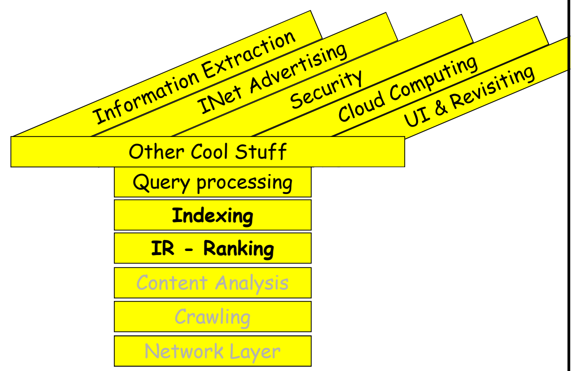
CSE 454

Infrmation Retrieval & Indexing

Class Overview



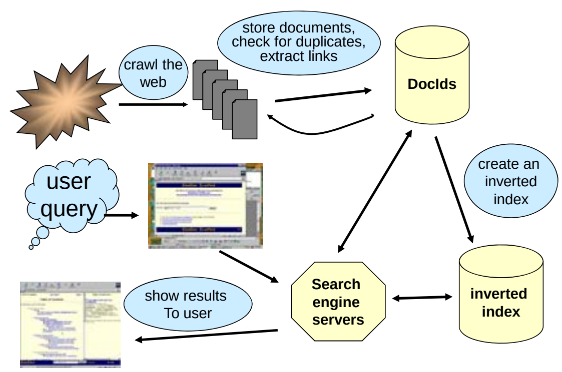
A Closeup View

10/19 – IR & Indexing

10/21 – Google & Alta Vista

10/26 – Pagerank

Standard Web Search Engine Architecture



Relevance

- **Complex concept that has been studied for some time**
  - Many factors to consider
  - People often disagree when making relevance judgments
- **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: fraction of retrieved docs that are relevant = P(relevant|retrieved)**
- **Recall: fraction of relevant docs that are retrieved = P(retrieved|relevant)**

	Relevant	Not Relevant
Retrieved	tp	fp
Not Retrieved	fn	tn

- **Precision P = tp/(tp + fp)**
- **Recall R = tp/(tp + fn)**

slide from Raghavan, Schütze, Larson

Precision & Recall

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

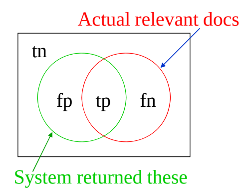
Proportion of selected items that are correct

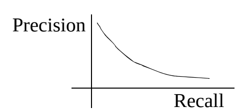
Recall $\frac{tp}{tp + fn}$

% of target items that were selected

Precision-Recall curve

Shows tradeoff





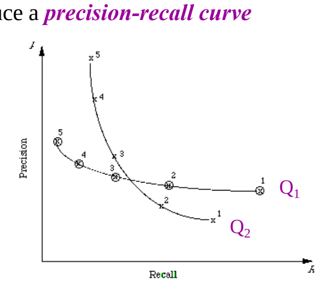
Precision/Recall

- **Can get high recall (but low precision)**
  - Retrieve all docs on all queries!
- **Recall is a non-decreasing function of the number of docs retrieved**
  - Precision usually decreases (in a good system)
- **Difficulties in using precision/recall**
  - Binary relevance
  - Should average over large corpus/query ensembles
  - Need human relevance judgements
  - Heavily skewed by corpus/authorship

slide from Raghavan, Schütze, Larson

Precision-Recall Curves

- **May return any # of results ordered by similarity**
- **By varying numbers of docs (levels of recall)**
  - Produce a ***precision-recall curve***



slide from Raghavan, Schütze, Larson

A combined measure: F

- **Combined measure assessing tradeoff is F measure (weighted harmonic mean):**

$$F = \frac{1}{\alpha\frac{1}{P} + (1-\alpha)\frac{1}{R}} = \frac{(\beta^2+1)PR}{\beta^2 P + R}$$

