15-853:Algorithms in the Real World

Indexing and Searching I (how google and the likes work)

Indexing and Searching Outline

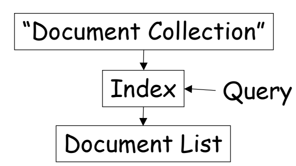
Introduction: model, query types
Inverted File Indices: Compression, Lexicon, Merging
Vector Models:
Latent Semantic Indexing:
Link Analysis: PageRank (Google), HITS
Duplicate Removal:

Indexing and Searching Outline

Introduction:
- model
- query types
- common techniques (stop words, stemming, …)

Inverted File Indices: Compression, Lexicon, Merging
Vector Models:
Latent Semantic Indexing:
Link Analysis: PageRank (Google), HITS
Duplicate Removal:

Basic Model

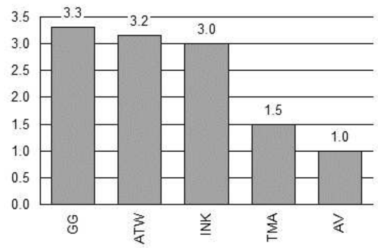


Applications:
- Web, mail and dictionary searches
- Law and patent searches
- Information filtering (e.g., NYT articles)

Goal: Speed, Space, Accuracy, Dynamic Updates

How big is an Index?

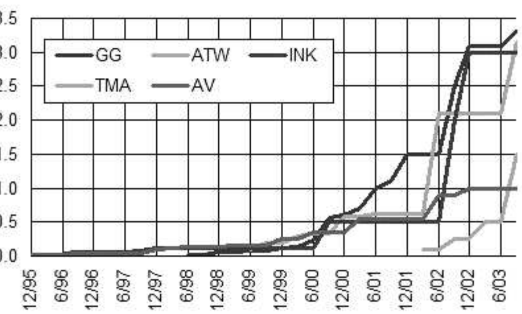
Billion Pages



Sep 2003, self proclaimed sizes (gg = google, atw = alltheweb, ink = inktomi, tma = teoma)

Source: Search Engine Watch

Sizes over time



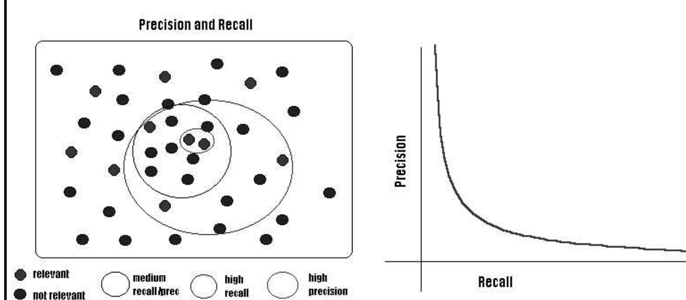
Precision and Recall

Precision: number retrieved that are relevant / total number retrieved

Recall: number relevant that are retrieved / total number relevant

Typically a tradeoff between the two.

Precision and Recall



Does the black or the blue circle have higher precision?

Main Approaches

Full Text Searching
- e.g. grep, agrep (used by many mailers)

Inverted File Indices
- good for short queries
- used by most search engines

Signature Files
- good for longer queries with many terms

Vector Space Models
- good for better accuracy
- used in clustering, SVD, …

Queries

Types of Queries on Multiple "terms"
- boolean (and, or, not, andnot)
- proximity (adj, within <n>)
- keyword sets
- in relation to other documents

And within each term
- prefix matches
- wildcards
- edit distance bounds

Technique used Across Methods

Case folding
London -> london

Stemming
compress = compression = compressed
(several off-the-shelf English Language stemmers are freely available)

Stop words
to, the, it, be, or, …
how about "to be or not to be"

Thesaurus
fast -> rapid

Other Methods

Document Ranking:
Returning an ordered ranking of the results
- A priori ranking of documents (e.g. Google)
- Ranking based on "closeness" to query
- Ranking based on "relevance feedback"

Clustering and "Dimensionality Reduction"
- Return results grouped into clusters
- Return results even if query terms does not appear but are clustered with documents that do

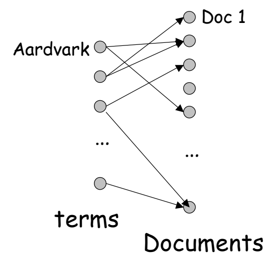
Document Preprocessing
- Removing near duplicates
- Detecting spam

