Faster TF-IDF

David Kauchak
cs160
Fall 2009

adapted from:
http://www.stanford.edu/class/cs276/handouts/lecture6-tfidf.ppt

Administrative

- Assignment 1
- Assignment 2
  - Look at the assignment by Wed!
  - New turnin procedure
- Class participation

Stoplist and dictionary size

Recap: Queries as vectors

- Represent the queries as vectors
- Represent the documents as vectors

- proximity = similarity of vectors

- What do the entries in the vector represent in the tf-idf scheme?

Recap: tf-idf weighting

- The tf-idf weight of a term is the product of its tf weight and its idf weight.

$$\mathrm{w}_{t,d} = \mathrm{tf}_{t,d} \times \log(N/\mathrm{df}_t)$$

- For each document, there is one entry for every term in the vocabulary
- Each entry in that vector is the tf-idf weight above

- How do we calculate the similarity?

Recap: cosine(query,document)

Dot product

Unit vectors

$$\cos(\vec{q},\vec{d}) = \frac{\vec{q} \bullet \vec{d}}{|\vec{q}||\vec{d}|} = \frac{\vec{q}}{|\vec{q}|} \bullet \frac{\vec{d}}{|\vec{d}|} = \frac{\sum_{i=1}^{|V|} q_i d_i}{\sqrt{\sum_{i=1}^{|V|} q_i^2}\sqrt{\sum_{i=1}^{|V|} d_i^2}}$$

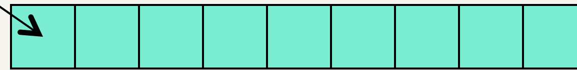
cos(*q*,*d*) is the cosine similarity of *q* and *d* … or, equivalently, the cosine of the angle between *q* and *d*.

Outline

- Calculating tf-idf score
- Faster ranking
- Static quality scores
- Impact ordering
- Cluster pruning

Calculating cosine similarity

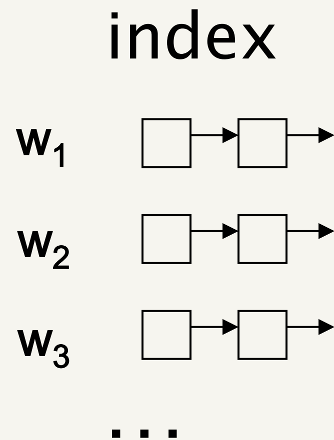
tf-idf entries



$$\cos(\vec{q},\vec{d}) = \frac{\sum_{i=1}^{|V|} q_i d_i}{\sqrt{\sum_{i=1}^{|V|} q_i^2}\sqrt{\sum_{i=1}^{|V|} d_i^2}}$$

- Traverse entries calculating the product
- Accumulate the vector lengths and divide at the end
- How can we do it faster if we have a sparse representation?

Calculating cosine tf-idf from index



- What should we store in the index?
- How do we construct the index?
- How do we calculate the document ranking?

$$w_{t,d} = \mathrm{tf}_{t,d} \times \log(N/\mathrm{df}_t)$$

$$\cos(\vec{q},\vec{d}) = \frac{\vec{q}}{|\vec{q}|} \bullet \frac{\vec{d}}{|\vec{d}|} = \frac{\sum_{i=1}^{|V|} q_i d_i}{\sqrt{\sum_{i=1}^{|V|} q_i^2}\sqrt{\sum_{i=1}^{|V|} d_i^2}}$$

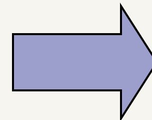
Index construction: collect documentIDs

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

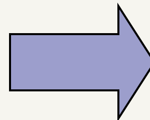


Term	Doc #
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

Index construction: sort dictionary

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

sort based on terms

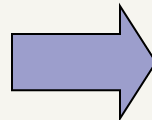


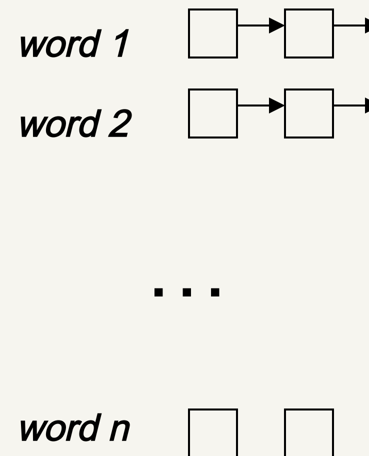
Term	Doc #
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

Index construction: create postings list

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

create postings lists from identical entries





$$w_{t,d} = \mathrm{tf}_{t,d} \times \log(N/\mathrm{df}_t)$$

Do we have all the information we need?

Obtaining tf-idf weights

- Store the tf initially in the index
- In addition, store the number of documents the term occurs in in the index

- How do we get the idfs?
  - We can either compute these on the fly using the number of documents in each term
  - We can make another pass through the index and update the weights for each entry
- Pros and cons of each approach?

