

Lecture 4

Ranking Search Results

Many thanks to Prabhakar Raghavan for sharing most content from the following slides

Recap of the previous lecture

- Index construction
 - Doing sorting with limited main memory
 - Parallel and distributed indexing
- Index compression
 - Space estimation
 - Dictionary compression
 - Postings compression

This lecture

- Scoring documents
 - Term weighting
 - Vector space scoring
- Efficiency considerations
 - Nearest neighbors and approximations
- Parametric and field searches
- Results summaries
 - Making our good results usable to a user
 - Snippet generation

Scoring and Ranking

Facts

- The average query length on current search engines is 2.4 words
- Over 40% of the user queries are single words
- About 80+% of the users look only at the first page of results, 95% look at the first two pages, almost everybody looks at the first three

Ranked retrieval

- Thus far, our queries have all been Boolean.
 - Documents either match or don't.
- Good for expert users with precise understanding of their needs and the collection.
 - Also good for applications: Applications can easily consume 1000s of results.
- Not good for the majority of users.
 - Most users incapable of writing Boolean queries (or they are, but they think it's too much work).
 - Most users don't want to wade through 1000s of results.
 - This is particularly true of web search.

Problem with Boolean search: feast or famine

- Boolean queries often result in either too few (=0) or too many (1000s) results.
- Query 1: “*standard user dlink 650*” → 200,000 hits
- Query 2: “*standard user dlink 650 no card found*”: 0 hits
- It takes a lot of skill to come up with a query that produces a manageable number of hits.
 - AND gives too few; OR gives too many

Ranked retrieval models

- Rather than a set of documents satisfying a query expression, in **ranked retrieval models**, the system returns an ordering over the (top) documents in the collection with respect to a query
- **Free text queries**: Rather than a query language of operators and expressions, the user's query is just one or more words in a human language
- In principle, there are two separate choices here, but in practice, ranked retrieval models have normally been associated with free text queries and vice versa

Feast or famine: not a problem in ranked retrieval

- When a system produces a ranked result set, large result sets are not an issue
 - Indeed, the size of the result set is not an issue
 - We just show the top k (≈ 10) results
 - We don't overwhelm the user
- Premise: the ranking algorithm works

Scoring as the basis of ranked retrieval

- We wish to return in order the documents most likely to be useful to the searcher
- How can we rank-order the documents in the collection with respect to a query?
- Assign a score – say in $[0, 1]$ – to each document
- This score measures how well document and query “match”.

Query-document matching scores

- We need a way of assigning a score to a query/document pair
- Let's start with a one-term query
- If the query term does not occur in the document: score should be 0
- The more frequent the query term in the document, the higher the score (should be)
- We will look at a number of alternatives for this.

Take 1: Jaccard coefficient

- Recall: A commonly used measure of overlap of two sets A and B
- $\text{jaccard}(A, B) = |A \cap B| / |A \cup B|$
- $\text{jaccard}(A, A) = 1$
- $\text{jaccard}(A, B) = 0$ if $A \cap B = 0$
- A and B don't have to be the same size.
- Always assigns a number between 0 and 1.

Jaccard coefficient: Scoring example

- What is the query-document match score that the Jaccard coefficient computes for each of the two documents below?
- Query: *ides of march*
- Document 1: *caesar died in march*
- Document 2: *the long march*

Issues with Jaccard for scoring

- It doesn't consider *term frequency* (how many times a term occurs in a document)
- Rare terms in a collection are more informative than frequent terms. Jaccard doesn't consider this information
- We need a more sophisticated way of normalizing for length
- Later in this lecture, we'll use $|A \cap B| / \sqrt{|A \cup B|}$
- . . . instead of $|A \cap B| / |A \cup B|$ (Jaccard) for length normalization.

Recall (Lecture 1): Binary term-document incidence matrix

	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

Each document is represented by a binary vector $\in \{0,1\}^{|M|}$

Term-document count matrices

- Consider the number of occurrences of a term in a document:
 - Each document is a count vector in \mathbb{N}^V : a column below

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	157	73	0	0	0	0
Brutus	4	157	0	1	0	0
Caesar	232	227	0	2	1	1
Calpurnia	0	10	0	0	0	0
Cleopatra	57	0	0	0	0	0
mercy	2	0	3	5	5	1
worser	2	0	1	1	1	0

Bag of words model

- Vector representation doesn't consider the ordering of words in a document
- *John is quicker than Mary and Mary is quicker than John have the same vectors*
- This is called the bag of words model.
- In a sense, this is a step back: The positional index was able to distinguish these two documents.
- We will look at “recovering” positional information later in this course.
- For now: bag of words model

Term frequency tf

- The term frequency $tf_{t,d}$ of term t in document d is defined as the number of times that t occurs in d .
- We want to use tf when computing query-document match scores. But how?
- Raw term frequency is not what we want:
 - A document with 10 occurrences of the term is more relevant than a document with 1 occurrence of the term.
 - But not 10 times more relevant.
- Relevance does not increase proportionally with term frequency.

NB: frequency = count in IR

Log-frequency weighting

- The log frequency weight of term t in d is

$$w_{t,d} = \begin{cases} 1 + \log_{10} \text{tf}_{t,d}, & \text{if } \text{tf}_{t,d} > 0 \\ 0, & \text{otherwise} \end{cases}$$

- $0 \rightarrow 0, 1 \rightarrow 1, 2 \rightarrow 1.3, 10 \rightarrow 2, 1000 \rightarrow 4$, etc.
- Score for a document-query pair: sum over terms t in both q and d .
- $\text{score} = \sum_{t \in q \cap d} (1 + \log \text{tf}_{t,d})$
- The score is 0 if none of the query terms is present in the document.

Document frequency

- Rare terms are more informative than frequent terms
 - Recall stop words
- Consider a term in the query that is rare in the collection (e.g., *arachnocentric*)
- A document containing this term is very likely to be relevant to the query *arachnocentric information*
- → We want a high weight for rare terms like *arachnocentric*.

Document frequency, continued

- Frequent terms are less informative than rare terms
- Consider a query term that is frequent in the collection (e.g., *high*, *increase*, *line*)
- A document containing such a term is more likely to be relevant than a document that doesn't
- But it's not a sure indicator of relevance.
- → For frequent terms, we want high positive weights for words like *high*, *increase*, and *line*
- But lower weights than for rare terms.
- We will use document frequency (df) to capture this.

idf weight

- df_t is the document frequency of t : the number of documents that contain t
 - df_t is an inverse measure of the informativeness of t
 - $df_t \leq N$
- We define the idf (inverse document frequency) of t by
$$idf_t = \log_{10} (N/df_t)$$
 - We use $\log (N/df_t)$ instead of N/df_t to “dampen” the effect of idf.

Will turn out the base of the log is immaterial.

idf example, suppose $N = 1$ million

term	df_t	idf_t
calpurnia	1	
animal	100	
sunday	1,000	
fly	10,000	
under	100,000	
the	1,000,000	

$$idf_t = \log_{10} (N/df_t)$$

There is one idf value for each term t in a collection.

Effect of idf on ranking

- Does idf have an effect on ranking for one-term queries, like
 - iPhone
- idf has no effect on ranking one term queries
 - idf affects the ranking of documents for queries with at least two terms
 - For the query **capricious person**, idf weighting makes occurrences of **capricious** count for much more in the final document ranking than occurrences of **person**.

Collection vs. Document frequency

- The collection frequency of t is the number of occurrences of t in the collection, counting multiple occurrences.

