

CS3245

Information Retrieval

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



Live Q&A
<https://pollev.com/jin>

Last Time: tf-idf weighting



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

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

- Best known weighting scheme in information retrieval
 - One of the easy but important things you should remember for IR
 - Increases with the number of occurrence 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

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

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



Computing cosine scores, redux

COSINESCORE(q)

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

*Consider only the terms
appearing in both q and d .*

```

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  $Scores[]$ 

```

Dot product

*Normalize by the (pre-computed)
document length only.*

Normalization

Today



Goal

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

Efficient cosine ranking



- Key observations
 - Users only checks the top results.
 - There are probably too many (relevant) documents in the first place.
- Given a collection of N documents and a query
 - Find K ($\ll N$) docs that are (likely to be) the "nearest" to the query based on cosine similarity.
- Efficient ranking
 - Simplify the processing
 - Possibly less accurate / exact



Faster cosine: unweighted query

- To simplify the computation of a single cosine, we can...
- Assume each query term has weight 1
 - i.e., $w_{t,q} = 1$ (no *tf*, nor *idf* factor; just Boolean presence)
 - Before: $\text{Scores}[d] += w_{t,d} \times w_{t,q}$
 - After: $\text{Scores}[d] += w_{t,d}$ ← *No expensive multiplication, only addition*
- But the bigger bottleneck is to process all N documents in the collection...

Let's shrink the collection...



- Full collection = N documents
- Documents that do not contain any query terms have zero cosine values
 - Q: emperor
 - Doc1: queen, Doc2: the emperor, ...
 - $\text{Score}(Q, \text{Doc1}) = 0$
- Such documents can be safely ignored...Let's call the remaining collection of documents J .



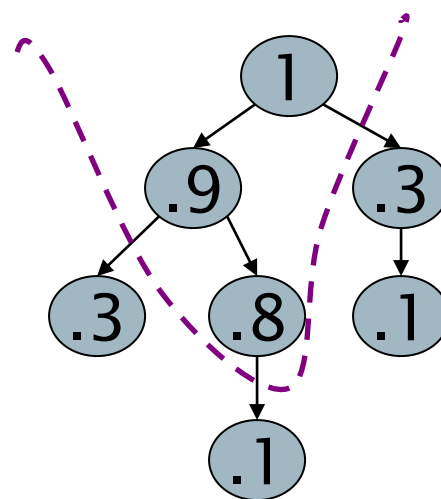
Optimizing the selection process

- What we need: Select **K** best out of **J**
 - Typically, $K \ll J$
 - Query: emperor
 - J (i.e., docs containing emperor) = 1M, but K could be just 100
- Sort and output top $K = O(J \log J + K)$
- Can we do better?

Use heaps for selecting top K

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

- For $J = 1M$, $K = 100$, this is about 5% of the cost of sorting and outputting (with log base 2)



Blanks on slides, you may want to fill in

Bottlenecks

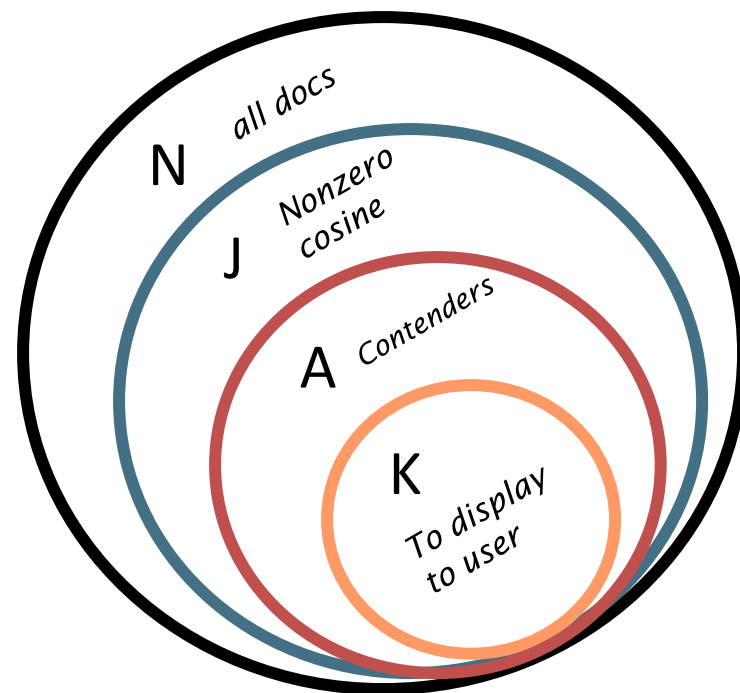


- Primary computational bottleneck in scoring: cosine computation
- Can we avoid doing this computation for all docs in J ?
 - Yes, we need to do some pruning.
- We may get it wrong sometimes but it is ok if we are not missing too many.
 - It is unlikely that the user really want **all** relevant documents.



