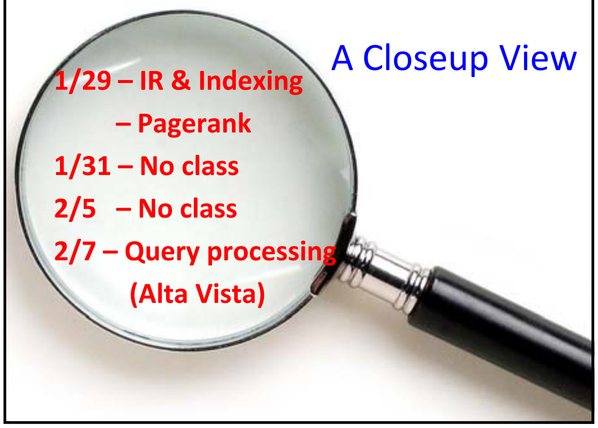
CSE 454

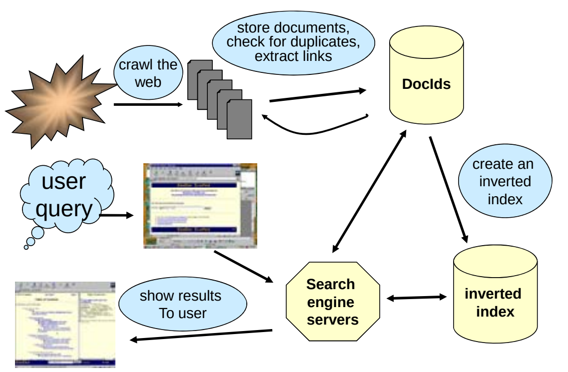
Infrmation Retrieval & Indexing



A Closeup View

1/29 – IR & Indexing
– Pagerank
1/31 – No class
2/5 – No class
2/7 – Query processing
(Alta Vista)

Standard Web Search Engine Architecture



Relevance

- **Complex concept – extensive study**
  – Less consensus
  – People often disagree on relevance
  – Many factors…
- **Retrieval models make various assumptions about relevance to simplify problem**
  – e.g., *topical* vs. *user* relevance
  – e.g., *binary* vs. *multi-valued* relevance

from Croft, Metzler, Strohman. © Addison Wesley

Retrieval Model Overview

- **Older models**
  – Boolean retrieval
  – Overlap Measures
  – Vector Space model
- **Probabilistic Models**
  – BM25
  – Language models
- **Combining evidence**
  – Inference networks
  – Learning to Rank

from Croft, Metzler, Strohman. © Addison Wesley

Test Corpora

TABLE 4.3 Common Test Corpora

Collection	*NDocs*	*NQrys*	*Size (MB)*	*Term/Doc*	*Q-D RelAss*
ADI	82	35			
AIT	2109	14	2	400	>10,000
CACM	3204	64	2	24.5	
CISI	1460	112	2	46.5	
Cranfield	1400	225	2	53.1	
LISA	5872	35	3		
Medline	1033	30	1		
NPL	11,429	93	3		
OSHMED	34,8566	106	400	250	16,140
Reuters	21,578	672	28	131	
TREC	740,000	200	2000	89-3543	» 100,000

slide from Raghavan, Schütze, Larson

Standard Benchmarks

- **National Institute of Standards +Testing (NIST)**
  - Has run large IR testbed for many years (TREC)
- **Reuters and other benchmark sets used**
- **"Retrieval tasks" specified**
  - sometimes as queries
- **Human experts mark, for each query and for each doc, "Relevant" or "Not relevant"**
  - or at least for subset that some system returned

