

CS3245

Information Retrieval

Lecture 8: A complete search system –
Scoring and results assembly



Last Time: 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
 - “One of the easy but important things you should remember about IR” – Min
 - Increases with the number of occurrences within a document
 - Increases with the rarity of the term in the collection



Last Time: Vector Space Model

- Key idea 1: represent both d and q as vectors
- Key idea 2: Rank documents according to their proximity (similarity) to the query in this space

Dot product

Unit vectors

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

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



Today

Goal

- Speeding up and shortcutting ranking
- Incorporating additional ranking information into VSM

Method

- Heuristics

Recap:

- An overview of the complete search system



Recap: 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:
 1. Computing a single cosine efficiently



Simpler case – unweighted queries

- No weighting on query terms
 - Assume each query term has weight 1
 - i.e. $w_{t,q} = 1$
(no tf, nor idf factor; just Boolean presence)
- Then for ranking, don't need to normalize query vector
 - Simpler version of algorithm from last week



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  $Scores[]$ 
```

No expensive multiplication;
now just addition

Figure 7.1 A faster algorithm for vector space scores.

Efficient cosine ranking



- Find the K docs in the collection “nearest” to the query $\Rightarrow K$ largest query-doc cosines.
- Efficient ranking:
 1. Computing a single cosine efficiently.
 2. Choosing the K largest cosine values efficiently.

Can we do this without computing all N cosines?

Computing the K largest cosines: selection vs. sorting



- Typically we want to retrieve the top K docs (in the cosine ranking for the query)
 - Don't need total order for **all** docs

Can we pick off docs with K highest cosines?

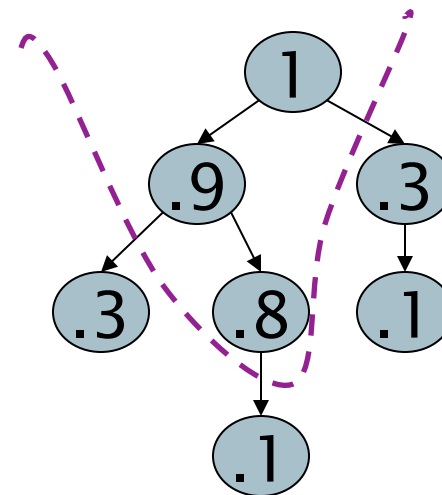
- Formal Problem Specification:
Let J = number of docs with nonzero cosines.
Then we seek the K best of these J



Use heaps for selecting top K

- Heap = Binary tree in which each node's value $>$ the values of children
- Takes $O(J)$ operations to construct, then each of K “winners” read off in $O(\log J)$ steps.

- For $J=1M$, $K=100$, this is about 10% of the cost of sorting



Blanks on slides, you may want to fill in



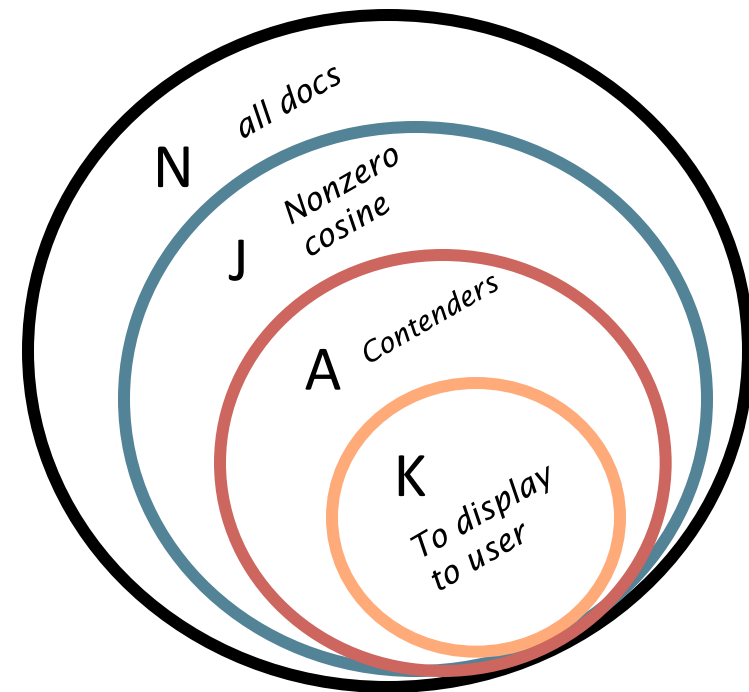
Bottlenecks

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



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 can also be used for other (non-cosine) scoring functions





Heuristic 1: Index elimination

- Basic algorithm: FastCosineScore of Fig 7.1 only considers docs containing at least one query term

- Extend this to a logical conclusion:
 - A. Only consider high idf query terms
 - B. Only consider docs containing many query terms



A. High-idf query terms only

- E.g., given 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
 - Similar in spirit to stopwording



B. Docs containing many query terms

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