Generic approach

- Find a set A of contenders, with $K < |A| \ll |J| \ll N$
 - 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.



Blanks on slides, you may want to fill in



Heuristic 1: Index elimination

- Basic algorithm: FastCosineScore of Fig 7.1 considers docs containing at least one query term (i.e., set J)
 - 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
- J will be large and the computation will be slow if
- We can in fact ignore part of the index (i.e., postings lists) based on the query.



1a. High-idf query terms only

- E.g., given a query such as ***catcher in the rye*** only accumulate scores from *catcher* and *rye*
- It is usually not important to match **in** and **the** anyway since they have low idfs.
- Benefit:
 - Postings of low *idf* terms have many docs → these (many) docs get eliminated from set *A* of contenders
 - Similar in spirit to stop word removal



1b. Docs containing many query terms

- Any doc with at least one query term is a candidate from 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 query terms
- Easy to implement in postings traversal

Example: Requiring 3 of 4 query terms

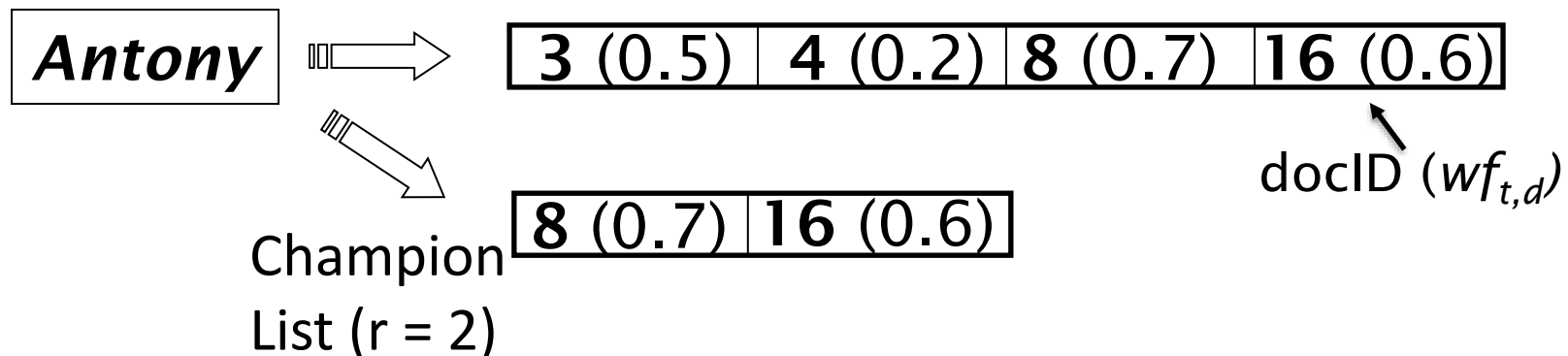
<i>Antony</i>	⇒	<table border="1"><tr><td>3</td><td>4</td><td>8</td><td>16</td><td>32</td><td>64</td><td>128</td><td></td></tr></table>	3	4	8	16	32	64	128	
3	4	8	16	32	64	128				
<i>Brutus</i>	⇒	<table border="1"><tr><td>2</td><td>4</td><td>8</td><td>16</td><td>32</td><td>64</td><td>128</td><td></td></tr></table>	2	4	8	16	32	64	128	
2	4	8	16	32	64	128				
<i>Caesar</i>	⇒	<table border="1"><tr><td>1</td><td>2</td><td>3</td><td>5</td><td>8</td><td>13</td><td>21</td><td>34</td></tr></table>	1	2	3	5	8	13	21	34
1	2	3	5	8	13	21	34			
<i>Calpurnia</i>	⇒	<table border="1"><tr><td>13</td><td>16</td><td>32</td><td></td><td></td><td></td><td></td><td></td></tr></table>	13	16	32					
13	16	32								

Scores only computed for docs 8, 16 and 32.

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
(a.k.a. fancy list or top docs for t)



