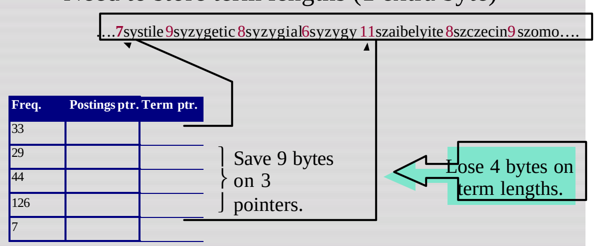
CS347

Lecture 3

April 16, 2001

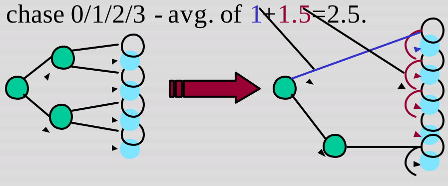
Blocking

- Store pointers to every *k*th on term string.
- Need to store term lengths (1 extra byte)



Impact on search

- Binary search down to 4-term block;
- Then linear search through terms in block.
- Instead of chasing 2 pointers before, now chase 0/1/2/3 - avg. of 1+1.5=2.5.



Wild-card queries

- ***mon****: find all docs containing any word beginning “mon”.
- Solution: **index** all *k*-grams occurring in any doc (any sequence of *k* chars).
- *e.g.*, from text “April is the cruelest month” we get the 2-grams (*bigrams*)
  - $ is a special word boundary symbol

a,ap,pr,ri,il,l,i,is,s,t,th,he,e,c,cr,ru,ue,el,le,es,st,t, m,mo,on,nt,h

Today's topics

- Index construction
  - time and strategies
- Dynamic indices - updating
- Term weighting and vector space indices

Somewhat bigger corpus

- Number of docs = n = 4M
- Number of terms = m = 1M
- Use Zipf to estimate number of postings entries:
- $n + n/2 + n/3 + \ldots. + n/m \sim n \ln m = 56\text{M}$ entries
- No positional info yet

Check for yourself

Index construction

- As we build up the index, cannot exploit compression tricks
  - parse docs one at a time, final postings entry for any term incomplete until the end
- At 10-12 bytes per postings entry, demands several hundred temporary megabytes

System parameters for design

- Disk seek ~ 1 millisecond
- Block transfer from disk ~ 1 microsecond per byte
- All other ops ~ 10 microseconds

Recall index construction

- Documents are parsed to extract words and these are saved with the Document ID.

Doc 1

I did enact Julius Caesar I was killed i' the Capitol; Brutus killed me.

Doc 2

So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious

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
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

- After all documents have been parsed the inverted file is sorted by terms

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
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

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
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

Bottleneck

- Parse and build postings entries one doc at a time
- To now turn this into a term-wise view, must sort postings entries by term (then by doc within each term)
- Doing this with random disk seeks would be too slow

If every comparison took 1 disk seek, and *n* items could be sorted with $n\log_2 n$ comparisons, how long would this take?

Sorting with fewer disk seeks

- 12-byte (4+4+4) records *(term, doc, freq)*.
- These are generated as we parse docs.
- Must now sort 56M such records by *term*.
- <u>Block</u> = 1M such 12-byte records, can "easily" fit a couple into memory.
- Will sort within blocks first, then merge multiple blocks.

Sorting 56 blocks of 1M records

- First, read each block and sort within:
  - Quicksort takes about 2 x (1M ln 1M) steps
- *Exercise: estimate total time to read each block from disk and and quicksort it.*
- 56 times this estimate - gives us 56 sorted *runs* of 1M records each.
- Need 2 copies of data on disk, throughout.

Merging 56 sorted runs

- Merge tree of $\log_2 56 \sim 6$ layers.
- During each layer, read into memory runs in blocks of 1M, merge, write back.
- Time estimate for disk transfer:
- $6 \times 56 \times (\underbrace{12M \times 10^{-6}}_{\text{disk block transfer time}}) \times 2 \sim 2$ hours.