- Example:

Word	Collection frequency	Document frequency
<i>insurance</i>	10440	3997
<i>try</i>	10422	8760

- Which word is a better search term (and should get a higher weight)?

tf-idf weighting

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

$$w_{t,d} = (1 + \log \text{tf}_{t,d}) \times \log_{10} (N / \text{df}_t)$$

- **Best known weighting scheme in information retrieval**
 - Note: the “-” in tf-idf is a hyphen, not a minus sign!
 - **Alternative names: tf.idf, tf x idf**
- Increases with the number of occurrences within a document
- **Increases with the rarity of the term in the collection**

Final ranking of documents for a query

$$\text{Score}(q, d) = \sum_{t \in q \cap d} \text{tf.idf}_{t,d}$$

Binary → count → weight matrix

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	5.25	3.18	0	0	0	0.35
Brutus	1.21	6.1	0	1	0	0
Caesar	8.59	2.54	0	1.51	0.25	0
Calpurnia	0	1.54	0	0	0	0
Cleopatra	2.85	0	0	0	0	0
mercy	1.51	0	1.9	0.12	5.25	0.88
worser	1.37	0	0.11	4.15	0.25	1.95

Each document is now represented by a real-valued vector of tf-idf weights $\in \mathbb{R}^{|V|}$

-> May also normalize for document length
(more on this later)

Documents as vectors

- So we have a $|V|$ -dimensional vector space
- Terms are axes of the space
- Documents are points or vectors in this space
- Very high-dimensional: tens of millions of dimensions when you apply this to a web search engine
- These are very sparse vectors - most entries are zero.

Queries as vectors

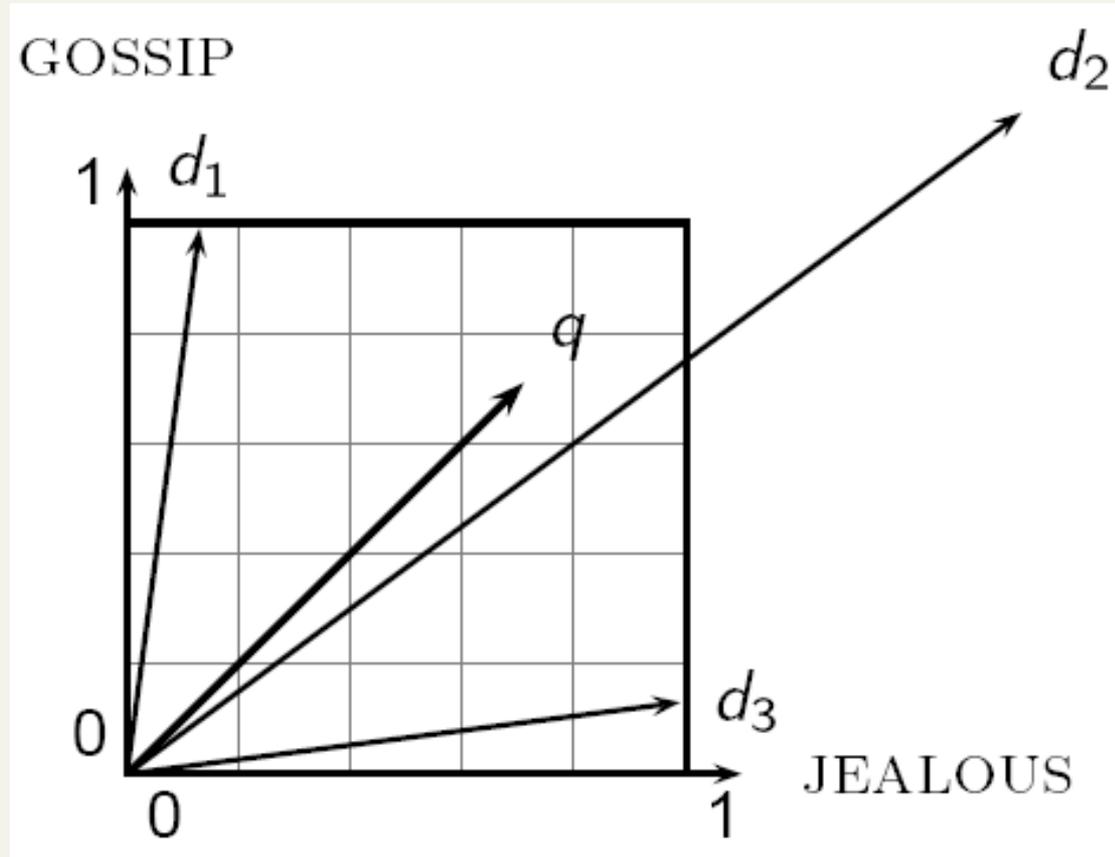
- [Key idea 1](#): Do the same for queries: represent them as vectors in the space
- [Key idea 2](#): Rank documents according to their proximity to the query in this space
- proximity = similarity of vectors
- proximity \approx inverse of distance
- **Recall: We do this because we want to get away from the you're-either-in-or-out Boolean model.**
- Instead: rank more relevant documents higher than less relevant documents

Formalizing vector space proximity

- First cut: distance between two points
 - (= distance between the end points of the two vectors)
- **Euclidean distance?**
- Euclidean distance is a bad idea . . .
- . . . because Euclidean distance is **large** for vectors of **different lengths**.

Why distance is a bad idea

The Euclidean distance between q and d_2 is large even though the distribution of terms in the query q and the distribution of terms in the document d_2 are very similar.



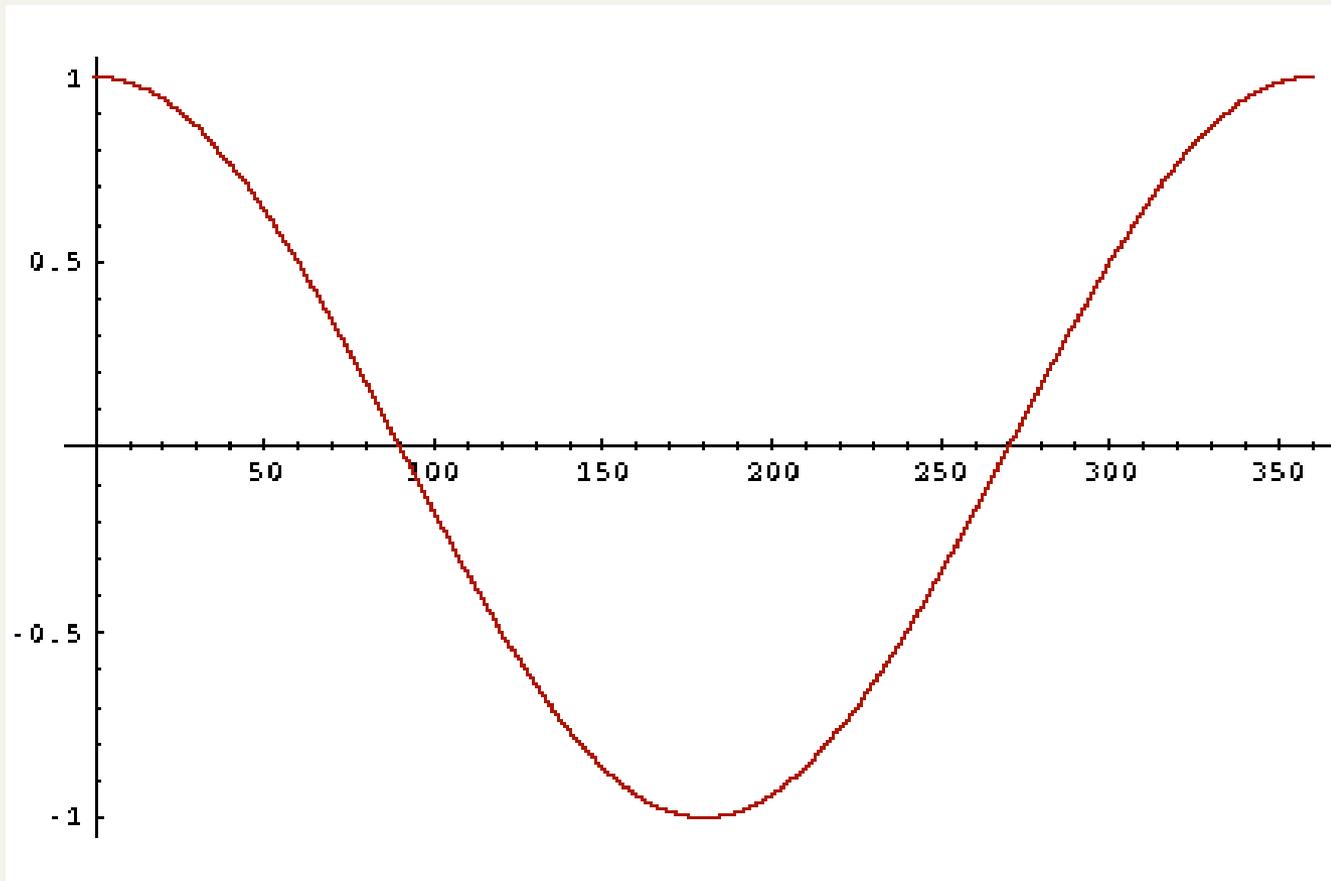
Use angle instead of distance

- Thought experiment: take a document d and append it to itself. Call this document d' .
- “Semantically” d and d' have the same content
- The Euclidean distance between the two documents can be quite large
- The angle between the two documents is 0, corresponding to maximal similarity.
- Key idea: Rank documents according to angle with query.

