Algoritmi per IR

Storage of the Posting Lists

Recall the Inverted index

Brutus	2	4	6	10	32			
the	1	2	3	5	8	13	21	34
Calpurnia	13	16						

Some recent results [WWW 2008]

- HTML collection takes 36Gb zipped
- pos + docsIDs ≈ 8Gb
- docIDs only ≈ 3Gb

Gap-coding for postings

- Sort the docIDs (two advantages....)
- Store gaps between consecutive docIDs:
  - ***Brutus***: 33, 47, 154, 159, 202 ...
    33, 14, 107, 5, 43 ...

Two advantages:

- Space: store smaller integers (clustering?)
- Speed: query requires just a scan

γ–code for integer encoding

0000.....0	x in binary
Length-1	

- x > 0 and Length = $\lfloor \log_2 x \rfloor + 1$

  e.g., 9 represented as <000,1001>.

- γ–code for x takes $2 \lfloor \log_2 x \rfloor + 1$ bits
  (ie. factor of 2 from optimal)

A rough analysis on Zipf law and γ-coding…

- Zipf Law: *k*th term occurs ≅ *c* n/k times, for a proper constant c depending on the data collection.
- Wiki EN consists of about 4Mil docs
- *γ*-coding on the k-th term costs:
  - *n/k* gaps, use $2\log_2 k + 1$ bits for each gap;
  - Log is *concave*, thus the maximum cost is for equal-gaps = $(2cn/k)\log_2 k + (cn/k)$ bits

This is about 170Mb vs more than 2.5 Gb of 22-bit

δ–code for integer encoding

γ(Length)	Offset

- Use γ-coding to reduce the length of the first field
- Useful for *medium*-sized integers

  e.g., 19 represented as <00,101,10011>.

- δ–coding *x* takes about

  $\lfloor \log_2 x \rfloor + 2\lfloor \log_2(\lfloor \log_2 x \rfloor + 1) \rfloor + 2$ bits.

- Optimal for $Pr(x) = 1/2x(\log x)^2$, and i.i.d integers

Variable-byte codes

- Wish to get very fast (de)compress → byte-align

e.g., $v=2^{14}+1$ → *binary(v)* = 100000000000001

1*000000*1 10000000 00000001

Note: We waste 1 bit per byte, and avg 4 for the first byte.

Golomb codes [7.54 bits on TREC12]

Unary(q+1)	Binary rest
[q times 0s] 1	r ≤ d ? r : (r+d)

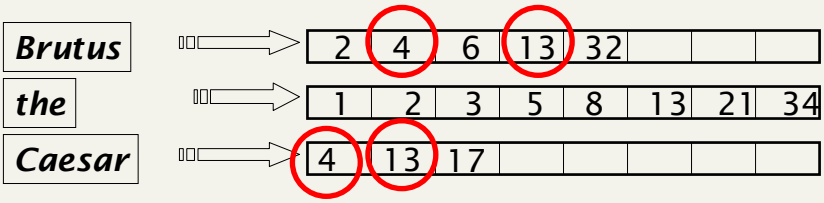
If r ≤ d use ⌊log k⌋ bits (*enough*), else use ⌈log k⌉ bits

- It is a parametric code: **depends on k**
- We set $d = 2^{\lfloor \log k \rfloor + 1} - k$, hence d may be represented in ⌊log k⌋ bits
- Quotient $q=\lfloor (v-1)/k \rfloor$, and the rest is $r= v - k * q - 1$
  - In the second case r > d, so that (r+d)/2 > d → Consequently, the first ⌊log k⌋ bits discriminate the case in decompression
- Useful when integers concentrated around k: k=3, v=4 → d=4, <01,0>
- Usually k ≈ 0.69 * *mean*(v) [Bernoulli model]
  - Optimal for $Pr(x) = p\,(1-p)^{x-1}$, where mean(x)=1/p, and i.i.d ints
- If k is a power-of-2, then we have the simpler Rice code

Algoritmi per IR

Query Processing

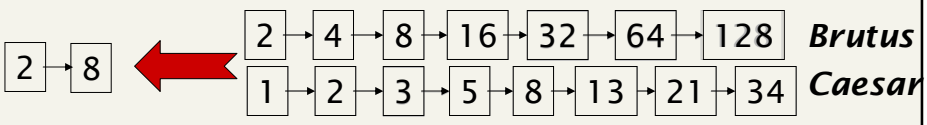
Query processing



1) Retrieve all pages matching the query
2) Order pages according to various scores:
   - Term position & freq (body, title, anchor,...)
   - Link popularity
   - User clicks or preferences

Pages matching the query

- Consider the query: ***Brutus*** *AND* ***Caesar***



If the list lengths are *m* and *n*, this takes O(*m*+*n*) time.

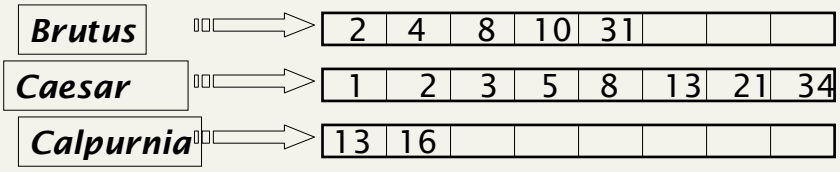
Crucial: postings sorted by docID

(one reason for doing this! Recall gap-coding).