- **People usually use balanced F_1 measure**
  - i.e., with $\beta = 1$ or $\alpha = ½$
- **Harmonic mean is conservative average**
  - See CJ van Rijsbergen, *Information Retrieval*

slide from Raghavan, Schütze, Larson

Other Measures

- **Precision at fixed recall**
  - This is perhaps the most appropriate thing for web search: all people want to know is how many good matches there are in the first one or two pages of results
- **11-point interpolated average precision**
  - The standard measure in the TREC competitions: Take the precision at 11 levels of recall varying from 0 to 1 by tenths of the documents, using interpolation (the value for 0 is always interpolated!), and average them

slide from Raghavan, Schütze, Larson

Boolean Retrieval

- **Two possible outcomes for query processing**
  - TRUE and FALSE
  - "exact-match" retrieval
  - simplest form of ranking
- **Query specified w/ Boolean operators**
  - AND, OR, NOT
  - proximity operators also used

from Croft, Metzler, Strohman. © Addison Wesley

Query

- **Which plays of Shakespeare contain the words *Brutus AND Caesar* but *NOT Calpurnia*?**

slide from Raghavan, Schütze, Larson

Term-document incidence

	Tempest	Hamlet	Othello	Macbeth
Antony	0	0	0	1
Brutus	0	1	0	0
Caesar	0	1	1	1
Calpurnia	0	0	0	0
Cleopatra	0	0	0	0
mercy	1	1	1	1
worser	1	1	1	0

1 if play contains word, 0 otherwise

slide from Raghavan, Schütze, Larson

Booleans over Incidence Vectors

- **So we have a 0/1 vector for each term.**
- **To answer query: take the vectors for *Brutus, Caesar* and *Calpurnia* (complemented) ➔ bitwise *AND*.**
- **110100 *AND* 110111 *AND* 101111 = 100100.**

slide from Raghavan, Schütze, Larson

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

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
  - Spelling
  - Numbers

Issues in what to index

Cooper's concordance of Wordsworth was published in 1911. The applications of full-text retrieval are legion: they include résumé scanning, litigation support and searching published journals on-line.

- ***Cooper's* vs. *Cooper* vs. *Coopers*.**
- ***Full-text* vs. *full text* vs. *{full, text}* vs. *fulltext.***
- ***résumé* vs. *resume*.**

slide from Raghavan, Schütze, Larson

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

Thesauri and Soundex

- **Handle synonyms and homonyms**
  - Hand-constructed equivalence classes
    - e.g., ***car*** = ***automobile***
    - ***your*** ≠ ***you're***
- **Index such equivalences?**
- **Or expand query?**

slide from Raghavan, Schütze, Larson

Spell Correction

- **Look for all words within (say) edit distance 3 (Insert/Delete/Replace) at query time**
  - *e.g.,* ***Alanis Morisette***
- **Spell correction is expensive and slows the query (up to a factor of 100)**
  - Invoke only when index returns zero matches?
  - What if docs contain mis-spellings?

slide from Raghavan, Schütze, Larson

Lemmatization

- **Reduce inflectional/variant forms to base form**
  - *am, are, is* → *be*
  - *car, cars, car's, cars'* → *car*

the boy's cars are different colors

→

the boy car be different color

slide from Raghavan, Schütze, Larson

Stemming

- **Reduce terms to their "roots" before indexing**
  - language dependent
  - e.g., ***automate(s), automatic, automation*** all reduced to ***automat***.

for example compressed and compression are both accepted as equivalent to compress.	➡	for exampl compres and compres are both accept as equival to compres.

slide from Raghavan, Schütze, Larson

Porter's algorithm

- **Common algorithm for stemming English**
- **Conventions + 5 phases of reductions**
  - phases applied sequentially
  - each phase consists of a set of commands
  - sample convention: *Of the rules in a compound command, select the one that applies to the longest suffix.*
- **Porter's stemmer available:** http//www.sims.berkeley.edu/~hearst/irbook/porter.html

slide from Raghavan, Schütze, Larson

Typical rules in Porter

- ***sses*** → ***ss***
- ***ies*** → ***i***
- ***ational*** → ***ate***
- ***tional*** → ***tion***

slide from Raghavan, Schütze, Larson

Challenges

- **Sandy**
- **Sanded**
- **Sander**

→ Sand ???