Do we have everything we need?

$$\cos(\vec{q},\vec{d}) = \frac{\vec{q}}{|\vec{q}|} \bullet \frac{\vec{d}}{|\vec{d}|} = \frac{\sum_{i=1}^{|V|} q_i d_i}{\sqrt{\sum_{i=1}^{|V|} q_i^2}\sqrt{\sum_{i=1}^{|V|} d_i^2}}$$

- Still need the document lengths
  - Store these in a separate data structure
  - Make another pass through the data and update the weights
- Benefits/drawbacks?

Computing cosine scores

- Similar to the merge operation
- Accumulate scores for each document

- float *scores*[N] = 0
- for each query term *t*
  - calculate $w_{t,q}$
  - for each entry in *t*'s postings list: *docID,* $w_{t,d}$
    - *scores[docID]* += $w_{t,q} * w_{t,d}$
- return top *k* components of scores

Efficiency

- What are the inefficiencies here?
  - Only want the scores for the top *k* but are calculating all the scores
  - Sort to obtain top k?

- float *scores*[N] = 0
- for each query term *t*
  - calculate $w_{t,q}$
  - for each entry in *t*'s postings list: *docID*, $w_{t,d}$
    - *scores[docID]* += $w_{t,q} * w_{t,d}$
- return top *k* components of scores

Outline

- Calculating tf-idf score
- Faster ranking
- Static quality scores
- Impact ordering
- Cluster pruning

Efficient cosine ranking

- What we're doing in effect: solving the *K*-nearest neighbor problem for a query vector
- In general, we do not know how to do this efficiently for high-dimensional spaces
- Two simplifying assumptions
  - Queries are short!
  - Assume no weighting on query terms and that each query term occurs only once
  - Then for ranking, don't need to normalize query vector

Computing cosine scores

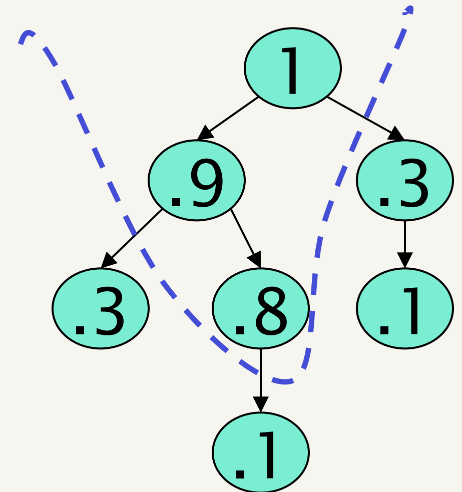
- Assume no weighting on query terms and that each query term occurs only once

---

- float *scores*[N] = 0
- for each query term *t*
  - for each entry in *t*'s postings list: *docID*, $w_{t,d}$
    - *scores[docID]* += $w_{t,d}$
- return top *k* components of scores

Selecting top K

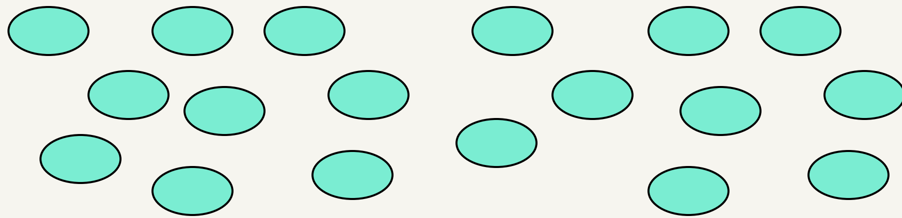
- We could sort the scores and then pick the top K
- What is the runtime of this approach?
  - O(N log N)
- Can we do better?
- Use a heap (i.e. priority queue)
  - Build a heap out of the scores
  - Get the top K scores from the heap
  - Running time?
    - O(N + K log N)
- For *N*=1M, *K*=100, this is about 10% of the cost of sorting



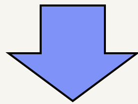
Inexact top K

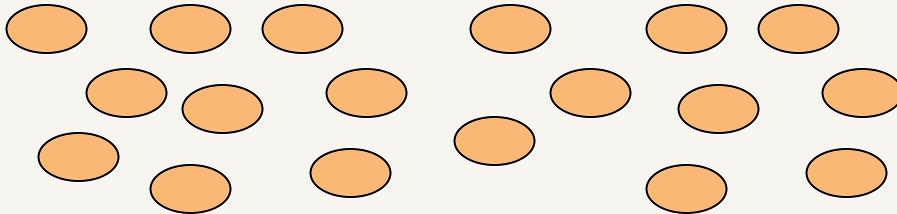
- What if we don't return the exactly the top K, but a set close to the top K?
  - User has a task and a query formulation
  - Cosine is a proxy for matching this task/query
  - If we get a list of *K* docs "close" to the top *K* by cosine measure, should still be ok

