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**
  - Handle dynamic pages, detect duplicates, detect spam
  - Strategies for crawl schedule and priority queue for crawl frontier

- **Extract & Clean**
  - Build and analyze Web graph, index all tokens or word stems

- **Index**
  - Fast top-k queries, query logging, auto-completion
  - Scoring function over many data and context criteria

- **Search**
  - GUI, user guidance, personalization

- **Rank**

- **Present**

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

Internet crisis: users still love search engines and have trust in the Internet

Documents

- Crawling

- Internet crisis
  - users
  - ... 

Extraction of relevant words

- Linguistic methods: stemming

- Statistically weighted features (terms)

- Thesaurus (Ontology)
  - Synonyms, Sub-/Super-Concepts

- Index (B⁺-tree)

- URLs

- Internet
  - Web
  - crisis
  - user
  - love
  - search
  - engine
  - trust
  - faith
  - ...

Extraction of relevant words

- Linguistic methods: stemming

- Statistically weighted features (terms)

- Thesaurus (Ontology)
  - Synonyms, Sub-/Super-Concepts

- Index (B⁺-tree)

- URLs

- Internet
  - Web
  - crisis
  - user
  - love
  - search
  - engine
  - trust
  - faith
  - ...

- Crawling
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 (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
automatically 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

Bookmarks
Semistructured Data
Web Retrieval
Soumen Chakrabarti
Susan Dumais Homepage
SIGIR 2000 TECHNICAL PROGRAM SCHEDULE
Byron Dom's home page
Weizi Meng's Home page
Towards a Highly-Scalable and Effective Metasearch Engine
http://www.heinzinger.com/momka/mnpublications.html
The Anatomy of a Search Engine

Data Mining
Johannes Gehrke's Publications
Data Mining and Knowledge Discovery Table of Contents
Knowledge Discovery in Databases and Data Mining

Crawler
Classifier
Link Analysis

seeds
training
re-training

high confidence
high authority

topic-specific archetypes

Root

Database Technology
Semistructured Data
Web Retrieval
Data Mining
Social Graphs

IRDM WS 2015
Vector Space Model for Content Relevance Ranking

**Ranking by descending relevance**

- **Search engine**
- **Query** $q \in [0,1]^{|F|}$ (set of weighted features)
- Documents are **feature vectors** (bags of words)

Similarity metric:

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

$\sum_{j=1}^{|F|} d_{ij} q_j$ 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

**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 \land d_{ij} \neq 0

sparse scalar product
(Many) \textit{tf*idf} Variants: Pivoted \textit{tf*idf} Scores

\begin{align*}
\text{tf} (d_i, t_j) &= \text{term frequency of term } t_j \text{ in doc } d_i \\
\text{df} (t_j) &= \text{document frequency of } t_j = \#\text{docs with } t_j \\
\text{idf} (t_j) &= N / \text{df}(t_j) \text{ with corpus size (total \#docs) } N \\
\text{dl} (d_i) &= \text{doc length of } d_i \text{ (avgdl: avg. doc length over all N docs)}
\end{align*}

\textbf{tf*idf score} for single-term query (\textbf{index weight}): \[
d_{ij} = (1 + \ln(1 + \ln(\text{tf}(d_i, t_j)))) \cdot \ln \frac{1+N}{\text{df}(t_j)} \quad \text{for } \text{tf}(d_i,t_j)>0, \ 0 \text{ else}
\]

\textbf{pivoted tf*idf score:} \[
d_{ij} = \frac{1+\ln(1+\ln(\text{tf}(d_i,t_j))))}{(1-s)+s\frac{\text{dl}(d_i)}{\text{avgdl}}} \cdot \ln \frac{1+N}{\text{df}(t_j)}
\]

also uses scalar product for score aggregation

\textit{tf*idf} scoring often works very well, but it has many ad-hoc tuning issues \rightarrow \text{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

