

Introduction to **Information Retrieval**

CS276

Information Retrieval and Web Search

Pandu Nayak and Prabhakar Raghavan

Lecture 7: Scoring and results assembly

Lecture 6 – I introduced a bug

- In my anxiety to avoid taking the log of zero, I rewrote

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

as

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

In fact this was unnecessary, since the zero case is treated specially above; net the FIRST version above is right.

Recap: tf-idf weighting

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

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

- Best known weighting scheme in information retrieval
- Increases with the number of occurrences within a document
- Increases with the rarity of the term in the collection

Recap: 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

Recap: cosine(query, document)

Dot product
Unit vectors

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

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

This lecture

- Speeding up vector space ranking
- Putting together a complete search system
 - Will require learning about a number of miscellaneous topics and heuristics

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[]
```

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

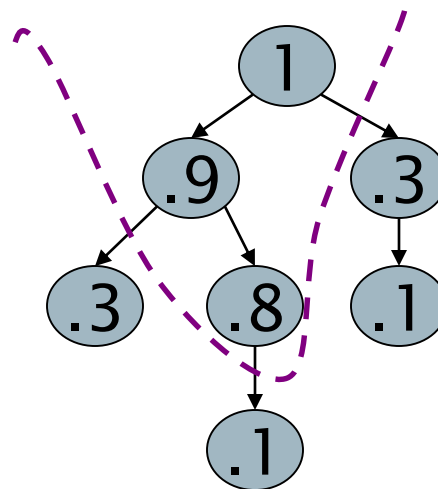
- No weighting on query terms
 - Assume each query term occurs only once
- Then for ranking, don't need to normalize query vector
 - Slight simplification of algorithm from Lecture 6

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 $J=1\text{M}$, $K=100$, 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 cosine computation algorithm 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