Current approach

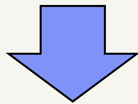


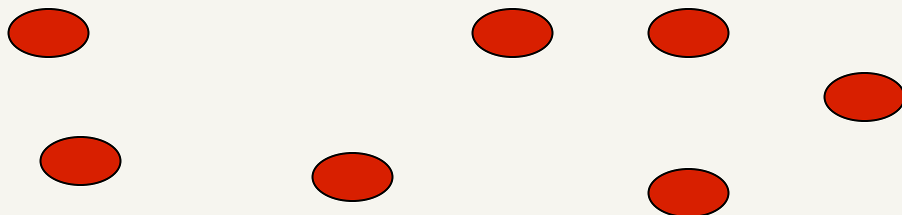
Documents





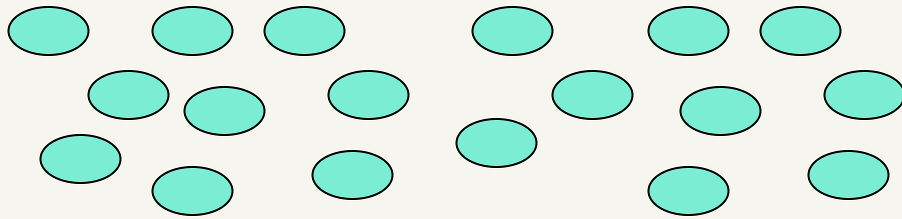
Score documents



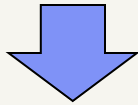


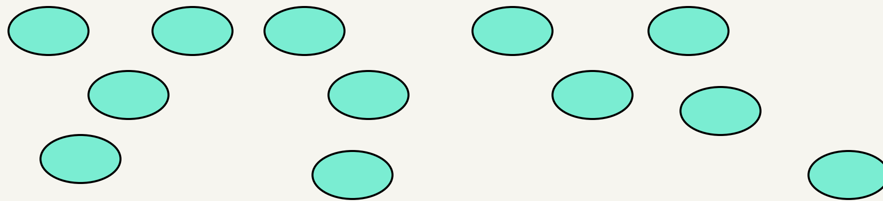
Pick top K

Approximate approach

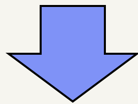


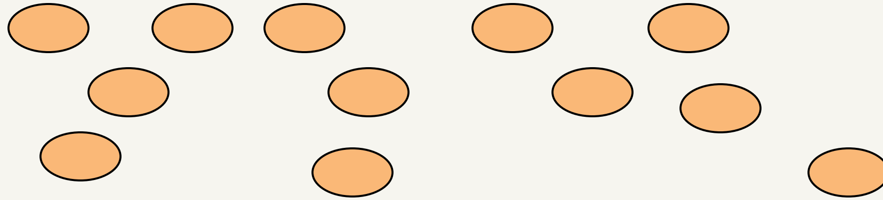
Documents



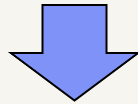


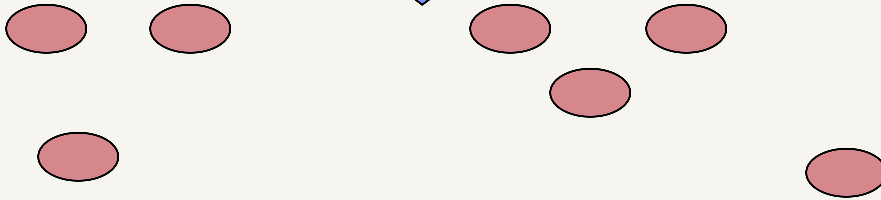
Select A candidates
K < A << N





Score documents in A

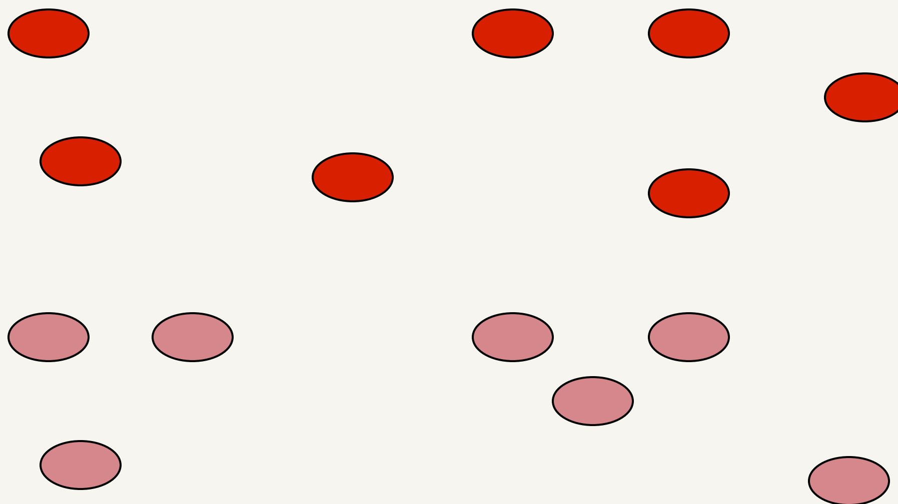




Pick top K in A

Exact vs. approximate

- Depending on how A is selected and how large A is, can get different results
- Can think of it as **pruning** the initial set of docs
- How might we pick A?