From angles to cosines

- The following two notions are equivalent.
 - Rank documents in increasing order of the angle between query and document
 - Rank documents in decreasing order of $\text{cosine}(\text{query}, \text{document})$
- Cosine is a monotonically decreasing function for the interval $[0^\circ, 180^\circ]$

From angles to cosines



- But how – *and why* – should we be computing cosines?

Length normalization

- A vector can be (length-) normalized by dividing each of its components by its length – for this we use the L_2 norm:
$$\|\vec{x}\|_2 = \sqrt{\sum_i x_i^2}$$
- Dividing a vector by its L_2 norm makes it a unit (length) vector (on surface of unit hypersphere)
- Effect on the two documents d and d' (d appended to itself) from earlier slide: they have identical vectors after length-normalization.
 - Long and short documents now have comparable weights

cosine(query, document)

Dot product

Unit vectors

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

q_i is the tf-idf weight of term i in the query

d_i is the tf-idf weight of term i in the document

$\cos(\vec{q}, \vec{d})$ is the cosine similarity of \vec{q} and \vec{d} ... or, equivalently, the cosine of the angle between \vec{q} and \vec{d} .

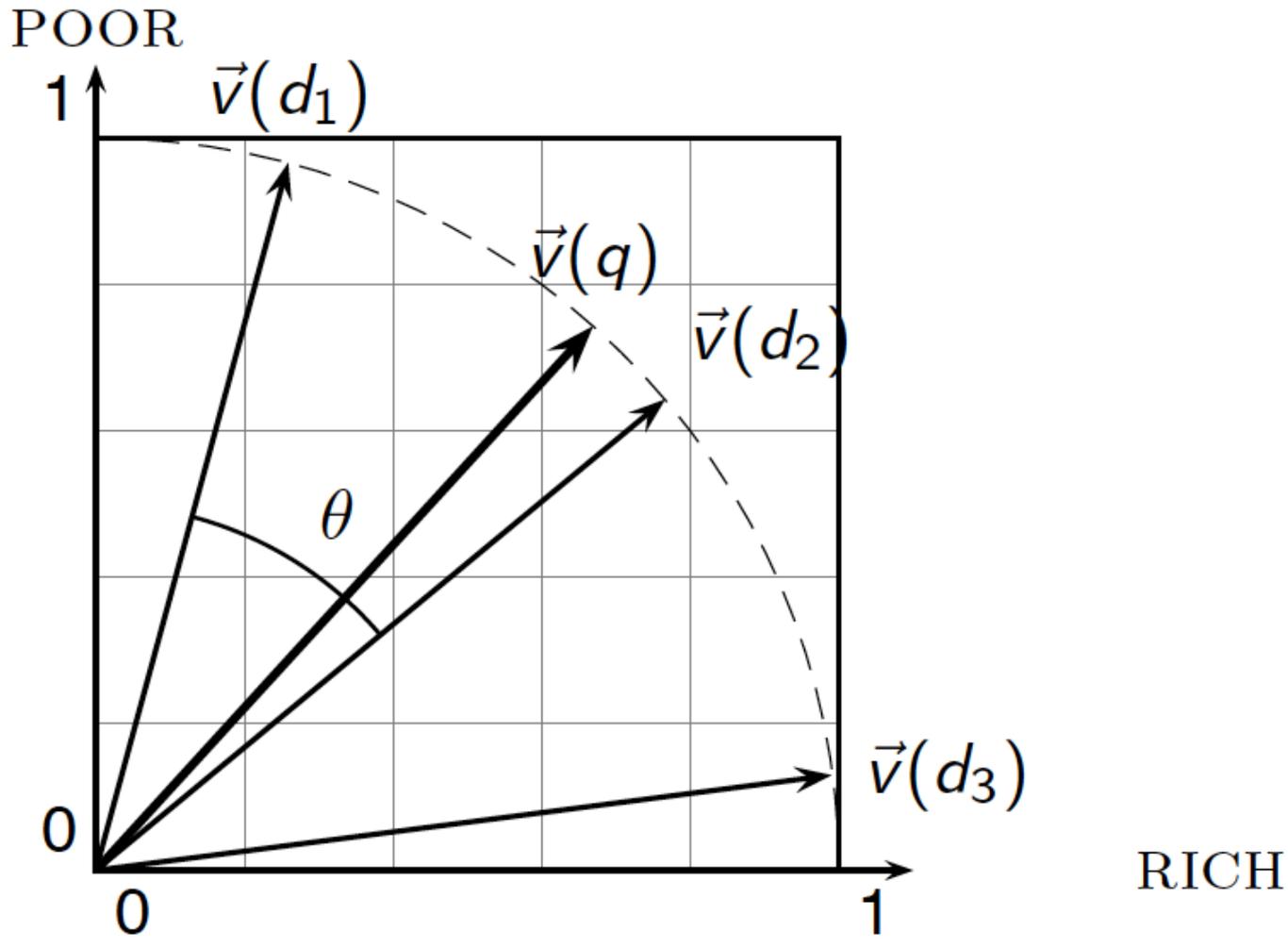
Cosine for length-normalized vectors

- For length-normalized vectors, cosine similarity is simply the dot product (or scalar product):

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

for q, d length-normalized.

Cosine similarity illustrated



Cosine similarity amongst 3 documents

How similar are
the novels

SaS: *Sense and
Sensibility*

PaP: *Pride and
Prejudice*, and

WH: *Wuthering
Heights*?

term	SaS	PaP	WH
affection	115	58	20
jealous	10	7	11
gossip	2	0	6
wuthering	0	0	38

Term frequencies (counts)

Note: To simplify this example, we don't do idf weighting.

3 documents example contd.

Log frequency weighting

term	SaS	PaP	WH
affection	3.06	2.76	2.30
jealous	2.00	1.85	2.04
gossip	1.30	0	1.78
wuthering	0	0	2.58

After length normalization

term	SaS	PaP	WH
affection	0.789	0.832	0.524
jealous	0.515	0.555	0.465
gossip	0.335	0	0.405
wuthering	0	0	0.588

$$\cos(\text{SaS}, \text{PaP}) \approx$$

$$0.789 \times 0.832 + 0.515 \times 0.555 + 0.335 \times 0.0 + 0.0 \times 0.0$$

$$\approx 0.94$$

$$\cos(\text{SaS}, \text{WH}) \approx 0.79$$

$$\cos(\text{PaP}, \text{WH}) \approx 0.69$$

Why do we have $\cos(\text{SaS}, \text{PaP}) > \cos(\text{SAS}, \text{WH})$?

