

Introduction to
Information Retrieval

Lecture 7: Scoring and results assembly

Recap: tf-idf weighting

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

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

- 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

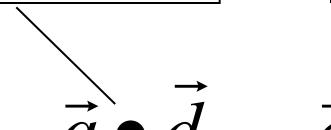
Recap: Queries as vectors

- 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)

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

Dot product Unit vectors



$\cos(\vec{q}, \vec{d})$ 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

Question: Why don't we just use the query processing methods for Boolean queries?

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

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 $\boxed{wf_{t,d}}$ 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.

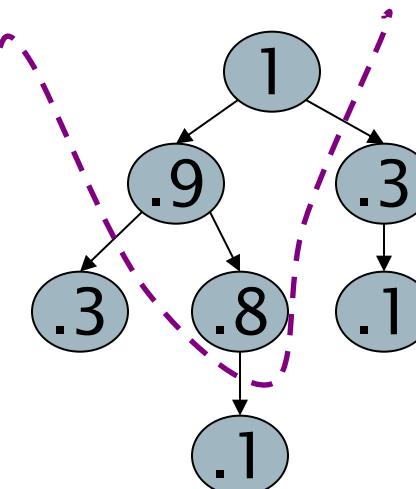
Computing the K largest cosines: selection vs. sorting

- 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 n of docs with nonzero cosines
 - We seek the K best of these n

http://en.wikipedia.org/wiki/Binary_heap

Use heap for selecting top $K / 1$

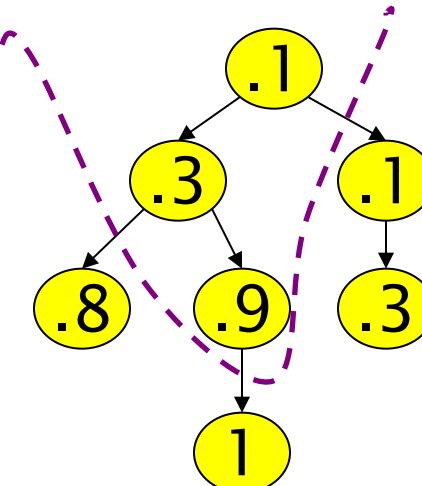
- Max-heap:
 - Binary tree in which each node's value > the values of children
- Takes $2n$ operations to construct, then each of K “winners” read off in $2\log n$ steps
- Total time is $O(n + K * \log(n))$; space complexity is $O(n)$
- For $n=1M$, $K=100$, this is about 10% of the cost of sorting.



http://en.wikipedia.org/wiki/Binary_heap

Use heap for selecting top $K / 2$

- What about using a min-heap?
- Use the min-heap to maintain the top k scores so far.
- For each new score, s , scanned:
 - H.push (s)
 - H.pop()
- Total time is $O(n * \log(k) + k * \log(k))$; space complexity is $O(k)$



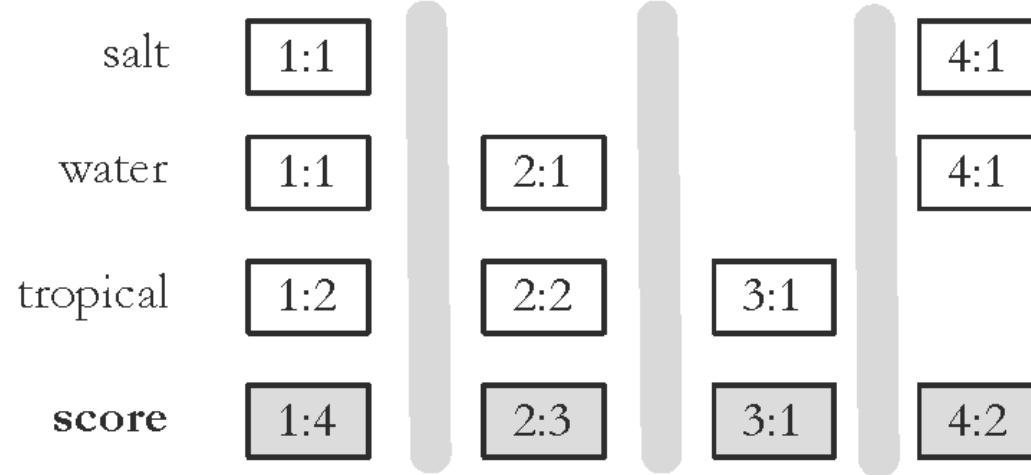
Quick Select

- QuickSelect is similar to QuickSort to find the top-K elements from an array
 - Takes $O(n)$ time (in expectation)
- Sorting the top-K items takes $O(K * \log(K))$ time
- Total time is $O(n + K * \log(K))$