B+ tree or hashmap

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

<table>
<thead>
<tr>
<th>crisis</th>
<th>...</th>
<th>Internet</th>
<th>...</th>
<th>trust</th>
</tr>
</thead>
<tbody>
<tr>
<td>17: 0.3</td>
<td>12: 0.5</td>
<td>11: 0.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>44: 0.4</td>
<td>14: 0.4</td>
<td>17: 0.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>52: 0.1</td>
<td>28: 0.1</td>
<td>28: 0.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>53: 0.8</td>
<td>44: 0.2</td>
<td>:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>55: 0.6</td>
<td>51: 0.6</td>
<td>52: 0.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 (⇒ 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 $x_i = \max\{s: s \text{ is a key in subtree } t_i\}$ a shorter router $y_i$ with $s_i \leq y_i < x_{i+1}$ for all $s_i$ in $t_i$ and all $s_{i+1}$ in $t_{i+1}$ is sufficient, for example, $y_i = \text{shortest string with the above property.}$ → 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. $tf(crisis, d_{123}) \times idf(crisis) = 3.75$)
  - **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**

<table>
<thead>
<tr>
<th>Posting</th>
<th>Document Identifier</th>
<th>Term Frequency</th>
<th>Offsets</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d_{123}$</td>
<td>2</td>
<td>[4, 14]</td>
<td></td>
</tr>
<tr>
<td>$d_{234}$</td>
<td>4</td>
<td>[47]</td>
<td></td>
</tr>
<tr>
<td>$d_{266}$</td>
<td>3</td>
<td>[1, 9, 20]</td>
<td></td>
</tr>
</tbody>
</table>

Payload: $tf$, offsets
Query Processing on Inverted Lists

**Merge Algorithm:**

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

*Given:* query $q = t_1 \ t_2 \ldots \ t_z$ with $z$ (conjunctive) keywords

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

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

index lists with (DocId, score)
sorted by DocId

<table>
<thead>
<tr>
<th>crisis</th>
<th>...</th>
<th>Internet</th>
<th>...</th>
<th>trust</th>
</tr>
</thead>
<tbody>
<tr>
<td>17: 0.3</td>
<td>12: 0.5</td>
<td>11: 0.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>44: 0.4</td>
<td>14: 0.4</td>
<td>17: 0.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>52: 0.1</td>
<td>28: 0.1</td>
<td>28: 0.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>53: 0.8</td>
<td>44: 0.2</td>
<td>:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| 55: 0.6 | 51: 0.6 | :
| : | 52: 0.3 | :

Precomputed scores (index weights) $s_i(d)$ for which $q_i \neq 0$
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 $q = q_1 \ldots q_k$ required

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

**mixed-mode** queries and **negations**: $q = q_1 q_2 q_3 +q_4 +q_5 -q_6$

**phrase** queries and **proximity** queries: $q = "q_1 q_2 q_3" q_4 q_5 \ldots$

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

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

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

see 11.4
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)

\[ d_{123}: \text{the giants played a fantastic season. it is not clear ...} \]

\[ d_{123} \quad dl:428 \quad content:< 1, 222, 127, 3, 897, 233, 0, 12, 6, 7, ... > \]
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

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: \( K=\text{docIds}, V=\text{contents}, L=\text{termIds}, W=\text{docIds} \)
- click log analysis: \( K=\text{logs}, V=\text{clicks}, L=\text{URLs}, W=\text{counts} \)
- web graph reversal: \( K=\text{docIds}, V=(s,t) \text{ outlinks}, L=t, W=(t,s) \text{ inlinks} \)
Map-Reduce Parallelism for Index Building

Map

Extractor

Intermediate files

Reduce

Extractor

Output files

input files

Map

Extractor

a..c

sort

a..c

merge

Intermediate files

Inverter

a..c

sort

merge

Inverter

u..z

sort

u..z

merge

output files
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, embarrassingly scalable, easy maintenance.
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

queries

Index Server

Query Processor

Inverted-List Caches

queries

Query-Result Caches

Index Server

Query Processor

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

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

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

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

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

\[
D( f \parallel g ) := \sum_x f(x) \log_2 \frac{f(x)}{g(x)}
\]

