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

What are the documents containing both “database” and “search”?
Keywords × documents

<table>
<thead>
<tr>
<th>All keywords</th>
<th>Document 1</th>
<th>Document 2</th>
<th>Document 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;cat&quot;</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>&quot;database&quot;</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>&quot;dog&quot;</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>&quot;search&quot;</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

1 means keyword appears in the document
0 means otherwise

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

Inverted lists

- Store the matrix by rows
- For each keyword, store an inverted list
  - \((\text{keyword}, \text{doc-id-list})\)
  - \("\text{database}"\), \(\{3, 7, 142, 857, \ldots\}\)
  - \("\text{search}"\), \(\{3, 9, 192, 512, \ldots\}\)
  - It helps to sort \(\text{doc-id-list}\) (why?)
- Vocabulary index on keywords
  - \(\text{B}^+\)-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"

- OR? NOT?
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
  - \( \{ \text{keyword}, \{ \langle \text{doc-id}, \text{number-of-occurrences} \rangle, \langle \text{doc-id}, \text{number-of-occurrences} \rangle, \ldots \} \} \)
- Proximity (and frequency)
  - \( \{ \text{keyword}, \{ \langle \text{doc-id}, \text{position-of-occurrence}_1 \rangle, \langle \text{position-of-occurrence}_2, \ldots \rangle, \langle \text{doc-id}, \text{position-of-occurrence}_{1}, \ldots \rangle, \ldots \} \} \)
  - 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

\[
\begin{align*}
  \text{hash}(\text{"database"}) &= 0110 \\
  \text{hash}(\text{"dog"}) &= 1100 \\
  \text{hash}(\text{"cat"}) &= 0010 \\
\end{align*}
\]

Does \( \text{doc}_1 \) contain "database"? 0110
Does \( \text{doc}_1 \) contain "dog"? 1100
Does \( \text{doc}_3 \) contain "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 (*TODX*, 1998)
- Problems of signature files
  - False positives
  - Hard to use because \( n, w \), 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 \(p\): \([p_1 f_1, \ldots, p_n f_n]\)
- Textual similarity between two pages \(p\) and \(q\) is defined to be \([p_1 f_1, \ldots, p_n f_n] \cdot [q_1 f_1, \ldots, q_n f_n] = p_1 q_1 f_1^2 + \ldots + p_n q_n f_n^2\)
  - \(q\) could be the query text

Why weight significance by IDF?

- 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

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

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
  \( \text{PageRank}(p) \leftarrow \sum_{q \in B(p)} (\text{PageRank}(q) / N(q)) \)
  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
- PageRank\( (p) = \frac{d}{N} \sum_{q \in B(p)} (\text{PageRank}(q) / N(q)) + (1 - d) \)

- 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

<table>
<thead>
<tr>
<th>plan</th>
<th>fancy</th>
<th>cap</th>
<th>imp</th>
<th>type</th>
<th>position</th>
<th>in URL/in title tag</th>
<th>in anchor weight for</th>
<th>within the page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>fancy</td>
<td>cap</td>
<td>imp</td>
<td>type</td>
<td>position</td>
<td>in URL/in title tag</td>
<td>in anchor weight for</td>
<td>within the page</td>
</tr>
</tbody>
</table>

- 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

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

\(S = \text{mississippi}\) \(q = \text{sip}\)

<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>mississippi</td>
<td>ippi</td>
<td>7</td>
</tr>
<tr>
<td>sissippi</td>
<td>sippi</td>
<td>4</td>
</tr>
<tr>
<td>sissippi</td>
<td>ississippi</td>
<td>1</td>
</tr>
<tr>
<td>sissippi</td>
<td>mississippi</td>
<td>0</td>
</tr>
<tr>
<td>sippi</td>
<td>pipi</td>
<td>8</td>
</tr>
<tr>
<td>ippi</td>
<td>sippi</td>
<td>6</td>
</tr>
<tr>
<td>pipi</td>
<td>sisissippi</td>
<td>3</td>
</tr>
<tr>
<td>pi</td>
<td>ssissippi</td>
<td>5</td>
</tr>
<tr>
<td>i</td>
<td>ssissippi</td>
<td>2</td>
</tr>
</tbody>
</table>

\(\alpha(q \cdot \log |S|)\)

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

- Remember how much of the query string has been matched
  \[ q = \text{sisterhood} \]

  ...  insipi...  Matched 3 characters
  ...  sisterhood...  Start checking from the 4th character
  ...  sistering...  Matched 5 characters

Another improvement

- Pre-compute the longest common prefix information between suffixes
  - For all (low, middle) and (middle, high) pairs that can come up in a binary search
    \[ q = \text{sisterhood} \]
    \[ \mathcal{O}(|q| + \log |S|) \]

  ...  insipi...  Matched 3 characters
  ...  sisterhood...  Matched 6 characters (pre-computed)
  ...  sistering...  Matched 6 characters

Suffix arrays versus inverted lists
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 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

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, …