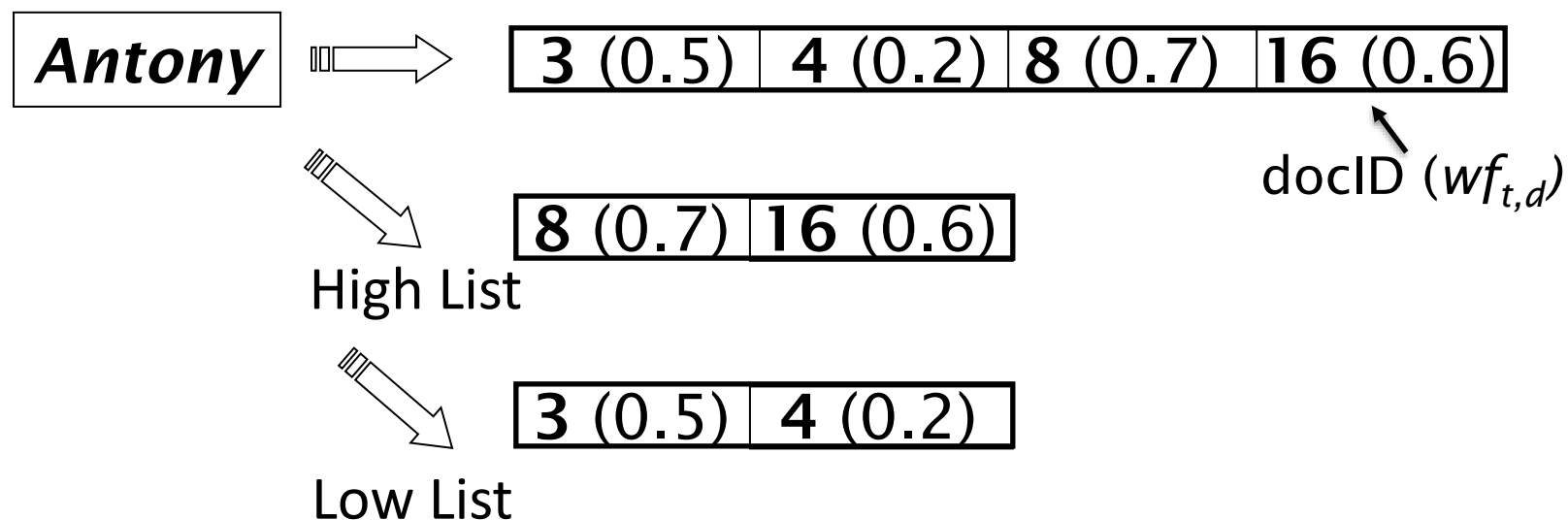
Heuristic 2: Champion lists



- 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
- Note that r has to be chosen at the indexing stage
 - Thus, it's possible that $r < K$

High and low lists

- For each term, we maintain two postings lists called *high* and *low*
 - Think of *high* as the champion



High and low lists



- 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



- Generalizing high-low lists into tiers
- Break postings up into a hierarchy of lists

Most important

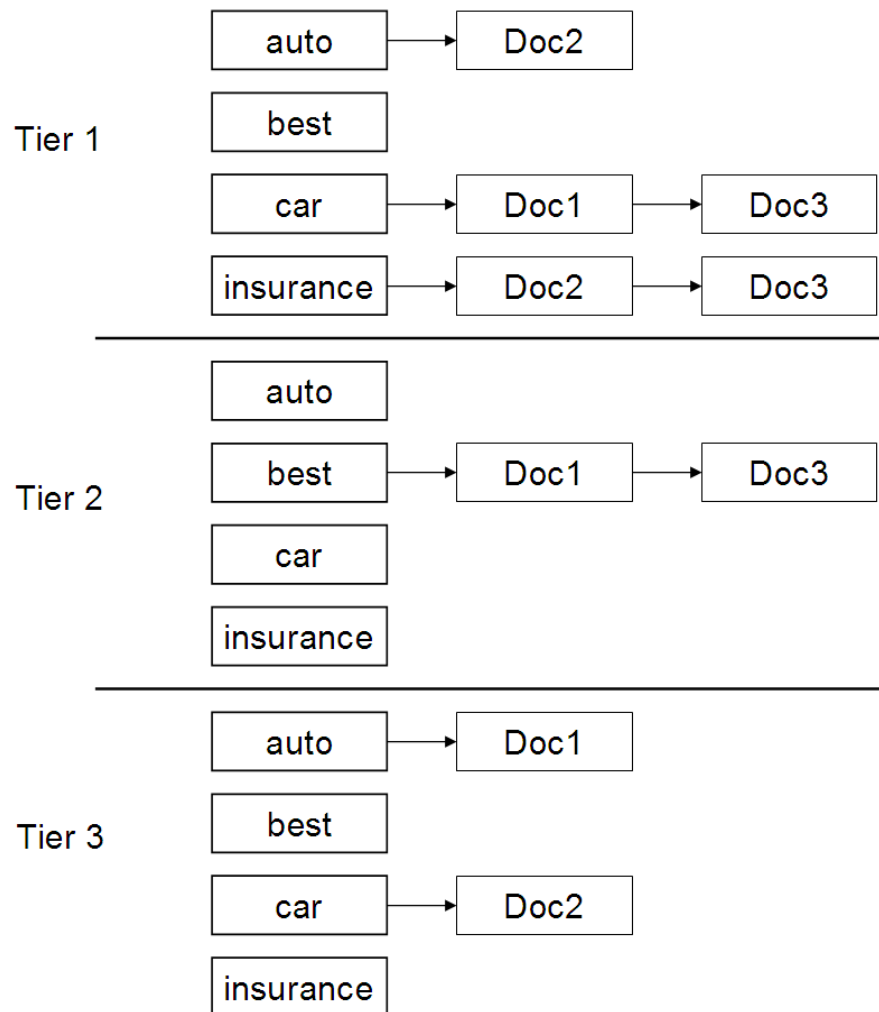
...

