In Boolean retrieval, queries have been evaluated using inverted indexes (aka inverted files)

Document collection:
- Document1 = {step, mankind}
- Document2 = {step, China}

Inverted index:
- step: {Document1, Document2}
- mankind: {Document1}
- China: {Document2}

Query:
- “mankind AND step”
Let's assume some document has to be indexed.

**Character Sequence Decoding**

- Many document formats can be converted into a plain text representation (possibly with some markup information) using special converters.
- **Example:** pdf2html (open source tool)

**Character Sequence Decoding (2)**

- Unfortunately, a text document's character encoding often is unknown or wrongly specified.
- There are many heuristics for auto-detecting encodings:
  - **Coding Scheme Method:** Exclude certain encodings by looking for illegal bytes or byte sequences, which are not defined within this encoding.
  - **Character Distribution Method:** Use statistics to exploit the fact that in any given language, some characters are used more often than other characters.
  - **Two-Char Sequence Distribution Method:** Exploit statistical information about 2-grams, i.e., look at pairs of adjacent characters.

**Character Sequence Decoding (3)**

- But even plain text can be problematic.
- A plain text document is a sequence of bytes.
- What does the following byte sequence mean?
  - “fährt” in UTF-8
  - “f hrt” in ASCII

**Character Sequence Decoding (4)**

- Sometimes, the documents to be indexed are very large and should be split up into smaller parts.
- **Examples:**
  - E-books (large PDFs)
  - E-mail collections, including attachments (e.g. in UNIX's mbox format)
  - again, this normally can be done using heuristics...
Y2K Around the World

As computers all over the world switched to 2000, few Y2K bugs were reported in several labs. [...]

**Tokenization:**
- Remove formatting information (e.g. HTML tags)
- Remove punctuation
- Carry out basic normalization (e.g. remove capitalization)
- **Goal:** Convert the text into a sequence of "tokens"

Tokenization is difficult!

Let's tokenize the following sentence!

"Mr. O’Neill thinks that the boys’ stories about Chile’s capital aren’t amusing.”

One token or two?
- Hewlett-Packard
- State-of-the-art
- Data base
- San Francisco
- York University vs. New York University
- Speed limit

Specific problems in other languages:
- This two Chinese characters can be treated as one word meaning “monk” or as sequence of two words meaning “and” and “still”

How to handle compounds?
- Donaudampfschiffahrtsgesellschaftskapitänsfrau
- Lebensversicherungsgesellschaftsfachangestellter

Normalization handles most of the problematic cases

Define **equivalence classes** of character sequences that get mapped to the same token
- U.S.A. and USA
- naïve and naive

Define these classes implicitly by **transformation rules**
- Omit all accents
- Remove periods between two characters, where there is no whitespace around (e.g. in U.S.A.)
- Do case folding, i.e. reduce all letters to lower case
- Maybe you need exceptions for names: windows ≠ Windows (Not important, since users ask queries in lowercase anyway)

Removal of stop words!

**Stop words:** Extremely common words, which are of little value in selecting which documents match a user’s query

**Examples:** a, an, and, are, as, at, be, by, for, from, has, he, in, is, its, of, on, that, the, to, was, were, which, will, with

“to be or not to be?”
For example, the most common stemmer is the Porter stemmer (Porter, 1980)

• By design, it fits the characteristics of English language
  – Idea: Suffixes in the English language are mostly made up of a combination of smaller and simpler suffixes

How does it work?
  – The algorithms run through five steps, one by one
  – In each step, several rules are applied that change the word’s suffix

A general strategy for stop words removal:
  – Sort tokens by collection frequency
  – Take top-k of this list as stop words

Alternatively, use a (possibly domain-specific) predefined stop word list

But:
  – One can handle large indexes by exploiting the statistics of language for compression, so the cost for including stop words is not high for modern systems
  – The trend goes to smaller stop word lists, e.g. 200–300 or even less

Of course, lemmatization would be the “right” thing and there are software tools for this task

But:
  – “Good” lemmatization is computationally expensive if very large document collections are to be processed
  – Gains of lemmatizers over stemmers in retrieval quality (mean average precision) are very modest for English (0–5%)
  – Larger gains have been reported for other languages, e.g. German, Spanish, and Finnish (10–30%)

In this lecture, we will discuss only stemming
Finally, we arrive at the bag-of-words representation

• A comparison of different stemmers:
  - Sample text:
    Such an analysis can reveal features that are not easily visible from the variations in the individual genes and can lead to a picture of expression that is more biologically transparent and accessible to interpretation
  - Porter stemmer:
    such an analysis can reveal features that are not easily visible from the variations in the individual genes and can lead to a picture of expression that is more biologically transparent and accessible to interpretation
  - Lovins stemmer:
    such an analysis can reveal features that are not easily visible from the variations in the individual genes and can lead to a picture of expression that is more biologically transparent and accessible to interpretation
  - Paice stemmer:
    such an analysis can reveal features that are not easily visible from the variations in the individual genes and can lead to a picture of expression that is more biologically transparent and accessible to interpretation