Exact

Approximate

Docs containing many query terms

- So far, we consider any document with at least one query term in it
- For multi-term queries, only compute scores for docs containing several of the query terms
  - Say, at least 3 out of 4
  - Imposes a “soft conjunction” on queries seen on web search engines (early Google)
- Easy to implement in postings traversal

3 of 4 query terms

Antony	3	4	8	16	32	64	128	
Brutus	2	4	8	16	32	64	128	
Caesar	1	2	3	5	8	13	21	34
Calpurnia	13	16	32					

Scores only computed for 8, 16 and 32.

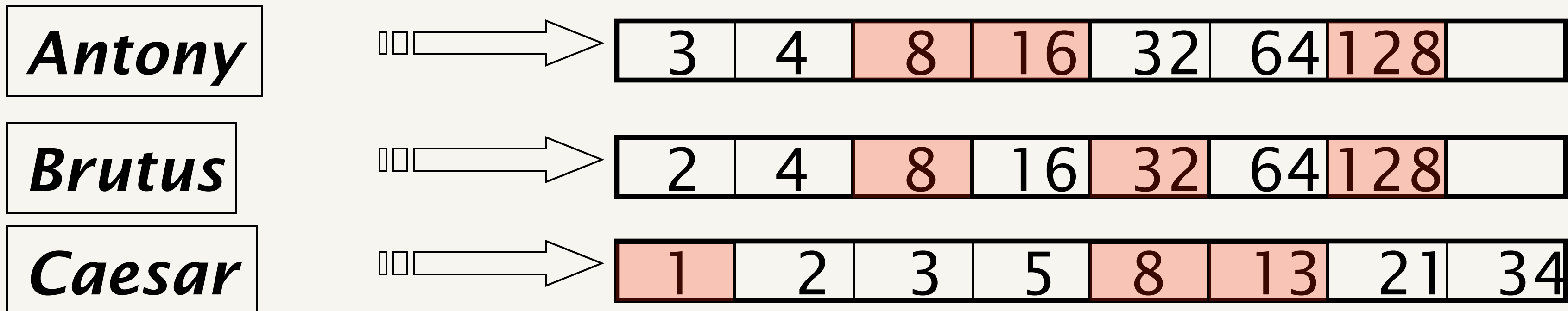
High-idf query terms only

- For a query such as *catcher in the rye*
- Only accumulate scores from *catcher* and *rye*
- Intuition: *in* and *the* contribute little to the scores and don't alter rank-ordering much
- Benefit:
  - Postings of low-idf terms have many docs → these (many) docs get eliminated from *A*

- Can we calculate this efficiently from the index?

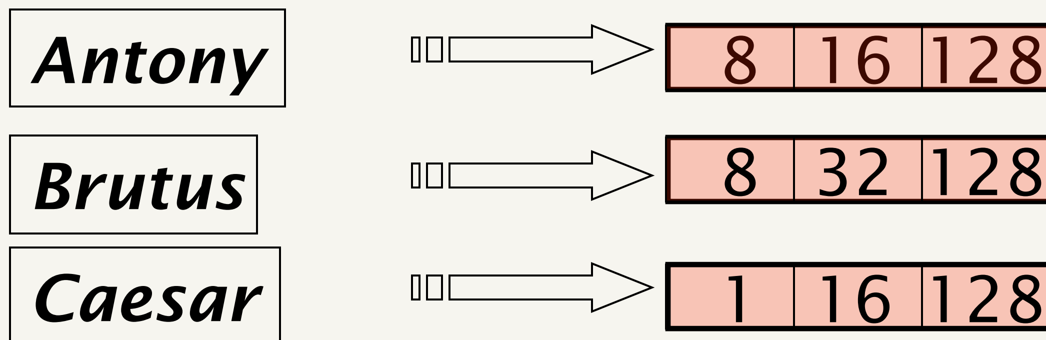
Champion lists

- **Precompute** for each dictionary term the *r* docs of highest weight in the term's postings
  - Call this the champion list for a term
  - (aka fancy list or top docs for a term)
- This must be done at index time



Champion lists

- At query time, only compute scores for docs in the champion list of some query term
  - Pick the *K* top-scoring docs from amongst these



- Are we guaranteed to always get K documents?