Query Processing

- Document-at-a-time
 - Calculates complete scores for documents by processing all term lists, one document at a time
- Term-at-a-time
 - Accumulates scores for documents by processing term lists one at a time
- Both approaches have optimization techniques that significantly reduce time required to generate scores
 - Distinguish between safe and heuristic optimizations

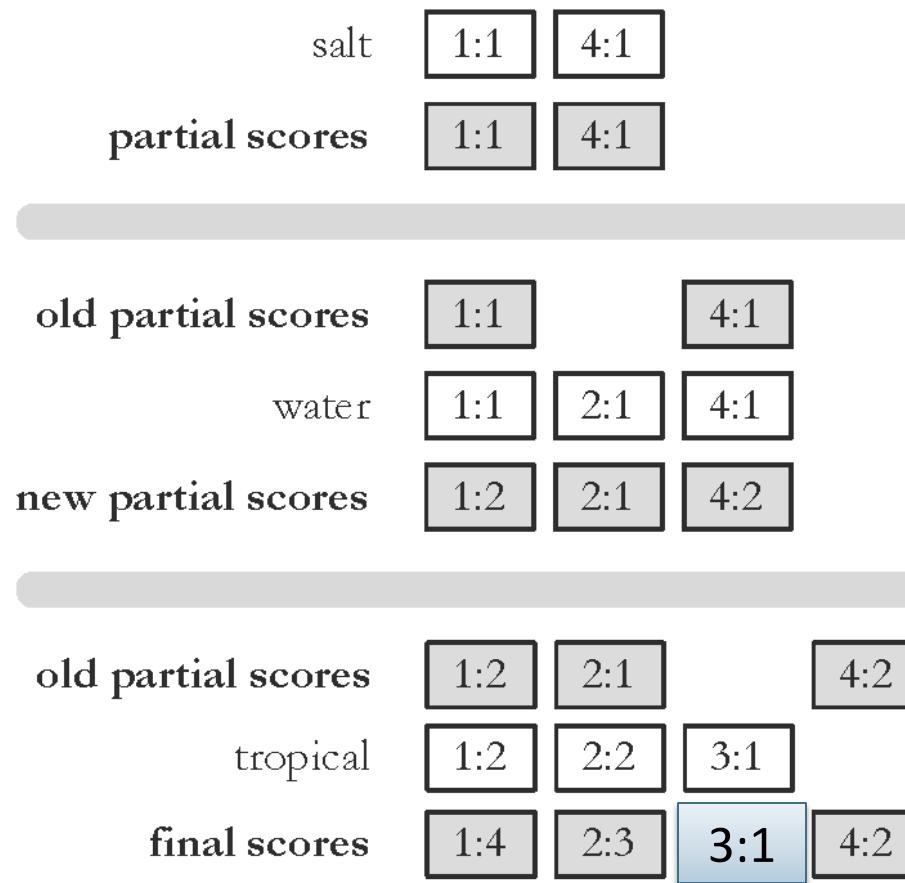
Document-At-A-Time



Document-At-A-Time

```
procedure DOCUMENTATATIMERETRIEVAL( $Q, I, f, g, k$ )
     $L \leftarrow \text{Array}()$ 
     $R \leftarrow \text{PriorityQueue}(k)$ 
    for all terms  $w_i$  in  $Q$  do
         $l_i \leftarrow \text{InvertedList}(w_i, I)$ 
         $L.\text{add}( l_i )$ 
    end for
    for all documents  $d \in I$  do
        for all inverted lists  $l_i$  in  $L$  do
            if  $l_i$  points to  $d$  then
                 $s_D \leftarrow s_D + g_i(Q)f_i(l_i)$            ▷ Update the document score
                 $l_i.\text{movePastDocument}( d )$ 
            end if
        end for
         $R.\text{add}( s_D, D )$ 
    end for
    return the top  $k$  results from  $R$ 
end procedure
```