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 \( g(x) \):

\[
\frac{1}{2} D(f \parallel g) + \frac{1}{2} D(g \parallel f)
\]

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

\[
H( f, g ) := H(f) + D(f \parallel g) = -\sum_x f(x) \log g(x)
\]
Compression

- Text is sequence of symbols (with specific frequencies)
- Symbols can be
  - letters or other characters from some alphabet $\Sigma$
  - 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: $H(d) = \sum p_i \log_2 \frac{1}{p_i}$

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 $\Sigma = \{A, B, C, D\}$

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

Optimal (prefix-free) code from Huffman tree:

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

Avg. code length: $0.5*1 + 0.25*2 + 2*0.125*3 = 1.75$ bits
Basic Compression: Huffman Coding

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

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

Optimal (prefix-free) code from Huffman tree:
- $A \rightarrow 0$
- $B \rightarrow 10$
- $C \rightarrow 110$
- $D \rightarrow 111$

Avg. code length: $0.6 \times 1 + 0.3 \times 2 + 0.05 \times 3 + 0.05 \times 3 = 1.5$ bits
Algorithm for Computing a Huffman Code

\[ n := |\Sigma| \]

priority queue \( Q := \Sigma \) sorted in ascending order by \( p(s) \) for \( s \in \Sigma \)

for \( i := 1 \) to \( n - 1 \) do

\[ z := \text{MakeTreeNode}( ) \]

\[ z.\text{left} := \text{ExtractMin}(Q) \]

\[ z.\text{right} := \text{ExtractMin}(Q) \]

\[ p(z) := p(z.\text{left}) + p(z.\text{right}) \]

\( \text{Insert}(Q, z) \)

od

return \( \text{ExtractMin}(Q) \)

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 → 0
B → 101
C → 100
D → 111
E → 1101
F → 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 $P[A]=0.4$, $P[B]=0.3$, $P[C]=0.2$, $P[D]=0.1$

$\rightarrow F(A)=0.4$, $F(B)=0.7$, $F(C)=0.9$, $F(D)=1.0$

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>$2^{-3}$</td>
</tr>
<tr>
<td>B</td>
<td>$2^{-2}$</td>
</tr>
<tr>
<td>C</td>
<td>$2^{-1}$</td>
</tr>
<tr>
<td>D</td>
<td></td>
</tr>
</tbody>
</table>

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` is 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 characters 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 offsets 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 \( \gamma \) coding:**
- **length**: \( \text{floor}(\log_2 x) \) in unary, followed by
- **offset**: \( x - 2^{\text{floor}(\log_2 x)} \) 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

<table>
<thead>
<tr>
<th>x</th>
<th>length (unary)</th>
<th>offset (binary)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>001</td>
<td>100</td>
</tr>
<tr>
<td>17</td>
<td>00001</td>
<td>10001</td>
</tr>
<tr>
<td>24</td>
<td>00001</td>
<td>11000</td>
</tr>
<tr>
<td>63</td>
<td>000001</td>
<td>11111</td>
</tr>
<tr>
<td>64</td>
<td>0000001</td>
<td>1000000</td>
</tr>
</tbody>
</table>

Note 1: as there are no gaps of size $x=0$, one typically encodes $x-1$

Note 2: a variant called $\delta$ coding uses $\gamma$ 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** $q = \text{floor}(x/M)$ – stored in unary code with $q+1$ bits
- **remainder** $r = x \mod M$ – stored in binary code with $\text{ceil}(\log_2 r)$ bits

let $b = \text{ceil}(\log_2 M)$ → 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

