Web Searching & Indexing

CPS 116
Introduction to Database Systems

Announcements (November 29)

- Homework #4 due today
- Course project demo Dec. 7-14
  - Each project gets a 30-minute slot with me and Yi
  - Watch for an email this weekend scheduling demo slots
- Final exam on Saturday, Dec. 15, 7-10pm
  - Again, open book, open notes
  - Focus on the second half of the course

Keyword search

- Google
- Web & Images & Groups
- Directory
- Google Search & I'm Feeling Lucky
- Advanced Search & Preferences & Language

ACM

- Association for Computing Machinery
- Founded in 1947, ACM is the world's first educational and scientific computing society
- Today our members—

CPS 216: Advanced Database Systems (Fall 2007)
Course Information

The Internet Movie Database (IMDb)... Search the Internet Movie Database. For more search options, please visit Search Central...

Database AND search

What are the documents containing both "database" and "search"?

Keywords × documents

- All documents
- All keywords
  - "a" 1 1 1 ...
  - "cat" 1 1 0 ...
  - "database" 0 0 1 ...
  - "dog" 0 1 0 ...
  - "search" 0 0 1 ...
- Inverted lists: store the matrix by rows
- Signature files: store the matrix by columns

Inverted lists

- Store the matrix by rows
- For each keyword, store an inverted list
  - (keyword, doc-id-list)
  - ("database", {3, 7, 142, 857, ...})
  - ("search", {3, 9, 192, 512, ...})
  - It helps to sort doc-id-list (why?)
- Vocabulary index on keywords
  - B+-tree or hash-based
- How large is an inverted list index?

Using inverted lists

- Documents containing "database"
  - Use the vocabulary index to find the inverted list for "database"
  - Return documents in the inverted list
- Documents containing "database" AND "search"
  - Return documents in the intersection of the two inverted lists
- OR? NOT?
  - Union and difference, respectively
What are “all” the keywords?
- All sequences of letters (up to a given length)?
  - … that actually appear in documents!
- All words in English?
- Plus all phrases?
  - Alternative: approximate phrase search by proximity
- Minus all stop words
  - They appear in nearly every document, e.g., a, of, the, it
  - Not useful in search
- Combine words with common stems
  - Example: database, databases
  - They can be treated as the same for the purpose of search

Frequency and proximity
- Frequency
  - \( \{ \text{keyword}, \{ \text{doc-id, number-of-occurrences}, \} \} \)
- Proximity (and frequency)
  - \( \{ \text{keyword}, \{ \text{doc-id, \{position-of-occurrence}, \} \} \} \)
  - When doing AND, check for positions that are near

Signature files
- Store the matrix by columns and compress them
- For each document, store a \( w \)-bit signature
- Each word is hashed into a \( w \)-bit value, with only
  \( s < w \) bits turned on
- Signature is computed by taking the bit-wise OR of the hash values of all words on the document

\[
\text{hash(“database”) = 0110} \quad \text{doc}_1 \text{ contains “database”: 0110}
\]
\[
\text{hash(“dog”) = 1100} \quad \text{doc}_2 \text{ contains “dog”: 1100}
\]
\[
\text{hash(“cat”) = 0010} \quad \text{doc}_3 \text{ contains “cat” and “dog”: 1110}
\]

- Some false positives; no false negatives

Bit-sliced signature files
- Motivation
  - To check if a document contains a
    word, we only need to check the
    bits that are set in the word’s hash value
  - So why bother retrieving all \( w \) bits of the signature?
  - Instead of storing \( n \) signature
    files, store \( w \) bit slices
  - Only check the slices that correspond to the set bits in the word’s hash value
  - Start from the sparse slices

Inverted lists versus signatures
- Inverted lists better for most purposes (TODS, 1998)
- Problems of signature files
  - False positives
  - Hard to use because \( r, w \), and the hash function need tuning to
    work well
  - Long documents will likely have mostly 1’s in signatures
  - Common words will create mostly 1’s for their slices
  - Difficult to extend with features such as frequency, proximity
- Saving grace of signature files
  - Sizes are tunable
  - Good for lots of search terms
  - Good for computing similarity of documents

Ranking result pages
- A single search may return many pages
  - A user will not look at all result pages
  - Complete result may be unnecessary
- Result pages need to be ranked
- Possible ranking criteria
  - Based on content
    - Number of occurrences of the search terms
    - Similarity to the query text
  - Based on link structure
    - Backlink count
    - PageRank
  - And more…
Textual similarity

- Vocabulary: \([w_1, \ldots, w_n]\)
- IDF (Inverse Document Frequency): \([f_1, \ldots, f_n]\)
  - \(f_i = 1 / \text{the number of times } w_i \text{ appears on the Web}\)
- Significance of words on page \(p\): \([p_1 f_1, \ldots, p_n f_n]\)
  - \(p_i\) is the number of times \(w_i\) appears on \(p\)
- Textual similarity between two pages \(p\) and \(q\) is defined to be \(\prod_{i=1}^{n} p_i f_i^2 \cdot \prod_{i=1}^{n} q_i f_i^2 = p_1 q_1 f_1^2 + \ldots + p_n q_n f_n^2\)
  - \(q\) could be the query text

Why weight significance by IDF?

