Web Search:
Indexing Web Pages

CPS 296.1

Topics in Database Systems

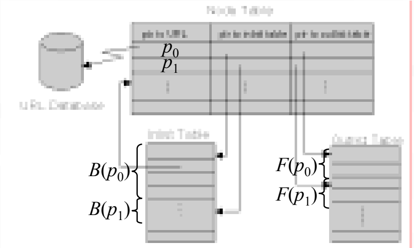
Indexing Web pages

- Indexing the link structure
  - AltaVista Connectivity Server case study
    - Bharat et al., "The Connectivity Server: Fast Access to Linkage Information on the Web." *WWW7*, 1998
  - Google case study
    - Brin and Page, "The Anatomy of a Large-Scale Hypertextual Web Serach Engine." *WWW7*, 1998
- Indexing the text

Indexing the link structure

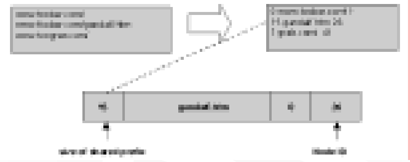
- Main problem: how to store the set of (p_i, p_j) pairs efficiently
- URL certainly identifies a page, but it is too long and has variable length
  - Storing $(\mathrm{URL}_i, \mathrm{URL}_j)$ has too much overhead and therefore slows down access
  - Use a more compact ID and support fast ID/URL conversion

Connectivity Server:
Node table



- ID is the index into the Node Table
- Each link (p_i, p_j) is stored twice

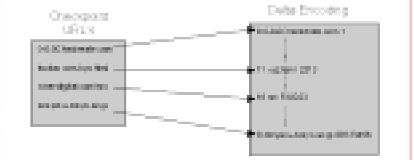
Connectivity Server:
Delta encoding of URL's



- This structure facilitates URL-to-ID translation
- URL's are sorted
- Delta encoding achieves 70% size reduction, but then you can only get a complete URL by starting from the first URL and applying all deltas!

Connectivity Server:
Checkpoint URL's

- Periodically checkpoint URL's by storing the complete URL strings
- A URL is computed by searching linearly from a checkpoint URL
- Pointers to URL (from the Node Table) actually point to checkpoint URL's



Connectivity Server:

Updates

- Tough because the structure is very tight
- Updates are processed in batch daily
- Each deletion is marked by a "tombstone" bit
  - Space is not reclaimed
- New URL's use a separate URL-to-ID translation structure
  - Implemented as a string search tree
- Indexes are completely rebuilt when too much space is wasted by deletions or the separate string search tree is too big

Google as of 1998

- Links database: pairs of docID's
- Document index
  - ISAM on docID
  - Fixed-width record containing document status, checksum, pointer to cached version, pointer to URL in DocInfo file
- DocInfo file
  - Heap file
  - Variable-length record containing URL and title
- URL-to-docID translation structure
  - List of URL checksums with corresponding docID's, sorted by checksum

Indexing Web pages

- Indexing the link structure
- ➢ Indexing the text
  - Inverted lists
  - Signature files
  - Suffix arrays

Words and pages

All pages

All words	Page 1	Page 2	Page 3		Page *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
	…	…	…	…	…

- Inverted lists: store the matrix by rows
- Signature files: store the matrix by columns

With compression, of course!

Inverted lists

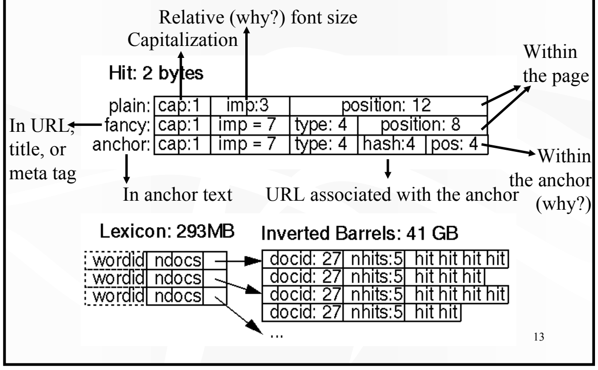
- For each word, store an inverted list
  - <*word*, *page-id-list*>
  - <"database", {3, 7, 142, 857, …}>
  - <"search", {3, 9, 192, 512, …}>
- A vocabulary index for looking up inverted list by word

- Example: find pages containing "database" and "search"
  - Use the vocabulary index to find the two inverted lists
  - Return the page ID's in the intersection, ordered by the ranks of these pages