Indexing and Searching Outline

Introduction: model, query types
Inverted File Indices:
- Index compression
- The lexicon
- Merging terms (unions and intersections)

Vector Models:
Latent Semantic Indexing:
Link Analysis: PageRank (Google), HITS
Duplicate Removal:

Documents as Bipartite Graph

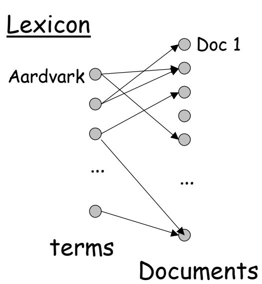


Called an "Inverted File" index
Can be stored using adjacency lists, also called
- posting lists (or files)
- inverted file entry

Example size of TREC database (Text REtrieval Conference)
- 538K terms
- 742K documents
- 333,856K edges

For the web, multiply by 10K

Documents as Bipartite Graph



Implementation Issues:
1. Space for posting lists
   these take almost all the space
2. Access to lexicon
   - btrees, tries, hashing
   - prefix and wildcard queries
3. Merging posting list
   - multiple term queries

1. Space for Posting Lists

Posting lists can be as large as the document data
- saving space and the time to access the space is critical for performance

We can compress the lists,
but, we need to uncompress on the fly.

Difference encoding:
Lets say the term elephant appears in documents:
[3, 5, 20, 21, 23, 76, 77, 78]
then the difference code is
[3, 2, 15, 1, 2, 53, 1, 1]

Some Codes

Gamma code:

if most significant bit of n is in location k, then

gamma(n) = 0^k n[k..0]

2 log(n) – 1 bits

Delta code:

gamma(k)n[k..0]

2 log(log(n)) + log(n) - 1 bits

Frequency coded:

base on actual probabilities of each distance

Global vs. Local Probabilities

Global:

- Count # of occurrence of each distance
- Use Huffman or arithmetic code

Local:

generate counts for each list

elephant: [3, 2, 1, 2, 53, 1, 1]

Problem: counts take too much space

Solution: batching

group into buckets by $\lfloor \log(\text{length}) \rfloor$

Performance

Global	bits/edge
Binary	20.00
Gamma	6.43
Delta	6.19
Huffman	5.83
Local	
Skewed Bernoulli	5.28
Batched Huffman	5.27

Bits per edge based on the TREC document collection

Total size = 333M * .66 bytes = 222Mbytes

2. Accessing the Lexicon

We all know how to store a dictionary, BUT…

- it is best if lexicon fits in memory---can we avoid storing all characters of all words
- what about prefix or wildcard queries?

Some possible data structures

- Front Coding
- Tries
- Perfect Hashing
- B-trees

Front Coding

Word	front coding
7,jezebel	**0,7,jezebel**
5,jezer	**4,1,r**
7,jezerit	**5,2,it**
6,jeziah	**3,3,iah**
6,jeziel	**4,2,el**
7,jezliah	**3,4,liah**

For large lexicons can save 75% of space
But what about random access?

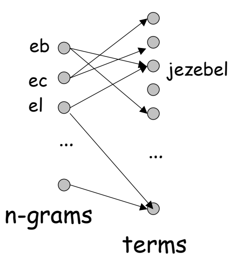
Prefix and Wildcard Queries

Prefix queries
- Handled by all access methods except hashing

Wildcard queries
- n-gram
- rotated lexicon

n-gram

Consider every block of n characters in a term:
e.g. 2-gram of jezebel -> `$j,je,ez,ze,eb,el,l$`



Break wildcard query into an n-grams and search.
e.g. `j*el` would
1. search for `$j,el,l$` as if searching for documents
2. find all potential terms
3. remove matches for which the order is incorrect

Rotated Lexicon

Consider every rotation of a term:
e.g. `jezebel ->`
`$jezebel, l$jezebe, el$jezeb, bel$jeze`

Now store lexicon of all rotations

Given a query find longest contiguous block (with rotation) and search for it:
e.g. `j*el` -> search for `el$j` in lexicon

Note that each lexicon entry corresponds to a single term
e.g. `ebel$jez` can only mean `jezebel`

3. Merging Posting Lists

Lets say queries are expressions over:
- and, or, andnot

View the list of documents for a term as a set:

Then

e_1 and e_2 -> S_1 intersect S_2

e_1 or e_2 -> S_1 union S_2

e_1 andnot e_2 -> S_1 diff S_2

Some notes:
- the sets are ordered in the "posting lists"
- S_1 and S_2 can differ in size substantially
- might be good to keep intermediate results
- persistence is important

Union, Intersection, and Merging

Given two sets of length n and m how long does it take for intersection, union and set difference?

