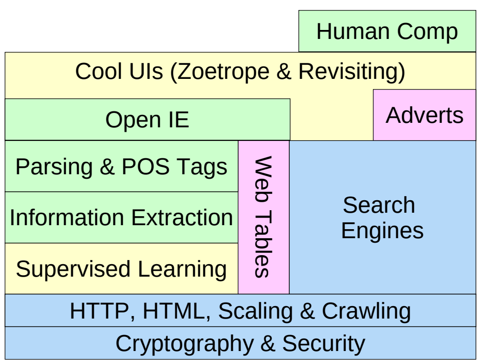
CSE 454

Indexing

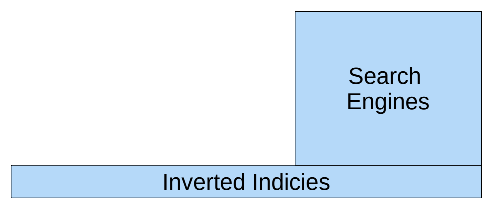
Todo

- **A bit repetitive – cut some slides**
- **Some inconsistencie – eg are positions in the index or not.**
- **Do we want nutch as case study instead of google?**

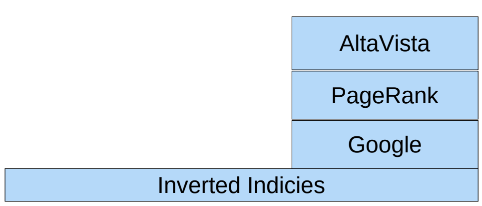
CSE 454 Overview



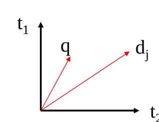
CSE 454 Overview



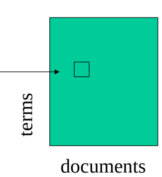
CSE 454 Overview



Review



- **Vector Space Representation**
  - Dot Product as Similarity Metric
- **TF-IDF for Computing Weights**
  - $w_{ij} = f(i,j) * \log(N/n_i)$
  - Where q = … $word_i$ …
  - N = |docs| n_i = |docs with $word_i$|



- **But How Process Efficiently?**

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.

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:

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

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
$sim(q, d_i) += q_j * w_{ij}$;
$sim(q, d_i) = sim(q, d_i) \cancel{*(1/|q|)} *(1/|d_i|)$; }
sort documents in descending similarities;
display the top k to the user;

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 (4562nd word)**
30	**Last entry is the last word**
36	**An inverted file is a list of positions by word!**

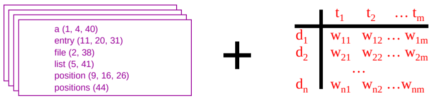
a (1, 4, 40)
entry (11, 20, 31)
file (2, 38)
list (5, 41)
position (9, 16, 26)
positions (44)
word (14, 19, 24, 29, 35, 45)
words (7)
4562 (21, 27)

INVERTED FILE

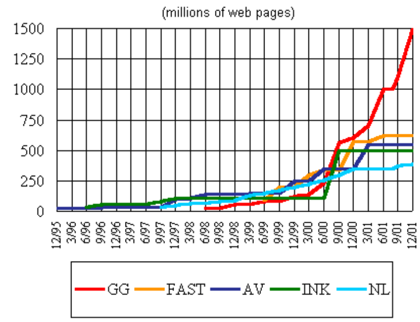
aka "Index"

Requirements for Search

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



Index Size over Time



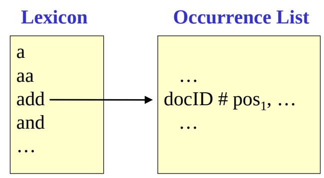
Number of indexed pages, self-reported
Google: 50% of the web?

Thinking about Efficiency

- **Clock cycle: 2 GHz**
  - Typically *completes* 2 instructions / cycle
    - ~10 cycles / instruction, but pipelining & parallel execution
  - Thus: 4 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 *20 Million* times slower !!!**
- **Store index in Oracle database?**
- **Store index using files and unix filesystem?**

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

LEXICON

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

OCCURENCE INDEX

DOCID OCCUR POS 1 POS 2 . . .

"jezebel" occurs
6 times in document 34,
3 times in document 44,
4 times in document 56 . . .

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

. . .

107	4	322	354	381	405		
232	6	15	195	248	1897	1951	2192
677	1	481					
713	3	42	312	802			

- **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**

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)

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) }

Efficient Retrieval

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

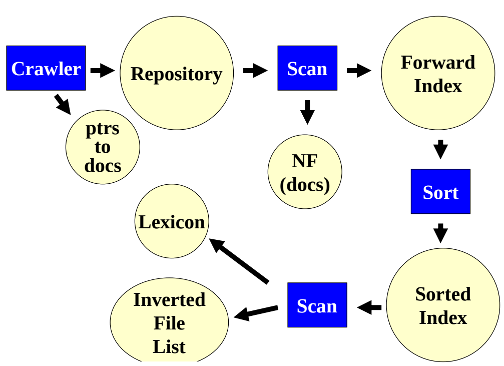
I(t1) = { (d1, 2), (d3, 1), (d4, 2) }
I(t2) = { (d1, 1), (d2, 2), (d4, 1), (d5, 2) }
I(t3) = { (d1, 1), (d2, 1), (d3, 1), (d4, 2) }
I(t4) = { (d2, 1), (d3, 1), (d4, 1), (d5, 1) }
I(t5) = { (d5, 2) }

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

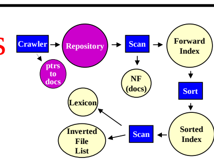
Efficiency versus Flexibility

- 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

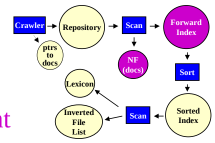


Creating Inverted Files



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

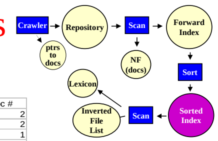


NF ~ Length of each document

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



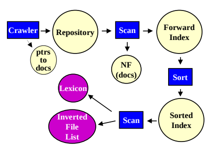
Term	Doc #
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1

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

Sorted Index

(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

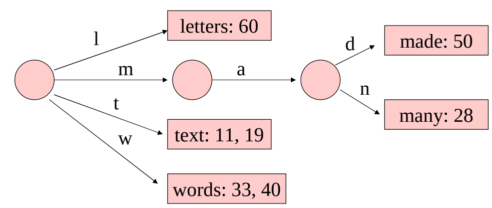
Inverted File List

. . .

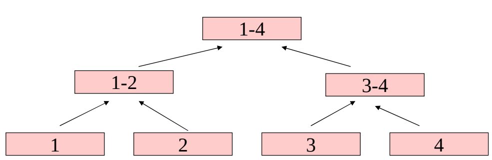
Lexicon Construction

- **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.



Memory Too Small?



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

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

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

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

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 $< f_t\,;\, d_1\,,\, d_2\,,\, d_3\,,\, ...\,,\, d_{f_t} >$

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

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

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.**

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