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”
Indexing:
“The process of assigning keywords to each document”

• These **keywords** are used as a (possibly intermediate) representation of each document

• **Some problems** we are faced with:
  – “<h1>Document heading</h1>” vs. “Document heading”
  – computer vs. computers vs. Computer vs. computer’s
  – aren’t vs. are not
Lecture 4: Indexing

1. Document Preparation
2. Index Construction
3. Query Evaluation

Some slides have been adapted from lecture notes by Hinrich Schütze, Universität Stuttgart.

Information Retrieval and Web Search Engines — Wolf-Tilo Balke and Joachim Selke — Technische Universität Braunschweig
As computers all over the world switched to 2000, few Y2K bugs were reported in several labs. […]
Let’s assume some document has to be indexed

**First step: Getting a textual representation**

Sounds easy, but might be pretty complicated

- Many different document formats:
  - DOC, plain text, PDF, HTML, XLS, PPT, RTF, XML, …

---

Y2K Around the World

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

Information Retrieval and Web Search Engines — Wolf-Tilo Balke and Joachim Selke — Technische Universität Braunschweig
Many document formats can be converted into a plain text representation (possibly with some markup information) using special converters.

Example: pdftohtml (open source tool)

Homework: Exercise 14a

- Given a collection, a query, and an IR system:
  - Collection: 20 relevant documents, 180 non-relevant
  - Found: 8 relevant documents, 10 non-relevant
  - Precision, recall, and fallout?

\[
\text{Precision} = \frac{\#(\text{relevant items retrieved})}{\#(\text{items retrieved})} = \frac{8}{18} = 0.44
\]
\[
\text{Recall} = \frac{\#(\text{relevant items retrieved})}{\#(\text{relevant items})} = \frac{8}{20} = 0.4
\]
\[
\text{Fallout} = \frac{\#(\text{nonrelevant items retrieved})}{\#(\text{nonrelevant items})} = \frac{10}{180} = 0.06
\]
• But even plain text can be problematic
• A plain text document is a sequence of bytes
• What does the following byte sequence mean?
  102 195 164 104 114 116
• This depends on the character encoding
  – A character encoding assigns byte sequences to characters
• It means:
  – “fährt” in UTF-8
  – “fÃ¤hrt” in ISO-8859-1
  – “fØhrt” in ASCII
    (ASCII is a 7-bit character encoding!)
• 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

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...
• **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”**

**DOCUMENT TEXT**

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

**TOKENIZATION**

y2k around the world as computers all over the world switched to 2000 few y2k bugs were reported in several labs […]
• **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
• Identical tokens?
  – 4/15/99
  – 15/4/99
  – Apr 15, 1999

• Identical tokens?
  – (0531) 391 3271
  – +49 531 391-3271
  – 531.391.3271
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?
  - Donaudampfschifffahrtsgesellschaftskapitänsfrau
  - 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, it, its, of, on, that, the, to, was, were, which, will, with

• “to be or not to be”?

Information Retrieval and Web Search Engines — Wolf-Tilo Balke and Joachim Selke — Technische Universität Braunschweig
• In classical IR systems, stop words have been rigorously deleted

• But stop words are needed for phrase queries, e.g. “King of Finland” or “As We May Think”

• For example, Google does not remove stop words:

```
Google

Results 1 - 10 of about 19,400 for "king of finland". (0.18 seconds)
```

Monarchy of Finland - Wikipedia, the free encyclopedia
(Redirected from King of Finland). Jump to: navigation, search ... of the late king Charles XII of Sweden, could be proclaimed as the King of Finland ... en.wikipedia.org/wiki/King_of_Finland - 28k - Cached - Similar pages

List of Finnish monarchs - Wikipedia, the free encyclopedia
5 Jan 2009 ... During Svinhufvud's regency, Prince Frederick Charles of Hesse was elected as the King of Finland on October 6, 1918 ... en.wikipedia.org/wiki/List_of_Finnish_rulers - 46k - Cached - Similar pages

Finland's King - TIME
The old man was Jean Julius Christian Sibelius, most famous of present-day composers and "Uncrowned King of Finland"; the occasion was his seventieth ... www.time.com/time/magazine/article/0,9171,756541,00.html - 37k - Cached - Similar pages
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
• **Lemmatization:**
  The process of grouping together the different **inflected forms** of a word, e.g. walking → walk,  better → good