slide from Raghavan, Schütze, Larson

Beyond Term Search

- **Phrases?**
- **Proximity: Find *Gates NEAR Microsoft*.**
  – Index must capture position info in docs.
- **Zones in documents: Find documents with (*author = Ullman*) *AND* (text contains *automata*).**

slide from Raghavan, Schütze, Larson

Ranking search results

- **Boolean queries give inclusion or exclusion of docs.**
- **Need to measure *proximity* from query to each doc.**
- **Whether docs presented to user are singletons, or a group of docs covering various aspects of the query.**

slide from Raghavan, Schütze, Larson

Ranking models in IR

- **Key idea:**
  – *We wish to return in order the documents most likely to be useful to the searcher*
- **To do this, we want to know which documents *best* satisfy a query**
  – An obvious idea is that if a document talks about a topic *more* then it is a better match
- **A query should then just specify terms that are relevant to the information need, without requiring that all of them must be present**
  – Document relevant if it has a lot of the terms

slide from Raghavan, Schütze, Larson

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

Binary term presence matrices

- **Record whether a document contains a word: document is binary vector in $\{0,1\}^v$**
- **Idea: Query satisfaction = overlap measure:**

$$|X \cap Y|$$

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	1	1	0	0	0	1
Brutus	1	1	0	1	0	0
Caesar	1	1	0	1	1	1
Calpurnia	0	1	0	0	0	0
Cleopatra	1	0	0	0	0	0
mercy	1	0	1	1	1	1
worser	1	0	1	1	1	0

slide from Raghavan, Schütze, Larson

Overlap matching

- **What are the problems with the overlap measure?**
- **It doesn't consider:**
  - Term frequency in document
  - Term scarcity in collection
    - (How many documents mention term?)
  - Length of documents

slide from Raghavan, Schütze, Larson

Many Overlap Measures

$\lvert Q \cap D \rvert$	Simple matching (coordination level match)
$2\frac{\lvert Q \cap D \rvert}{\lvert Q \rvert + \lvert D \rvert}$	Dice's Coefficient
$\frac{\lvert Q \cap D \rvert}{\lvert Q \cup D \rvert}$	Jaccard's Coefficient
$\frac{\lvert Q \cap D \rvert}{\lvert Q \rvert^{\frac{1}{2}} \times \lvert D \rvert^{\frac{1}{2}}}$	Cosine Coefficient
$\frac{\lvert Q \cap D \rvert}{\min(\lvert Q \rvert, \lvert D \rvert)}$	Overlap Coefficient

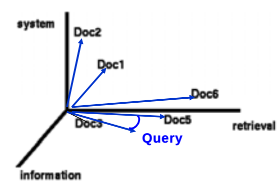
slide from Raghavan, Schütze, Larson

Documents as vectors

- **Each doc *j* can be viewed as a vector of *tf* values, one component for each term**
- **So we have a vector space**
  - terms are axes
  - docs live in this space
  - even with stemming, may have 20,000+ dimensions
- **(The corpus of documents gives us a matrix, which we could also view as a vector space in which words live – transposable data)**

slide from Raghavan, Schütze, Larson

Vector Space Representation



Documents that are close to query
(measured using vector-space metric)
=> returned first.

slide from Raghavan, Schütze, Larson

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?

Copyright © Weld 2002-2007

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
  - (And we are ignoring rotation and transfer times)
- **Disk is *40 Million* times slower !!!**

Copyright © Weld 2002-2007

Retrieval

Document-term matrix

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

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

Most w_{ij}'s will be zero.

Copyright © Weld 2002-2007

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:

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

Copyright © Weld 2002-2007

Naïve Retrieval (continued)

Method 1: Compare q with documents directly

initialize all $sim(q, d_i) = 0$;
for each document d_i $(i = 1, \ldots, n)$
  { **for each** term t_j $(j = 1, \ldots, m)$
    if t_j appears in both q and d_i
      $sim(q, d_i) \mathrel{+}= q_j * w_{ij}$;
    $sim(q, d_i) = sim(q, d_i)\ \cancel{*(1/|q|)}\ *(1/|d_i|)$; }
sort documents in descending similarities;
display the top k to the user;

Copyright © Weld 2002-2007

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

Copyright © Weld 2002-2007

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

Copyright © Weld 2002-2007

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 (4562nd 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)

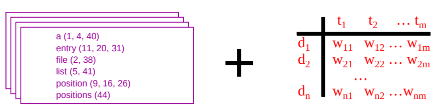
INVERTED FILE

aka "Index"

Copyright © Weld 2002-2007

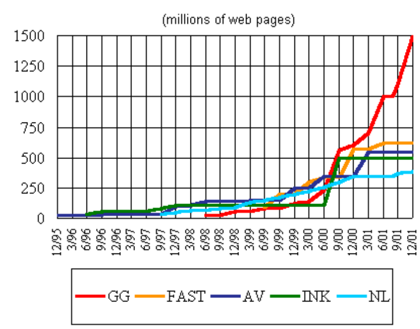
Requirements for Search

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



Copyright © Weld 2002-2007

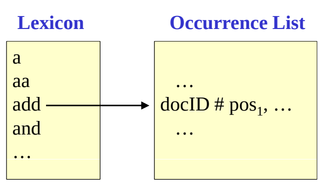
Index Size over Time



Now >> 50 Billion Pages

Copyright © Weld 2002-2007

How Store Index?