• PDF (portable document format)
  - The standard for exchanging digital documents
  - ISO 15930, 19005, 24517, 32000 etc.
  - Documents are collections of objects (characters, vector-based graphics, bitmap images, ...)
  - Objects are positioned by absolute coordinates

• “Pseudo-PDFs”
  - Non-digital documents, scanned or photographed
  - Rule of thumb: Everything published before 1995
    - Journal publications
    - Research reports
    - Project reports
    - ...
• **PDF-to-text converters** are of no help
  — Only extract the text objects from PDF
  — pdftotext, PDFBox, PDFExtractor, PDFTextStream

• **Screen reader and OCR software**
  — Input can be arbitrary — bitmap images
  — Tries to find regions, lines, words, and characters in the image

---

**Solution: Extraction Tools?**

**The Perfect Document**

• “Perfect” documents can be processed easily
  — Single column, well-structured
  — Direct output of the authoring or production process
  — Or converted from other digital formats (XML, Word, LaTeX, ...)
  — Not much markup

• The only problem: segmentation!

---

**Segmentation**

**More Problematic Sources**

• Problematic documents:
  — Compound documents (pictures + text)
  — Complex layouts (magazines, journals)
  — Complex markup (math + chemical formulas)

• Already a lot of problems!

---

**Markup**

3. Suppose there are two events, x and y, in question with m possibilities for the first and n for the second. Let \( p_{ij} \) be the probability of the joint occurrence of \( i \) for the first and \( j \) for the second. The entropy of the joint event is:

\[
H_{x \times y} = - \sum_{i,j} p_{ij} \log p_{ij}
\]

while

\[
H_x = - \sum_{i} p_i \log p_i, \quad H_y = - \sum_{j} p_j \log p_j
\]

It is easily shown that:

\[
H_{x \times y} = H_x + H_y - \sum_{i,j} p_{ij} \log \frac{p_{ij}}{p_i p_j}
\]

Table with a reference to entities mentioned in the table:

“Basic” reaction scheme

Long chemical entity names (row span)

Figure with a reference to one special step in the table above (2c)
Lecture 3: Indexing

1. Document Preparation
2. Index Construction
3. Query Evaluation
4. Properties of Document Collections

Some slides have been adapted from (Zobel and Moffat, 2006)

Index Construction

• Building an inverted index looks easy:
  1. Assign an ID to each document: docID
  2. Run the document preparation process on each document
  3. Compile a list of all index terms
  4. Assign an ID to each index term: termID
  5. Create a list of all (termID, docID, tf) triplets
  6. Sort this list: Primarily by termID, secondarily by docID

• Essentially, this corresponds to a matrix transposition

Let’s have a look at an example...

Example (2)

<table>
<thead>
<tr>
<th>df</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Then, we get the following (termID, docID, tf) triplets:

(5, 1, 3), (7, 1, 1), (8, 1, 1), (10, 1, 1), (1, 2, 2), (3, 2, 1), (4, 2, 1), (8, 2, 2), (1, 3, 1), (4, 3, 1), (5, 3, 1), (8, 3, 1), (10, 3, 1), (5, 4, 1), (7, 4, 1), (10, 1, 1), (5, 5, 3), (7, 5, 2), (2, 6, 1), (5, 6, 1), (6, 6, 1), (9, 6, 1)

Example (3)

(5, 1, 3), (7, 1, 1), (8, 1, 1), (10, 1, 1), (1, 2, 2), (3, 2, 1), (4, 2, 1), (8, 2, 2), (1, 3, 1), (4, 3, 1), (5, 3, 1), (8, 3, 1), (10, 3, 1), (5, 4, 1), (7, 4, 1), (10, 1, 1), (5, 5, 3), (7, 5, 2), (2, 6, 1), (5, 6, 1), (6, 6, 1), (9, 6, 1)

Now, sort: Primarily by termID, secondarily by docID

(1, 2, 2), (1, 3, 1), (2, 6, 1), (3, 2, 1), (4, 2, 1), (4, 3, 1), (5, 1, 3), (5, 3, 1), (5, 4, 1), (5, 5, 3), (6, 6, 1), (7, 1, 1), (7, 4, 1), (7, 5, 2), (8, 1, 1), (8, 2, 2), (8, 3, 1), (8, 4, 1), (9, 4, 1), (9, 6, 1), (10, 1, 1), (10, 3, 1)
The inverted index:

Example (4)

