Search Engines

WS 2009 / 2010

Lecture 6, Thursday November 26th, 2009
(Prefix Search)

Prof. Dr. Hannah Bast
Chair of Algorithms and Data Structures
Department of Computer Science
University of Freiburg

UNI FREIBURG

Overview of today's Lecture

- Everything about Prefix Search
  - how to realize it …

    … using an ordinary inverted index
  - how to realize it efficiently …

    … using a special kind of index
  - what all prefix search is good for
    - for example, synonym search

Prefix Search Demo

- Obvious advantages
  - type less
  - find more
  - find out what words there are in the collection
- Less obvious advantages
  - many advanced search features reduce to prefix search
  - synonym search
  - error tolerant search
  - database-like search
  - semantic search

Prefix Search via an Inverted Index

- Binary search on the (sorted) vocabulary
  - let the size of the vocabulary be n
  - for example bas*
  - time ~ $\log_2 n$ to find the first match (locate bas)
  - time ~ $\log_2 n$ to find the last match (locate bat)
  - so time ~ $\log_2 n$ overall
  - for n = 100 million ≈ 2^27 … $\log_2 n$ is 27
  - one string comparison takes ≈ 1 µsec
  - so a fraction of 1 msec even for large vocabularies
  - but only works if vocabulary fits into memory
  - but: 100 millions words take up ≈ 1GB

about
aware
banks
base
based
bases
basics
basis
bruno
cache
call
cases
…

Permuterm Index

- What if we allow the * in any place
  - for example ba*s
  - should find banks, bases, basics, and basis
  - no longer a range of words (worst case: * in the beginning)
  - scanning the whole vocabulary is too expensive
    - for n = 100 million → 100 seconds
- Idea: Permuterm index
  - append a $ to each word
  - add all permutations for each word
  - for example, for base$ add each of

    base$, ase$b, seba, ebas, $base

Permuterm Index

- Assume three-word vocabulary with
  - banks, base, basics
- Permuterm index
  - simply all permutations sorted
  - each permutation points to the inverted list of the word of which it is a permutation
    (no need to duplicate the lists for each permutation)
  - now for the query ba*s find matches for s$ba*

Permuterm Index

- Efficiency
  - blowup in vocabulary size is about a factor of 8
  - a factor of 8 increases $\log_2 n$ by 3
  - so no problem for the binary searches
  - but a very large vocabulary might not fit into memory anymore
  - note that the size of the inverted lists remains the same (we did not copy the lists, just pointed to them)
- Data structure for very large vocabularies
  - the B-Tree
  - with todays memory, depth 2 is usually enough

[maybe draw picture of B-tree on separate slide]

Permuterm Index

- How about more than one * ?
  - for example in*ma*tik
  - should find informatik
- Simple trick
  - first collapse to one * as in in*tik
  - we already know how to handle this query
  - but this will find a (typically strict) superset of matches
  - for example, will also find intervallarithmetik
  - anyway, the number of matches will be relatively small
  - so just go over them, and filter out the false positives

N-Gram Index

- Can we do with less space than Permuterm?
  - YES WE CAN!
- Idea: Index not the words, but n-grams of the words
  - n-grams of a word = the substrings of length n
  - for example, the 3-grams of $informatik$ are

    $inf, nfo, for, rma, mat, ati, tik, tik$
- N-gram Index
  - Variant 1: let each n-gram point to the words that contain it
  - Variant 2: let each n-gram point to the union of the inverted lists of the doc ids containing it

N-Gram Index

- Why more space-efficient than Permuterm index?
  - because many n-grams are common to many words (whereas the permutations were unique)
  - and anyway, the number of 3-grams is bounded
    - say $128 = 2^7$ symbols which occur at all
    - then at most 2^{21} 3-grams with these symbols
- How to query with the n-gram index?
  - for example, search in*tik
  - contains n-grams $in, tik, and ik$
  - boolean query for $in AND tik AND ik$
  - again, must post-filter … why?

Merging the Inverted Lists

- Whatever we do …
  - … be it binary search, Permuterm, n-Gram index
  - we end up with a large number of inverted lists (one for each word matching the wildcard query)
  - these have to be merged (now it's really merge, not intersection)
  - merging k sorted lists with a total of n elements
    - takes time $n \cdot \log k$

K-Way Merge

UNI FREIBURG

- Algorithm
  - for each of the k lists maintain the current position
  - in each step determine the smallest of the elements at the k current positions
  - output that element and advance by one in that list
  - requires the following data structure
    - at each point have k elements
    - be able to return the smallest of these … fast
    - and replace it with a new one
    - this is called a (fixed-size) priority queue

Priority Queue of fixed size k

- A fixed size priority queue can be easily realized with a heap data structure
  - at each time maintain the heap property:

    each element is larger than its parent

[show example of a heap with 8 elements]

Priority Queue of fixed size k

- Analysis
  - the heap obviously achieves time ~ log k per replace-min

    (a typical priority queue will support get-min, delete-min, and insert separately, but here we only need replace-min)
  - so merging k lists with a total of n elements can be done

    in time $k \cdot \log n$
  - could it possibly be done (asymptotically) faster?
  - No! (At least not comparison-based) Why?
  - otherwise we could sort faster than $n \cdot \log n$