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



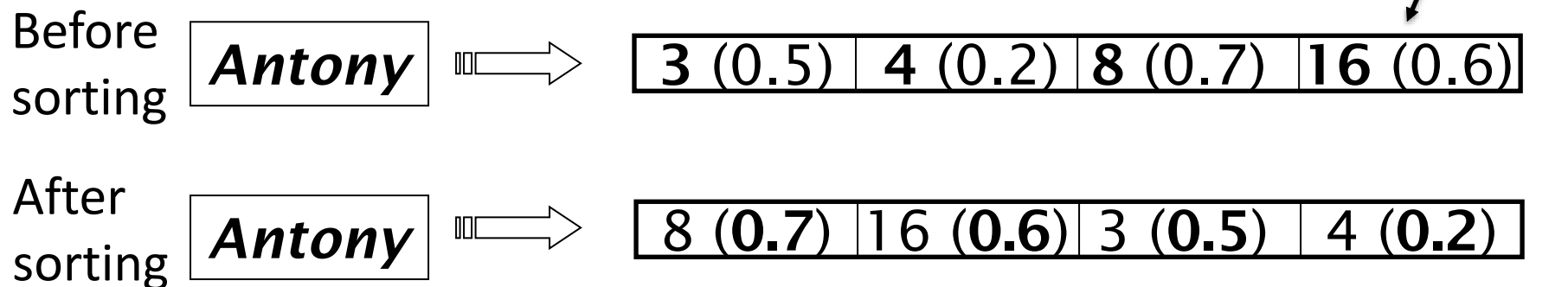
Example tiered index



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

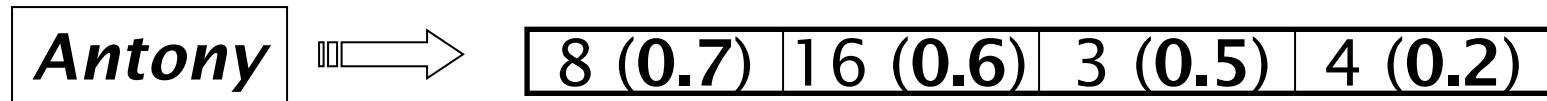
Heuristic 3: Impact-ordered postings

- We only want to compute scores for docs for which $wf_{t,d}$ is high enough
- We **sort** each postings list by $wf_{t,d}$





3a. Early termination

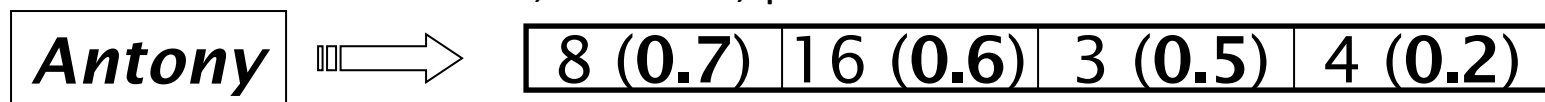


- When traversing t 's postings (sorted by $wf_{t,d}$), stop early after either
 - a fixed number of r docs
 - $wf_{t,d}$ drops below some threshold

The score contribution ($wf_{t,d} * wf_{t,q}$) is likely to be too low beyond these.
- Take the union of the resulting sets of docs
 - One set from the postings of each query term
- Compute only the scores for docs in this union

3b. idf-ordered query terms

- Consider the postings of query terms in order of decreasing *idf*
 - Query: *story Caesar Antony*
 - Order of processing: *Antony Caesar story*
- Skip low-idf query terms completely (e.g., ignore *story*) ← Similar to 1a
- Move on to the next query term once the score contribution ($wf_{t,d} * wf_{t,q}$) is low (e.g., ≤ 0.5)



E.g., if the query term weight of Antony is **0.9**, skip to Caesar after checking the 3rd document.