Assume elements are taken from a total order (<)

Very similar to merging two sets A and B, how long does this take?

What is a lower bound?

Union, Intersection, and Merging

Lower Bound:
- There are *n* elements of A and *n + m* positions in the output they could belong
- Number of possible interleavings: $\binom{n+m}{n}$
- Assuming comparison based model, the decision tree has that many leaves
- Max depth is at least log of number of leaves
- Assuming m < n: $\log\binom{n+m}{n} \in \Omega\left(m\log\left(\frac{n+m}{m}\right)\right)$

Merging: Upper bounds

Brown and Tarjan show an O(m log((n + m)/m)) upper bound using 2-3 trees with cross links and parent pointers. Very messy.

We will take different approach, and base an implementation on two operations: split **and** join

Split and Join can then be implemented on many different kinds of trees. We will describe an implementation based on treaps.

Split and Join

Split(S,v) :

Split S into two sets

$S_<$ = {s ∈ S | s < v} and $S_>$ = {s ∈ S | s > v}.

Also return a flag which is true if v ∈ S.

- Split({7,9,15,18,22}, 18) → {7,9,15},{22},True

Join($S_<$, $S_>$) :

Assuming ∀ $k_<$ ∈ $S_<$, $k_>$ in $S_>$: $k_<$ < $k_>$

returns $S_<$ ∪ $S_>$

- Join({7,9,11},{14,22}) → {7,9,11,14,22}

Time for Split and Join

Split(S,v) → ($S_<$, $S_>$),flag **Join($S_<$, $S_>$)** → S

Naively:
- T = O(|S|)

Less Naively:
- T = O(log|S|)

What we want:
- T = O(log(min(|$S_<$|, |$S_>$|))) -- **can be shown**
- T = O(log |$S_<$|) -- **will actually suffice**

Will also use

isEmpty(S) → boolean
- True if the set **S** is empty

first(S) → e
- returns the least element of S
- **first({2,6,9,11,13}) → 2**

{e} → S
- creates a singleton set from an element

We assume they can both run in O(1) time.

An ADT with 5 operations!

Union with Split and Join

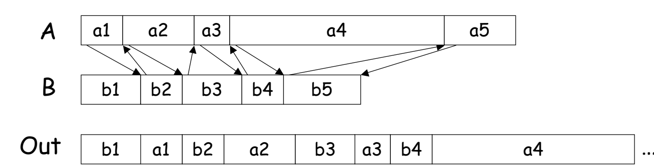
Union(S_1, S_2) =

if **isEmpty(S_1)** then return **S_2**

else

($S_{2<}$, $S_{2>}$, fl) = Split(S_2, first(S_1))

return **Join($S_{2<}$, Union($S_{2>}$, S_1))**



Runtime of Union

Out

o1	o2	o3	o4	o5	o6	o7	o8	...

$T_{union} = O(\sum_i \log |o_i| + \sum_i \log |o_i|)$

Splits Joins

Since the logarithm function is concave, this is maximized when blocks are as close as possible to equal size, therefore

$T_{union} = O(\sum_{i=1}^{m} \log \lceil n/m + 1 \rceil)$

$= O(m \log ((n+m)/m))$

Intersection with Split and Join

Intersect(S_1, S_2) =
  if **isempty**(S_1) then return ∅
  else
    ($S_{2<}$, $S_{2>}$, **flag**) = **Split**(S_2, **first**(S_1))
    if **flag** then
      return **Join**({**first**(S_1)}, **Intersect**($S_{2>}$, S_1))
    else
      return **Intersect**($S_{2>}$, S_1)

Efficient Split and Join

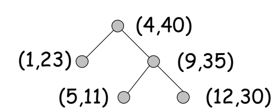
Recall that we want: **T = O(log |$S_<$|)**

How do we implement this efficiently?

Treaps

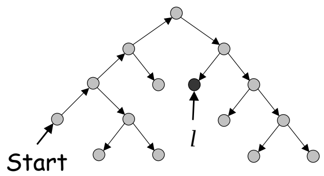
Every key is given a "random" priority.
- keys are stored in-order
- priorities are stored in heap-order

e.g. (key,priority) : (1,23), (4,40), (5,11), (9,35), (12,30)



If the priorities are unique, the tree is unique.

