15-853: Algorithms in the Real World

Indexing and Searching I
- Introduction
- Inverted Indices

Outline for next few classes

Inverted Indices (used by all search engines)
- Compression
- The lexicon
- Merging terms (unions and intersections)

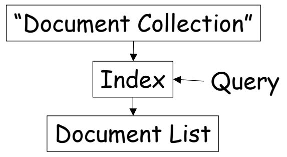
Vector Models

Latent Semantic Indexing

Link Analysis:
- PageRank (Google)
- HITS

Duplicate Removal

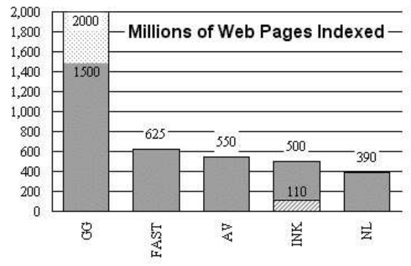
Basic Model



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?



Dec 2001, self proclaimed sizes (gg = google)

Source: Search Engine Watch

Main Approaches

Full text searching
- e.g. grep, agrep (used by many mailers)

Inverted 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

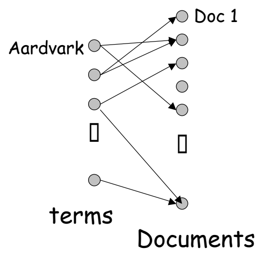
Case folding
London -> london

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

Documents as Bipartite Graph



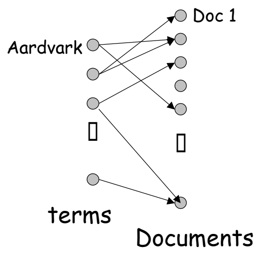
Called an "Inverted File Index"

Can be stored using adjacency lists, also called
- posting lists (or files)
- inverted file entry

Example size of TREC
- 538[illegible] terms
- [illegible]2[illegible] documents
- 333,85[illegible] edges

For the Web, multiply by [illegible]

Documents as Bipartite Graph



Implementation Issues:

□□ **Space □or posting lists**

these take almost all the space

□□ **Access to lexicon**

– btrees, tries, hashing

– prefix and wildcard □ueries

□□ **Merging posting list**

– multiple term □ueries

1. Space for Posting Lists

□osting lists can be as large as the document data

– saving space and the time to access the space is critical for performance

□ e can compress the lists,
but, □e need to uncompress on the □ly□

Di□□erence encoding:

Lets say the term <u>elephant</u> appears in documents:

□□, □, □□, □□, □□, □□, □□, □□□

then the difference code is

□□, □, □□, □, □, □□, □, □□

Some Codes

***Gamma code*:**

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

gamma(n) = 0^{k-1} n□k..0□

2 log(n) – 1 bits

Delta code:

gamma(k)n□k..0□

2 log(log(n)) □ log(n) - 1 bits

□re□uency coded:

base on actual probabilities of each distance

Global vs. Local Probabilities

Global:

– Count □ of occurneces of each distance

– □ se 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 ⌊log(length) ⌋

Performance

Global	bits□edge
Binary	□□□□□
Gamma	□□□□
Delta	□□□□
Huffman	□□□□
Local	
Skewed Bernoulli	□□□□
Batched Huffman	□□□□

□its per edge based on the □R□□ document collection

□otal si□e □ □□□M □ □□□ bytes □ □□□Mbytes

2. Accessing the Lexicon

□ e all kno□ ho□ to store a dictionary, □U□□

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

Some possible data structures

- □ront Coding
- Tries
- Perfect Hashing
- B-trees

□ront Coding

□ ord	□ront coding
□,□ezebel	0,□,□ezebel
5,□ezer	□,1,r
□,□ezerit	5,2,it
□,□eziah	3,3,iah
□,□eziel	□,2,el
□,□ezliah	3,□,liah

□or large lexicons can save □□□ o□ space

□ut □hat about random access□

Prefix and Wildcard Queries

□re□ix □ueries

- Handled by all access methods except hashing

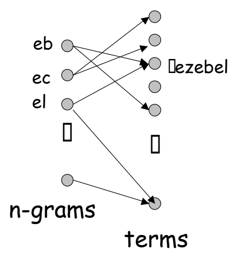
□ ildcard □ueries

- 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. filter matches for which the order does not match

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 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, Merging

Given two sets of length <u>n</u> and <u>m</u> 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?

