Announcements (February 17)

- Homework #2 due in two weeks
- Reading assignments for this and next week
  - “The” query processing survey by Graefe
  - Due next Wednesday
- Midterm and course project proposal in three weeks

Keyword search

- Google
- Web | Images | Groups | Directory
- Google Search | I’m Feeling Lucky
- Advanced Search | Preferences | Language

- 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

Course Information

Course Description / Time and Place / Books

Resources: Staff...

The Internet Movie Database (IMDb) ...

... Search the Internet Movie Database. For more search options, please visit Search central...

Inverted lists

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

Keywords × documents

- Inverted lists: store the matrix by rows
- Signature files: store the matrix by columns
- With compression, of course!

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; not useful in search
  - Example: a, of, the, it
- Combine words with common stems
  - They can be treated as the same for the purpose of search
  - Example: database, databases

Frequency and proximity

- Frequency
  - \{\text{keyword}, \{\text{doc-id, number-of-occurrences}\}, \{\text{doc-id, number-of-occurrences}\}, \ldots \}\}
- Proximity (and frequency)
  - \{\text{keyword}, \{\text{doc-id, position-of-occurrence}_1, \ldots \}, \{\text{doc-id, position-of-occurrence}_2, \ldots \}, \ldots \}\}
  - 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

\[
\text{hash}(\text{"database"}) = 0110 \\
\text{hash}(\text{"dog"}) = 1100 \\
\text{hash}(\text{"cat"}) = 0010
\]

\(\text{doc}_1\) contains "database": 0110
\(\text{doc}_2\) contains "dog": 1100
\(\text{doc}_3\) 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 \(s, w\), 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): \[ f_1, \ldots, f_n \]
  \[ f_i = \log_2 (\text{total # of docs} / \text{# of docs containing } w_i) \]
- TF (Term Frequency): \[ p_1, \ldots, p_n \]
  \[ p_i = \text{# of times } w_i \text{ appears on } p \]
- Significance of words on page \( p \): \[ p_1 f_1, \ldots, p_n f_n \]
  \[ q \text{ could be the query text} \]

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

Calculating naive PageRank

- Initially, set all PageRank’s to 1; then evaluate
  \[ \text{PageRank}(p) = \sum_{q \in B(p)} \left( \frac{\text{PageRank}(q)}{N(q)} \right) \]
  repeatedly until the values converge (i.e. a fixed point is reached)
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
- $\text{PageRank}(p) = d \cdot \sum_{q \in B(p)} \left( \frac{\text{PageRank}(q)}{N(q)} \right) + (1 - d)$

- 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
  - 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

Suffix arrays (SODA, 1990)

- Another index for searching text
- Conceptually, to construct a suffix array for string $S$
  - Enumerate all $|S|$ suffixes of $S$
  - Sort these suffixes in lexicographical order
- To search for occurrences of a substring
  - Do a binary search on the suffix array

Suffix array example

$S = \text{mississippi} \quad q = \text{sip}$

<table>
<thead>
<tr>
<th>Suffixes</th>
<th>Sorted suffixes</th>
<th>Suffix array</th>
</tr>
</thead>
<tbody>
<tr>
<td>mississippi</td>
<td>i</td>
<td>10</td>
</tr>
<tr>
<td>ississippi</td>
<td>ippi</td>
<td>7</td>
</tr>
<tr>
<td>ssissippi</td>
<td>issippi</td>
<td>4</td>
</tr>
<tr>
<td>sissippi</td>
<td>issippi</td>
<td>1</td>
</tr>
<tr>
<td>issippi</td>
<td>issippi</td>
<td>0</td>
</tr>
<tr>
<td>sippi</td>
<td>i</td>
<td>9</td>
</tr>
<tr>
<td>sippi</td>
<td>ippi</td>
<td>8</td>
</tr>
<tr>
<td>ippi</td>
<td>issippi</td>
<td>6</td>
</tr>
<tr>
<td>ppi</td>
<td>issippi</td>
<td>3</td>
</tr>
<tr>
<td>pi</td>
<td>issippi</td>
<td>5</td>
</tr>
<tr>
<td>i</td>
<td>issippi</td>
<td>2</td>
</tr>
</tbody>
</table>

No need to store the suffix strings; just store where they start

$O(|q| \cdot \log |S|)$
One improvement

- Remember how much of the query string has been matched

  \[ q = \text{sisterhood} \]

  \begin{align*}
  \text{low: } & \text{sissipi...} \quad \text{Matched 3 characters} \\
  \text{middle: } & \text{sisterhood...} \quad \text{Start checking from the 4th character} \\
  \text{high: } & \text{sistering...} \quad \text{Matched 6 characters}
  \end{align*}

Another improvement

- Pre-compute the longest common prefix information between suffixes

  \[ q = \text{sisterhood} \quad O(|q| + \log |S|) \]

  \begin{align*}
  \text{low: } & \text{sissipi...} \quad \text{Matched 3 characters} \\
  \text{middle: } & \text{sisterhood...} \quad \text{Start checking from the 7th character} \\
  \text{high: } & \text{sistering...} \quad \text{Matched 6 characters (pre-computed)}
  \end{align*}

Suffix arrays versus inverted lists

- Suffix arrays are more powerful because they index all substrings (not just words)
  - No problem with long phase searches
  - No problem if there is no word boundary
  - No problem with a huge vocabulary of words

- Suffix arrays use more space than inverted lists?
  - Check out compressed suffix arrays (STOC 2000)

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

  \begin{align*}
  \text{root: } & \text{a} \quad \text{b} \quad \text{c} \quad \text{d} \quad \text{e} \\
  \text{path: } & \text{a} \quad \text{b} \quad \text{c} \\
  \text{leaves: } & \text{a} \quad \text{b} \quad \text{c} \quad \text{d} \quad \text{e}
  \end{align*}

  \begin{align*}
  \text{What's the max fan-out?}
  \end{align*}

- 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 the same queries as a suffix array
  - 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 large 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
  - Good exercise: put them in a GiST!

- Two general ways to index for substring queries
  - Index words: inverted lists, signature files
  - Index all suffixes: suffix array, suffix tree, Pat tree

- Web search and information retrieval go beyond substring queries
  - TF/IDF, PageRank, …