• **Stemming:**
  Trying to do lemmatization using crude heuristics, usually without knowledge of the word’s context or any rules of grammar, e.g. walking → walk,  but better → better
• 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**
• The most common stemmer is the **Porter stemmer** (Porter, 1980)

• It is designed to fit 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 runs through **five steps**, one by one
  
  — In each step, several **rules** are applied that change the word’s **suffix**
Some (simplified!) examples of rules used in the Porter stemmer:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSES → SS</td>
<td>caresses → caress</td>
</tr>
<tr>
<td>IES → I</td>
<td>ponies → poni</td>
</tr>
<tr>
<td>S →</td>
<td>cats → cat</td>
</tr>
<tr>
<td>ING →</td>
<td>motoring → motor</td>
</tr>
<tr>
<td>Y → I</td>
<td>happy → happi</td>
</tr>
<tr>
<td>ATIONAL → ATE</td>
<td>relational → relate</td>
</tr>
<tr>
<td>FULNESS → FUL</td>
<td>hopefulness → hopeful</td>
</tr>
<tr>
<td>ICAL → IC</td>
<td>electrical → electric</td>
</tr>
<tr>
<td>ABLE →</td>
<td>adjustable → adjust</td>
</tr>
<tr>
<td>ATE →</td>
<td>activate → activ</td>
</tr>
</tbody>
</table>
Some transformations made by the Porter stemmer:

<table>
<thead>
<tr>
<th>Input word</th>
<th>Stemmed word</th>
</tr>
</thead>
<tbody>
<tr>
<td>gen</td>
<td>gen</td>
</tr>
<tr>
<td>gender</td>
<td>gender</td>
</tr>
<tr>
<td>genders</td>
<td>gender</td>
</tr>
<tr>
<td>general</td>
<td>gener</td>
</tr>
<tr>
<td>generally</td>
<td>gener</td>
</tr>
<tr>
<td>generals</td>
<td>gener</td>
</tr>
<tr>
<td>generation</td>
<td>gener</td>
</tr>
<tr>
<td>generations</td>
<td>gener</td>
</tr>
<tr>
<td>generative</td>
<td>gener</td>
</tr>
<tr>
<td>generosity</td>
<td>generos</td>
</tr>
<tr>
<td>generous</td>
<td>gener</td>
</tr>
<tr>
<td>genitive</td>
<td>genit</td>
</tr>
<tr>
<td>genitivo</td>
<td>genitivo</td>
</tr>
</tbody>
</table>
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.
There are some additional things apart from stemming that could be done during this step.

Take care of umlauts and accents:
- Usually, just remove them (e.g. ä → a) or transliterate them (e.g. ä → ae)
- But be careful: unbeschränkt ≠ unbeschränkt

Take care of synonyms, e.g. auto → car
- Again, be very careful when doing this!
Finally, we arrive at the bag-of-words representation.

The preparation of original documents is finished now, but we need to talk about efficient data structures for managing the inverted indexes...
Preparing chemical documents is particularly problematic

- A lot of information is encoded in figures
- Text references figures and figures may contain tabular data
- Names (e.g. of substances) implicitly contain structural information

The preparation process (for PDFs):

- Extracting text
- Recognize chemical entities and reactions within text, tables, and figures
- Derive structural data from named entities
• 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
    • ...
Solution: Extraction Tools?

• **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
“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!
Information Retrieval and Web Search Engines — Wolf-Tilo Balke and Joachim Selke — Technische Universität Braunschweig
Problematic documents:

- Compound documents (pictures + text)
- Complex layouts (magazines, journals)
- Complex markup (math + chemical formulas)

Already a lot of problems!

The copper(I)-catalyzed 1,3-dipolar cycloaddition [33-35] of organic azides and alkynes (also called "click chemistry") resulting in the formation of 1,2,3-triazoles has become an increasingly attractive area [79]. According to literature [33-35], the Cu(I) species can be used directly (e.g. CuI) or generated by oxidation of a Cu(0) or reduction of a Cu(II) species. Catalysis by the CuI is known to yield exclusively the 1,4-disubstituted regioisomer [33,34]. First, the N-(p-methoxyphenyl)-1-(trifluoromethyl)propargylamine was reacted with benzyl azide in the presence of CuI (10 mol%) and showed good reactivity with completion of the reaction within 24 h, whereas the use of Cu50 mg acetone afforded the cycloadduct in low yield. The reaction was then carried out with different propargylamines (N-(p-methoxyphenyl)- and N-benzyl) and various azides at room temperature in acetonitrile within 24 h which afforded the compounds 1a-4 with good yields (67-93%) after purification by column chromatography. The results are summarized in Table 1.

