Indexing: Part IV

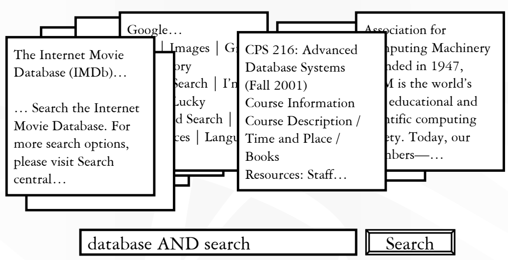
CPS 216

Advanced Database Systems

Announcements

- Homework #2 due in two days (February 26)
  - Typo corrected in Problem 5
  - You may work in groups of three, but then you must complete the optional part of either 8(c) or 8(d)
- Midterm next Monday (March 3)
  - Everything up to (including) today's lecture
  - Open-book, open-notes
- Course project proposal due in 9 days (March 5)
  - By email to `junyang@cs.duke.edu`
- Reading assignment
  - Two papers on cache-sensitive indexing, by Rao and Ross, *VLDB* 1999 and *SIGMOD* 2000

Keyword search



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

Keywords × documents

All documents

All keywords	Document 1	Document 2	Document 3	…	Document *n*
"a"	1	1	1	…	1
"cat"	1	1	0	…	0
"database"	0	0	1	…	0
"dog"	0	1	0	…	1
"search"	0	0	1	…	0
	…	…	…	…	…

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
  - ⟨*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; 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
  - ⟨*keyword*, { ⟨*doc-id*, *number-of-occurrences*⟩,
    ⟨*doc-id*, *number-of-occurrences*⟩,
    … }⟩
- Proximity (and frequency)
  - ⟨*keyword*, { ⟨*doc-id*, ⟨$position\text{-}of\text{-}occurrence_1$,
    $position\text{-}of\text{-}occurrence_2$, …⟩,
    ⟨*doc-id*, ⟨$position\text{-}of\text{-}occurrnece_1$, …⟩⟩,
    … }⟩
  - 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

hash("database") = 0110
hash("dog") = 1100
hash("cat") = 0010

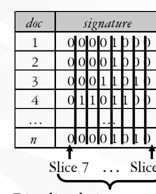
doc_1 contains "database": 0110
doc_2 contains "dog": 1100
doc_3 contains "cat" and "dog": 1110

Does doc_3 contain "database"?

☞ 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



Bit-sliced signature files

Starting to look like an inverted list again!

Inverted lists versus signatures

- Inverted lists are 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
- Saving grace of signature files
  - Good for lots of search terms
  - Good for computing similarity of documents

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 = mississippi q = sip

Suffixes:	Sorted suffixes:	Suffix array:
mississippi	i	10
ississippi	ippi	7
ssissippi	issippi	4
sissippi	ississippi	1
issippi	mississippi	0
ssippi	⇨ pi	9
sippi	ppi	8
ippi	⇨ sippi	6
ppi	⇨ sissippi	3
pi	ssippi	5
i	ssissippi	2

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 = sisterhood

…
low: ⇨ sissipi… Matched 3 characters
…
middle: ⇨ sisterhood… Start checking from the 4th character
…
high: ⇨ 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 = sisterhood $O(|q| + \log |S|)$

…
low: ⇨ sissipi… Matched 3 characters
…
middle: ⇨ sisterhood… Start checking from the 6th character
… Matched 5 characters (pre-computed)
high: ⇨ sistering… Matched 5 characters
…

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)