Using and storing the index

Review: Model

- **Document**: sequence of \{terms + attributes\}
  - equivalently, set of \{(term, attributes)\} pairs
  - positions of a term are attributes
- **Query**: sequence of terms
  - Can make more complicated: Advanced search
- **Satisfying**: most common now: \text{AND} model
  - for Web, terms “contained” in doc. includes:
    - in anchor text of pointers to this doc from other docs
    - in URL
- **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: \text{a postings list}
  - Document entry is list of positions at which term occurs and attributes for each occurrence: \text{a posting}
- Keep summary term information
- Keep summary document information

Index structure

\[
\begin{align*}
term_1 &: \text{(doc ID, \{(position, attributes)\})} \\
      & \quad \text{(position, attributes),} \\
      & \quad \text{(doc ID, \{(position, attributes)\})} \\
      & \quad \text{(position, attributes),} \\
      & \quad \text{…} \\
      & \quad \text{(position, attributes) \}) \\
\text{…} \\
term_2 &: \text{(doc ID, \{(position, attributes)\})} \\
      & \quad \text{(position, attributes),} \\
      & \quad \text{…} \\
      & \quad \text{(position, attributes) \}) \\
\text{…}
\end{align*}
\]

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
- Issue of efficient retrieval
Basic retrieval algorithms

• One term
• AND of several terms
• OR of several terms
• NOT term
• proximity

Basic postings list processing: Merging posting lists

• Have two lists must coordinate
  – Find shared entries and do “something”
  – “something” changes for different operations
  • Set operations UNION? INTERSECTION? DIFFERENCE? …

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(other terms), take a difference
  – a list merge (similar to AND)
• Proximity
  – a list merge (similar to AND)

Algorithms for Merging Postings Lists: sorted lists

• Lists sorted by some key
  – same key both lists
• Read both lists in “parallel”
  – Classic list merge: (sorted list₁, sorted list₂) ⇒ sorted set union
  – General merge: if no duplicates, get time |L₁|+|L₂|
• 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

Algorithms for Merging Postings Lists: unsorted lists

• Read 2nd list over and over - once for each entry on 1st list
  – computationally expensive
time O(|L₁|*|L₂|) 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

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
Keys within document list
Processing within document posting

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

Computing document score

- “On fly” - as find each satisfying document
- Separate phase after build list of satisfying documents
- For either, must sort docs by score

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

Limting 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

Data structure for inverted index?

How access individual terms and each associated postings list?

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
Example B+ Tree

order = 2: 2 to 4 search keys per interior node

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

Disk-based B+ trees for large data sets

- Each leaf is file page (block) on disk
- Each interior node is file page on disk
- Keep top of tree in buffer (RAM)
- Typical sizes:
  - m ~ 200;
  - average fanout ~ 267
  - Height 4 gives ~ 5 billion entries

prefix key B+ trees

- Save space
- Each interior node key is shortest prefix of word needed 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