Some optimization

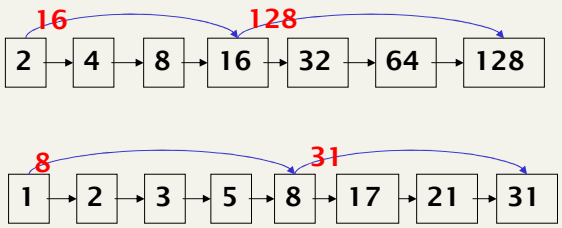
Best order for query processing ?

- Shorter lists first…



Query: ***Brutus*** *AND* ***Calpurnia*** *AND* ***Caesar***

Use skip pointers



- To skip postings that will not figure in the search results, take 16 vs 8.

Lucene
stores one out of 16

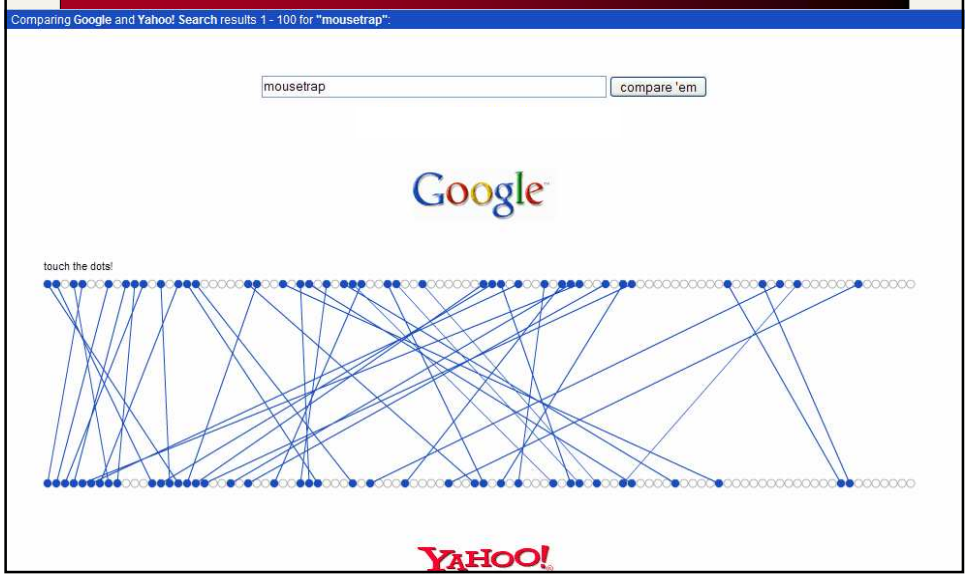
Phrase queries

- Expand the posting lists with word positions
  - ***to***:
    - *2*:1,17,74,222,551; *4*:8,16,190,429,433; *7*:13,23,191; ...
  - ***be***:
    - *1*:17,19; *4*:17,191,291,430,434; *5*:14,19,101; ...
- Larger space, about 3-4 times more

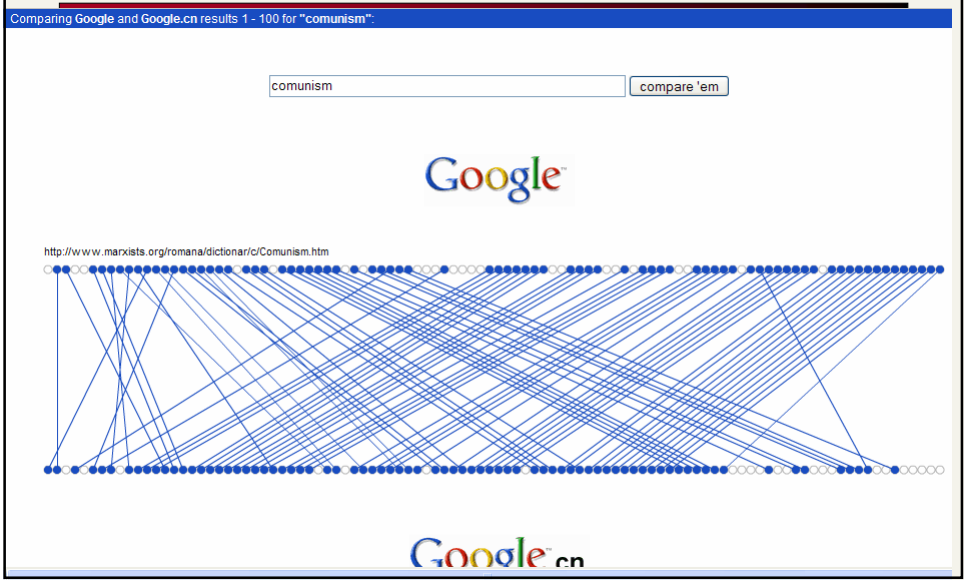
Algoritmi per IR

Ranking

The big fight: find the best *ranking*...



