Review: Model

- Document: sequence of (terms + attributes)
- Query: sequence of terms
  - Can make more complicated: Advanced search
- Satisfying: in current search engines, documents “containing” all terms
  - AND model
  - “containing” includes anchor text of pointers to this doc from other docs
- Ranking: wide open function of document and terms

Review: Inverted Index

- For each term, keep list of document entries, one for each document in which it appears: a postings list
  - Document entry is list of positions at which term occurs and attributes for each occurrence: a posting
- Keep summary term information
- Keep summary document information

Consider “advanced search” queries

To know if satisfied need:

<table>
<thead>
<tr>
<th>Content</th>
<th>Meta-data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phrases</td>
<td>Language</td>
</tr>
<tr>
<td>OR</td>
<td>Geographic region</td>
</tr>
<tr>
<td>NOT</td>
<td>File format</td>
</tr>
<tr>
<td>Numeric range</td>
<td>Date published</td>
</tr>
<tr>
<td>Where in page</td>
<td>From specific domain</td>
</tr>
<tr>
<td></td>
<td>Specific licensing rights</td>
</tr>
<tr>
<td></td>
<td>Filtered by “safe search”</td>
</tr>
</tbody>
</table>

Retrieval of satisfying documents

- Inverted index will allow retrieval for content queries
- Keep meta-data on docs for meta-data queries
  - Need length even for tf.idf
- Issue of efficient retrieval

Basic retrieval algorithms?

- One term
- AND of several terms
- OR of several terms
- NOT term
- proximity
Basic retrieval algorithms

- One term:
  - look up posting list in (inverted) index
- AND of several terms:
  - Intersect posting lists of the terms: a list merge
- OR of several terms:
  - Union posting lists of the terms
- NOT term
  - If terms AND NOT(terms), take a difference
- Proximity
  - a list merge (similar to AND)

Merging posting lists

- Have two lists must coordinate
  - Find shared entries and do something
- Algorithms?

Algorithms: unsorted lists

- Read 2nd list over and over - once for each entry on 1st list
  - computationally expensive
  - time $O(|L_1| \times |L_2|)$ where $|L|$ length list $L$
- Build hash table on entry values;
  - insert entries of one list, then other;
  - look for collisions
  - must have good hash table
  - unwanted collisions expensive
- Sort lists; use algorithm for sorted lists
  - often lists on disk: external sort
  - can sort in $O(|L| \log |L|)$ operations

Algorithms: sorted lists

- Lists sorted by some entry ID: Read both lists in "parallel"
  - Classic list merge algorithm for sorted lists
  - must be no duplicates to get time $|L_1| + |L_2|$
- Build lists so sorted
  - pay cost at most once
  - must be able to binary search!
- If only one list sorted, can do binary search of sorted list for entries of other list
  - can’t binary search disk
  - External sort

Sort keys for documents

- For posting lists, entries are documents
  - What value is used to sort?
  - Unique document IDs
  - can still be duplicate documents
  - consider for Web when consider crawling
  - document scoring function that is independent of query
  - PageRank, HITS authority
  - sort on document IDs as secondary key
  - allows for approximate "highest k" retrieval
  - approx. k highest ranking doc.s for a query

Sort keys within document list

- Proximity of terms
  - merge lists of terms occurrences within 1 doc.
- Sort on term position