Heuristic 4: Cluster pruning – preprocessing



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

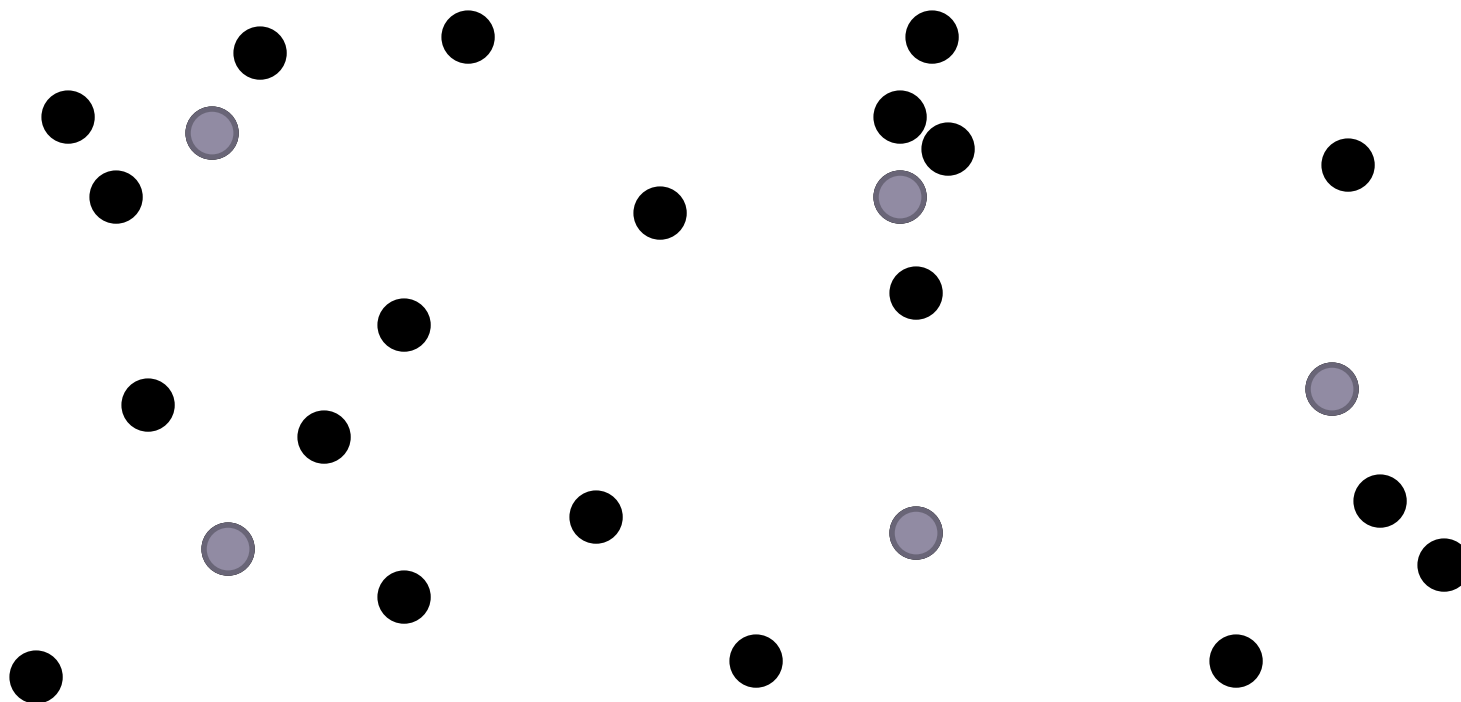
Why choose leaders at random?