Work out for yourself how these transfers are staged, and the total time for *merging*.

Large memory indexing

- Suppose instead that we had 1GB of memory for the above indexing task.
- *Exercise: how much time to index?*
- In practice, spidering interlaced with indexing.
  - Spidering bottlenecked by WAN speed.

Improving on merge tree

- Compressed temporary files
  - compress terms in temporary dictionary runs
- Merge more than 2 runs at a time
  - maintain heap of candidates from each run

Dynamic indexing

- Docs come in over time
  - postings updates for terms already in dictionary
  - new terms added to dictionary
- Docs get deleted

Simplest approach

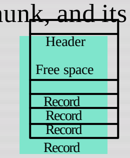
- Maintain "big" main index
- New docs go into "small" auxiliary index
- Search across both, merge results
- Deletions
  - Invalidation bit-vector for deleted docs
  - Filter docs output on a search result
- Periodically, re-index into one main index

More complex approach

- Fully dynamic updates
- Only one index at all times
  - No big and small indices
- Active management of a pool of space

Fully dynamic updates

- Inserting a (variable-length) record
  - a typical postings entry
- Maintain a pool of (say) 64KB *chunks*
- Chunk header maintains metadata on records in chunk, and its free space



Global tracking

- In memory, maintain a global record address table that says, for each record, the chunk it's in.
- Define one chunk to be current.
- Insertion
  - if current chunk has enough free space
    - extend record and update metadata.
  - else look in other chunks for enough space.
  - else open new chunk.

Changes to dictionary

- New terms appear over time
  - cannot use a static perfect hash for dictionary
- OK to use term char string w/pointers from postings as in lecture 2.

Digression: food for thought

- What if a doc consisted of *components*
  - Each component has its own *access control list.*
- Your search should get a doc only if your query meets one of its components that <u>you</u> have access to.
- More generally: doc assembled from *computations* on components.
- Welcome to the real world … more later.

Weighting terms

- Relative importance of
  - 0 vs. 1 occurrence of a term in a doc
  - 1 vs. 2 occurrences
  - 2 vs. 3 occurrences …
- (The Kandy-Kolored Tangerine-Flake Streamline Baby)

Weighting should depend on term

- Which of these tells you more about a doc?
  - 10 occurrences of ***hernia***?
  - 10 occurrences of ***the***?

Properties of weights

- Assign a weight to each term in each doc
  - Increases with the number of occurrences *within* a doc
  - Increases with the "rarity" of the term *across* the whole corpus

tf x idf weights

- *tf* x *idf* measure:
  - term frequency (*tf*)
    - measure of term density in a doc
  - inverse document frequency (*idf*)
    - measure of rarity across corpus
- Goal: assign a *tf* x *idf* weight to each term in each document

tf x idf

$$w_{ij} = tf_{ij} \times \log(n / n_i)$$

What is the wt of a term that occurs in all of the docs?

tf_{ij} = frequency of term i in document j

n = total number of documents

n_i = the number of documents that contain term i

$idf_i = \log\left(\frac{n}{n_i}\right)$ = inverse document frequency of term i

Doc as vector

- Each doc *j* can now be viewed as a vector of *tf*×*idf* 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 10000+ dimensions

Example

Doc 1:
Beauty is truth and truth beauty.

Doc 2:
A thing of beauty is a joy forever.

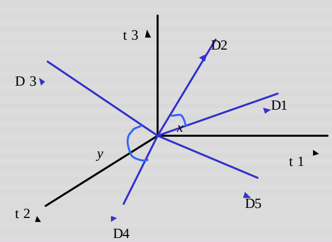
Term	tf in Doc1	tf in Doc2	idf	tfxidf: Doc 1	tfxidf: Doc 2
beauty	0.33	0.125	0	0	0
is	0.16666	0.125	0	0	0
truth	0.33333	0	1	0.33333	0
and	0.16666	0	1	0.16666	0
a	0	0.25	1	0	0.25
thing	0	0.125	1	0	0.125
of	0	0.125	1	0	0.125
joy	0	0.125	1	0	0.125
forever	0	0.125	1	0	0.125

Note: *idf* (and thus *tf* x *idf*) can exceed 1.