As expected the new triazoles were formed in a fully regioselective manner affording the 1,4-regioisomers as highlighted from NOE experiments on compound 2c (Figure 1). A strong correlation was observed between the hydrogen H3 and H4, respectively. The structure of the other compounds 2a-4 was assigned by analogy with 2c.

In our goal to study the influence of the CF3 group on the conformations of pyridinoureas, we applied our strategy to the 2-amino-4,5-difluorotri fluoromethylpropargylamine bearing the removable (R)-phenylglycinol chiral auxiliary (Schemes 2) [30-32].

The reaction was carried out under the same condition with stoichiometric CuI and afforded the cycloadduct 4 in good yield (79%) and as a single isomer without any racemization. This compound can easily afford the free amino ester which is a promising trifluorooroxymethyl building block for the synthesis of new triazole-based trifluoroethyl oligomers.
The copper(I)-catalyzed 1,3-dipolar cycloaddition [33-35] of organic azides and alkynes (also called "click chemistry") resulting in the formation of 1,2,3-triazoles has become an increasingly attractive area [30]. According to the literature [33-35], the Cu(I) species can be used directly (e.g., CuI), or generated by oxidation of a Cu(II) or reduction of a Cu(I) species. Catalysis by the CuI is known to yield exclusively the 1,4-disubstituted regiosomer [33-35]. First, the 

2a–i: Correspond to entities explained elsewhere

Table with a reference to one special step in the table above (2c)

References to entities mentioned in the table

Long chemical entity names (row span)

"Basic" reaction scheme

Table with figure of a reaction

Figure with a reference to one special step in the table above (2c)
3. Suppose there are two events, $x$ and $y$, in question with $m$ possibilities for the first and $n$ for the second. Let $p(i,j)$ be the probability of the joint occurrence of $i$ for the first and $j$ for the second. The entropy of the joint event is

$$H(x,y) = - \sum_{i,j} p(i,j) \log p(i,j)$$

while

$$H(x) = - \sum_{i} p(i,j) \log \sum_{j} p(i,j)$$

$$H(y) = - \sum_{j} p(i,j) \log \sum_{i} p(i,j).$$

It is easily shown that...
• Scanned or photographed documents with a rich structure – ... in bad quality
• Can only be processed with OCR software – Usually, a lot of errors
Lecture 4: Indexing

1. Document Preparation
2. Index Construction
3. Query Evaluation

Some slides have been adapted from (Zobel and Moffat, 2006)
• 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…
• Our **example collection** of six documents:
  1. The old night keeper keeps the keep in the town
  2. In the big old house in the big old gown
  3. The house in the town had the big old keep
  4. Where the old night keeper never did sleep
  5. The night keeper keeps the keep in the night
  6. And keeps in the dark and sleeps in the light

• **Case-folding, stopping, and stemming reduces the vocabulary to ten index terms:**
  1. big 4. house 7. night 10. town
  2. dark 5. keep 8. old
  3. gown 6. light 9. sleep
Then, we get the following \((\text{termID}, \text{docID}, \text{tf})\) triplets:

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

(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), (5, 6, 1), (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:

<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), (3, 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 Inversion

Documents

Parsed document

..., dark, ..., keep, night, ....

In-memory index

dark

keep

night

Runs on disk

merge

Final index

read

update

flush
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
• The problem of building the index essentially is solved
  – OK, Google uses massive replication and data distribution
    but these are very special requirements…

• Now, how to store an inverted index on disk?
• Since disk accesses are very expensive (e.g. compared to
  computations), there are **two major requirements:**

  Keep the index as small as possible!

  Read as little data as possible from disk!