slide from Raghavan, Schütze, Larson

Precision & Recall

Precision

$$\frac{tp}{tp + fp}$$

Proportion of selected items that are correct

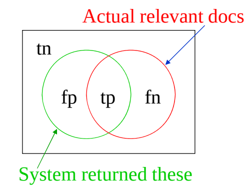
Recall

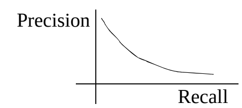
$$\frac{tp}{tp + fn}$$

% of target items that were selected

Precision-Recall curve

Shows tradeoff





Boolean Retrieval

- **Advantages**
  - Results are predictable, relatively easy to explain
  - Many different features can be incorporated
  - Efficient processing since many documents can be eliminated from search
- **Disadvantages**
  - Effectiveness depends entirely on user
  - Simple queries usually don't work well
  - Complex queries are difficult
  - Brittle with user errors (eg misspelling)

from Croft, Metzler, Strohman. © Addison Wesley

Interlude

- **Better Models Coming Soon:**
  - Vector Space model
  - Probabilistic Models
    - BM25
    - Language models
- **Shared Issues – What to Index**
  - Punctuation
  - Case Folding
  - Stemming
  - Stop Words
  - Numbers
  - Font size, titles, anchor text

Punctuation

- ***Ne'er*: use language-specific, handcrafted "locale" to normalize.**
- ***State-of-the-art*: break up hyphenated sequence.**
- ***U.S.A.* vs. *USA* - use locale.**
- ***a.out***

slide from Raghavan, Schütze, Larson

Numbers

- **3/12/91**
- **Mar. 12, 1991**
- **55 B.C.**
- **B-52**
- **100.2.86.144**
  - Generally, don't index as text
  - Creation dates for docs

slide from Raghavan, Schütze, Larson

Case folding

- Reduce all letters to lower case
- Exception: upper case in mid-sentence
  - *e.g., **General Motors***
  - ***Fed*** vs. ***fed***
  - ***SAIL*** vs. ***sail***

slide from Raghavan, Schütze, Larson

Ranking search results

- Boolean queries give inclusion or exclusion of docs.
- Need to assess *quality* of results
  - First attempt: **OVERLAP** between query and document

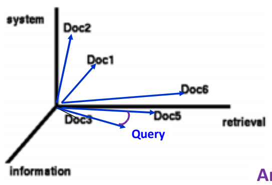
$$|X \cap Y|$$

Words in query — Words in document

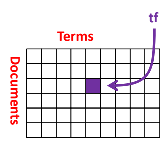
- What's missing?

Vector Space Model

- Each term defines an axis
  Even with stemming, may have 20,000+ dimensions
- Each doc is a vector of *frequency* values
  One "tf" component for each term
- Treat query as just another document
  How measure *distance* in this vector space??



Equivalently



Are all dimensions equivalent?

TF x IDF

$$w_{ik} = tf_{ik} * \log(N / n_k)$$

T_k = term k in document D_i

tf_{ik} = frequency of term T_k in document D_i

idf_k = inverse document frequency of term T_k in C

$$idf_k = \log\left(\frac{N}{n_k}\right)$$

N = total number of documents in the collection C

n_k = the number of documents in C that contain T_k

slide from Raghavan, Schütze, Larson

BM25

Popular and effective ranking algorithm based on binary independence model

- adds document and query term weights

$$\sum_{i \in Q} \log \frac{(r_i+0.5)/(R-r_i+0.5)}{(n_i-r_i+0.5)/(N-n_i-R+r_i+0.5)} \cdot \frac{(k_1+1)f_i}{K+f_i} \cdot \frac{(k_2+1)qf_i}{k_2+qf_i}$$

- N = number of doc, n_i = num containing term I
- R, r_i = encode relevance info (if avail, otherwise = 0)
- f_i = freq of term I in doc; qf_i = freq in doc
- k_1, k_2 and K are parameters, values set empirically
  - k_1 weights tf component as f_i increases
  - k_2 = weights query term weight
  - K normalizes

adapted from Croft, Metzler, Strohman. © Addison Wesley

Simple Formulas

But How Process Efficiently?