- Fast
- Leaders reflect data distribution

Cluster pruning visualization



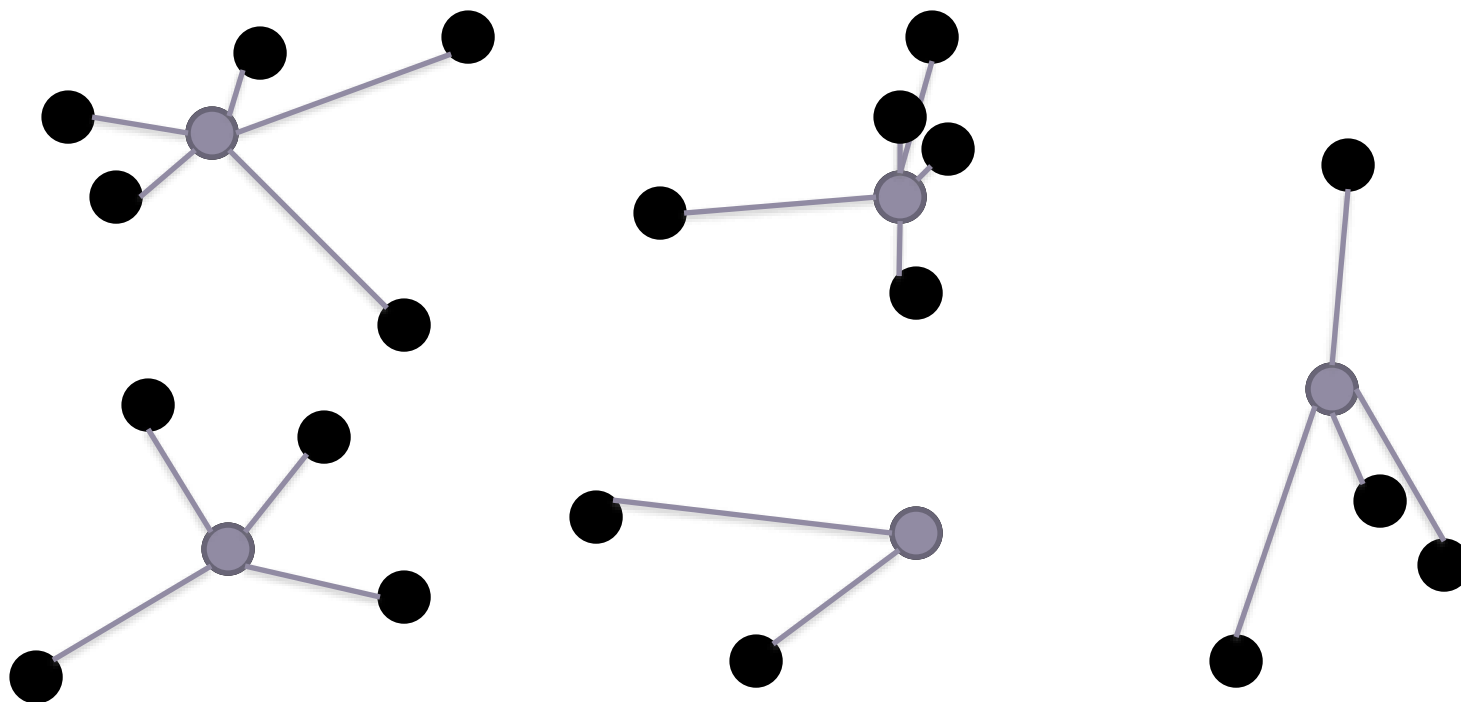
1. Offline: Choose \sqrt{N} leaders



Cluster pruning visualization



2. Associate documents to leaders to form clusters





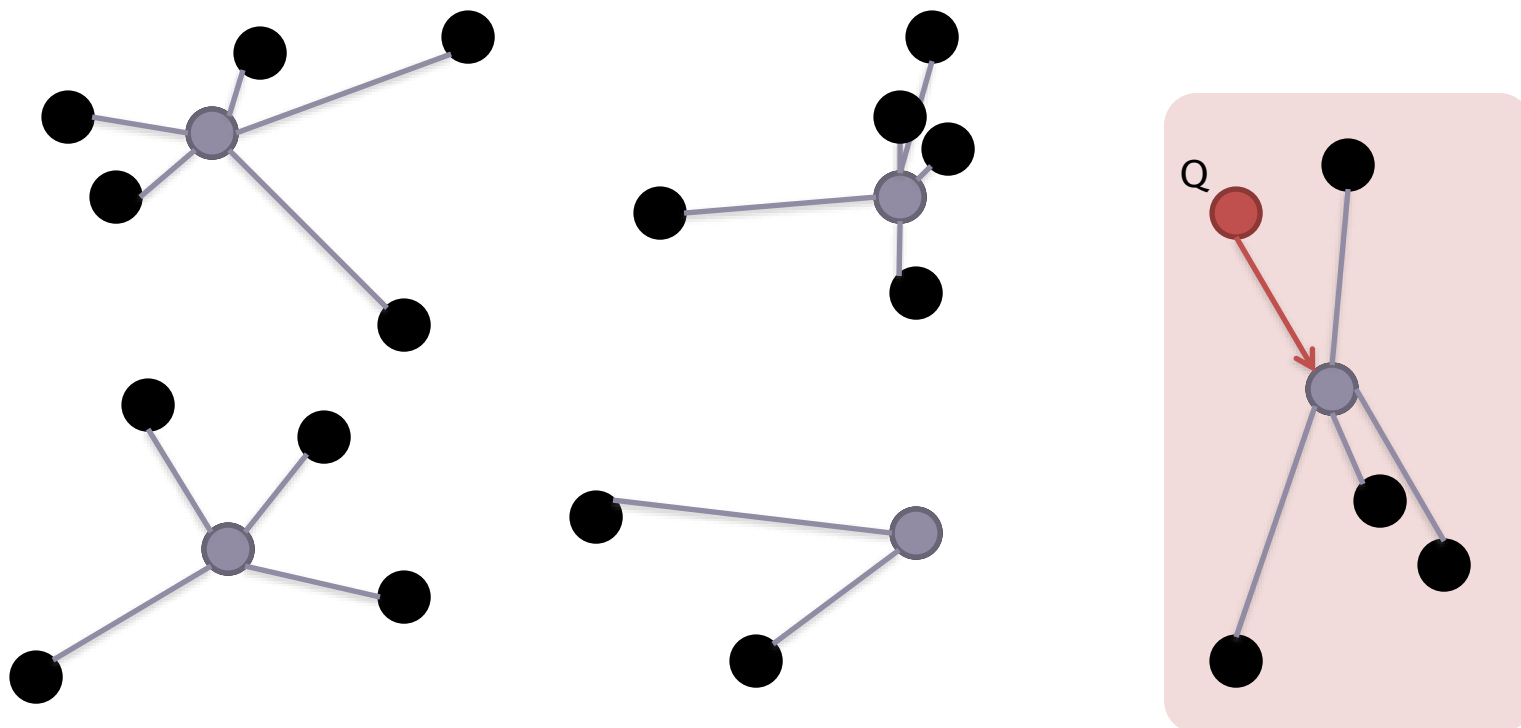
Cluster pruning – query processing

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

Cluster pruning visualization



3. Online: Associate query to a leader (cluster)





Clustering pruning variants



- Have each follower attached to b_1 nearest leaders
- From query, find b_2 nearest leaders and their followers
- b_1 affects preprocessing step at indexing time
- b_2 affects query processing step at run time

To think about: How do these parameters affect the retrieval results?

