Using and storing the index

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
  – eliminate duplicates: a list merge
• NOT term
  – If terms AND NOT(terms), take a difference
  – a list merge (similar to AND)
• 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 $O(|L_1|*|L_2|)$ 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
  – maybe get sorted order “naturally”
• If only one list sorted, can do binary search of sorted list for entries of other list
  – Must be able to binary search! - rare!
  • can’t binary search disk

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 docs for a query

Sort keys within document list

Processing within document posting

• Proximity of terms
  – merge lists of terms occurrences within 1 doc.
• Sort on term position
Data structure for inverted index?

- Sorted array:
  - Binary search if can keep in memory
  - High overhead for additions
- Hashing
  - Fast look-up
  - Collisions
- Search trees: B+-trees
  - Maintain balance - always log look-up time
  - Can insert and delete

B+-trees

- All index entries are at leaves
- Order $m$ B+ tree has $m$ to $2m$ children for each interior node
- Look up: follow root to leaf by keys in interior nodes
- Insert:
  - Find leaf in which belongs
  - If leaf full, split
  - Split can propagate up tree
- Delete:
  - Merge or redistribute from too-empty leaf
  - Merge can propagate up tree

B+ trees used for large data sets

- Leaves are file pages on disk
- Each interior node is file page on disk
- Keep top of tree in buffer (RAM)
  - $m$ is typically 200; average fanout $\sim 267$
    - Height 4 gives $\sim 5$ Billion entries
- Save more space: prefix B+ trees for words
  - Each interior node key is shortest prefix of word
    that need to distinguish which child pointer to follow
  - Allows more keys per interior node
    - Higher fanout
    - Fanout determined by what can fit; keep at least 1/2 full

Another tree structure: tries

- Strictly for character strings
- Each edge out of node labeled with one character
- Follow path root to leaf to spell word
- Leaf contain data for word
  - Usually pointer
Web query processing: limiting size

- For Web-scale collections, may not process complete posting list for each term in query
  - at least not initially
- Need docs sorted first on global (static) quantity
  - why not by term frequency for doc?
- Only take first k docs on each term list
  - k depends on query - how?
  - k depends on how many want to be able to return
    - Google: 1000 max returns
  - Flaws w/ partial retrieval from each list?
- Other limits? query size
  - Google: 32 words max query size

Limiting size with term-based sorting

- Can sort docs on postings list by score of term
  - term frequency ++
- Lose linear merge - salvage any?
- Tiered index:
  - tier 1: docs with highest term-based scores, sorted by ID or global quantity
  - tier 2: docs in next bracket of score quality, sorted
  - etc.
  - need to decide size or range of brackets
- If give up AND of query terms, can use idf too
  - only consider terms with high idf = rarer terms

Example

Tries: remarks

- Large height
  - slow look-up
  - can contract strings without fanout
- More useful for lexicon construction