Thinking about Efficiency

- **Clock cycle: 4 GHz**
  - Typically *completes* 2 instructions / cycle
    - ~10 cycles / instruction, but pipelining & parallel execution
  - Thus: 8 billion instructions / sec
- **Disk access: 1-10ms**
  - Depends on seek distance, published average is 5ms
  - Thus perform 200 seeks / sec
  - (Ignoring rotation and transfer times)
- **Disk is *40 Million* times slower !!!**

Retrieval

Document-term matrix

	t_1	t_2	. . .	t_j	. . .	t_m	nf
d_1	w_{11}	w_{12}	. . .	w_{1j}	. . .	w_{1m}	$1/\vert d_1\vert$
d_2	w_{21}	w_{22}	. . .	w_{2j}	. . .	w_{2m}	$1/\vert d_2\vert$
. . .	. . .	. . .	. . .	. . .	. . .	. . .	
d_i	w_{i1}	w_{i2}	. . .	w_{ij}	. . .	w_{im}	$1/\vert d_i\vert$
. . .	. . .	. . .	. . .	. . .	. . .	. . .	
d_n	w_{n1}	w_{n2}	. . .	w_{nj}	. . .	w_{nm}	$1/\vert d_n\vert$

w_{ij} is the weight of term t_j in document d_i

Most w_{ij}'s will be zero.

Naïve Retrieval

Consider query $Q = (q_1, q_2, \ldots, q_j, \ldots, q_n)$, nf = $1/|q|$.

How evaluate Q?

(i.e., compute the similarity between q and every document)?

Method 1: Compare Q with every doc.

Document data structure:

$d_i : ((t_1, w_{i1}), (t_2, w_{i2}), \ldots, (t_j, w_{ij}), \ldots, (t_m, w_{im}), 1/|d_i|)$

- **Only terms with positive weights are kept.**
- **Terms are in alphabetic order.**

Query data structure:

$Q : ((t_1, q_1), (t_2, q_2), \ldots, (t_j, q_j), \ldots, (t_m, q_m), 1/|q|)$

Observation

- Method 1 is not efficient
  - Needs to access most non-zero entries in doc-term matrix.
- Solution: Use Index (Inverted File)
  - Data structure to permit fast searching.
- Like an Index in the back of a text book.
  - Key words --- page numbers.
  - E.g, "Etzioni, 40, 55, 60-63, 89, 220"
  - Lexicon
  - Occurrences

Search Processing (Overview)

1. **Lexicon search**
   - E.g. looking in index to find entry
2. **Retrieval of occurrences**
   - Seeing where term occurs
3. **Manipulation of occurrences**
   - Going to the right page

Simple Index for One Document

FILE

POS	
1	**A file is a list of words by position**
10	**First entry is the word in position 1 (first word)**
20	**Entry 4562 is the word in position 4562 (4562^{nd} word)**
30	**Last entry is the last word**
36	**An inverted file is a list of positions by word!**

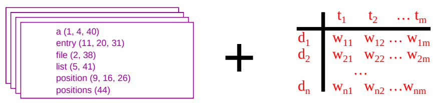
a (1, 4, 40)
entry (11, 20, 31)
file (2, 38)
list (5, 41)
position (9, 16, 26)
positions (44)
word (14, 19, 24, 29, 35, 45)
words (7)
4562 (21, 27)

INVERTED FILE

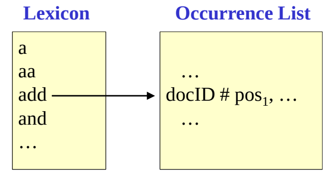
aka "Index"

Requirements for Search

- **Need index structure**
  - Must handle multiple documents
  - Must support phrase queries
  - Must encode TF/IDF values
  - Must minimize disk seeks & reads



How Store Index?

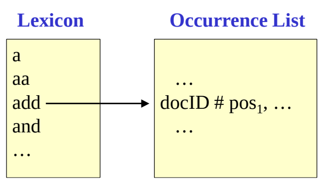


Oracle Database?