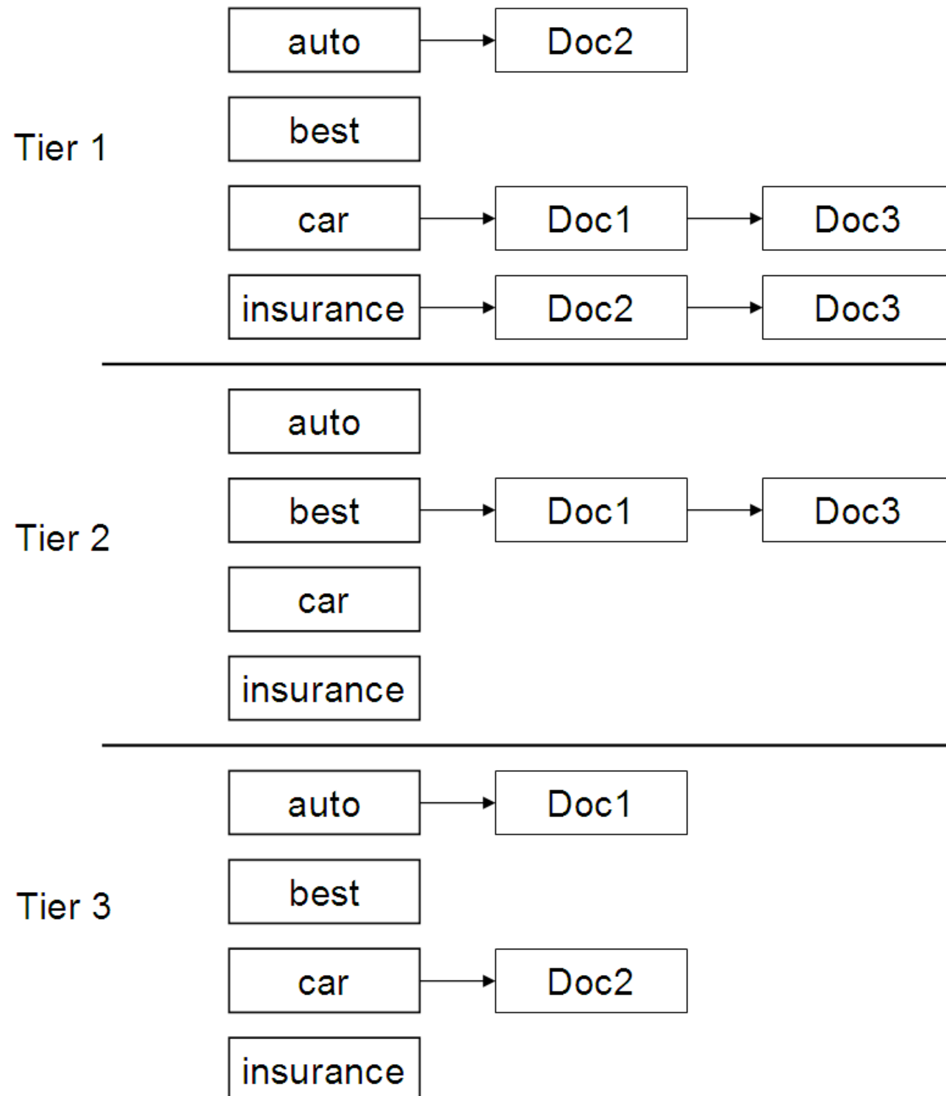
High and low lists

- For each term, we maintain two postings lists called *high* and *low*
  - Think of *high* as the champion list
- When traversing postings on a query, only traverse *high* lists first
  - If we get more than *K* docs, select the top *K* and stop
  - Else proceed to get docs from the *low* lists
- A means for segmenting index into two tiers

Tiered indexes

- Break postings up into a hierarchy of lists
  - Most important
  - …
  - Least important
- Inverted index thus broken up into tiers of decreasing importance
- At query time use top tier unless it fails to yield *K* docs
  - If so drop to lower tiers

Example tiered index



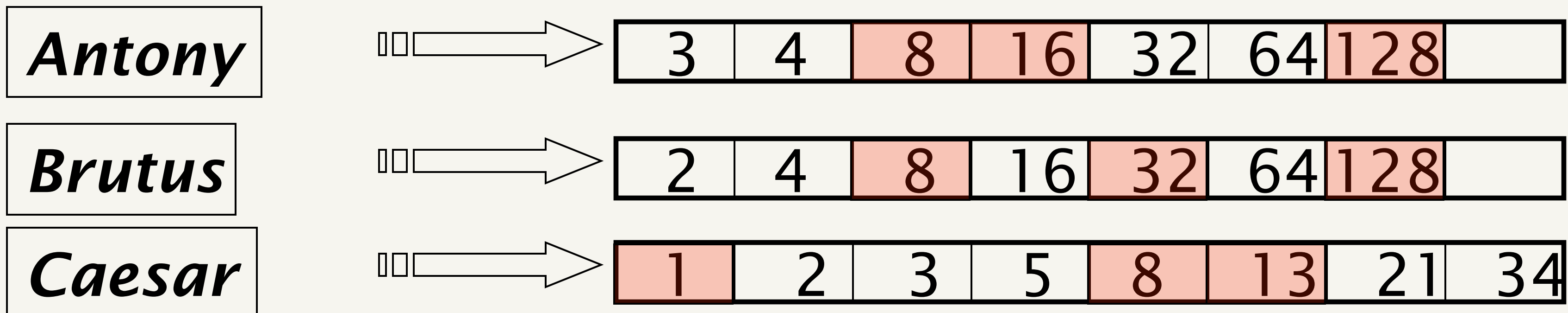
Quick review

- Rather than selecting the best K scores from all N documents
  - Initially filter the documents to a smaller set
  - Select the K best scores from this smaller set

- Methods for selecting this smaller set
  - Documents with <u>more than one</u> query term
  - Terms with high IDF
  - Documents with the highest weights

Discussion

- How can Champion Lists be implemented in an inverted index? How do we modify the data structure?