Example: Requiring 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 docs 8, 16 and 32.

Blanks on slides, you may want to fill in



Heuristic 2: 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)
 - For tf-idf weighting this just means
- **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



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



Tiered indexes

- Break postings up into a hierarchy of lists

Most important

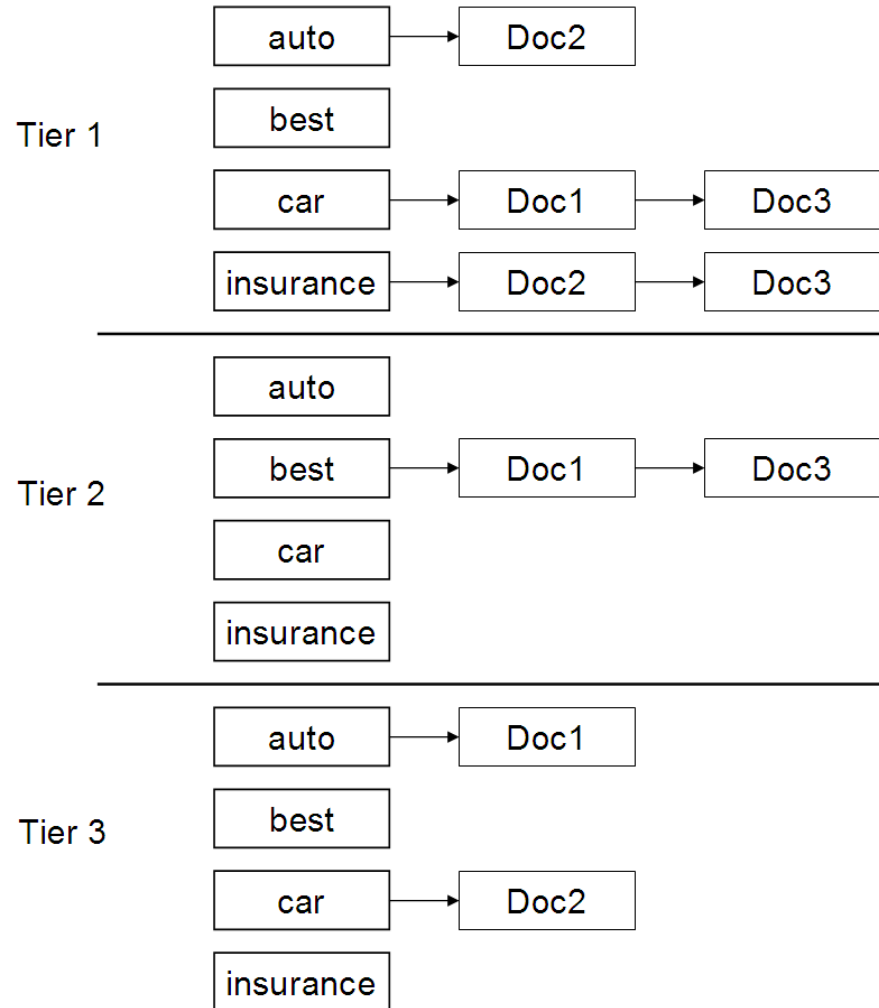
...

Least important

- Inverted index thus broken up into tiers of decreasing importance
- At query time, use only top tier unless insufficient to get K docs.
If so, drop to lower tiers
- Generalization of high-low lists



Example tiered index



To think about:
What information would be useful to use to determine tiers?



Heuristic 3: 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 $wf_{t,d}$
- Problem: not all postings in a common order!
(Concurrent traversal then not possible)
- **How do we compute scores in order to pick off top K ?**
Two ideas:
 - A. Early Termination
 - B. IDF Ordered Terms



A. Early termination

- Sort t 's postings by descending $wf_{t,d}$ value
- 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



B. 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 weighting but also other net scores

Heuristic 4: 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.

Why choose leaders at random?

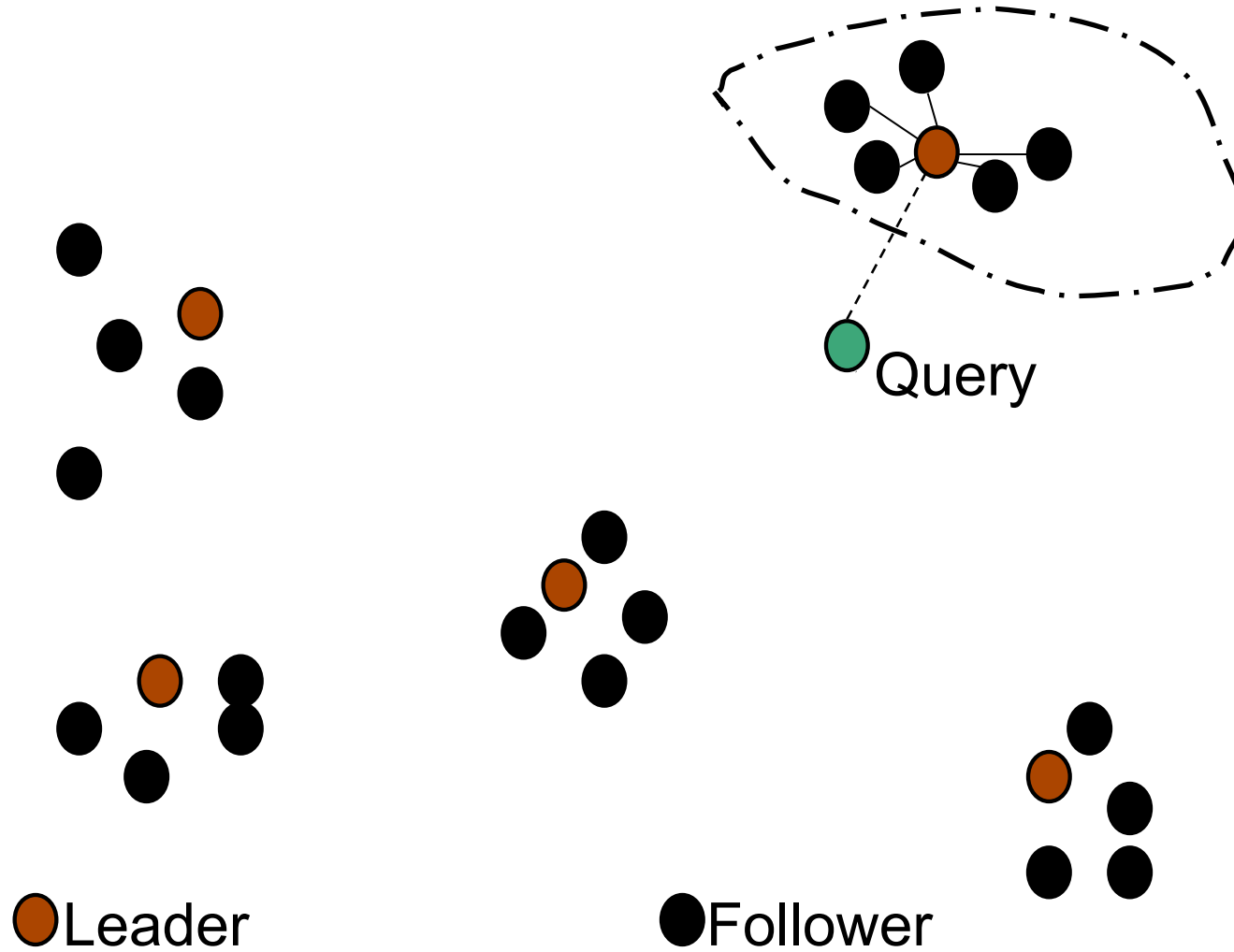
- Fast
- Leaders reflect data distribution



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