Ordering of an inverted list

- Should we sort an inverted list by ID or by rank (assuming page ID's are not ordered by the ranks of the pages)?
- By ID?
  - With multiple search terms, intersection = merging
  - But the result always needs to be ranked!
- By rank?
  - With a single search term, just return the list
  - With multiple search terms, intersection is more difficult (involving testing set membership)

Google's inverted lists (1998)



Google's search algorithm (1998)

- Final ranking is not just PageRank
- 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

- Once a certain number of pages (40,000) have been identified in the result set, they are ranked and the top *k* are returned to the user

Signature files

- For each page, 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 page

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 page contains a word, we just 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

page	*signature*							
1	0	0	0	0	1	0	0	0
2	0	0	0	0	1	0	0	0
3	0	0	0	1	1	0	1	0
4	0	1	1	0	1	1	0	0
…				…				
n	0	0	0	0	1	0	1	0

Slice 7 … Slice 0

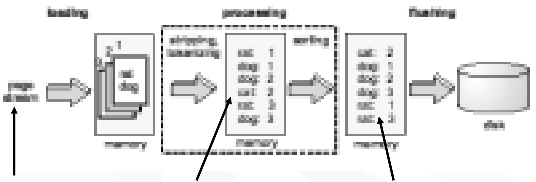
Bit-sliced signature files

It's starting to look like an inverted list again!

Inverted lists versus signature files

- Zobel et al., "Inverted Files versus Signature Files for Text Indexing." *TODS*, 1998
- Inverted lists are better for most purposes
- Problems of signature files
  - False positives
  - Hard to use because *s*, *w*, and the hash function need tuning to work well
  - Long pages 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 pages

Building inverted lists for Web



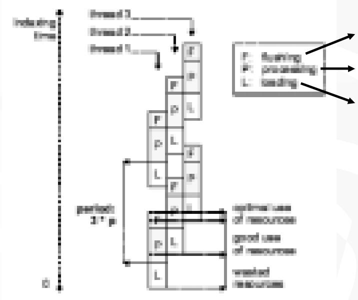
Each indexer gets a stream of pages from a crawler or a page repository

Postings are naturally grouped by page and sorted by position

In each run, postings are ordered by word, page ID, position

Size of a run is limited by the amount of memory
Multiple sorted runs need to merge-sorted

Pipelining index building



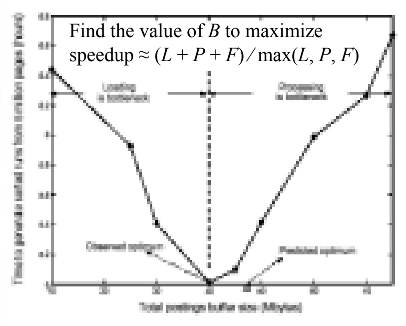
Disk, memory

CPU, memory

Network, memory

If the size of a run is *B*, the amount of memory required for pipelining is 3*B*

Choosing the size of a run



Find the value of *B* to maximize speedup $\approx (L + P + F) / \max(L, P, F)$

Some thoughts:

The analysis does not consider the work left for merging; what if optimal *B* is small compared with the amount of memory available?

Collecting global statistics

- Global statistics (e.g., total number of occurrences of each word on the Web) are needed to evaluate certain ranking criteria (e.g., IDF-based similarity)
- Main ideas
  - Collecting statistics "for free" (no additional I/O's)
    - During flushing or merging
  - Local (early) aggregation to reduce communication
    - For example, count the number of occurrences for each word in each inverted list, and then sum up the counts

Index partitioning

Partition the inverted lists across multiple servers

- IF_L: partition by pages
  - Each server indexes a subset of pages
- IF_G: partition by words
  - Each server indexes a subset of the vocabulary

- IF_L beats IF_G
  - Tomasic & Garcia-Molina, "Performance of Inverted Indices in Shared-Nothing Distributed Text Document Information Retrieval Systems." *PDIS*, 1993

IF_L

- A search involves all servers
- Servers always return pages in the final result
  - No communication is "wasted"
  - Just need to merge-sort the returned pages by rank
- Load is evenly distributed
- If a server goes down, searches will still work but may miss some pages
  - Probably okay for Web searches

IF_G

- A search involves all (one or more) servers indexing the search terms
- Each servers may return pages not in the final result
  - Some communication is "wasted"
  - Need to merge-intersect-sort the returned pages by rank
- Load distribution depends on search term frequencies
- If a server goes down, searches containing certain words cannot processed any more
  - Not okay

Suffix arrays

- Another index for searching text
  - Manber and Myers, “Suffix Arrays: a New Method for On-Line String Searches.” *SODA*, 1990
- Conceptually, to construct a suffix array for a 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
  - Grossi and Vitter, “Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching.” *STOC*, 2000