<u>Lower Bound:</u>

- There are n elements of A and n + m positions in the output they could belong
- choose (n + m, n) possibilities
- assuming comparison based model, the decision tree has that many leaves and depth log of that
- Assuming m < n this give $\Omega(m \log ((n + m)/n))$

Merging: Upper bounds

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

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

Split and Join

Split(S,v) : Split S into two sets $S_< = \{s \in S \mid s < v\}$ and $S_> = \{s \in S \mid s > v\}$. Also return a flag which is true if $v \in S$.

- Split({7,9,15,18,22}, 18) $\rightarrow$ {7,9,15},{22},True

Join($S_<$, $S_>$) : Assuming $\forall\, k_< \in S_<$, $k_>$ in $S_>$: $k_< < k_>$, it returns $S_< \cup S_>$

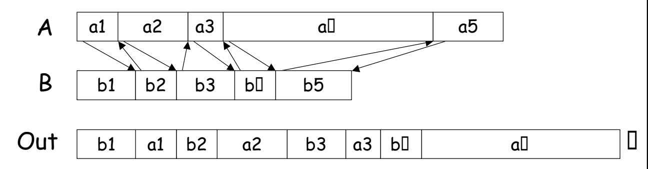
- Join({7,9,11},{14,22}) $\rightarrow$ {7,9,11,14,22}

Time for both:

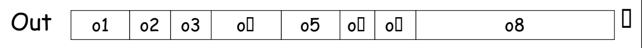
- **O(log(min(|$S_<$|, |$S_>$|))),** can be shown
- **O(log |$S_<$|),** will suffice for us (shown later)

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



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

Splits **Joins**

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

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

Intersection with Split and Join

```
**Intersect**($S_1$, $S_2$) =
  **if** isempty($S_1$) **then return** $\emptyset$
  **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

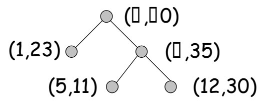
Recall that we want: $T = O(\log |S_1|)$

How do we implement this efficiently?

Treaps

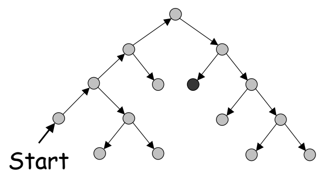
Every key is given a "random" priority.
- keys are stored in-order
- priorities are stored in heap-order

e.g. (key,priority) : (1,23), ([illegible],[illegible]0), (5,11), ([illegible],35), (12,30)



If the priorities are unique, the tree is unique.

Left Spinal Treap



Time to split = length from Start to split location

We will show that this is O(log L) in the expected case, where L is the path length between Start and the split location

Time to join is the same

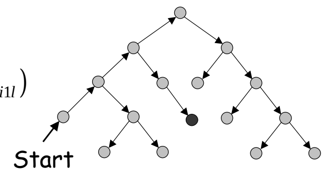
Analysis

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

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

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

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



Analysis Continued

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

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

□roo□:

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

Analysis Continued

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

Can similarly show that:

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

Therefore the epected path length and runtime for split and □oin is O(log l).

Similar techni□ue can be used for other properties of Treaps.

And back to Inverted Indices

□ e sho□ ed ho□ to take Unions and Intersections, but □reaps are not very space e□□icient□

Idea: i□ priorities are in the range □□ □□□) then any node □ith priority □ □ □ α is stored compressed□

α represents □raction o□ uncompressed nodes□