or containment is above threshold**

# Efficient Duplicate Detection in Large Corpora [Broder et al. 1997]

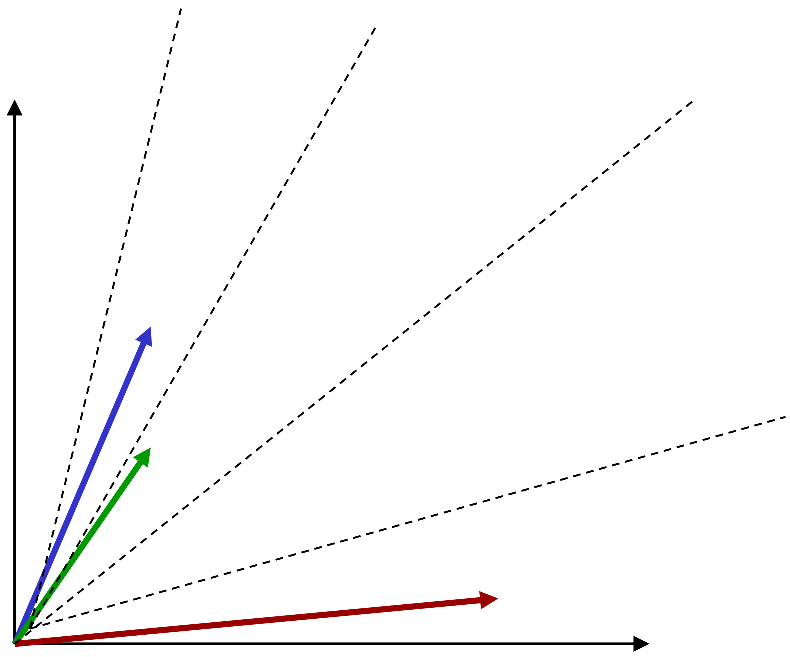
**avoid comparing all pairs of docs**

**Solution:**

1) **for each doc compute shingle-set and MIPs**
2) **produce (shingleID, docID) sorted list**
3) **produce (docID1, docID2, shingleCount) table with counters for common shingles**
4) **Identify (docID1, docID2) pairs with shingleCount above threshold and add (docID1, docID2) edge to graph**
5) **Compute connected components of graph (union-find) → these are the near-duplicate clusters**

Trick for additional speedup of steps 2 and 3:
- compute super-shingles (meta sketches) for shingles of each doc
- docs with many common shingles have common super-shingle w.h.p.

# Similarity Search by Random Hyperplanes
## [Charikar 2002]



similarity measure: cosine

- generate random hyperplanes with normal vector h
- test if $d$ and $d'$ are on the same side of the hyperplane

$$P\,[\,\mathrm{sign}(h^T d) = \mathrm{sign}(h^T d')\,] = 1 - \mathrm{angle}(d, d') \,/\, (\pi/2)$$

# Summary of Chapter 11

- indexing by **inverted lists**:
- posting lists in doc id order (or score impact order)
  - partitioned across server farm for scalability
- major space and time savings by **index compression**:

  Huffman codes, variable-bit Gamma and Golomb coding
- **similarity search** based on edit distances and N-gram overlaps
- efficient similarity search by min-hash signatures

**Happy Holidays and Merry Christmas!**

# Additional Literature for Chapter 11


- S. Brin, L. Page: The Anatomy of a Large-Scale Hypertextual Web Search Engine. Computer Networks 30(1-7), 1998
- M. McCandless, E. Hatcher, O. Gospodnetic: Lucene in Action, Manning 2010
- C. Gormley, Z. Tong: Elasticsearch – The Definitive Guide, O'Reilly 2015
- E.C. Dragut, W. Meng, C.T. Yu: Deep Web Query Interface Understanding and Integration. Morgan & Claypool 2012
- F. Menczer, G. Pant, P. Srinivasan: Topical web crawlers: Evaluating adaptive algorithms. ACM Trans. Internet Techn. 4(4): 378-419 (2004)
- J. Zobel, A. Moffat: Inverted files for text search engines. ACM Computing Surveys 38(2), 2006
- X. Long, T. Suel: Three-Level Caching for Efficient Query Processing in Large Web Search Engines, WWW 2005
- F. Transier, P. Sanders: Engineering basic algorithms of an in-memory text search engine. ACM Trans. Inf. Syst. 29(1), 2010

# Additional Literature for Chapter 11


- J. Dean, S. Ghemawat: MapReduce: Simplified Data Processing in Large Clusters, OSDI 2004
- T. White: Hadoop – The Definitive Guide, O‘Reilly 2015
- J. Lin, C. Dyer: Data-Intensive Text Processing with MapReduce, Morgan & Claypool 2010
- J. Dean: Challenges in Building Large-Scale Information Retrieval Systems, WSDM 2009, http://videolectures.net/wsdm09_dean_cblirs/
- D. Peng, F. Dabek: Large-scale Incremental Processing Using Distributed Transactions and Notifications, OSDI 2010
- A.Z. Broder, S.C. Glassman, M.S. Manasse, G. Zweig: Syntactic Clustering of the Web. Computer Networks 29(8-13): 1157-1166 (1997)
- M. Henzinger: Finding near-duplicate web pages: a large-scale evaluation of algorithms. SIGIR 2006: 284-291