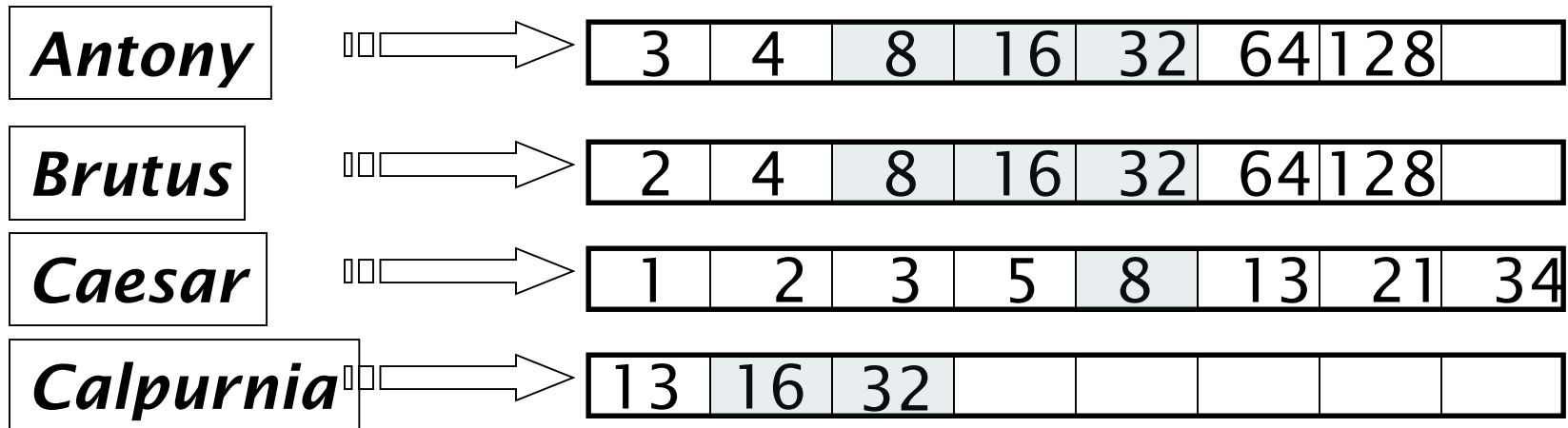
High-idf query terms only

- 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



Scores only computed for docs 8, 16 and 32.

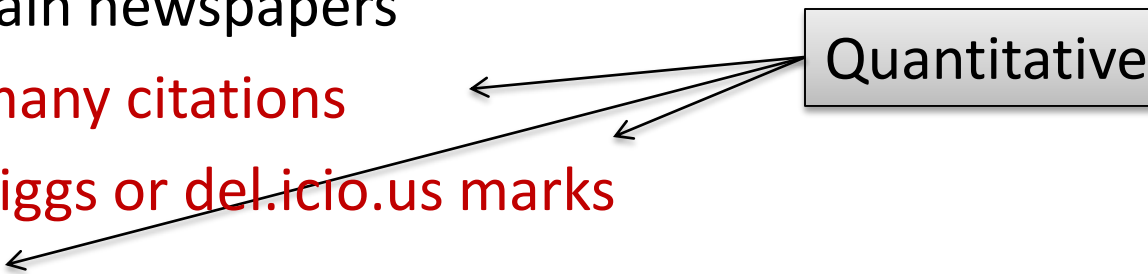
Champion lists

- 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 bitly's, diggs 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
 - 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

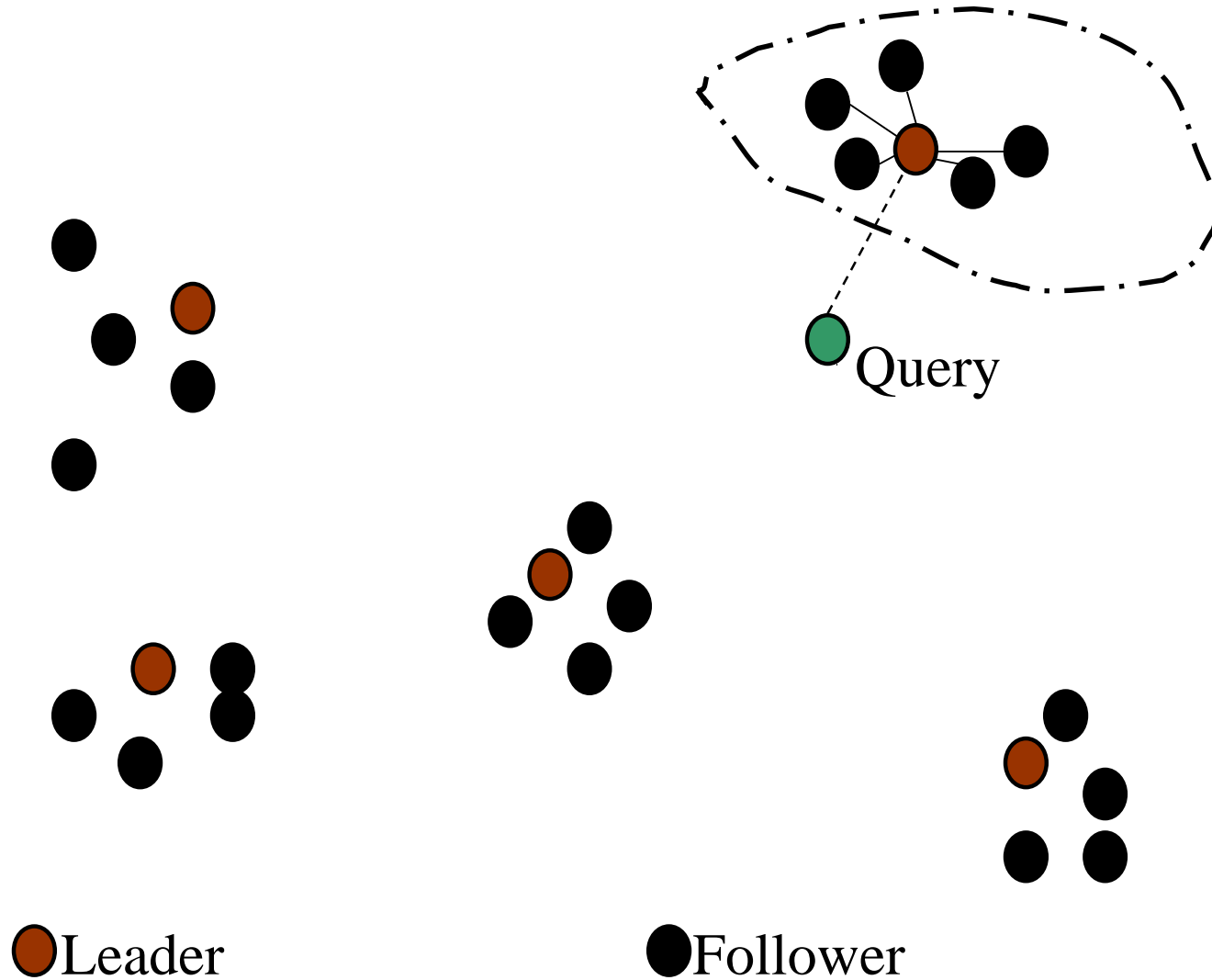
Cluster pruning: preprocessing

- 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 recurse 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