Computing cosine scores

COSINESCORE(q)

```
1  float Scores[ $N$ ] = 0
2  float Length[ $N$ ]
3  for each query term  $t$ 
4  do calculate  $w_{t,q}$  and fetch postings list for  $t$ 
5      for each pair( $d, tf_{t,d}$ ) in postings list
6          do  $Scores[d] + = w_{t,d} \times w_{t,q}$ 
7  Read the array Length
8  for each  $d$ 
9  do  $Scores[d] = Scores[d] / Length[d]$ 
10 return Top  $K$  components of Scores[]
```

tf-idf weighting has many variants

Term frequency		Document frequency		Normalization	
n (natural)	$tf_{t,d}$	n (no)	1	n (none)	1
l (logarithm)	$1 + \log(tf_{t,d})$	t (idf)	$\log \frac{N}{df_t}$	c (cosine)	$\frac{1}{\sqrt{w_1^2 + w_2^2 + \dots + w_M^2}}$
a (augmented)	$0.5 + \frac{0.5 \times tf_{t,d}}{\max_t(tf_{t,d})}$	p (prob idf)	$\max\{0, \log \frac{N - df_t}{df_t}\}$	u (pivoted unique)	$1/u$
b (boolean)	$\begin{cases} 1 & \text{if } tf_{t,d} > 0 \\ 0 & \text{otherwise} \end{cases}$			b (byte size)	$1/CharLength^\alpha$, $\alpha < 1$
L (log ave)	$\frac{1 + \log(tf_{t,d})}{1 + \log(\text{ave}_{t \in d}(tf_{t,d}))}$				

Columns headed 'n' are acronyms for weight schemes.

Why is the base of the log in idf immaterial?

Weighting may differ in queries vs documents

- Many search engines allow for different weightings for queries vs. documents
- **SMART Notation:** denotes the combination in use in an engine, with the notation *ddd.qqq*, using the acronyms from the previous table
- A very standard weighting scheme is: **Inc.ltn**
 - Document: logarithmic tf (**l as first character**), no idf and cosine normalization
 - Query: logarithmic tf (**l in leftmost column**), idf (**t in second column**), no normalization ...

tf-idf example: Inc.ltn

Document: *car insurance auto insurance*

Query: *best car insurance*

Term	Query						Document				Prod
	tf-raw	tf-wt	df	idf	wt	n'lize	tf-raw	tf-wt	wt	n'lize	
auto	0	0	5000	2.3	0	0	1	1	1	0.52	0
best	1	1	50000	1.3	1.3	0.34	0	0	0	0	0
car	1	1	10000	2.0	2.0	0.52	1	1	1	0.52	0.27
insurance	1	1	1000	3.0	3.0	0.78	2	1.3	1.3	0.68	0.53

Exercise: what is N , the number of docs?

$$\text{Doc length} = \sqrt{1^2 + 0^2 + 1^2 + 1.3^2} \approx 1.92$$

$$\text{Score} = 0+0+0.27+0.53 = 0.8$$

Efficient cosine ranking

- Find the K docs in the collection “nearest” to the query $\Rightarrow K$ largest query-doc cosines.
- Efficient ranking:
 - Computing a single cosine efficiently.
 - Choosing the K largest cosine values efficiently.
 - Can we do this without computing all N cosines?

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
- But it is solvable for short queries, and standard indexes support this well

Special case – unweighted queries

- No weighting on query terms
 - Assume each query term occurs only once
- Then for ranking, don't need to normalize query vector

Faster cosine: unweighted query

```
FASTCOSINESCORE( $q$ )
1  float  $Scores[N] = 0$ 
2  for each  $d$ 
3  do Initialize  $Length[d]$  to the length of doc  $d$ 
4  for each query term  $t$ 
5  do calculate  $w_{t,q}$  and fetch postings list for  $t$ 
6     for each pair( $d, tf_{t,d}$ ) in postings list
7     do add  $wf_{t,d}$  to  $Scores[d]$ 
8  Read the array  $Length[d]$ 
9  for each  $d$ 
10 do Divide  $Scores[d]$  by  $Length[d]$ 
11 return Top  $K$  components of  $Scores[]$ 
```

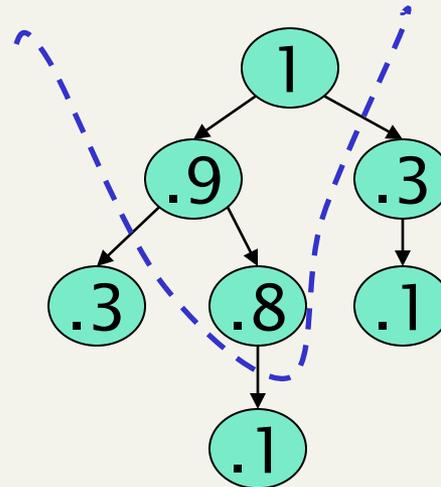
Figure 7.1 A faster algorithm for vector space scores.

Computing the K largest cosines: selection vs. sorting

- Typically we want to retrieve the top K docs (in the cosine ranking for the query)
 - not to totally order all docs in the collection
- Can we pick off docs with K highest cosines?
- Let J = number of docs with nonzero cosines
 - We seek the K best of these J

Use heap for selecting top K

- Binary tree in which each node's value $>$ the values of children
- Takes $2J$ operations to construct, then each of K “winners” read off in $2\log J$ steps.
- For $J=1M$, $K=100$, this is about 10% of the cost of sorting.



Bottlenecks

- Primary computational bottleneck in scoring: cosine computation
- **Can we avoid all this computation?**
- Yes, but may sometimes get it wrong
 - a doc *not* in the top K may creep into the list of K output docs
 - Is this such a bad thing?

Cosine similarity is only a proxy

- User has a task and a query formulation
- Cosine matches docs to query
- Thus cosine is anyway a proxy for user happiness
- If we get a list of K docs “close” to the top K by cosine measure, should be ok

Generic approach

- Find a set A of *contenders*, with $K < |A| \ll N$
 - A does not necessarily contain the top K , but has many docs from among the top K
 - Return the top K docs in A
- Think of A as pruning non-contenders
- The same approach is also used for other (non-cosine) scoring functions
- Will look at several schemes following this approach

Index elimination

- Basic algorithm FastCosineScore only considers docs containing at least one query term
- Take this further:
 - Only consider high-idf query terms
 - Only consider docs containing many query terms

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 so don't alter rank-ordering much
- Benefit:
 - Postings of low-idf terms have many docs → these (many) docs get eliminated from set *A* of contenders

Docs containing many query terms

- Any doc with at least one query term is a candidate for the top K output list
- 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

<i>Antony</i>	→	3	4	8	16	32	64	128	
<i>Brutus</i>	→	2	4	8	16	32	64	128	
<i>Caesar</i>	→	1	2	3	5	8	13	21	34
<i>Calpurnia</i>	→	13	16	32					

Scores only computed for docs 8, 16 and 32.

Champion lists

- Precompute for each dictionary term t , the r docs of highest weight in t 's postings
 - Call this the champion list for t
 - (aka fancy list or top docs for t)
- Note that r has to be chosen at index build time
 - Thus, it's possible that $r < K$
- 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

Exercises

- How do Champion Lists relate to Index Elimination? Can they be used together?
- How can Champion Lists be implemented in an inverted index?
 - Note that the champion list has nothing to do with small docIDs

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
- **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)**

Quantitative



Modeling authority

- Assign to each document a *query-independent quality score* in $[0, 1]$ to each document d
 - Denote this by $g(d)$
- Thus, a quantity like the number of citations is scaled into $[0, 1]$
 - Exercise: suggest a formula for this.