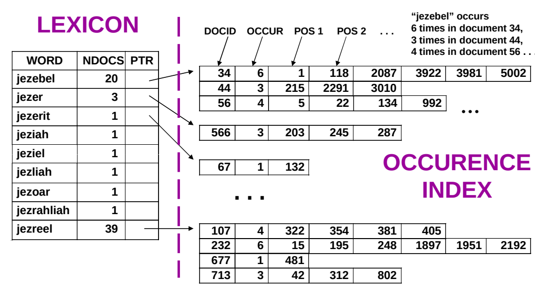
Unix File System?

The Solution

- **Inverted Files for Multiple Documents**
  - Broken into Two Files
- **Lexicon**
  - Hashtable on disk (one read)
  - Nowadays: stored in main memory
- **Occurrence List**
  - Stored on Disk
  - "Google Filesystem"



Inverted Files for Multiple Documents



- **One method. Alta Vista uses alternative**

Many Variations Possible

- **Address space (flat, hierarchical)**
- **Record term-position information**
- **Precalculate TF-IDF info**
- **Stored header, font & tag info**
- **Compression strategies**

Other Features Stored in Index

- **Page Rank**
- **Query word in color on page?**
- **# images on page**
- **# outlinks on page**
- **URL length**
- **Page edit recency**
- **Page Classifiers (20+)**
  - Spam
  - Adult
  - Actor
  - Celebrity
  - Athlete
  - Product / review
  - Tech company
  - Church
  - Homepage
  - ….

Amit Singhai says Google uses over 200 such features [NY Times 2008-06-03]

Using Inverted Files

Some data structures:

Lexicon: a hash table for all terms in the collection.

	. . . . . .
t_j	pointer to $I(t_j)$
	. . . . . .

- **Inverted file lists previously stored on disk.**
- **Now fit in main memory**

The Lexicon

- **Grows Slowly (Heap's law)**
  - $O(n^{\beta})$ where n=text size; β is constant ~0.4 – 0.6
  - E.g. for 1GB corpus, lexicon = 5Mb
  - Can reduce with stemming (Porter algorithm)
- **Store lexicon in file in lexicographic order**
  - Each entry points to loc in occurrence file (aka inverted file list)

Using Inverted Files

Several data structures:

2. For each term t_j, create a list (**occurrence file list**) that contains all document ids that have t_j.

   $I(t_j) = \{ (d_1, w_{1j}),$
   $(d_2, \ldots$
   $\ldots \}$

- **d_i is the document id number of the i^{th} document.**
- **Weights come from freq of term in doc**
- **Only entries with non-zero weights are kept.**

More Elaborate Inverted File

Several data structures:

2. For each term t_j, create a list (**occurrence file list**) that contains all document ids that have t_j.

   $I(t_j) = \{ (d_1, \text{freq}, \text{pos}_1, \ldots \text{pos}_k),$
   $(d_2, \ldots$
   $\ldots \}$

- **d_i is the document id number of the i^{th} document.**
- **Weights come from freq of term in doc**
- **Only entries with non-zero weights are kept.**

Inverted files continued

More data structures:

3. **Normalization factors** of documents are pre-computed and stored similarly to lexicon

   nf[i] stores $1/|d_i|$.

Retrieval Using Inverted Files

initialize all **sim(q, d_i)** = 0
for each term **t_j** in **q**
 find **I(t)** using the hash table
 for each **(d_i, w_{ij})** in **I(t)**
  sim(q, d_i) += $q_j * w_{ij}$
for each (relevant) document **d_i**
 sim(q, d_i) = sim(q, d_i) ∗ nf[i]
sort documents in descending similarities and display the top **k** to the user;

