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"

Recap: Inverted Indexes

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

Indexing

"The process of assigning keywords to each document"

Document Preparation

INPUT DOCUMENT

DOCUMENT TEXT

Y2K Around the World

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

DOCUMENT REPRESENTATION

y2k (2), world (2), computer (1), switch (1), 2000 (1), bug (1), report (1), lab (1)

TOKENIZATION

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

FILTRATION

y2k world computers world switched 2000 y2k bugs reported labs [...]

STEMMING

y2k world computer world switched 2000 y2k bug report lab [...]

Character Sequence Decoding

DOCUMENT TEXT

Y2K Around the World

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, ...
Character Sequence Decoding (2)

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

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?
  
<table>
<thead>
<tr>
<th>102</th>
<th>195</th>
<th>164</th>
<th>104</th>
<th>114</th>
<th>116</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>2</td>
<td>5</td>
<td>8</td>
<td>8</td>
<td>6</td>
</tr>
</tbody>
</table>

  - This depends on the character encoding.
  - A character encoding assigns byte sequences to characters.
  - It means:
    - “fährt” in UTF-8
    - “Aşhtr” in ISO-8859-1
    - “fährt” in ASCII (ASCII is a 7-bit character encoding!)

Character Sequence Decoding (4)

- 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.

Tokenization

- 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

Tokenization (2)

- **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 (3)

- Identical tokens?
  - 4/15/99
  - 15/4/99
  - Apr 15, 1999

- Identical tokens?
  - (0531) 391 3271
  - +49 531 391-3271
  - 531.391.3271

Tokenization (4)

- 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?
  - Donau dampfschifffahrtsgesellschaftskapitänsfrau
  - Lebensversicherungsgesellschaftsfachangestellter

Tokenization (5)

- **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 naïve
- 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)

Filtration

- **Normalisation** 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 naïve
- 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)

Filtration (2)

- 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:

Filtration (3)

- **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
Some (simplified!) examples of rules used in the Porter stemmer:

- **Rule**: SSES → SS
  - **Example**: caresses → caress
- **Rule**: IES → I
  - **Example**: ponies → poni
- **Rule**: S →
  - **Example**: cats → cat
- **Rule**: ING
  - **Example**: motorizing → motor
- **Rule**: Y → Y
  - **Example**: happy → happy
- **Rule**: TIONAL → ATE
  - **Example**: relational → relate
- **Rule**: FULNESS → FUL
  - **Example**: hopefulfulness → hopeful
- **Rule**: ICAL → IC
  - **Example**: electrical → electic
- **Rule**: ABLE
  - **Example**: adjustable → adjust
- **Rule**: ATE
  - **Example**: activate → activ

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 feature that are not easily visible from the variation in the individual genes and can lead to a picture of expression that is more biologically transparent and accessible to interpret
- **Lovins stemmer**: such an analysis can reveal feature that are not easily visible from the variation in the individual genes and can lead to a picture of expression that is more biologically transparent and accessible to interpret
- **Paice stemmer**: such an analysis can reveal feature that are not easily visible from the variation in the individual genes and can lead to a picture of expression that is more biologically transparent and accessible to interpret
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)
  - But be careful: unbeschränkt ≠ unbefristet
- 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
- 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
  - ...
More Problematic Sources

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

- Already a lot of problems!

Segmentation

Textual output

Markup

3. Suppose there are two events, \( X \) and \( Y \), in question with \( m \) possibilities for the first and \( n \) for the second. Let \( p_{i,j} \) 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

\[
H(X) = - \sum_{i} p_{i,\cdot} \log p_{i,\cdot}
\]

\[
H(Y) = - \sum_{j} p_{\cdot,j} \log p_{\cdot,j}
\]

It is easily shown that

Long chemical entity names (row span)

References to entities mentioned in the table

2a-c: Correspond to entities explained elsewhere

Table with figure of a reaction

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

“Basic” reaction scheme

The Ultimate Horror

- 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: \text{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: \text{termID}
5. Create a list of all \((\text{termID}, \text{docID}, \text{tf})\) triplets
6. Sort this list: Primarily by \text{termID}, secondarily by \text{docID}

Essentially, this corresponds to a matrix transposition

Let’s have a look at an example…

Index Construction

Example (2)

<table>
<thead>
<tr>
<th>\text{tf}</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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

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 \text{termID}, secondarily by \text{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)

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

The inverted index:

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

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, stemming, and stemming reduces the vocabulary to ten index terms:
1. \text{big}
2. \text{dark}
3. \text{gown}
4. \text{house}
5. \text{keep}
6. \text{light}
7. \text{night}
8. \text{old}
9. \text{sleep}
10. \text{town}

Now, sort: Primarily by \text{termID}, secondarily by \text{docID}

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

Large Collections

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 an 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

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

• 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!

Index Representations (2)

• 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 11111111110
• Another variable-bit 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 21 + 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>1100</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>110010</td>
</tr>
<tr>
<td>100</td>
<td>...</td>
<td>11111010101000</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 \( \Pr(x) \approx 1 / (2^x^2) \)
- Of course, there are many other codes available...

Variable-Length Codes (4)

- Using gamma coding, our example posting list becomes:

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

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

Storing Gaps

- 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>step</td>
<td>(1, 3), (2, 1), (1, 1), (1, 3), (6, 1)</td>
</tr>
</tbody>
</table>

- Gaps usually are much smaller than document IDs...

Storing Gaps (2)

- What do we get?

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

- 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)

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:

  Note that in addition to this we need some coding mechanism to
  indicate whether an entry contains a skip pointer or not…

Skip Lists (4)

- 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…

Dynamic Indexing

- 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

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…

Lecture 4: Indexing

1. Document Preparation
2. Index Construction
3. Query Evaluation
Vector Space Retrieval

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

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

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!!