Left Spinal Treap



Time to split = length of path from Start to split location l

We will show that this is O(log L) in the expected case, where L is the number of keys between **Start** and ***l*** (inclusive). 10 in the example.

Time to Join is the same

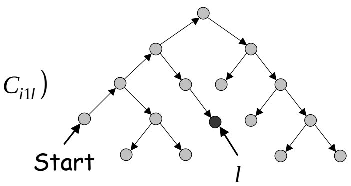
Analysis

P_i = lenght of path from Start to i $\qquad p_i = Ex[P_i]$

$$A_{ij} = \begin{cases} 1 & x_i \text{ ancestor of } x_j \\ 0 & \text{otherwise} \end{cases} \qquad a_{ij} = Ex[A_{ij}]$$

$$C_{ilm} = \begin{cases} 1 & x_i \text{ common ancestor of } x_l \text{ and } x_m \\ 0 & \text{otherwise} \end{cases} \qquad c_{ilm} = Ex[C_{ilm}]$$

$$P_l = \sum_{i=1}^{l} A_{i1} + \sum_{i=1}^{n} \left(A_{il} - C_{i1l}\right)$$



Analysis Continued

$$Ex[P_l] = p_l = \sum_{i=1}^{l} a_{i1} + \sum_{i=1}^{n} \left(a_{il} - c_{i1l}\right)$$

Lemma: $a_{ij} = \dfrac{1}{|i-j|+1}$

Proof:

1. i is an ancestor of j iff i has a greater priority than all elements between i and j, inclusive.
2. there are |i-j|+1 such elements each with equal probability of having the highest priority.

Analysis Continued

$$\sum_{i=1}^{l} a_{i1} = \sum_{i=1}^{l} \frac{1}{|i-1|+1} = \sum_{i=1}^{l} \frac{1}{i}$$
$$< 1 + \ln l \text{ (harmonic number } H_l)$$

Can similarly show that:

$$\sum_{i=1}^{n} \left(a_{il} - c_{i1l}\right) = O(\log l)$$

Therefore the expected path length and runtime for split and join is O(log l).

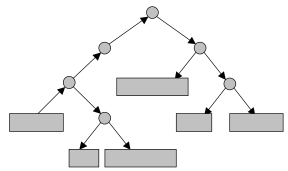
Similar technique can be used for other properties of Treaps.

And back to "Posting Lists"

We showed how to take Unions and Intersections, but Treaps are not very space efficient.

Idea: if priorities are in the range [0..1) then any node with priority < $1 - \alpha$ is stored compressed.

α represents fraction of uncompressed nodes.



Case Study: AltaVista

How AltaVista implements indexing and searching, or at least how they did in 1998.

Based on a talk by A. Broder and M. Henzinger from AltaVista. Henzinger is now at Google, Broder is at IBM.

- The index (posting lists)
- The lexicon
- Query merging (or, and, andnot queries)

The size of their whole index is about 30% the size of the original documents it encodes.

AltaVista: the index

All documents are concatenated together into one sequence of terms (stop words removed).

- This allows proximity queries
- Other companies do not do this, but do proximity tests in a postprocessing phase
- Tokens separate documents

Posting lists contain pointers to individual terms in the single "concatenated" document.

- Difference encoded

Use Front Coding for the Lexicon

AltaVista: the lexicon

The Lexicon is front coded.

- Allows prefix queries, but requires prefix to be at least 3 characters (otherwise too many hits)

AltaVista: query merging

Support expressions on terms involving:
AND, OR, ANDNOT and NEAR

Implement posting list with an abstract data type called an "**Index Stream Reader**" (ISR).

Supports the following operations:

- `loc()` : current location in ISR
- `next()` : advance to the next location
- `seek(k)` : advance to first location past k

AltaVista: query merging (cont.)

Queries are decomposed into the following operations:

`Create`	: term $\rightarrow$ ISR	ISR for the term
`Or`	: ISR * ISR $\rightarrow$ ISR	Union
`And`	: ISR * ISR $\rightarrow$ ISR	Intersection
`AndNot`	: ISR * ISR $\rightarrow$ ISR	Set difference
`Near`	: ISR * ISR $\rightarrow$ ISR	Intersection, almost

Note that all can be implemented with our Treap Data structure.

I believe (from private conversations) that they use a two level hierarchy that approximates the advantages of balanced trees (e.g. treaps).