Net score

- Consider a simple total score combining cosine relevance and authority
- $\text{net-score}(q, d) = g(d) + \text{cosine}(q, d)$
 - Can use some other linear combination than an equal weighting
 - Indeed, any function of the two “signals” of user happiness – more later
- Now we seek the top K docs by net score

Top K by net score – fast methods

- First idea: Order all postings by $g(d)$
- **Key: this is a common ordering for all postings**
- Thus, can concurrently traverse query terms' postings for
 - Postings intersection
 - Cosine score computation
- **Exercise: write pseudocode for cosine score computation if postings are ordered by $g(d)$**

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
 - Short of computing scores for all docs in postings

Champion lists in $g(d)$ -ordering

- Can combine champion lists with $g(d)$ -ordering
- Maintain for each term a champion list of the r docs with highest $g(d) + \text{tf-idf}_{td}$
- Seek top- K results from only the docs in these champion lists

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
- Can be used even for simple cosine scores, without global quality $g(d)$
- A means for segmenting index into two tiers

Impact-ordered postings

- We only want to compute scores for docs for which $wf_{t,d}$ is high enough
- We sort each postings list by $wf_{t,d}$
- Now: not all postings in a common order!
- How do we compute scores in order to pick off top K ?
 - Two ideas follow

1. Early termination

- When traversing t 's postings, stop early after either
 - a fixed number of r docs
 - $wf_{t,d}$ 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

2. 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 some other net scores

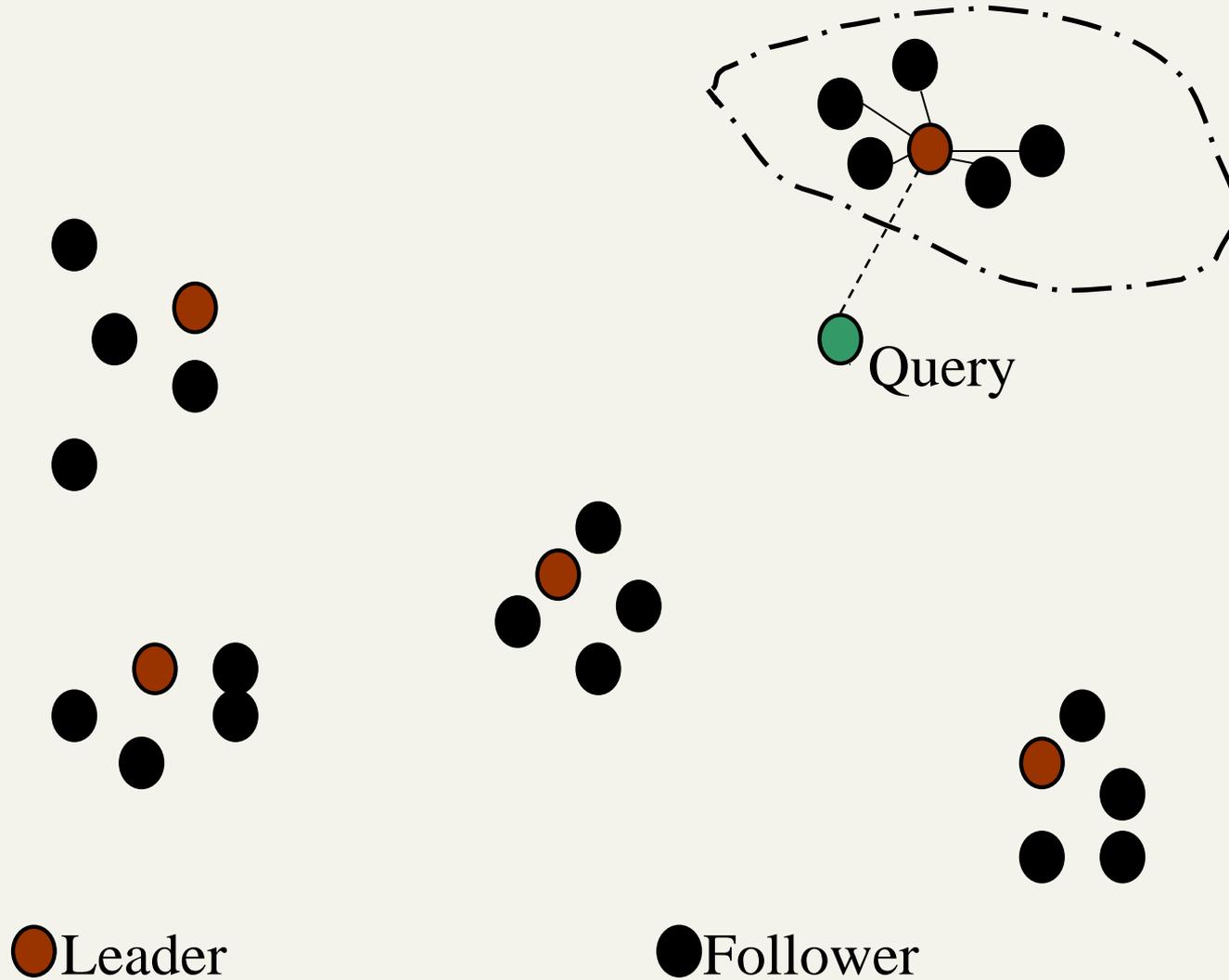
Cluster pruning: preprocessing

- Pick \sqrt{N} *docs* at random: call these *leaders*
- For every other doc, pre-compute nearest leader
 - Docs attached to a leader: its *followers*;
 - Likely: each leader has $\sim \sqrt{N}$ 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



Why use random sampling

- Fast
- Leaders reflect data distribution

General variants

- Have each follower attached to $b_1=3$ (say) nearest leaders.
- From query, find $b_2=4$ (say) nearest leaders and their followers.
- Can recur on leader/follower construction.

Exercises

- To find the nearest leader in step 1, how many cosine computations do we do?
 - Why did we have \sqrt{N} in the first place?
- What is the effect of the constants $b1$, $b2$ on the previous slide?
- Devise an example where this is *likely to fail* – i.e., we miss one of the K nearest docs.
 - *Likely* under random sampling.

Parametric and zone indexes

- Thus far, a doc has been a sequence of terms
- In fact documents have multiple parts, some with special semantics:
 - Author
 - Title
 - Date of publication
 - Language
 - Format
 - etc.
- These constitute the metadata about a document

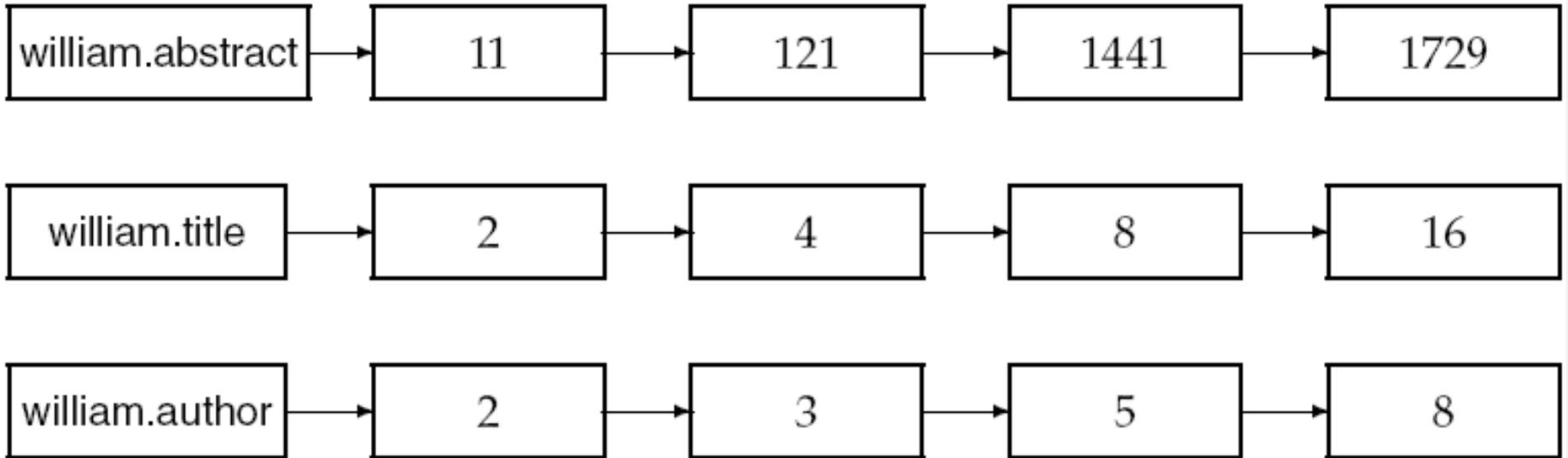
Fields

- We sometimes wish to search by these metadata
 - E.g., find docs authored by William Shakespeare in the year 1601, containing *alas poor Yorick*
- Year = 1601 is an example of a field
- Also, author last name = shakespeare, etc
- Field or parametric index: postings for each field value
 - Sometimes build range trees (e.g., for dates)
- Field query typically treated as conjunction
 - (doc *must* be authored by shakespeare)