Why turn docs into vectors?

- First application: Query-by-example
  - – Given a doc *D*, find others “like” it.
- Now that *D* is a vector, find vectors (docs) “near” it.

Intuition



Postulate: Documents that are “close together” in vector space talk about the same things.

Desiderata for proximity

- If *D1* is near *D2*, then *D2* is near *D1*.
- If *D1* near *D2*, and *D2* near *D3*, then *D1* not far from *D3*.
- No doc is closer to *D* than *D* itself.

First cut

- Distance between *D1* and *D2* is the length of the vector |*D1*-*D2*|.
  - Euclidean distance
- Why is this not a great idea?

tf x idf normalization

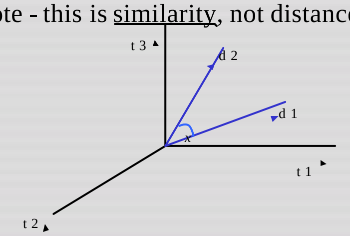
- Normalize the term weights
  - longer documents are not given more weight

$$w_{tj} = \frac{tf_{tj} \log(n/n_t)}{\sqrt{\sum_{i=1}^{m} (tf_{ij})^2 [\log(n/n_i)]^2}}$$

Now all docs have the same vector lengths.

Cosine similarity

- Distance between vectors *D1,D2 captured* by the cosine of the angle *x* between them.
- Note - this is similarity, not distance.



Cosine similarity

Cosine similarity of D_j, D_k:

$$sim(D_j, D_k) = \sum_{i=1}^{m} w_{ij} \times w_{ik}$$

Aka normalized inner product.

$$D_1 = (0.8, 0.6)$$
$$D_2 = (0.707, 0.707)$$
$$Q = (0.5, 0.866)$$
$$\cos a_1 = 0.9196$$
$$\cos a_2 = 0.965762$$

So D_2 adjudged closer to query document Q.

Cosine similarity exercises

- *Exercise: Rank the following by decreasing cosine similarity:*
  - Two docs that have only frequent words ***(the, a, an, of)*** in common.
  - Two docs that have no words in common.
  - Two docs that have many rare words in common ***(wingspan, tailfin).***

What's the real point of using vector spaces?

- **<u>Key</u>**: A user's query can be viewed as a (very) short document.
- Query becomes a vector in the same space as the docs.
- Can measure each doc's proximity to it.
- Natural measure of scores/ranking - no longer Boolean.

Back to our example

Doc 1:
Beauty is truth and truth beauty.

Doc 2:
A thing of beauty is a joy forever.

Term	tf in Doc1	tf in Doc2	idf	tfxidf: Doc 1	tfxidf: Doc 2
beauty	0.33	0.125	0	0	0
is	0.16666	0.125	0	0	0
truth	0.33333	0	1	0.33333	0
and	0.16666	0	1	0.16666	0
a	0	0.25	1	0	0.25
thing	0	0.125	1	0	0.125
of	0	0.125	1	0	0.125
joy	0	0.125	1	0	0.125
forever	0	0.125	1	0	0.125

On the query ***truth forever*** Doc 1 scores 0.16666, while Doc 2 scores 0.0625.

Vector space issues

+ Can rank docs
+ Uniform view of docs and queries

- Cannot insist all query terms be present in docs retrieved
  - queries are not Boolean
- Each axis treated as independent: ***dog, canine***
- Computation/selection of top cosines

Notions from linear algebra for next lecture

- Matrix, vector
- Matrix transpose and product
- Rank
- Eigenvalues and eigenvectors.

Resources, and beyond

- MG 5, MIR 2.5.3.
- Next steps
  - Computing cosine similarity efficiently.
  - Dimension reduction.
  - Bayesian nets.
  - Clustering docs into groups of similar docs.
  - Classifying docs automatically.