Outline

- Calculating tf-idf score
- Faster ranking
- Static quality scores
- Impact ordering
- Cluster pruning

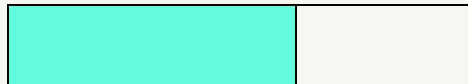
Static quality scores

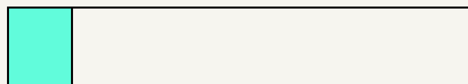
- We want top-ranking documents to be both ***relevant*** and ***authoritative***
- *Relevance* is being modeled by cosine scores
- *Authority* is typically a query-independent property of a document
- What are some examples of authority signals?
  - Wikipedia among websites
  - Articles in certain newspapers
  - A paper with many citations
  - Many diggs, Y!buzzes or del.icio.us marks
  - Pagerank

Modeling authority

- Assign to each document a *query-independent* <u>quality score</u> in [0,1] denoted *g(d)*
- Thus, a quantity like the number of citations is scaled into [0,1]

- Google PageRank





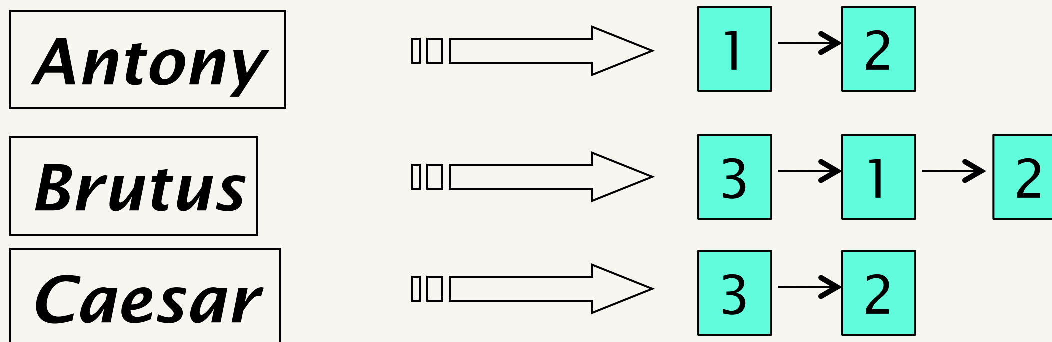
Net score

- We want a total score that combines cosine relevance and authority
  - net-score(q,d) = $g(d)$ + cosine(q,d)
  - Can use some other linear combination than an equal weighting
  - Indeed, any function of the two "signals" of user happiness
- Now we seek the top K docs by net score

- Doing this exactly, is similar to incorporating document length normalization

Top *K* by net score – fast methods

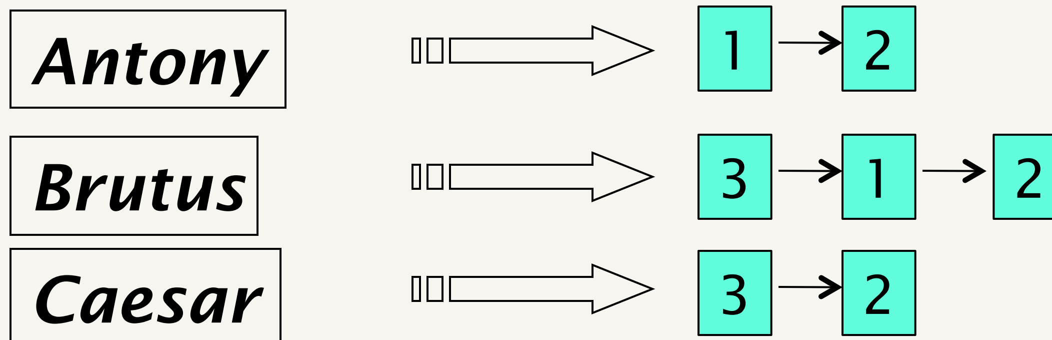
- Order all postings by *g(d)*
- Is this ok? Does it change our merge/traversal algorithms?
  - Key: this is still a common ordering for all postings



g(1) = 0.5, g(2) = .25, g(3) = 1

Why order postings by *g(d)*?

- Under *g(d)*-ordering, top-scoring docs likely to appear early in postings traversal
- In time-bound applications (say, we have to return whatever search results we can in 50 ms), this allows us to stop postings traversal early



g(1) = 0.5, g(2) = .25, g(3) = 1

Champion lists in *g(d)*-ordering

- We can still use the notion of champion lists…

- Combine champion lists with *g(d)*-ordering
- Maintain for each term a champion list of the *r* docs with highest *g(d)* + tf-idf_{td}
- Seek top-*K* results from only the docs in these champion lists

Outline

- Calculating tf-idf score
- Faster ranking
- Static quality scores
- Impact ordering
- Cluster pruning

Impact-ordered postings

- Why do we need a common ordering of the postings list?
  - Allows us to easily traverse the postings list and check for intersection
- Is that required for our tf-idf traversal algorithm?