Ranking: Google *vs* Google.cn



Algoritmi per IR

Text-based Ranking

(1° generation)

Similarity between binary vectors

- Document is binary vector X,Y in $\{0,1\}^v$
- Score: *overlap measure*

$$|X \cap Y|$$

What's wrong ?

	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

Normalization

- Dice coefficient (wrt avg #terms):

$$2|X \cap Y|/(|X| + |Y|)$$

NO, triangular

- Jaccard coefficient (wrt possible terms):

$$|X \cap Y|/|X \cup Y|$$

OK, triangular

What's wrong in doc-similarity ?

Overlap matching doesn't consider:

- Term frequency in a document
  - Talks more of t ? Then t should be weighted more.
- Term scarcity in collection
  - ***of*** commoner than ***baby bed***
- Length of documents
  - score should be normalized

A famous "weight": tf-idf

$$w_{t,d} = tf_{t,d} \times \log(n / n_t)$$

$tf_{t,d}$ = Frequency of term t in doc d = $\#occ_t / |d|$

$$idf_t = \log\left(\frac{n}{n_t}\right)$$

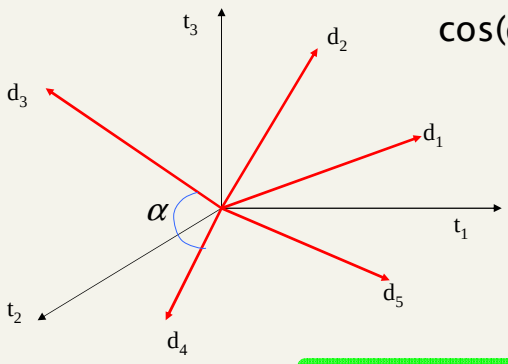
where n_t = #docs containing term t
n = #docs in the indexed collection

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	13,1	11,4	0,0	0,0	0,0	0,0
Brutus	3,0	8,3	0,0	1,0	0,0	0,0
Caesar	2,3	2,3	0,0	0,5	0,3	0,3
Calpurnia	0,0	11,2				
Cleopatra	17,7	0,0				
mercy	0,5	0,0				
worser	1,2	0,0	0,6	0,6	0,6	0,0

Vector Space model

A graphical example

Easy to Spam



$\cos(\alpha) = v \bullet w \,/\, ||v|| * ||w||$

Sophisticated algos to find top-k docs for a query Q

The user query is a very short doc

Postulate: Documents that are "close together" in the vector space talk about the same things. Euclidean distance sensible to vector length !!

MG book

$$w_{t,d} = 1 + \log tf_{t,d}$$

$$w_{t,q} = idf_t = \log(1 + \frac{n}{n_t})$$

- |d| may be precomputed, |q| may be considered as a constant;

$$sim(q,d) = \frac{\sum_t w_{t,q} \times w_{t,d}}{|d|} = \frac{\sum_t (1 + \log tf_{t,d}) \times idf_t}{|d|}$$

Vector spaces and other operators

- Vector space OK for bag-of-words queries
  - Clean metaphor for similar-document queries
  - Not a good combination with operators: Boolean, wild-card, positional, proximity

- ***First generation*** **of search engines**
  - **Invented before "spamming" web search**

Relevance Feedback: Rocchio

- User sends his/her query Q
- Search engine returns its best results
- User marks some results as *relevant* and resubmits query plus *marked* results (repeat the query !!)
- Search engine exploit this refined knowledge of the user need to return more relevant results.

$$\vec{Q}_{new} = \alpha \times \vec{Q}_{orig} + \beta \times \frac{1}{R} \sum_{D \in Rel} \vec{D} - \gamma \times \frac{1}{N-R} \sum_{D \notin Rel} \vec{D}$$

Approximate top-k results

- Preprocess: Assign to each term, its *m* best documents

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	13.1	11.4	0.0	0.0	0.0	0.0
Brutus	3.0	8.3	0.0	1.0	0.0	0.0
Caesar	2.3	2.3	0.0	0.5	0.3	0.3
Calpurnia	0.0	11.2	0.0	0.0	0.0	0.0
Cleopatra	17.7	0.0	0.0	0.0	0.0	0.0
mercy	0.5	0.0	0.7	0.9	0.9	0.3
worser	1.2	0.0	0.6	0.6	0.6	0.0

- Search:
  - If |Q| = *q* terms, merge their preferred lists ($\leq mq$ answers).
  - Compute COS between Q and these docs, and choose the top *k*.

Need to pick *m*>*k* to work well empirically.

Now SE use tf-idf PLUS PageRank (PLUS other weights)

Advanced: *Fancy-hits* heuristic

- Preprocess:
  - Assign docID by decreasing PR weight
  - Define FH(t) = *m* docs for t with highest tf-idf weight
  - Define IL(t) = the rest (i.e. incr docID = decr PR weight)
- Idea: a document that scores high should be in FH or the front of IL
- Search for a t-term query:
  - First FH: Take the common docs of their FH
    - compute the score of these docs and keep the top-*k* docs.
  - Then IL: scan ILs and check the common docs
    - Compute the score and possibly insert them into the top-*k*.
    - Stop when m docs have been checked or the PR score < some threshold.