Information Retrieval

- scope, basic concepts
- system architectures, modes of operation

Scope of IR:
Getting data relevant to user information needs from:
- free text (unstructured)
- hypertext (WWW)
- semi-structured (XML)

Recently also:

Of the linguistic levels:

- lexical, syntax, semantics

It uses (almost) only the first:
The basic building block of queries is the term מילה.
(For hypertext - links are used in query processing)

Architecture A: text pattern match
- Full text
- No indices
- Query = regular expression
- Query processing: read docs sequentially, match with the pattern
- Results: show matching docs

Example: grep on UNIX

Properties:
- Can be used with compression
- Practical for "small" db's (~ few MB's)

Architecture B: Hierarchical browsing
- DB: a hierarchy of categories, with associated bib records | full text docs
- "Query": browsing the hierarchy

Examples:
  - yahoo
  - bow

Issues: manual | (semi-) automatic construction of:
- Hierarchy
- Classification of entries

Properties:
- Browsing takes time/effort
- Restricted coverage
**Properties**: (mostly disadvantages)
- End users have difficulties translating info needs to boolean queries (may require use of distributive laws for and, or...)
- Precise boolean formulation often large/complex
- No standard, for syntax, additional operations...
- Extreme situations possible:
  - Answer too small – not clear how to weaken query
  - Answer too large – requires browsing & filtering, or transforming query to return less answers (not easy)
- Terms cannot be weighted by user
- Often require information professional to help users

**Architecture D**: ranked queries (actually, answers)
- DB: full text | bib records | abstracts | ...
- Query: lists of (possibly weighted) terms
- Result: docs, ranked by similarity (~relevance)
- Query processing:
  - List of terms – retrieve doc id’s, of docs containing some terms
    ( # of term occurrences in doc, significance of term,...)

A popular system architecture, underlies many commercial products & search engines

**Properties**: 
- Query = doc (list of term occurrences)
- Easy to write/change queries (no expertise needed)
- More relevant answers are shown first (provided system performs well)
- User’s feedback on relevance, can be used to improve query – relevance feedback (find similar pages)
- Possible to miss some relevant docs
- Possible to include some/many irrelevant docs

**Index structure for boolean/ranked**:
- Inverted file
- DB is ~ a boolean sparse matrix:
  - d-t(i,j) = 1 if tj occurs in di, 0 if not
  - A doc di is a boolean (sparse) vector (easily obtained)
- “Transposing the matrix”, we obtain
  - t-d(j,i) = 1/0 if tj occurs/occurs not in di
  - A term tj is ~ a boolean (sparse) vector
  - vector space model

Sparse: $10^5 - 10^7$ terms, doc ~ $10^5 - 10^9$ terms, $10^5 - 10^9$ docs

**Side trip**: on measuring quality of answer

**Precision** = \[
\frac{\text{# of relevant docs in answer}}{|\text{answer}|}
\]

**Recall** = \[
\frac{\text{# of relevant docs in answer}}{\text{# of relevant in DB}}
\]

**Issues**:
- How is relevance measured? (need users’ opinions)
  substitute: user satisfaction (e.g., # of clicks)
- Even if relevance is available, how is recall measured?
Typical inverted file organization:

- **Lexicon**: the terms, organized for fast search. Often can reside in MM.
  For each term:
  - (optional) global data such as weight/total # of occs
  - Address of inverted list on disk
- **Inverted lists file (on disk)** – allows to retrieve each inverted list
- **The DB of documents**
- **Also need a docId-to-docAddress mapper**

A compact representation of the vector of \( t_j \) is an inverted list (postings list):

- **docId** (logical pointer) for each doc that contains \( t_j \)
- (optional, for ranked) # of occurrences (or some measure of frequency, significance) of \( t_j \) in doc
- (optional for proximity) position of each occurrence of \( t_j \) in doc:
  - Section/paragraph/sentence
  - Word #, char #

(Typical inverted file organization:

- **3 level architecture**:
  - **Lexicon**: the terms, organized for fast search. Often can reside in MM.
  - **Inverted lists file (on disk)** – allows to retrieve each inverted list
  - **The DB of documents**
  - **Also need a docId-to-docAddress mapper**

Advantages of the organization:

- Can find fast lexicon entries for query terms (would be much slower if lexicon combined with IL’s)
- Can retrieve fast the IL’s for these terms
- Then manipulate docId’s before deciding which documents to retrieve (boolean, proximity, ranking, can be done on docId’s lists)
- Compression reduces storage costs, more importantly I/O costs (up to 70% of space)
  - can compress IL’s and DB separately, using appropriate methods for each

Processing of boolean queries:

- **List of terms**: retrieve their IL’s, intersect them
- **And**: intersect lists
- **Or**: union lists
- **And not**: take relative complement

Finally retrieve the docs

Proximity: use positional information in IL’s for filtering, if present

Proximity: use positional information in IL’s for filtering, if present

or: retrieve docs then filter as they arrive

How is similarity computed?

Assume \( n \) terms

- **The query** \( q \) is an \( n \)-vector \((q_1, q_2, \ldots, q_n)\)
- **A doc** \( d \) is an \( n \)-vector \((d_1, d_2, \ldots, d_n)\)

(The \( q_i, d_i \) can be 0/1 or 0/real numbers -- frequencies/weights of terms in documents / query)

The distance \( \text{dist}(q,t) \) computed as follows:

\[
\cos(q,t) = \frac{\sum q_i d_i}{\sqrt{\sum q_i^2} \sqrt{\sum d_i^2}}
\]

This measures the angle between the vectors in \( n \)-dimensional space

Processing of ranked queries:

Naively:

- Retrieve & merge IL’s of all terms (like or)
- Compute similarity / distance of query to each doc
- Present to user ranked by similarity

What is naïve about the above?
IR topics for the course:

- Terms and weights:
  - what are terms
  - which weights give good results
- Index -- inverted file (efficient) construction
  - Process many GB's DB's in hours
- Other index structures (signature files)
  - are inverted files the best solution?
- Efficient & fast processing of ranked answers
  - Which weights to use in the cosine formula?
  - Fast computation of similarity for many docs
  - Approximate • find highly relevant docs early

- Compression:
  - Disk capacities grow, collections grow faster
  - Main cost is I/O cost; compression reduces it
  - CPU speeds grow at a faster rate than I/O speeds – pays to always use compression for disk data – I/O rate vs. CPU usage tradeoff
  - Tailor compression methods for different kinds of data: DB docs, IL's, images, ...
- Search engines
  - How can link information improve answer quality
- Relevance feedback:
  - How can user feedback be used to improve answer?
- Answer quality & presentation:
  - Revisit precision, recall
  - Clustering for improved presentation