- float *scores*[N] = 0
- for each query term *t*
  - for each entry in *t*'s postings list: *docID*, $w_{t,d}$
    - *scores[docID]* += $w_{t,d}$
- return top *k* components of scores

Impact-ordered postings

- The ordering no long plays a role
- Our algorithm for computing document scores "accumulates" scores for each document

- Idea: sort each postings list by $w_{t,d}$
- Only compute scores for docs for which $w_{t,d}$ is high enough
- Given this ordering, how might we construct A when processing a query?

Impact-ordering: early termination

- When traversing a postings list, stop early after either
  - a fixed number of *r* docs
  - $w_{t,d}$ drops below some threshold
- Take the union of the resulting sets of docs
  - One from the postings of each query term
- Compute only the scores for docs in this union

Impact-ordering: idf-ordered terms

- When considering the postings of query terms
- Look at them in order of decreasing idf
  - High idf terms likely to contribute most to score
- As we update score contribution from each query term
  - Stop if doc scores relatively unchanged
- Can apply to cosine or other net scores

Outline

- Calculating tf-idf score
- Faster ranking
- Static quality scores
- Impact ordering
- Cluster pruning

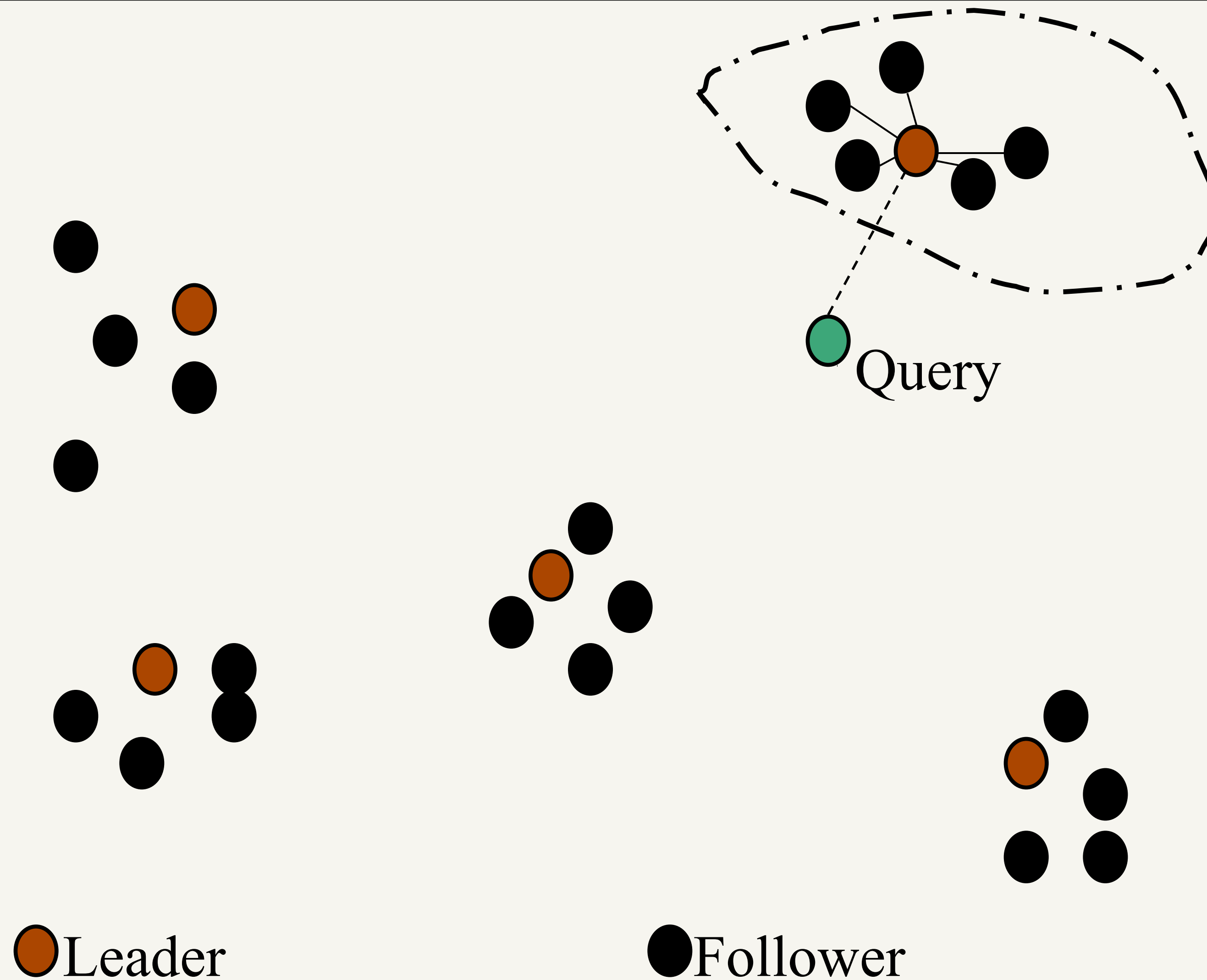
Cluster pruning: preprocessing

- Pick √N *docs,* call these *leaders*
- For every other doc, pre-compute nearest leader
  - Docs attached to a leader are called *followers*
  - Likely: each leader has ~ √*N* followers