Clustering Pruning Variants



- Have each follower attached to $b1$ nearest leaders
- From query, find $b2$ nearest leaders and their followers
- $b1$ affects preprocessing step at indexing time
- $b2$ affects query processing step at run time
- Think about how these parameters affect precision, recall

Incorporating Additional Information: 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, retweets or del.icio.us bookmarks
 - PageRank score

Quantitative

A red rectangular box containing the word 'Quantitative' has four red arrows pointing to the following items in the list: 'A paper with many citations', 'Many diggs, retweets or del.icio.us bookmarks', 'PageRank score', and 'Wikipedia among websites'.

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 $[0,1]$



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
-
- 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**



Wait a second. We previously said documents need to be in order of docID to be merged efficiently. Why does this not violate it?

- Thus, can concurrently traverse query terms' postings for
 - Postings intersection
 - Cosine score computation

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

Combining Ideas: 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 $g(d) + \text{tf-idf}_{t,d}$ instead of just $\text{tf-idf}_{t,d}$
- Seek top- K results from only the docs in these champion lists



Parametric and zone indexes

(Back to Chapter 6 skipped last week)

- Thus far, a doc has been a sequence of terms
- Documents often have multiple parts, with different semantics:
 - Author, Title, Date of publication, etc.
- These constitute the metadata about a document
- We sometimes wish to search by these metadata
 - E.g., find docs authored by William Shakespeare in the year 1601, containing *alas poor Yorick*

Fields

Will be used
in HW#4



- 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 (B-tree) trees (e.g., for dates)
- Field query typically treated as conjunction
 - (doc *must* be authored by shakespeare)

Zone

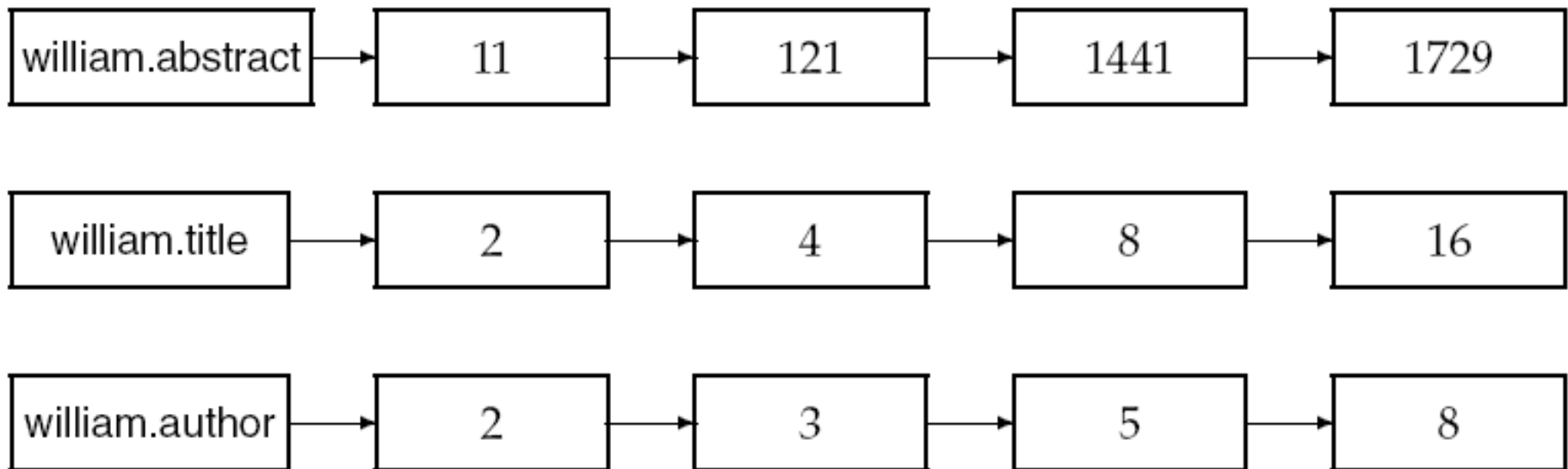
Also used in
HW#4



- 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*”

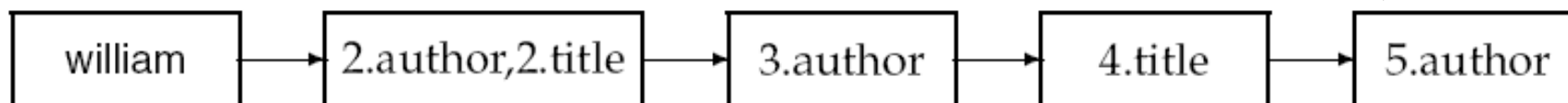
Two methods for zone indexing

Alternative 1:



Encode zones in dictionary vs. postings.

Alternative 2:





Query term proximity

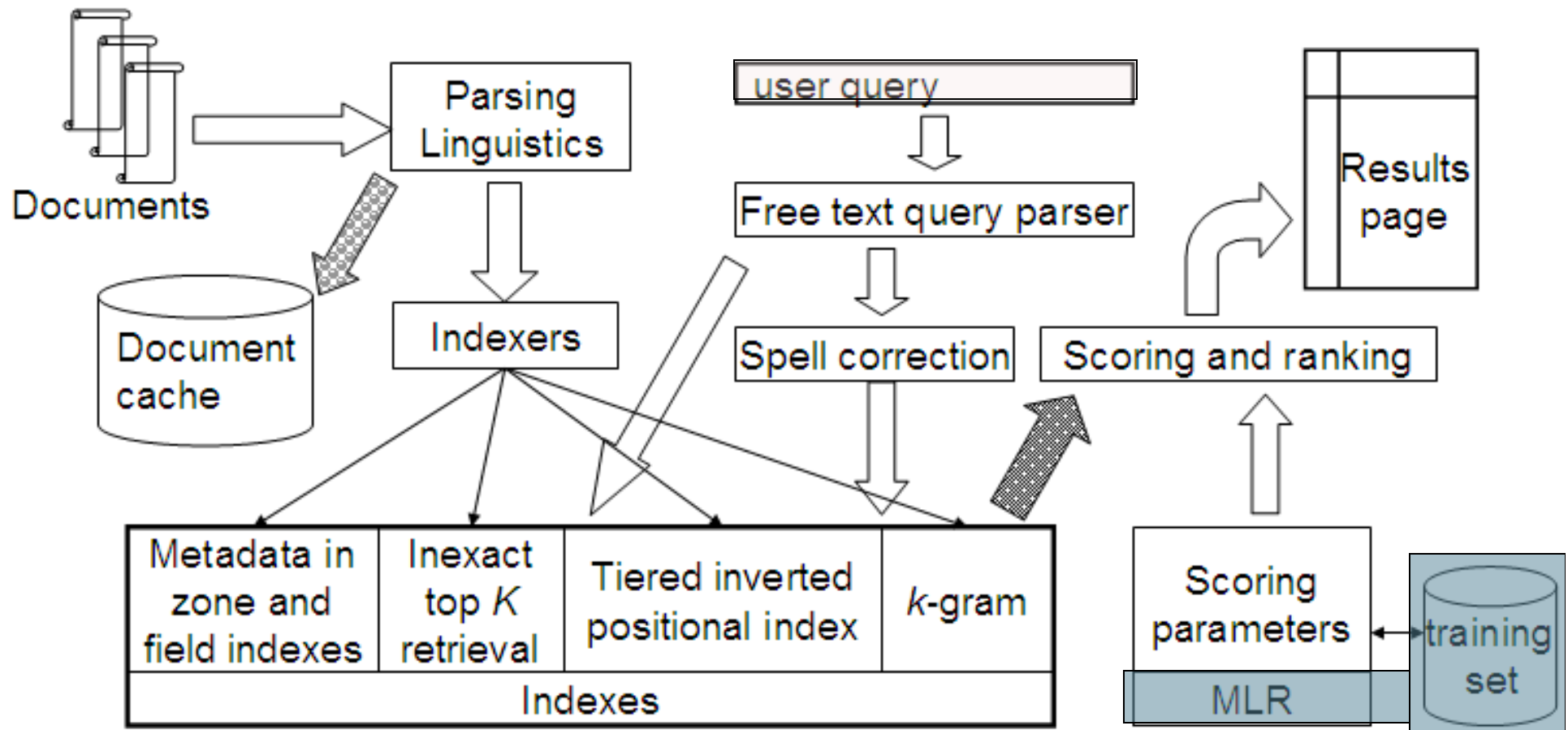
- Free text queries: just a set of terms typed into the query box – common on the web
- Users prefer docs where the query terms occur close to each other
- Let w be the smallest window in a doc containing all query terms, e.g.,
- For the query *strained mercy* the smallest window in the doc *The quality of mercy is not strained* is 4.



Query parsers

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

Putting it all together



Won't be covering these blue modules in this course



Summary

Making the Vector Space Model more effective and efficient to compute

- Incorporating other ranking information $G(d)$
- Approximating the actual correct results
- Skipping unnecessary documents

In actual data: dealing with zones and fields, query term proximity

Resources for today

- IIR 7, 6.1