Term-At-A-Time



Term-At-A-Time

```
procedure TERMATATIMERETRIEVAL( $Q, I, f, g, k$ )
```

$A \leftarrow \text{HashTable}()$

// accumulators

$L \leftarrow \text{Array}()$

$R \leftarrow \text{PriorityQueue}(k)$

for all terms w_i in Q **do**

$l_i \leftarrow \text{InvertedList}(w_i, I)$

$L.\text{add}(l_i)$

end for

for all lists $l_i \in L$ **do**

while l_i is not finished **do**

$d \leftarrow l_i.\text{getCurrentDocument}()$

// A_d contains partial score

$A_d \leftarrow A_d + g_i(Q)f(l_i)$

$l_i.\text{moveToNextDocument}()$

end while

end for

for all accumulators A_d in A **do**

$s_D \leftarrow A_d$ ▷ Accumulator contains the document score

$R.\text{add}(s_D, D)$

end for

return the top k results from R

end procedure

Optimization Techniques

- Term-at-a-time uses more memory for accumulators, but accesses disk more efficiently
- Two classes of optimization
 - Read less data from inverted lists
 - e.g., skip lists
 - better for simple feature functions
 - Calculate scores for fewer documents
 - e.g., conjunctive processing
 - better for complex feature functions

Conjunctive Processing

- Requires the result document containing all the query terms (i.e., conjunctive Boolean queries)
 - More efficient
 - Can also be more effective for short queries
 - Default for many search engines
- Can be combined with both DAAT and TAAT

```

1: procedure TERMATATIMERETRIEVAL( $Q, I, f, g, k$ )
2:    $A \leftarrow \text{HashTable}()$ 
3:    $L \leftarrow \text{Array}()$ 
4:    $R \leftarrow \text{PriorityQueue}(k)$ 
5:   for all terms  $w_i$  in  $Q$  do
6:      $l_i \leftarrow \text{InvertedList}(w_i, I)$ 
7:      $L.\text{add}( l_i )$ 
8:   end for
9:   for all lists  $l_i \in L$  do
10:    while  $l_i$  is not finished do
11:      if  $i = 0$  then
12:         $d \leftarrow l_i.\text{getCurrentDocument}()$ 
13:         $A_d \leftarrow A_d + g_i(Q)f(l_i)$ 
14:      else
15:         $d \leftarrow l_i.\text{getCurrentDocument}()$ 
16:         $d \leftarrow A.\text{getNextDocumentAfter}(d)$ 
17:         $l_i.\text{skipForwardTo}(d)$ 
18:        if  $l_i.\text{getCurrentDocument}() = d$  then
19:           $A_d \leftarrow A_d + g_i(Q)f(l_i)$ 
20:        else
21:           $A.\text{remove}(d)$ 
22:        end if
23:      end if
24:    end while
25:  end for
26:  for all accumulators  $A_d$  in  $A$  do
27:     $s_D \leftarrow A_d$             $\triangleright$  Accumulator contains the document score
28:     $R.\text{add}( s_D, D )$ 
29:  end for
30:  return the top  $k$  results from  $R$ 
31: end procedure

```

Conjunctive Term-at-a-Time

```

1: procedure DOCUMENTATATIMERETRIEVAL( $Q, I, f, g, k$ )
2:    $L \leftarrow \text{Array}()$ 
3:    $R \leftarrow \text{PriorityQueue}(k)$ 
4:   for all terms  $w_i$  in  $Q$  do
5:      $l_i \leftarrow \text{InvertedList}(w_i, I)$ 
6:      $L.\text{add}( l_i )$ 
7:   end for
8:   while all lists in  $L$  are not finished do
9:     for all inverted lists  $l_i$  in  $L$  do
10:      if  $l_i.\text{getCurrentDocument}() > d$  then
11:         $d \leftarrow l_i.\text{getCurrentDocument}()$ 
12:      end if
13:    end for
14:    for all inverted lists  $l_i$  in  $L$  do  $l_i.\text{skipForwardToDocument}(d)$ 
15:      if  $l_i$  points to  $d$  then
16:         $s_d \leftarrow s_d + g_i(Q)f_i(l_i)$             $\triangleright$  Update the document score
17:         $l_i.\text{movePastDocument}( d )$ 
18:      else
19:        break
20:      end if
21:    end for
22:     $R.\text{add}( s_d, d )$ 
23:  end while
24:  return the top  $k$  results from  $R$ 
25: end procedure