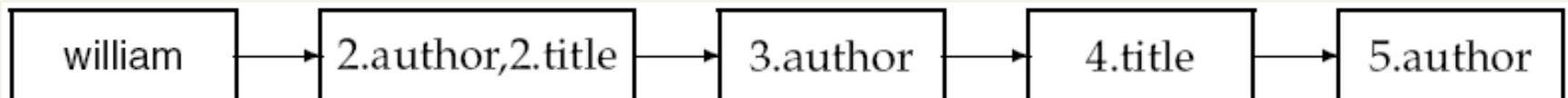
Zone

- A zone is a region of the doc that can contain an arbitrary amount of text e.g.,
 - Title
 - Abstract
 - References ...
- Build inverted indexes on zones as well to permit querying
- E.g., “find docs with *merchant* in the title zone and matching the query *gentle rain*”

Example zone indexes



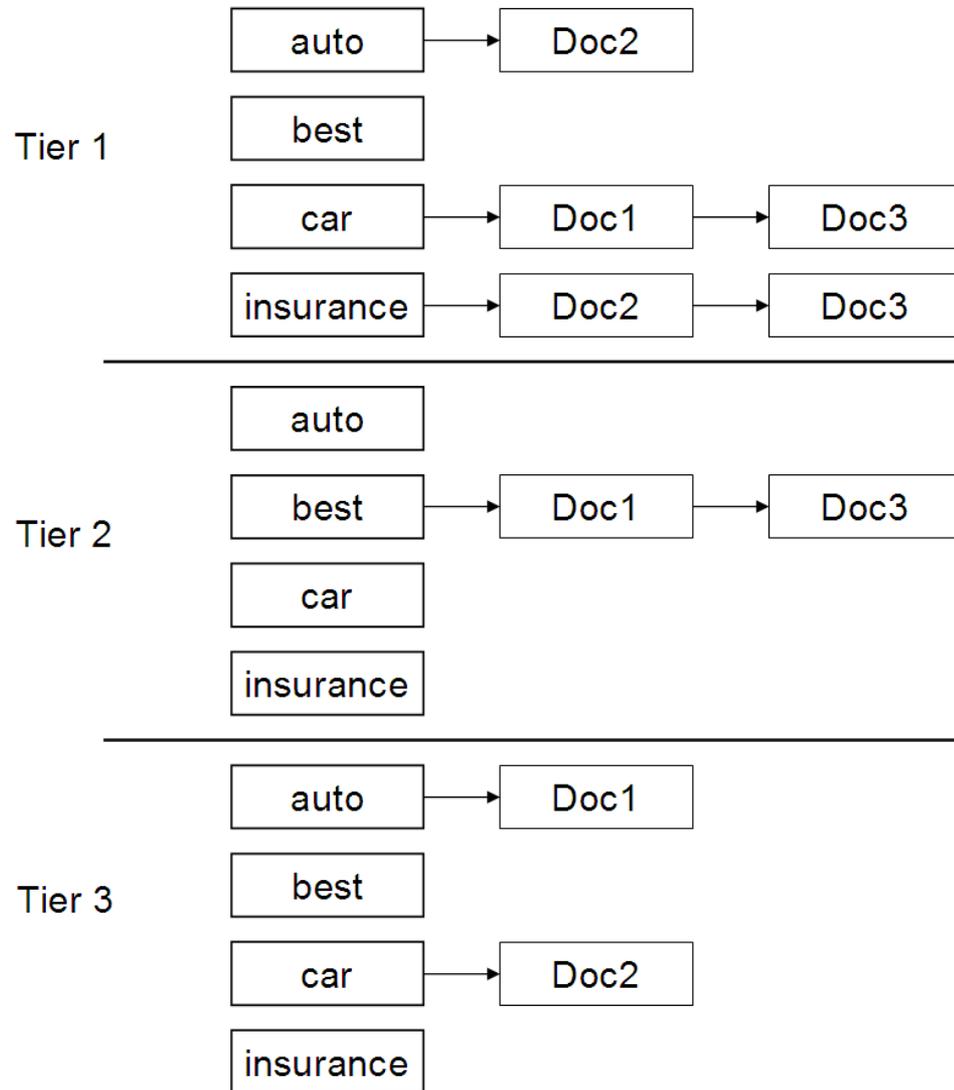
Encode zones in dictionary vs. postings.



Tiered indexes

- Break postings up into a hierarchy of lists
 - Most important
 - ...
 - Least important
- Can be done by $g(d)$ or another measure
- 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



Query term proximity

- Free text queries: just a set of terms typed into the query box – common on the web
- Users prefer docs in which query terms occur within close proximity of each other
- Let w be the smallest window in a doc containing all query terms, e.g.,
- For the query *strained mercy* the smallest window in the doc *The quality of mercy is not strained* is 4 (words)
- Would like scoring function to take this into account – how?

Query parsers

- Free text query from user may in fact spawn one or more queries to the indexes, e.g. query *rising interest rates*
 - Run the query as a phrase query
 - If $<K$ docs contain the phrase *rising interest rates*, run the two phrase queries *rising interest* and *interest rates*
 - If we still have $<K$ docs, run the vector space query *rising interest rates*
 - Rank matching docs by vector space scoring
- This sequence is issued by a query parser

Aggregate scores

- We've seen that score functions can combine cosine, static quality, proximity, etc.
- How do we know the best combination?
- Some applications – expert-tuned
- Increasingly common: machine-learned

Some Considerations on Scoring and Ranking

Digression: spamming indices

- This was all invented before the days when people were in the business of spamming web search engines. Consider:
 - Indexing a sensible passive document collection vs.
 - An active document collection, where people (and indeed, service companies) are shaping documents in order to maximize scores

Interaction: vectors and phrases

- Scoring phrases doesn't fit naturally into the vector space world:
 - *"tangerine trees" "marmalade skies"*
 - Positional indexes don't calculate or store tf.idf information for *"tangerine trees"*
- Biword indexes treat certain phrases as terms
 - For these, we can pre-compute tf.idf.
 - Theoretical problem of correlated dimensions
- Problem: we cannot expect end-user formulating queries to know what phrases are indexed
- We can use a positional index to boost or ensure phrase occurrence