Cluster pruning: query processing

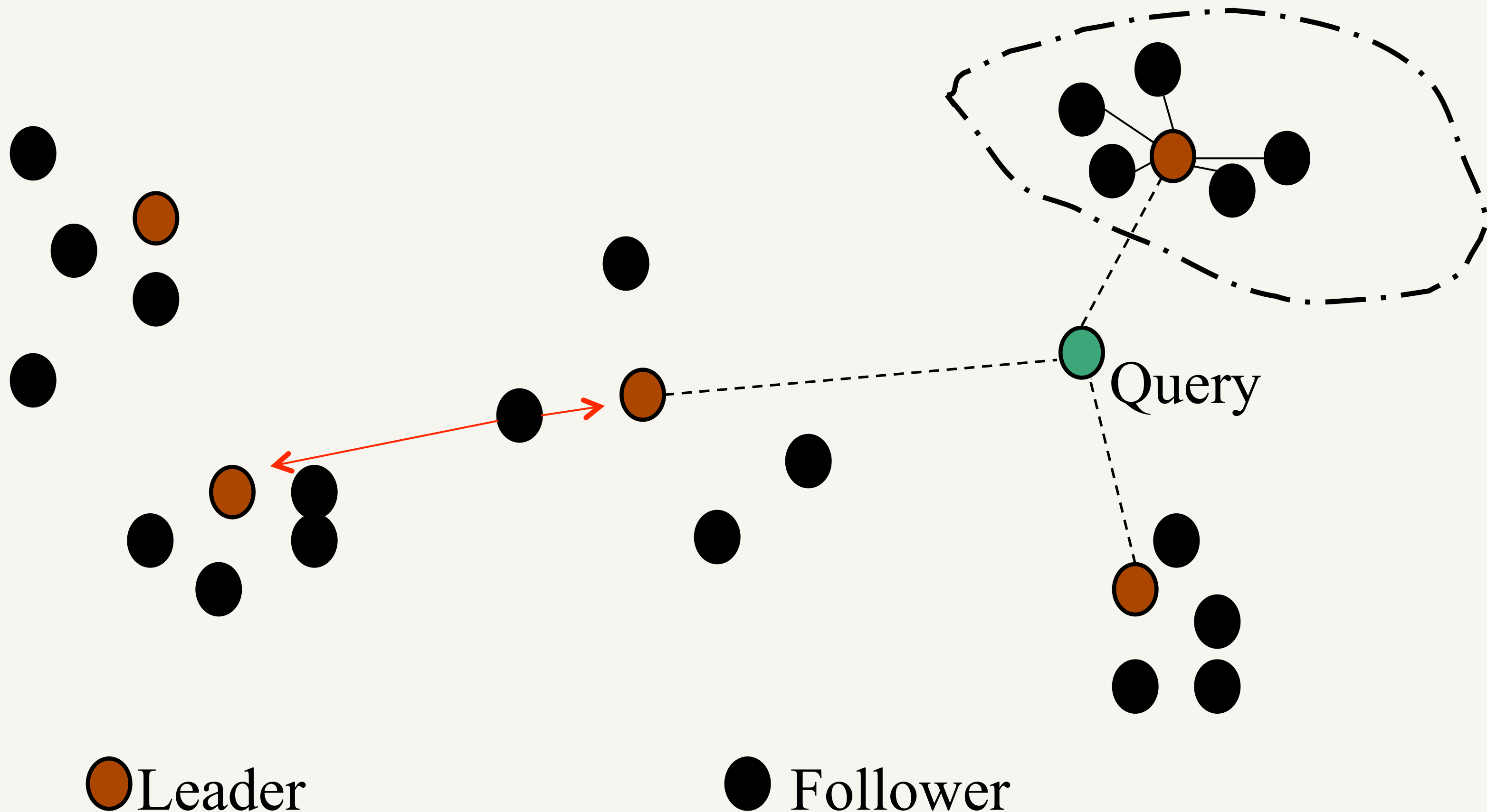
- Process a query as follows:
  - Given query *Q*, find its nearest *leader L*
  - Seek *K* nearest docs from among *L*'s followers

Visualization



Cluster pruning variants

- Have each follower attached to b_1 (e.g. 2) nearest leaders
- From query, find b_2 (e.g. 3) nearest leaders and their followers



Can Microsoft's Bing, or Anyone, Seriously Challenge Google?

- Will it ever be possible to dethrone Google as the leader in web search?
- What would a search engine need to improve upon the model Google offers?
- Is Bing a serious threat to Google’s dominance?