<table>
<thead>
<tr>
<th>x</th>
<th>q</th>
<th>bits(q)</th>
<th>r</th>
<th>bits(r)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>33</td>
<td>3</td>
<td>0001</td>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>57</td>
<td>5</td>
<td>000001</td>
<td>7</td>
<td>0111</td>
</tr>
<tr>
<td>99</td>
<td>9</td>
<td>0000000001</td>
<td>9</td>
<td>1001</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>x</th>
<th>q</th>
<th>bits(q)</th>
<th>r</th>
<th>bits(r)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>000</td>
</tr>
<tr>
<td>33</td>
<td>3</td>
<td>0001</td>
<td>3</td>
<td>011</td>
</tr>
<tr>
<td>57</td>
<td>5</td>
<td>000001</td>
<td>7</td>
<td>1101</td>
</tr>
<tr>
<td>99</td>
<td>9</td>
<td>0000000001</td>
<td>9</td>
<td>1111</td>
</tr>
</tbody>
</table>
Practical Index Compression: Layout of Index postings

One block (with n postings):

- Delta to last docId in block
- #docs in block: n
- n-1 docId deltas: Rice\(_k\) encoded
- n values tf: Gamma encoded
- tf attributes: Huffman encoded
- tf positions: Huffman encoded

[Jeff Dean (Google): WSDM'09]
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
Hamming distance of strings $s_1, s_2 \in \Sigma^*$ with $|s_1| = |s_2|$:
number of different characters (cardinality of $\{i : s_{1i} \neq s_{2i}\}$)

Levenshtein distance (edit distance) of strings $s_1, s_2 \in \Sigma^*$:
minimal number of editing operations on $s_1$
(replacement, deletion, insertion of a character)
to change $s_1$ into $s_2$

For $\text{edit} (i, j)$: Levenshtein distance of $s_1[1..i]$ and $s_2[1..j]$ 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 $s_{1i} \neq s_{2j}$, 0 otherwise
→ efficient computation by dynamic programming
Example for Levenshtein edit distance:

\( \text{grate}[1..i] \rightarrow \text{great}[1..j] \)

<table>
<thead>
<tr>
<th></th>
<th>g</th>
<th>r</th>
<th>e</th>
<th>a</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>r</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>a</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>t</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>e</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

\[
\text{edit} (s[1..i], t[1..j]) = \min \{ \\
\quad \text{edit} (s[1..i-1], t[1..j]) + 1, \\
\quad \text{edit} (s[1..i], t[1..j-1]) + 1, \\
\quad \text{edit} (s[1..i-1], t[1..j-1]) + \text{diff} (s[i], t[j]) \}
\]
Similarity Measures on Strings (2)

**Damerau-Levenshtein distance** of strings \( s_1, s_2 \in \Sigma^* \): minimal number of replacement, insertion, deletion, or transposition operations (exchanging two adjacent characters) for changing \( s_1 \) into \( s_2 \)

For edit \((i, j)\): Damerau-Levenshtein distance of \( s_1[1..i] \) and \( s_2[1..j] \):

\[
\text{edit}(0, 0) = 0, \quad \text{edit}(i, 0) = i, \quad \text{edit}(0, j) = j \\
\text{edit}(i, j) = \min \{ \text{edit}(i-1, j) + 1, \\
\qquad \text{edit}(i, j-1) + 1, \\
\qquad \text{edit}(i-1, j-1) + \text{diff}(i, j), \\
\qquad \text{edit}(i-2, j-2) + \text{diff}(i-1, j) + \text{diff}(i, j-1) + 1 \} \\
\text{with } \text{diff}(i, j) = 1 \text{ if } s_{1i} \neq s_{2j}, \ 0 \text{ otherwise}
\]
Similarity based on N-Grams

Determine for string $s$ the set or bag of its N-Grams:
$$G(s) = \{ \text{substrings of } s \text{ with length } N \}$$
(often trigrams are used, i.e. $N=3$)