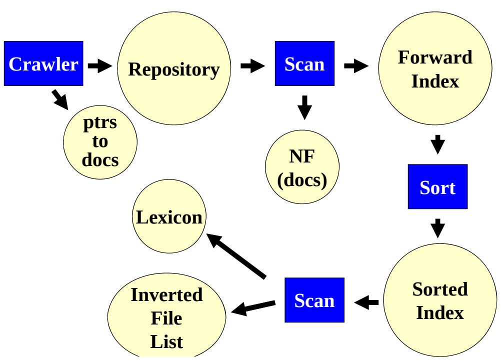
Observations about Method 2

- If doc d **doesn't contain** any term of query q, then d **won't be considered** when evaluating q.
- Only **non-zero** entries in the columns of the document-term matrix which correspond to query terms … are used to evaluate the query.
- Computes the similarities of multiple documents simultaneously (w.r.t. each query word)

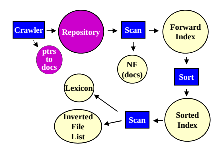
Efficiency versus Flexibility

- Storing computed document weights is good for efficiency, but bad for flexibility.
  - **Recomputation needed if TF and IDF formulas change and/or TF and DF information changes.**
- Flexibility improved by storing raw TF, DF information, but efficiency suffers.
- A compromise
  - **Store pre-computed TF weights of documents.**
  - **Use IDF weights with query term TF weights instead of document term TF weights.**

How Inverted Files are Created

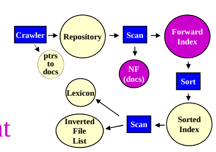


Creating Inverted Files



Repository
- File containing all documents downloaded
- Each doc has unique ID
- Ptr file maps from IDs to start of doc in repository

Creating Inverted Files

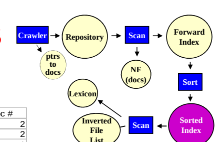


NF ~ Length of each document

Forward Index

Term	Doc #	Pos
I	1	1
did	1	2
enact	1	3
julius	1	4
caesar	1	5
I	1	6
was	1	7

Creating Inverted Files



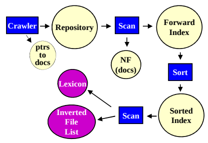
Sorted Index

Term	Doc #
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1

Term	Doc #
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1

(positional info as well)

Creating Inverted Files



Lexicon

WORD	NDOCS	PTR
jezebel	20	
jezer	3	
jezerit	1	
jeziah	1	
jeziel	1	
jezliah	1	
jezoar	1	
jezrahliah	1	
jezreel	39	

DOCID OCCUR POS 1 POS 2 . . .

34	6	1	118	2087	3922	3981	5002
44	3	215	2291	3010			
56	4	5	22	134	992		

566	3	203	245	287

67	1	132

Inverted File List

. . .

Compression

- **What Should We Compress?**
  – Repository
  – Lexicon
  – Inv Index
- **What properties do we want?**
  – Compression ratio
  – Compression speed
  – Decompression speed
  – Memory requirements
  – Pattern matching on compressed text
  – Random access

Inverted File Compression

Each inverted list has the form $< f_t \,;\, d_1, d_2, d_3, \ldots, d_{f_t} >$

A naïve representation results in a storage overhead of $(f + n) * \lceil \log N \rceil$

This can also be stored as $< f_t; d_1, d_2 - d_1, \ldots, d_{f_t} - d_{f_t - 1} >$

Each difference is called a d-gap. Since $\sum (d - gaps) \le N$,

each pointer requires fewer than $\lceil \log N \rceil$ bits.

Trick is encoding …. since worst case ….

Assume d-gap representation for the rest of the talk, unless stated otherwise

Slides adapted from Tapas Kanungo and David Mount, Univ Maryland

Text Compression

Two classes of text compression methods

- Symbolwise (or statistical) methods
  – **Estimate probabilities of symbols - modeling step**
  – **Code one symbol at a time - coding step**
  – **Use shorter code for the most likely symbol**
  – **Usually based on either arithmetic or Huffman coding**
- Dictionary methods
  – **Replace fragments of text with a single code word**
  – **Typically an index to an entry in the dictionary.**
    - **eg: Ziv-Lempel coding: replaces strings of characters with a pointer to a previous occurrence of the string.**
  – **No probability estimates needed**

Symbolwise methods are more suited for coding d-gaps