<table>
<thead>
<tr>
<th>Term</th>
<th>Posting List</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: big</td>
<td>(2, 2), (3, 1)</td>
</tr>
<tr>
<td>2: dark</td>
<td>(6, 1)</td>
</tr>
<tr>
<td>3: gown</td>
<td>(2, 1)</td>
</tr>
<tr>
<td>4: house</td>
<td>(2, 1)</td>
</tr>
<tr>
<td>5: keep</td>
<td>(1, 3), (3, 1), (4, 1), (5, 3), (6, 1)</td>
</tr>
<tr>
<td>6: light</td>
<td>(6, 1)</td>
</tr>
<tr>
<td>7: night</td>
<td>(1, 1), (4, 1), (5, 2)</td>
</tr>
<tr>
<td>8: old</td>
<td>(1, 1), (2, 2), (3, 1), (4, 1)</td>
</tr>
<tr>
<td>9: sleep</td>
<td>(4, 1), (6, 1)</td>
</tr>
<tr>
<td>10: town</td>
<td>(1, 1), (3, 1)</td>
</tr>
</tbody>
</table>

Building the inverted index isn't difficult if the whole document collection fits in main memory.

Now, let’s get serious: Typical collections are very large...

What can we do?
- Sort-based inversion:
  Use a external (i.e. disk-based) sorting algorithm that works on compressed disk blocks (for performance reasons)
- Merge-based inversion:
  Read and index documents in memory until a fixed capacity is exceeded; when memory is full, the index is flushed to disk and merged with the index already stored on disk

Merge-based indexing has several advantages:
- It is practical for collections of all sizes
- It even scales well and operates effectively in as little as 100 MB of main memory
- Disk space overheads can be restricted to a small fraction of the final index
- With clever data compression methods, the number of merge runs can be reduced further

Index Representations

A simple implementation of an inverted index:
- Use 32-bit integers for document identifiers
- Use 16-bit integers for term frequencies

Then, the posting list for term “keep” will be stored on disk like this:
Furthermore, fixed-width integers are not space-efficient. Furthermore, fixed-width integers are not space-efficient. Half of all values are the number 1. Another variable-length code is Elias' gamma code:

- To store the integer $x > 0$, it is factored into $2^a + b$, where $a = \lceil \log_2(x) \rceil$ and $0 \leq b < 2^a$.
- The codeword is formed as the concatenation of $a + 1$ represented in unary and $b$ represented in fixed-width binary of width $a$.
- Example: Encode 12 as 1110100.

Variable-Length Codes (4)

<table>
<thead>
<tr>
<th>Term</th>
<th>Posting List</th>
</tr>
</thead>
<tbody>
<tr>
<td>5: keep</td>
<td>(1, 3), (3, 1), (4, 1), (5, 3), (6, 1)</td>
</tr>
</tbody>
</table>

Using gamma coding, our example posting list becomes:

- 28 bit (compared to 240 using a fixed-length encoding)

Storing Gaps

<table>
<thead>
<tr>
<th>Term</th>
<th>Posting List (with Gaps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5: keep</td>
<td>(1, 3), (2, 1), (1, 1), (1, 3), (1, 1)</td>
</tr>
</tbody>
</table>

- Gaps usually are much smaller than document IDs.
- To allow even better compression ratios, we can store gaps instead of document IDs.

Variable-Length Codes (3)

- The efficiency of each code depends on the distribution of input numbers to be encoded.
- The unary code allows optimal space efficiency if the input distribution is given by $Pr(x) \approx 1/(2^x)$.
- Example: Encode 12 by the bit sequence 11111110.

- The gamma code is optimal for $Pr(x) = 1/(2x^2)$.
- Of course, there are many other codes available...

Variable-Length Codes (2)

<table>
<thead>
<tr>
<th>Value</th>
<th>Unary</th>
<th>Gamma</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>110</td>
<td>1100</td>
</tr>
<tr>
<td>4</td>
<td>1110</td>
<td>11100</td>
</tr>
<tr>
<td>5</td>
<td>11110</td>
<td>111100</td>
</tr>
<tr>
<td>6</td>
<td>111110</td>
<td>1111100</td>
</tr>
<tr>
<td>10</td>
<td>11111110</td>
<td>111111100</td>
</tr>
<tr>
<td>100</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Variable-Length Codes (3)

- The simplest variable-bit infinite code is unary:
  - Represent the integer $x > 0$ as $x-1$ "1" bits followed by a terminating "0" bit.
- Example: Encode 12 by the bit sequence 11111110.

- Some more examples:
  - The unary code allows optimal space efficiency if the input distribution is given by $Pr(x) = 1/(2^x)$.
  - Half of all values are the number 1.
  - A quarter are 2.
  - An eighth are 4.
- Example: Encode 12 as 1110100.

- Of course, there are many other codes available...