```

Conjunctive Document-at-a-Time

Threshold Methods

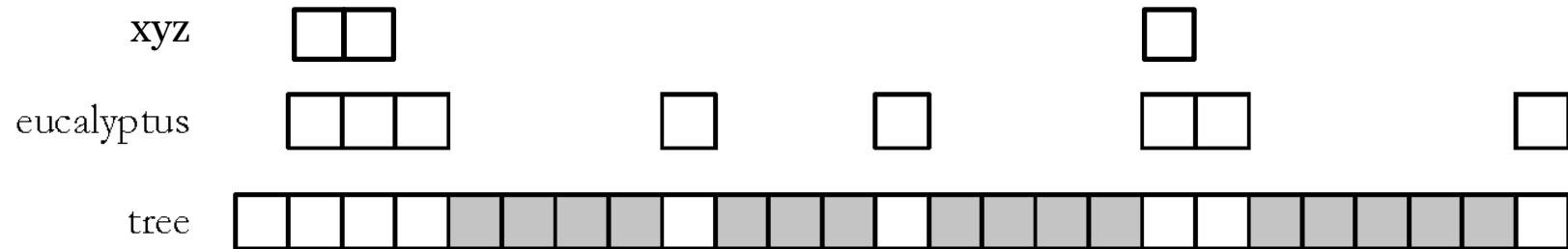
- Threshold methods use number of top-ranked documents needed (k) to optimize query processing
 - for most applications, k is small
- For any query, there is a *minimum score* that each document needs to reach before it can be shown to the user
 - score of the k th-highest scoring document
 - gives *threshold* τ
 - optimization methods estimate τ' to ignore documents

Threshold Methods

- For document-at-a-time processing, use score of lowest-ranked document so far for τ'
 - for term-at-a-time, have to use k_{th} -largest score in the accumulator table
- *MaxScore* method compares the maximum score that remaining documents could have to τ'
 - *safe* optimization in that ranking will be the same without optimization

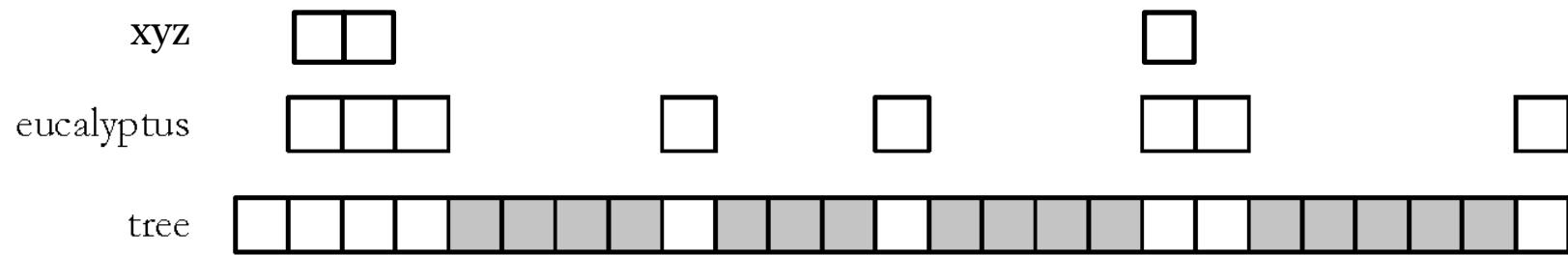
Better than the example in the textbook. See my Note 2 too.

MaxScore Example



- Compute max term scores, μ_t , of each list and sort them in decreasing order (fixed during query processing)
- Assume $k = 3$, τ' is lowest score of the **current** top- k documents
- If $\mu_{tree} < \tau'$ → any doc that scores higher than τ' must contain **at least one of** the first two keywords (aka **required term set**)
 - Use postings lists of required term set to “drive” the query processing
 - Will only check **some of** the white postings in the list of “tree” to compute score → at least all gray postings are skipped.