- Without IDF weighting, the similarity measure would be dominated by the stop words
- “the” occurs frequently on the Web, so its occurrence on a particular page should be considered less significant
- “engine” occurs infrequently on the Web, so its occurrence on a particular page should be considered more significant

Problems with content-based ranking

- Many pages containing search terms may be of poor quality or irrelevant
  - Example: a page with just a line “search engine”
- Many high-quality or relevant pages do not even contain the search terms
  - Example: Google homepage
- Page containing more occurrences of the search terms are ranked higher; spamming is easy
  - Example: a page with line “search engine” repeated many times

Backlink

- A page with more backlinks is ranked higher
- Intuition: Each backlink is a “vote” for the page’s importance
- Based on local link structure; still easy to spam
  - Create lots of pages that point to a particular page

Google’s PageRank

- Main idea: Pages pointed by high-ranking pages are ranked higher
  - Definition is recursive by design
  - Based on global link structure; hard to spam
- Naïve PageRank
  - \(N(p)\): number of outgoing links from page \(p\)
  - \(B(p)\): set of pages that point to \(p\)
  - \(\text{PageRank}(p) = \sum_{q \in B(p)} \text{PageRank}(q) / N(q)\)
  - Each page \(p\) gets a boost of its importance from each page that points to \(p\)
  - Each page \(q\) evenly distributes its importance to all pages that \(q\) points to

Calculating naïve PageRank

- Initially, set all PageRank’s to 1; then evaluate \(\text{PageRank}(p) \leftarrow \sum_{q \in B(p)} \text{PageRank}(q) / N(q)\)
  - repeated until the values converge (i.e. a fixed point is reached)

<table>
<thead>
<tr>
<th>Page</th>
<th>N</th>
<th>M</th>
<th>A</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>Netscape</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Amazon</td>
<td>1</td>
<td>0.5</td>
<td>1</td>
<td>0.625</td>
</tr>
<tr>
<td>Microsoft</td>
<td>1</td>
<td>1.3</td>
<td>1</td>
<td>1.2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>p</th>
<th>N</th>
<th>M</th>
<th>A</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.5</td>
<td>0</td>
<td>0.5</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0.5</td>
<td>0.5</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0.5</td>
<td>0.625</td>
<td>0.5</td>
<td>0.6</td>
</tr>
<tr>
<td>3</td>
<td>0.375</td>
<td>0.375</td>
<td>0.6875</td>
<td>0.6</td>
</tr>
<tr>
<td>4</td>
<td>1.2</td>
<td>1.25</td>
<td>1.25</td>
<td>1.2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>p</th>
<th>N</th>
<th>M</th>
<th>A</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.5</td>
<td>0</td>
<td>0.5</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0.5</td>
<td>0.5</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0.5</td>
<td>0.625</td>
<td>0.5</td>
<td>0.6</td>
</tr>
<tr>
<td>3</td>
<td>0.375</td>
<td>0.375</td>
<td>0.6875</td>
<td>0.6</td>
</tr>
<tr>
<td>4</td>
<td>1.2</td>
<td>1.25</td>
<td>1.25</td>
<td>1.2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>p</th>
<th>N</th>
<th>M</th>
<th>A</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.5</td>
<td>0</td>
<td>0.5</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0.5</td>
<td>0.5</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0.5</td>
<td>0.625</td>
<td>0.5</td>
<td>0.6</td>
</tr>
<tr>
<td>3</td>
<td>0.375</td>
<td>0.375</td>
<td>0.6875</td>
<td>0.6</td>
</tr>
<tr>
<td>4</td>
<td>1.2</td>
<td>1.25</td>
<td>1.25</td>
<td>1.2</td>
</tr>
</tbody>
</table>
Random surfer model

- A random surfer
  - Starts with a random page
  - Randomly selects a link on the page to visit next
  - Never uses the “back” button

- PageRank($p$) measures the probability that a random surfer visits page $p$

Problems with the naïve PageRank

- Dead end: a page with no outgoing links
  - A dead end causes all importance to “leak” eventually out of the Web

- Spider trap: a group of pages with no links out of the group
  - A spider trap will eventually accumulate all importance of the Web

Practical PageRank

- $d$: decay factor
- PageRank($p$) =
  \[ d \cdot \sum_{q \in B(p)} \left( \frac{\text{PageRank}(q)}{N(q)} \right) + (1 - d) \]

- Intuition in the random surfer model
  - A surfer occasionally gets bored and jumps to a random page on the Web instead of following a random link on the current page

Google (1998)

- Inverted lists in practice contain a lot of context information
  - Capitalization
  - Relative font size in URL/title/meta tag
  - In anchor text
  - Within the page

- PageRank is not the final ranking
  - Type-weight: depends on the type of the occurrence
    - For example, large font weights more than small font
  - Count-weight: depends on the number of occurrences
    - Increases linearly first but then tapers off
  - For multiple search terms, nearby occurrences are matched together and a proximity measure is computed
    - Closer proximity weights more

Trie: a string index

- A tree with edges labeled by characters
- A node represents the string obtained by concatenating all characters along the path from the root

- Compact trie: replace a path without branches by a single edge labeled by a string

Suffix tree

- Index all suffixes of a large string in a compact trie
  - Can support arbitrary substring matching
  - Internal nodes have fan-out $\geq 2$ (except the root)
  - No two edges out of the same node can share the same first character

To get linear space
- Instead of inlining the string labels, store pointers to them in the original string
  - Bad for external memory
Patricia trie, Pat tree, String B-tree

A Patricia trie is just like a compact trie, but
- Instead of labeling each edge by a string, only label by the first character and the string length
- Leaves point to strings
  - Faster search (especially for external memory) because of inlining of the first character
  - But must validate answer at leaves for skipped characters
- A Pat tree indexes all suffixes of a string in a Patricia trie
- A String B-tree uses a Patricia trie to store and compare strings in B-tree nodes

Summary

- General tree-based string indexing tricks
  - Trie, Patricia trie, String B-tree
- Two general ways to index for substring queries
  - Index words: inverted lists, signature files
  - Index all suffixes: suffix tree, Pat tree, suffix array (not covered)
- Web search and information retrieval go beyond substring queries
  - IDF, PageRank, …