Incorporating Additional Information: Static quality scores



- We want top-ranking documents to be both *relevant* and *authoritative*
 - *Relevance* is being modeled by cosine scores
 - *Quality* is typically a query-independent property of a document
- Examples of quality signals
 - Wikipedia among websites
 - Articles in certain newspapers
 - A paper with many citations
 - Many views, retweets, favs, bookmark saves
 - PageRank score

Quantitative

A light purple box with the word 'Quantitative' inside is positioned to the right of the list. Three arrows point from the box to the items: 'A paper with many citations', 'Many views, retweets, favs, bookmark saves', and 'PageRank score'.

Net score



- Assign to each document a quality score $g(d)$ in $[0,1]$
 - E.g., PageRank
- Combine cosine relevance and quality
$$\textit{net-score}(q,d) = g(d) + \textit{cos}(q, d)$$
 - Can use some other linear combination than an equal weighting
- Now we seek the top K docs by net-score

Incorporating Additional Information: 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 document containing all query terms, e.g.,
 - Given the query *open day*:
 - For the document *open the next day*, the size of w is 4.
 - For the document *national day open house*, the size of w is 2.

Query term proximity



- Collect candidates by running one or more queries to the indexes, and then rank.

- e.g., *NUS open day*
 1. Run it as a phrase query (e.g., using a positional index)
 2. If $< K$ docs contain the phrase *NUS open day*, run the two phrase queries "*NUS open*" and "*open day*"
 3. If we still have $< K$ docs, run the vector space query *NUS open day*
 4. Rank matching docs by vector space scoring combining all information (possibly including proximity score w)

Incorporating Additional Information: Parametric and zone indexes



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 T.S. Raffles in the year 1818, with *Dutch East India Company* in the title

Fields



- **Year = 1818** is an example of a field
 - Also, **author = T.S. Raffles**, etc
 - with a **finite set** of possible values
- 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
 - find docs authored by T.S. Raffles in the year 1818... =
 - doc *must* be authored by T.S. Raffles AND in the year 1818.

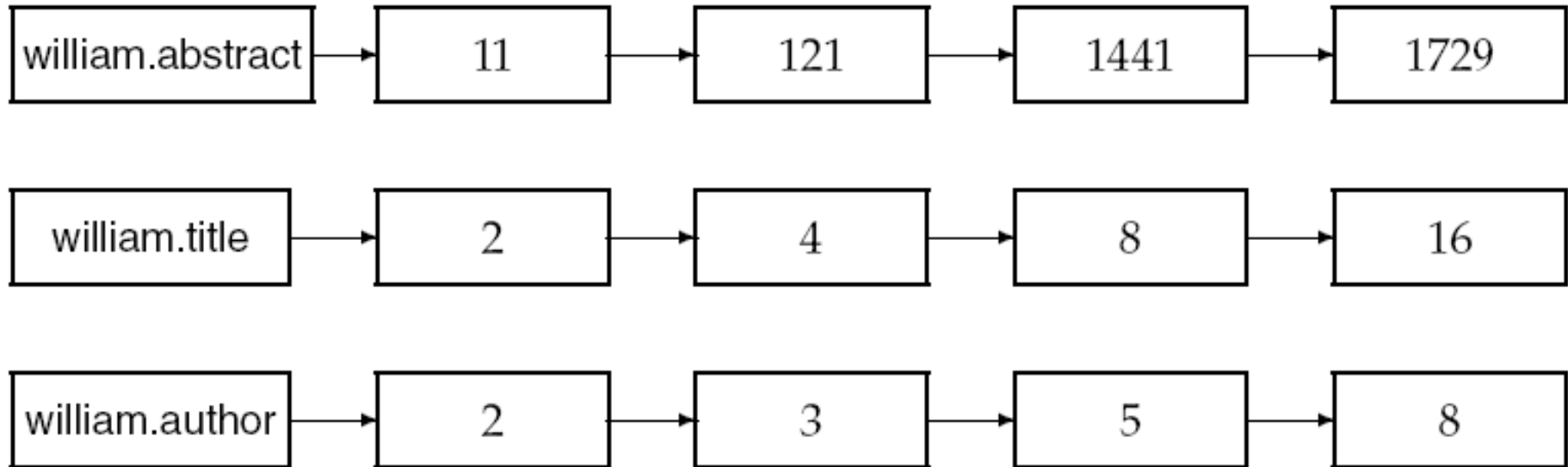
Zone



- 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 Dutch East India Company in the title