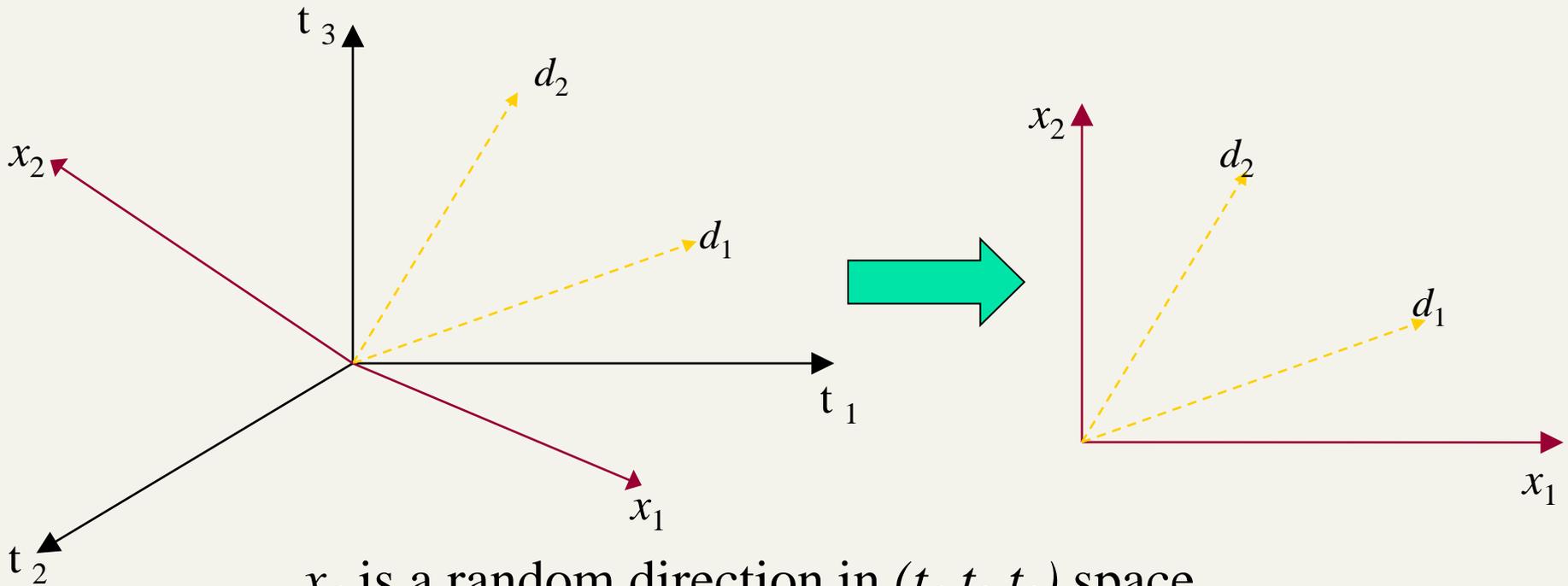
Dimensionality reduction

- What if we could take our vectors and “pack” them into fewer dimensions (say 50,000→100) while preserving distances?
- (Well, almost.)
 - Speeds up cosine computations.
- Two methods:
 - Random projection.
 - “Latent semantic indexing”.

Random projection onto $k \ll m$ axes

- Choose a random direction x_1 in the vector space.
- For $i = 2$ to k ,
 - Choose a random direction x_i that is orthogonal to x_1, x_2, \dots, x_{i-1} .
- Project each document vector into the subspace spanned by $\{x_1, x_2, \dots, x_k\}$.

E.g., from 3 to 2 dimensions



x_1 is a random direction in (t_1, t_2, t_3) space.
 x_2 is chosen randomly but orthogonal to x_1 .

Dot product of x_1 and x_2 is zero.

Guarantee

- With high probability, relative distances are (approximately) preserved by projection.

Computing the random projection

- Projecting n vectors from m dimensions down to k dimensions:
 - Start with $m \times n$ matrix of terms \times docs, A .
 - Find random $k \times m$ orthogonal projection matrix R .
 - Compute matrix product $W = R \times A$.
- j^{th} column of W is the vector corresponding to doc j , but now in $k \ll m$ dimensions.

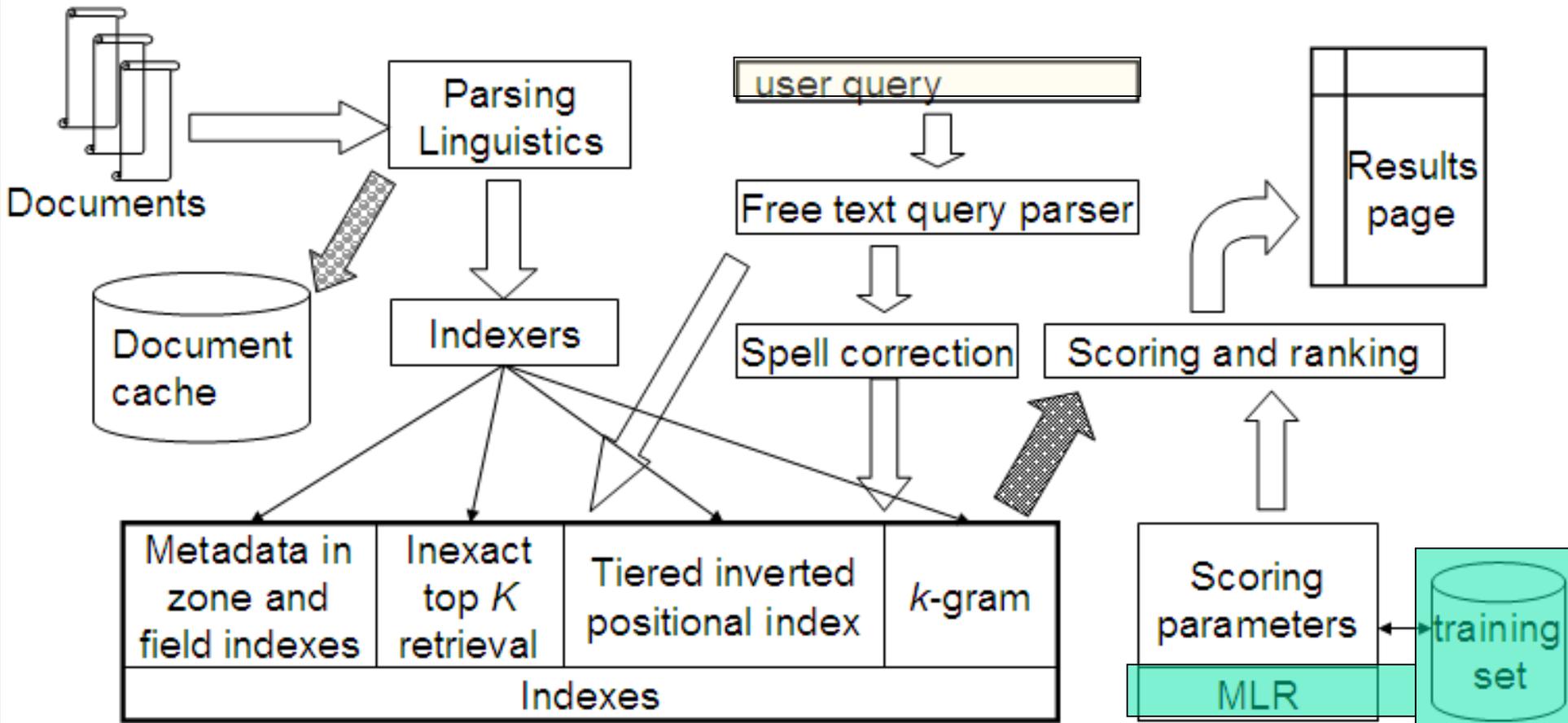
Cost of computation

- This takes a total of kmn multiplications.
- Expensive.
- *Question: by projecting from 50,000 dimensions down to 100, are we really going to make each cosine computation faster?*

Latent semantic indexing (LSI)

- Another technique for dimension reduction
- Random projection was *data-independent*
- LSI on the other hand is *data-dependent*
 - Eliminate redundant axes
 - Pull together “related” axes – hopefully
 - *car* and *automobile*

Putting it all together



Results summaries

Summaries

- Having ranked the documents matching a query, we wish to present a results list
- Most commonly, the document title plus a short summary
- The title is typically automatically extracted from document metadata
- What about the summaries?

Summaries

- Two basic kinds:
 - Static
 - Dynamic
- A static summary of a document is always the same, regardless of the query that hit the doc
- Dynamic summaries are *query-dependent* and attempt to explain why the document was retrieved for the query at hand (aka **snippets**)

Static summaries

- In typical systems, the static summary is a subset of the document
- Simplest heuristic: the first 50 (or so – this can be varied) words of the document
 - Summary cached at indexing time
- More sophisticated: extract from each document a set of “key” sentences
 - Simple NLP heuristics to score each sentence
 - Summary is made up of top-scoring sentences.
- Most sophisticated: NLP used to synthesize a summary
 - Seldom used in IR; cf. **text summarization** work

Dynamic summaries

- Present one or more “windows” within the document that contain several of the query terms
 - “KWIC” snippets: Keyword in Context presentation
- Generated in conjunction with scoring
 - If query found as a phrase, the/some occurrences of the phrase in the doc
 - If not, windows within the doc that contain multiple query terms
- The summary itself gives the entire content of the window – all terms, not only the query terms – how?