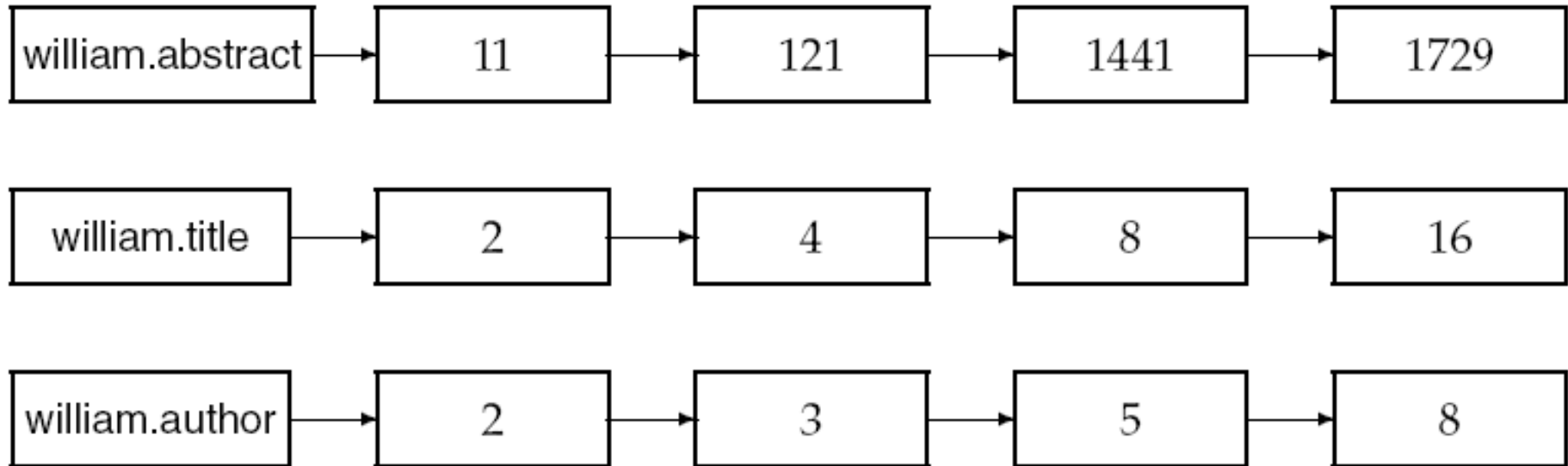
Fields

- 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)

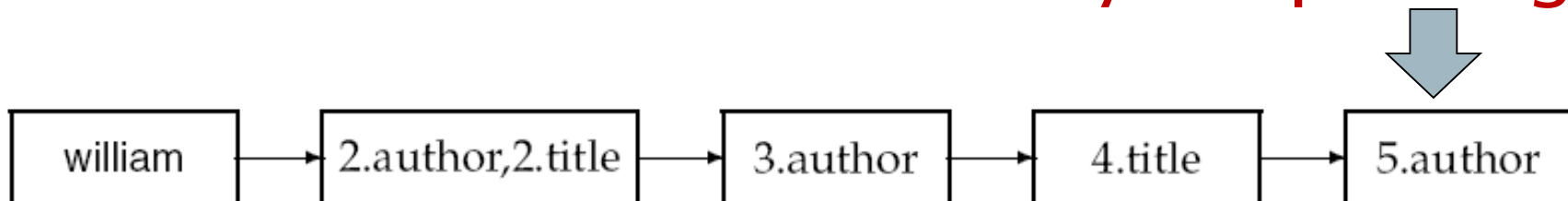
Zone

- 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



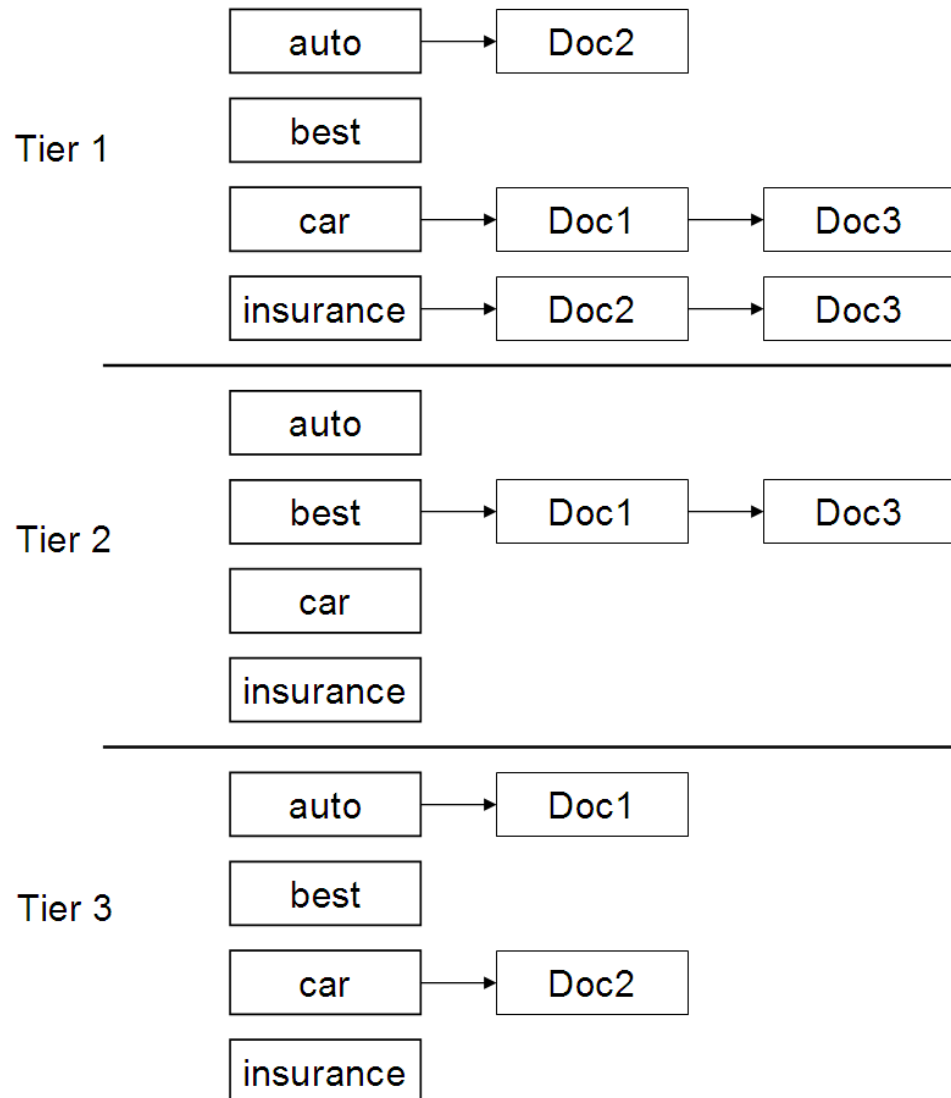
Encode zones in dictionary vs. postings.



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?

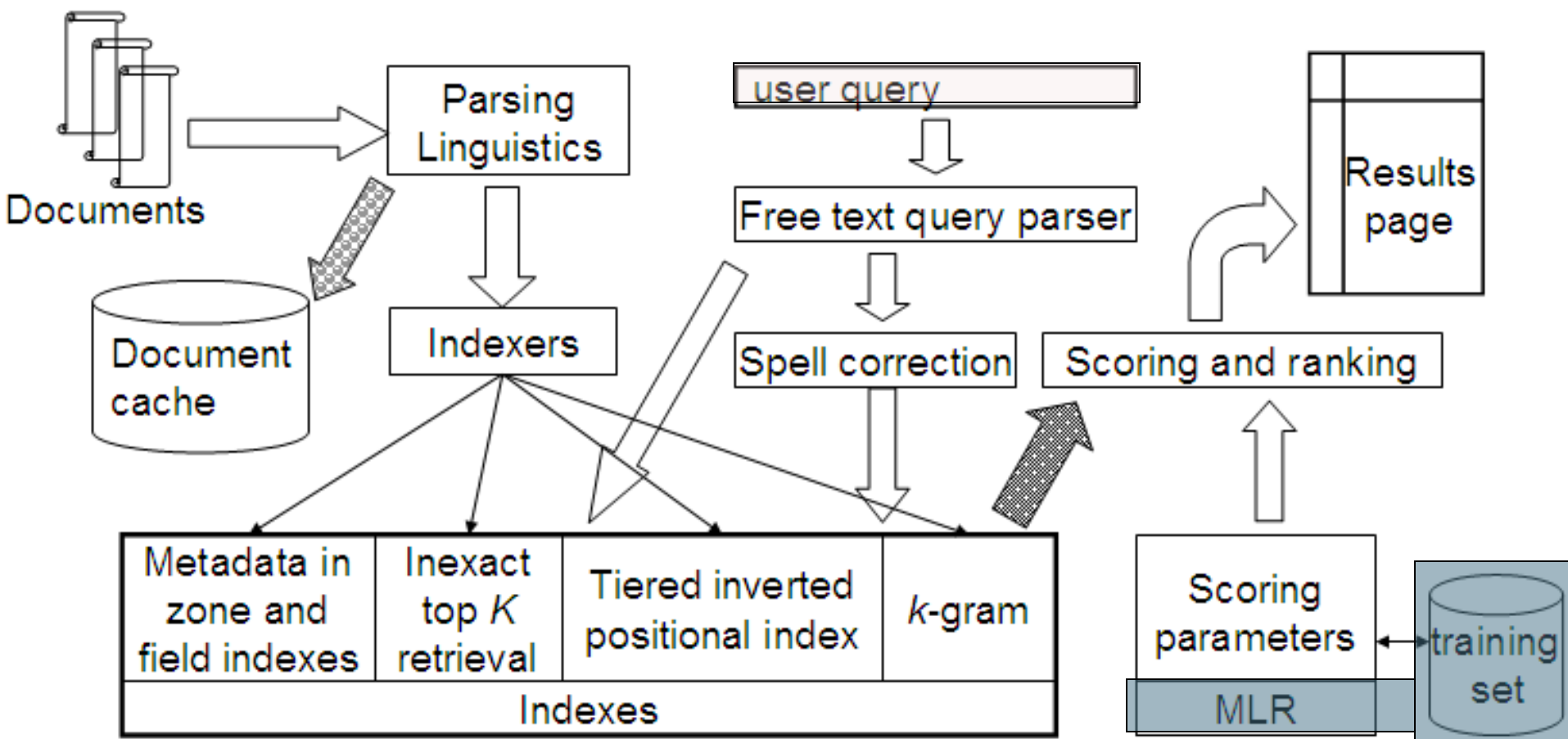
Query parsers

- 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**
 - See May 19th lecture

Putting it all together



Resources

- IIR 7, 6.1