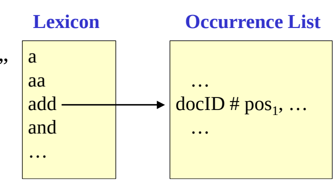


Oracle Database?

Unix File System?

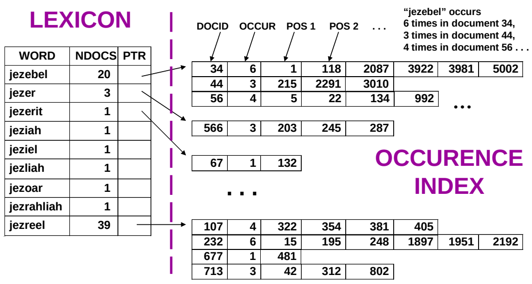
The Solution

- **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"



Copyright © Weld 2002-2007

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, freq, pos_1, … pos_k),
   (d_2, …
   … }

   – **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)

Efficient Retrieval

Example (Method 2): Suppose

q = { (t1, 1), (t3, 1) }, 1/|q| = 0.7071

d1 = { (t1, 2), (t2, 1), (t3, 1) }, nf[1] = 0.4082
d2 = { (t2, 2), (t3, 1), (t4, 1) }, nf[2] = 0.4082
d3 = { (t1, 1), (t3, 1), (t4, 1) }, nf[3] = 0.5774
d4 = { (t1, 2), (t2, 1), (t3, 2), (t4, 2) }, nf[4] = 0.2774
d5 = { (t2, 2), (t4, 1), (t5, 2) }, nf[5] = 0.3333
I(t1) = { (d1, 2), (d3, 1), (d4, 2) }
I(t2) = { (d1, 1), (d2, 2), (d4, 1), (d5, 2) }
I(t3) = { (d1, 1), (d2, 1), (d3, 1), (d4, 2) }
I(t4) = { (d2, 1), (d3, 1), (d4, 1), (d5, 1) }
I(t5) = { (d5, 2) }

q = { (t1, 1), (t3, 1) }, 1/|q| = 0.7071

Efficient Retrieval

d1 = { (t1, 2), (t2, 1), (t3, 1) }, nf[1] = 0.4082
d2 = { (t2, 2), (t3, 1), (t4, 1) }, nf[2] = 0.4082
d3 = { (t1, 1), (t3, 1), (t4, 1) }, nf[3] = 0.5774
d4 = { (t1, 2), (t2, 1), (t3, 2), (t4, 2) }, nf[4] = 0.2774
d5 = { (t2, 2), (t4, 1), (t5, 2) }, nf[5] = 0.3333

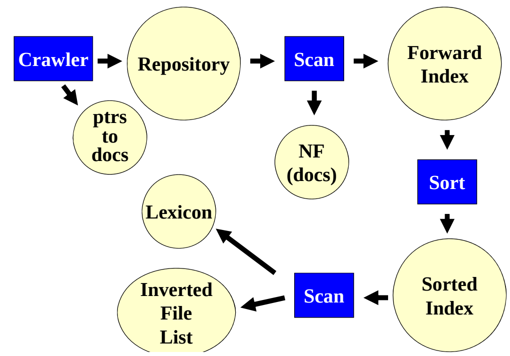
I(t1) = { (d1, 2), (d3, 1), (d4, 2) }
I(t2) = { (d1, 1), (d2, 2), (d4, 1), (d5, 2) }
I(t3) = { (d1, 1), (d2, 1), (d3, 1), (d4, 2) }
I(t4) = { (d2, 1), (d3, 1), (d4, 1), (d5, 1) }
I(t5) = { (d5, 2) }