Generating dynamic summaries

- If we have only a positional index, we cannot (easily) reconstruct context surrounding hits
- If we *cache the documents* at index time, can run the window through it, cueing to hits found in the positional index
 - E.g., positional index says “the query is a phrase in position 4378” so we go to this position in the cached document and stream out the content
- Most often, cache a fixed-size prefix of the doc
 - Note: Cached copy can be outdated

Dynamic summaries

- Producing good dynamic summaries is a tricky optimization problem
 - The real estate for the summary is normally small and fixed
 - Want short item, so show as many KWIC matches as possible, and perhaps other things like title
 - Want snippets to be long enough to be useful
 - Want linguistically well-formed snippets: users prefer snippets that contain complete phrases
 - Want snippets maximally informative about doc
- But users really like snippets, even if they complicate IR system design

Căutare

Aproximativ 2.260.000.000 (de) rezultate (0,13 secunde)

Toate

Imagini

Videoclipuri

Știri

Cărți

Mai multe

București

Schimbați locația

Rețeaua web

Pagini scrise în română

Pagini din România

Pagini străine traduse

Oricând

Din ultima oră

Din ultimele 24 de ore

Din ultimele 3 zile

Săptămâna trecută

Luna trecută

Anul trecut

Interval personalizat...

Toate rezultatele

Site-uri cu imagini

Mai multe instrumente

Sfat: [Căutați numai rezultate în Română](#). Puteți specifica limba dvs. de căutare în [Preferințe](#).

Căutări similare: [engine cheat](#) [engine 5.4 download](#)

[Engine - Wikipedia, the free encyclopedia](#)

<en.wikipedia.org/wiki/Engine> - Traducerea acestei pagini [+?](#)

An **engine** or motor is a machine designed to convert energy into useful mechanical motion. Heat **engines**, including internal combustion **engines** and external ...

[Diesel engine - Internal combustion engine - Motorcycle engine - Aircraft engine](#)

[Internal combustion engine - Wikipedia, the free encyclopedia](#)

[en.wikipedia.org/wiki/Internal_combustion_e...](en.wikipedia.org/wiki/Internal_combustion_engine) - Traducerea acestei pagini [+?](#)

The internal combustion **engine** is an **engine** in which the combustion of a ...

[+](#) Afișați mai multe rezultate de la wikipedia.org

[Animated Engines](#)

<www.animatedengines.com/> - Traducerea acestei pagini [+?](#)

Animated illustrations show the innerworkings of a variety of Steam, Stirling, and Internal Combustion **engines**.

[HowStuffWorks "How Car Engines Work"](#)

<www.howstuffworks.com/engine.htm> - Traducerea acestei pagini [+?](#)

A car **engine** is one of the most amazing machines we use on a daily basis. Learn all about car **engines** and the different types of combustion **engines**.

[Imagini pentru engine - Raportați imagini](#)[Engine Service Design](#)

<www.enginegroup.co.uk/> - Traducerea acestei pagini [+?](#)

Engine is a service design consultancy helping organisations to evolve and innovate services that attract and keep customers, increase revenue and build brand ...

[How A Car Engine Works - 3D Animation - Video](#)

www.metacafe.com/.../how_a_car_engine_works_3d_animatio... [+?](#)

26 Iun 2007 - 5 min

(Empty description). Watch Video about No Description, **Engines**. How by Metacafe.com.

[V 12 Modellmotor RC Engine the original Video ! - YouTube](#)

<www.youtube.com/watch?v=mutb7KgA9NM> [+?](#)

4 Mar 2007 - 2 min - Încărcat de boxermotorrad

Testrun- this is the original Video from the owner, topspeed 6000RPM:6 -8 HP--87 ccm -has got the whole **Engine** for further ...

Mai multe videoclipuri pentru **engine** »

[ENGINE HOUSE, Service, piese si accesorii, tuning 4x4, scuturi ...](#)

<enginehouse.ro/> [+?](#)

Piese si accesorii pentru masini 4x4, pick-up si off-road.

[Best Engine - service auto Timisoara - Acasa](#)

<www.bestengine.ro/> [+?](#)

Service auto Timisoara - electrica auto Timisoara, incarcari clima Timisoara, polishare auto Timisoara, tinichigerie Timisoara Acasa.

Anunțuri

[Quality rebuilt engines](#)

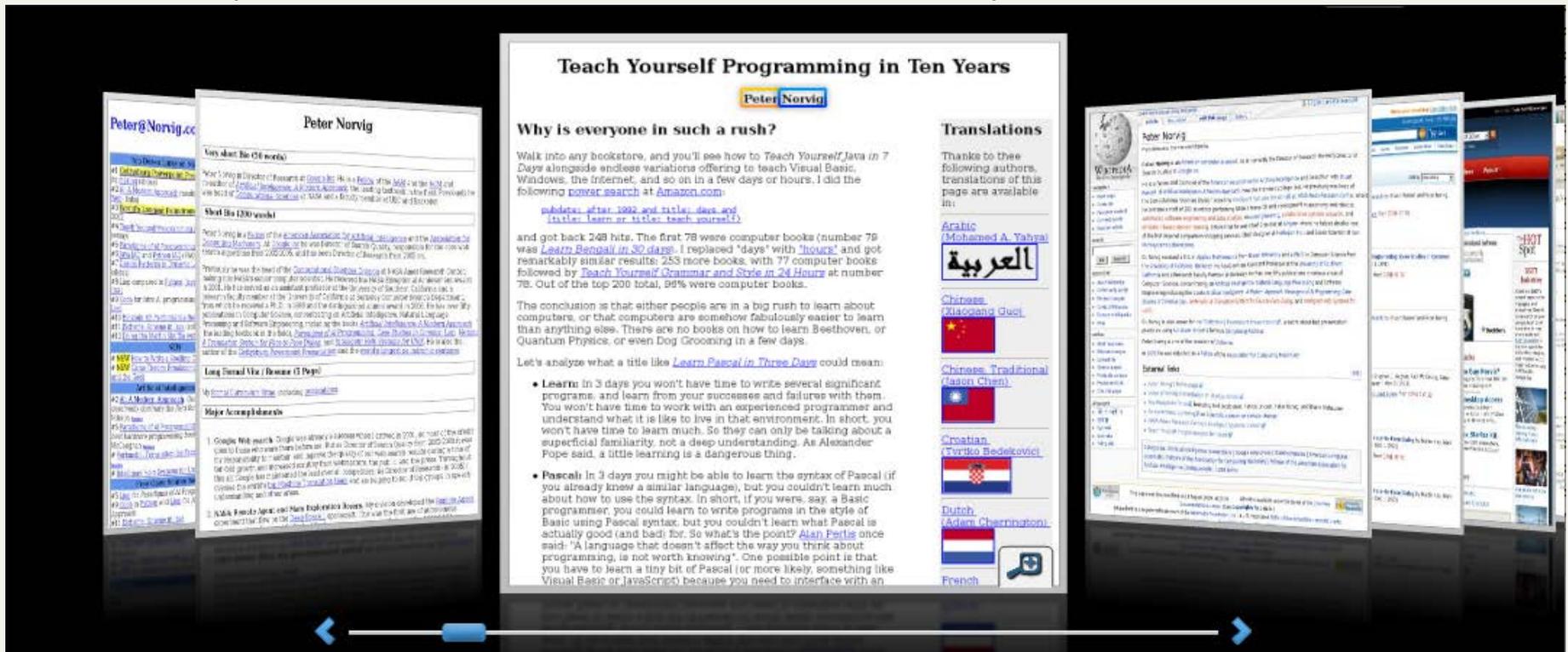
www.rebuilt.com [+?](#)

Great prices on 1400 different boat, RV, import & hi-perf engines

Publicați aici anunțul dvs. »

Alternative results presentations

- An active area of HCI research
- An alternative: <http://www.searchme.com/> / copies the idea of Apple's Cover Flow for search results
 - (Note: searchme went out of business)



Resources for today's lecture

- Term weighting and cosine similarity tutorial for SEO folk!
 - <http://www.miislita.com/information-retrieval-tutorial/cosine-similarity-tutorial.html>
- Other similarity metrics:
 - <http://www.lans.ece.utexas.edu/~strehl/diss/node52.html>