• Since computational power comes at (almost) no cost,
  **effective data compression** is our first way to go!
• 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:

<table>
<thead>
<tr>
<th>Term</th>
<th>Posting List</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>5: keep</td>
<td>(1, 3), (3, 1), (4, 1), (5, 3), (6, 1)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
• A typical inverted index:
  – Some document IDs occur very frequently in the whole index
  – Most document IDs occur very rarely
• Here, fixed-width integers are not space-efficient
• Furthermore, fixed-width integers limit the number of documents that can be stored…
• **Variable-length codes solve both problems**
  – They can encode **arbitrary large numbers**
  – They can be constructed to **store small values with little storage cost**, at the expense of large values
  – This perfectly fits our needs if document with many index entries get small IDs
Variable-Length Codes

• 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 111111111110

• Another variable-bit code is **Elias’ gamma** code:
  
  - To store the integer $x > 0$, it is factored into $2^a + b$, where $a = \lfloor \log_2(x) \rfloor$ 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 $12 = 2^3 + 4$ as 1110100
### Variable-Length Codes (2)

- **Some more examples:**

<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>101</td>
</tr>
<tr>
<td>4</td>
<td>1110</td>
<td>11000</td>
</tr>
<tr>
<td>5</td>
<td>11110</td>
<td>11001</td>
</tr>
<tr>
<td>6</td>
<td>111110</td>
<td>11010</td>
</tr>
<tr>
<td>10</td>
<td>11111110</td>
<td>1110010</td>
</tr>
<tr>
<td>100</td>
<td>...</td>
<td>1111110100100</td>
</tr>
<tr>
<td>1000</td>
<td>...</td>
<td>111111110111101000</td>
</tr>
</tbody>
</table>
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 \( \Pr(x) = 2^{-x} \)
  – Half of all values are the number 1
  – A quarter are 2
  – ...
• The gamma code is optimal for \( \Pr(x) \approx \frac{1}{2x^2} \)
• Of course, there are many other codes available...
Using gamma coding, our example posting list becomes:

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

- **28 bit** (compared to 240 using a fixed-length encoding)
To allow even better compression ratios, we can store gaps instead of document IDs:

<table>
<thead>
<tr>
<th>Term</th>
<th>Posting List (with Gaps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<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…
What do we get?

- **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.
Skip Lists

• We already have seen that compression can help us a lot
• Now: How to reduce the number of disk accesses?

• As we have seen in Boolean retrieval, an important operation is intersecting posting lists
  — Inverted index:
    • step: {Document1, Document2}
    • mankind: {Document1}
    • China: {Document2}
  — Query:
    • “mankind AND step”

• How to speed up this operation?
**Skip Lists (2)**

<table>
<thead>
<tr>
<th>Term</th>
<th>Posting List (no Gaps)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>...</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></td>
<td>...</td>
</tr>
</tbody>
</table>

- **Another example:**
  When trying to answer the query “keep AND light,” we have to scan through the two posting lists shown above
  
  - **Problem:** We have to scan the whole posting list of “keep” to finally reach document 6; we know that we can ignore every document having a smaller ID than 6
  
  - **Is there any way to skip some of these postings?**
**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:**

```
1 bit 3 bit 3 bit 1 bit 5 bit 1 bit 5 bit 1 bit 10 bit 14 bit
1 3 10 3 1 4 1 14 5 3 6 1
```

Note that in addition to this we need some coding mechanism to indicate whether an entry contains a skip pointer or not…
• Skip lists can speed up intersection operations massively if the posting lists are sorted by document ID

• 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…
• 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
  – If the auxiliary index gets too large, it is merged with the main index
Lecture 4: Indexing

1. Document Preparation
2. Index Construction
3. Query Evaluation

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

Information Retrieval and Web Search Engines — Wolf-Tilo Balke and Joachim Selke — Technische Universität Braunschweig
• **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…
• 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>2: dark</td>
<td>(6, 1)</td>
</tr>
<tr>
<td>5: keep</td>
<td>(1, 3), (3, 1), (4, 1), (5, 3), (6, 1)</td>
</tr>
<tr>
<td>7: night</td>
<td>(1, 1), (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:

<table>
<thead>
<tr>
<th>Doc 1</th>
<th>Doc 2</th>
<th>Doc 3</th>
<th>Doc 4</th>
<th>Doc 5</th>
<th>Doc 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 + 1 = 4</td>
<td>0</td>
<td>1</td>
<td>1 + 1 = 2</td>
<td>3 + 2 = 5</td>
<td>1 + 1 = 2</td>
</tr>
</tbody>
</table>
• 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...**
The complete retrieval process:

Query Processing

Query → Vocabulary → Inverted lists

- dark
- keep
- night

Inverted lists → Accumulators → Document weights → Similarity scores

Accumulators → divide by to get and then select the highest values

Add contributions
• 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
• 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

- Latent Semantic Indexing
- Next lecture is on May 11!