After t1 is processed:
sim(q, d1) = 2, sim(q, d2) = 0,
sim(q, d3) = 1
sim(q, d4) = 2, sim(q, d5) = 0
After t3 is processed:
sim(q, d1) = 3, sim(q, d2) = 1,
sim(q, d3) = 2
sim(q, d4) = 4, sim(q, d5) = 0
After normalization:
sim(q, d1) = .87, sim(q, d2) = .29,
sim(q, d3) = .82
sim(q, d4) = .78, sim(q, d5) = 0

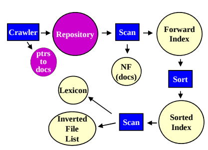
Efficiency versus Flexibility

- 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

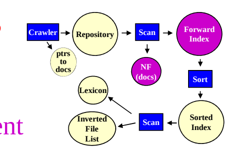


Creating Inverted Files



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

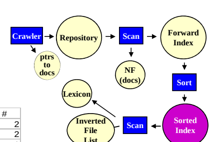


NF ~ Length of each document

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



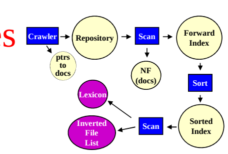
Sorted Index

(positional info as well)

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

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

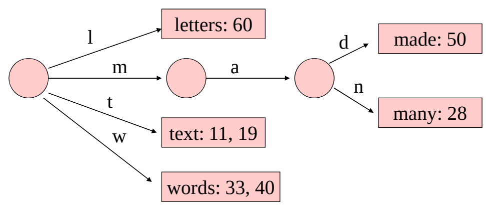
Inverted File List

. . .

Lexicon Construction

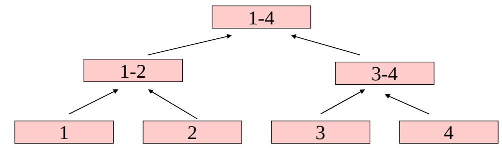
- **Build Trie (or hash table)**

1 6 9 11 17 19 24 28 33 40 46 50 55 60
This is a text. A text has many words. Words are made from letters.



Copyright © Weld 2002-2007

Memory Too Small?



- **Merging**
  - When word is shared in two lexicons
  - Concatenate occurrence lists
  - O(n1 + n2)
- **Overall complexity**
  - O(n log(n/M))

Copyright © Weld 2002-2007

Stop lists

- **Language-based stop list:**
  - words that bear little meaning
  - 20-500 words
  - http://www.dcs.gla.ac.uk/idom/ir_resources/linguistic_utils/stop_words
- **Subject-dependent stop lists**
- **Removing stop words**
  - From document
  - From query

From Peter Brusilovsky Univ Pittsburg INFSCI 2140

Copyright © Weld 2002-2007

Stemming

- **Are there different index terms?**
  - retrieve, retrieving, retrieval, retrieved, retrieves…
- **Stemming algorithm:**
  - (retrieve, retrieving, retrieval, retrieved, retrieves) ⇨ retriev
  - Strips prefixes of suffixes (-s, -ed, -ly, -ness)
  - Morphological stemming

Copyright © Weld 2002-2007

Stemming Continued

- Can reduce vocabulary by ~ 1/3
- C, Java, Perl versions, python, c#
  www.tartarus.org/~martin/PorterStemmer
- Criterion for removing a suffix
  - Does "a document is about w_1" mean the same as
  - a "a document about w_2"
- Problems: sand / sander & wand / wander
- Commercial SEs use giant in-memory tables

Copyright © Weld 2002-2007

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

Copyright © Weld 2002-2007

Inverted File Compression

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

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

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

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

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

Copyright © Weld 2002-2007

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

Copyright © Weld 2002-2007

Classifying d-gap Compression Methods:

- **Global: each list compressed using same model**
  - **non-parameterized**: probability distribution for d-gap sizes is predetermined.
  - **parameterized**: probability distribution is adjusted according to certain parameters of the collection.
- **Local: model is adjusted according to some parameter, like the frequency of the term**
- **By definition, local methods are parameterized.**

Copyright © Weld 2002-2007

Conclusion

- **Local methods best**
- **Parameterized global models ~ non-parameterized**
  - Pointers not scattered randomly in file
- **In practice, best index compression algorithm is:**
  - Local Bernoulli method (using Golomb coding)
- **Compressed inverted indices usually faster+smaller than**
  - Signature files
  - Bitmaps

Local < Parameterized Global < Non-parameterized Global

Not by much

Copyright © Weld 2002-2007