Efficiency of the Approaches so far

- Summary of what we have seen so far:
  - space consumption can be an issue for Permuterm
  - finding (a superset of) the matching words is very fast
  - but then we have to merge all these inverted lists
  - that is very, very, very expensive
    - cost is $C \cdot \log_2 k \cdot$ total size of inverted lists
    - k can easily become 128 → $\log_2 k = 7$
    - $C \approx 5$ compared to a simple scan → $C \cdot \log_2 k \approx 50$
    - total size of inverted are a factor of, say, 2 - 5 larger than a typical inverted list of a single word
    - that is, prefix search several 100 times more expensive than an ordinary keyword search

The HYB index

- HYB is the index behind our CompleteSearch engine
- Simple idea behind HYB
  - precompute inverted lists for unions of words
  - in the following let words be capital letters: A, B, C, …
  - along with each doc id, we now also have to store the word because of which that doc id is in the list

list for **A-D**	1	3	3	5	5	6	7	8	8	9	11	11	11	12	13	15
	D	A	C	A	B	A	C	A	D	A	A	B	C	A	C	A
list for **E-J**	2	2	3	3	4	4	7	7	8	8	9	9	11			
	E	F	G	J	H	I	I	E	F	G	H	J	I			
list for **K-N**	1	1	2	3	4	5	6	6	6	8	9	9	9	10	10	
	L	N	M	N	N	K	L	M	N	M	K	L	M	K	L	

The HYB Index

list for	1	3	3	5	5	6	7	8	8	9	11	11	11	12	13	15
A-D	D	A	C	A	B	A	C	A	D	A	A	B	C	A	C	A
list for	2	2	3	3	4	4	7	7	8	8	9	9	11			
E-J	E	F	G	J	H	I	I	E	F	G	H	J	I			
list for	1	1	2	3	4	5	6	6	6	8	9	9	9	10	10	
K-N	L	N	M	N	N	K	L	M	N	M	K	L	M	K	L	

- How do we do prefix search here?
  - simply find the enclosing blocks (typically only one)
  - scan the block and filter out false-positives

    (that's why we need to store the words along with the doc ids)
  - we are avoiding the merge overhead here

    (which gave the bulk of the cost before: a factor of ≈ 50)

The HYB Index

list for **A-D**	1 D	3 A	3 C	5 A	5 B	6 A	7 C	8 A	8 D	9 A	11 A	11 B	11 C	12 A	13 C	15 A
list for **E-J**	2 E	2 F	3 G	3 J	4 H	4 I	7 I	7 E	8 F	8 G	9 H	9 J	11 I			
list for **K-N**	1 L	1 N	2 M	3 N	4 N	5 K	6 L	6 M	6 N	8 M	9 K	9 L	9 M	10 K	10 L	

- Timewise, this looks good …
  - … but what about the space?
  - we know that lists of sorted doc ids can be compressed well
  - but the list of words are going to completely spoil it, right?

The HYB index — Space Analysis

- Let us analyze
  - the entropy of the ordinary inverted index (INV)
  - the entropy of the HYB index
  - (we already know that the inverted index has great space complexity)
  - (we already know that we can achieve compression close to the entropy)
- Notation
  - let n_i denote the size of the inverted list of the i-th word
  - so the sum of all n_i is just N = total number of all occurrences
  - we will not assume anything about the n_i
  - let n be the total number of documents

Entropy of the INV index

- We will show that the entropy of INV is close to

$$\Sigma\, n_i \cdot \left(1/\ln 2 + \log_2(n/n_i)\right)$$

[prove it live]

Entropy of the HYB index

- We will show that the entropy of HYB is at most

$$\Sigma\, n_i \cdot \big((1+\varepsilon)/\ln 2 + \log_2(n/n_i)\big)$$

where $\varepsilon \cdot n$ is the average number of doc ids in a block

[prove it live]

Synonym Search

- Naïve solution
  - have a synonym dictionary = thesaurus
  - at query time look up each query word in the thesaurus
  - if it's there, replace it with a disjunction of all its synonyms
  - for example uni AND studieren could become

    uni OR universität OR hochschule AND

    studieren OR lernen OR abhängen
  - same problem as for prefix search
    - expanding the query is not hard
    - but, again, computing the union of all the inverted lists (could again be very many) is very expensive

Synonym Search

- Idea: do it via prefix search
  - give each group of synonyms a group id
  - uni, universität, hochschule, etc. get the id 174
  - studieren, lernen, abhängen, etc. get the id 99
  - now in your vocabulary prepend the group id to each word

    syngroup:174:uni
    syngroup:174:universität
    syngroup:99:studieren
  - at query time, determine the group id for each query
  - and replace uni studieren by syngroup:174:* syngroup:99:*
  - if you don't want synonym search just replace the * by the query word, as in syngroup:174:uni syngroup:99:studieren

Prefix Search is extremely universal

- Basically everything can be done with prefix search
  - prefix search
  - autocompletion
  - synonym search
  - error-tolerant search
  - database-like search
  - semantic search
  - factorize arbitrarily large numbers
  - failure-safe lecture recording
  - automatic exercise sheet solving
  - and many more …