Distance of strings $s_1$ and $s_2$:
$$|G(s_1)| + |G(s_2)| - 2|G(s_1) \cap G(s_2)|$$

Example:
$G(\text{rodney}) = \{ \text{rod, odn, dne, ney} \}$
$G(\text{rhodnee}) = \{ \text{rho, hod, odn, dne, nee} \}$
distance (rodney, rhodnee) = $4 + 5 - 2*2 = 5$

Alternative similarity measures:
- **Jaccard coefficient**: $\frac{|G(s_1) \cap G(s_2)|}{|G(s_1) \cup G(s_2)|}$
- **Dice coefficient**: $2 \frac{|G(s_1) \cap G(s_2)|}{(|G(s_1)| + |G(s_2)|)}$
N-Gram Indexing for Similarity Search

Theorem (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:

\[ \text{edit}(s, t) \leq d \Rightarrow |Ngrams(s) \cap Ngrams(t)| \geq |s| - (N - 1) - dN \]

→ 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

\[ \text{edit}(s,t) \leq d \quad \Rightarrow \quad \text{overlap}(s,t) \geq |s| - (N-1) - dN \]
\[ \text{overlap}(s,t) < |s| - (N-1) - dN \quad \Rightarrow \quad \text{edit}(s,t) > d \]

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

\( t_1 = \text{ababababab}, |t_1| = 10 \)
\( t_2 = \text{abacdefaba}, |t_2| = 10 \)
\( t_3 = \text{ababaaababa}, |t_3| = 11 \)
\( t_4 = \text{abababb}, |t_4| = 7 \)
\( t_5 = \text{ababaaabbbb}, |t_5| = 11 \)

**Task:** find all \( t_i \) with \( \text{edit}(s,t_i) \leq 2 \)

\( \rightarrow \) prune all \( t_i \) with \( \text{edit}(s,t_i) > 2 = d \)

\( \rightarrow \) overlapBound = \( |s| - (N-1) - dN \)
\( = 6 \) (for \( N=2 \))

\( \rightarrow \) prune all \( t_i \) with \( \text{overlap}(s,t_i) < 6 \)

\( N=2: \)
\( \text{Ngrams}(t_1) = \{\text{ab}(5), \text{ba}(4)\} \)
\( \text{Ngrams}(t_2) = \{\text{ab}(2), \text{ba}(2), \text{ac}, \text{cd}, \text{de}, \text{ef}, \text{fa}\} \)
\( \text{Ngrams}(t_3) = \{\text{ab}(4), \text{ba}(4), \text{aa}(2)\} \)
\( \text{Ngrams}(t_4) = \{\text{ab}(3), \text{ba}(2), \text{bb}\} \)
\( \text{Ngrams}(t_5) = \{\text{ab3}, \text{ba}(2), \text{aa}(2)\text{bb}(3)\} \)

\( \rightarrow \) prune \( t_2, t_4, t_5 \) because \( \text{overlap}(s,t_j) < 6 \) for these \( t_j \)
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 \( \min(\pi(D)) \), \( \min(\pi(D')) \) for random permutations \( \pi \)
    of N-gram sets D and D' of docs d and d'
    and test \( \min(\pi(D)) = \min(\pi(D')) \)
Min-Wise Independent Permutations (MIPs) aka. Min-Hash Method

MIPs are unbiased estimator of resemblance:

\[
P \left[ \min \{ h(x) | x \in A \} = \min \{ h(y) | y \in B \} \right] = \frac{|A \cap B|}{|A \cup B|}
\]

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

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

- M. McCandless, E. Hatcher, O. Gospodnetic: Lucene in Action, Manning 2010
- 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
- J. Lin, C. Dyer: Data-Intensive Text Processing with MapReduce, Morgan & Claypool 2010
- D. Peng, F. Dabek: Large-scale Incremental Processing Using Distributed Transactions and Notifications, OSDI 2010