Variable-Length Codes (4)

<table>
<thead>
<tr>
<th>Term</th>
<th>Posting List</th>
</tr>
</thead>
<tbody>
<tr>
<td>5: keep</td>
<td>(1, 3), (3, 1), (4, 1), (5, 3), (6, 1)</td>
</tr>
</tbody>
</table>

Using gamma coding, our example posting list becomes:

• 28 bit (compared to 240 using a fixed-length encoding)
Simple (but inefficient) solution:

- Skip lists can speed up intersection operations massively if the posting lists are sorted by document ID
- We already have seen that compression can help us a lot
- Now: How to reduce the number of disk accesses?

16 bit (compared to 28 and 240, respectively)

- Using gap storage, even large posting lists can be stored efficiently, which enables us to abstain from stop words
- What do we get?

**Skip Lists (2)**

<table>
<thead>
<tr>
<th>Term</th>
<th>Posting List (no Gaps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>keep</td>
<td>(1, 3), (2, 3), (3, 4)</td>
</tr>
<tr>
<td>light</td>
<td>(6, 1)</td>
</tr>
</tbody>
</table>

**Skip Lists (3)**

- Idea: Skip Lists
  - Evenly spaced, add some skip pointers to the list
  - Every skip pointer consists of a number of bits that can be skipped to reach a different entry
  - This skipped entries do not have to be accessed from disk
  - Our gamma-coded posting list (no gaps) with skip pointers that allow skipping every other entry:

**Skip Lists (4)**

- Skip lists can speed up intersection operations massively if the posting lists are sorted by document ID
- How to handle changes to document collection?
  - New documents
  - Updated documents
  - Deleted documents
- Simple (but inefficient) solution:
  - Rebuild the index from scratch
- Better solution:
  - Keep an auxiliary in-memory index that keeps track of all changes

- Where to place skip pointers?
  - A heuristic says that they should be placed every $\sqrt{k}$ postings if $k$ is the number of list entries…
**Lecture 3: Indexing**

1. Document Preparation
2. Index Construction
3. Query Evaluation
4. Properties of Document Collections

Some slides have been adapted from (Zobel and Moffat, 2006)

---

**Vector Space Retrieval**

- **Query** = “dark keep night”
- **Vector representation:** Simple term frequencies
  - Query vector = (0, 1, 0, 0, 1, 0, 1, 0, 0, 0)
- **Similarity measure:** Simple scalar product

<table>
<thead>
<tr>
<th>Term</th>
<th>Posting List</th>
</tr>
</thead>
<tbody>
<tr>
<td>dark</td>
<td>(k, 1)</td>
</tr>
<tr>
<td>keep</td>
<td>(1, 3), (4, 1), (5, 3), (6, 1)</td>
</tr>
<tr>
<td>night</td>
<td>(1, 3), (4, 1), (5, 2)</td>
</tr>
</tbody>
</table>

- Process query by scanning though the three lists and add up term frequencies for each occurring document
- This gives the following final scores:
  - Doc 1: 3 + 1 = 4
  - Doc 2: 0
  - Doc 3: 1 + 3 = 4
  - Doc 4: 3 + 2 = 5
  - Doc 5: 1 + 1 = 2
  - Doc 6: 1 + 2 = 3

- Due to the nature of the scalar product, we only need to add up scores for any non-zero query component
- To support more advanced vector representations, simply add some more information to the posting lists
  - For example, to support TF-IDF, store each term’s IDF at the beginning of its corresponding posting list
- Furthermore, we can avoid reading all affected posting lists completely, by sorting the postings by their TF
  - This yields a significant speed-up of query processing
- Similar approaches can be used to process queries for other retrieval models...

---

**Query Processing**

- **Boolean retrieval:**
  Process queries as we already have discussed it

- **Vector space retrieval:**
  Answer the query “dark, keep, night” by scanning through the postings lists for “dark,” “night,” and “keep,” while accumulating scores for each document
  - Let’s have a look at an example…

---

**Phrase Queries**

- A special type of queries are phrase queries
- **Example:** “King of Finland”
- **Three strategies** to process phrase queries:
  - **Postprocessing:**
    - Initially, ignore the word order and do Boolean retrieval;
    - In a second step, search through the documents found and return only the ones containing the phrase
  - **Store word positions:**
    - Add word positions to each posting so that the locations of terms in documents can be checked during query evaluation
  - **Partial phrase indexes:**
    - Create a (partial) index containing phrases

---
Phrase Queries (2)

- The three strategies complement each other and usually are applied in combination
  - Create a phrase index for phrases containing frequent words
    (phrases containing rare words can be found easily by using the other two approaches)
  - Store word positions for every word and phrase
    (can be done efficiently using compression)
  - If for some reason there are postings without word positions, postprocess all document founds by doing a phrase search

Next Lecture

- Probabilistic retrieval models