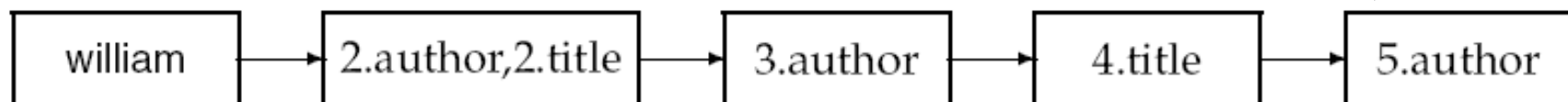
Two methods for zone indexing

Alternative 1:

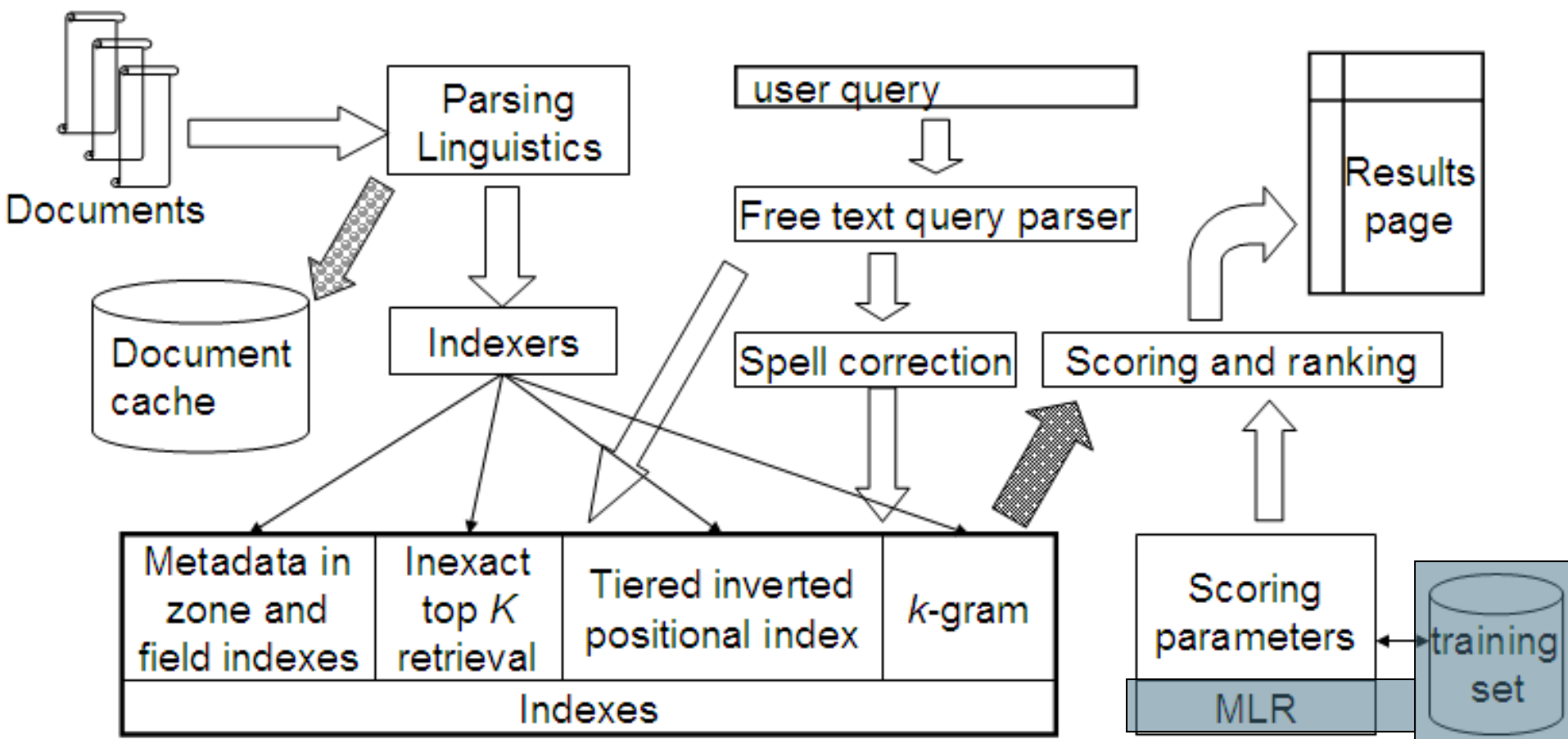


Encode zones in dictionary vs. postings.

Alternative 2:



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 additional information

Resources for today

- IIR 7, 6.1