MaxScore



Other Approaches

- Early termination of query processing
 - ignore high-frequency word lists in term-at-a-time
 - ignore documents at end of lists in doc-at-a-time
 - *unsafe* optimization
- List ordering
 - order inverted lists by quality metric (e.g., PageRank) or by partial score
 - makes unsafe (and fast) optimizations more likely to produce good documents

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

- Justifications
 - User has a task and a query formulation
 - Cosine matches docs to query
 - Thus cosine is anyway a proxy for user happiness
- Approximate query processing
 - 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 of Fig 7.1 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

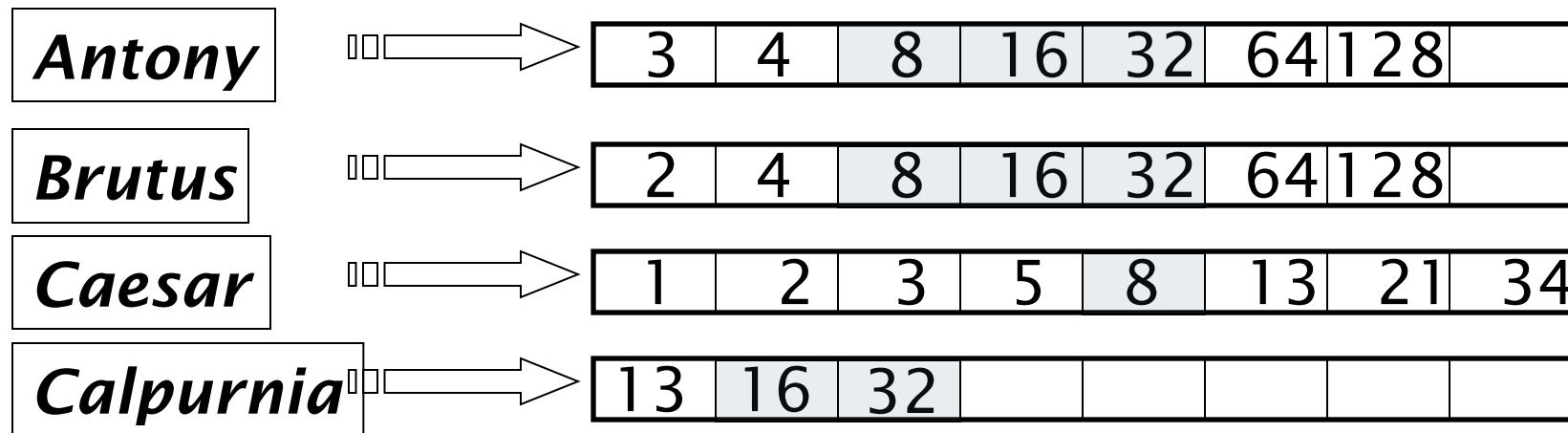
- For a query such as *catcher in the rye*
- Only accumulate scores from *catcher* and *rye*
- Intuition: *in* and *the* contribute little to the scores and so don't alter rank-ordering much
- Benefit:
 - Postings of low-idf terms have many docs → these (many) docs get eliminated from set *A* of contenders

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

Can generalize to WAND method (safe)

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 $A = \bigcup_{t \in Q} \text{ChampionList}(t)$
 - Pick the K top-scoring docs from amongst these

Inspired by “fancy lists” of Google:

<http://infolab.stanford.edu/~backrub/google.html>

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)
-
- The diagram consists of five red text items listed under the 'Examples of authority signals' heading. From each of these items, a thin black arrow points upwards and to the right towards a light gray rectangular box with a black border. Inside the box, the word 'Quantitative' is written in a black sans-serif font. The arrows originate from the bottom of each red text item and converge towards the center of the box.

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
- $\text{net-score}(q,d) = g(d) + \cosine(q,d)$
 - 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 all the *high* lists first
 - If we get more than K docs, select the top K and stop
 - Only union the high lists
 - 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

Why $N^{1/2}$ leaders?

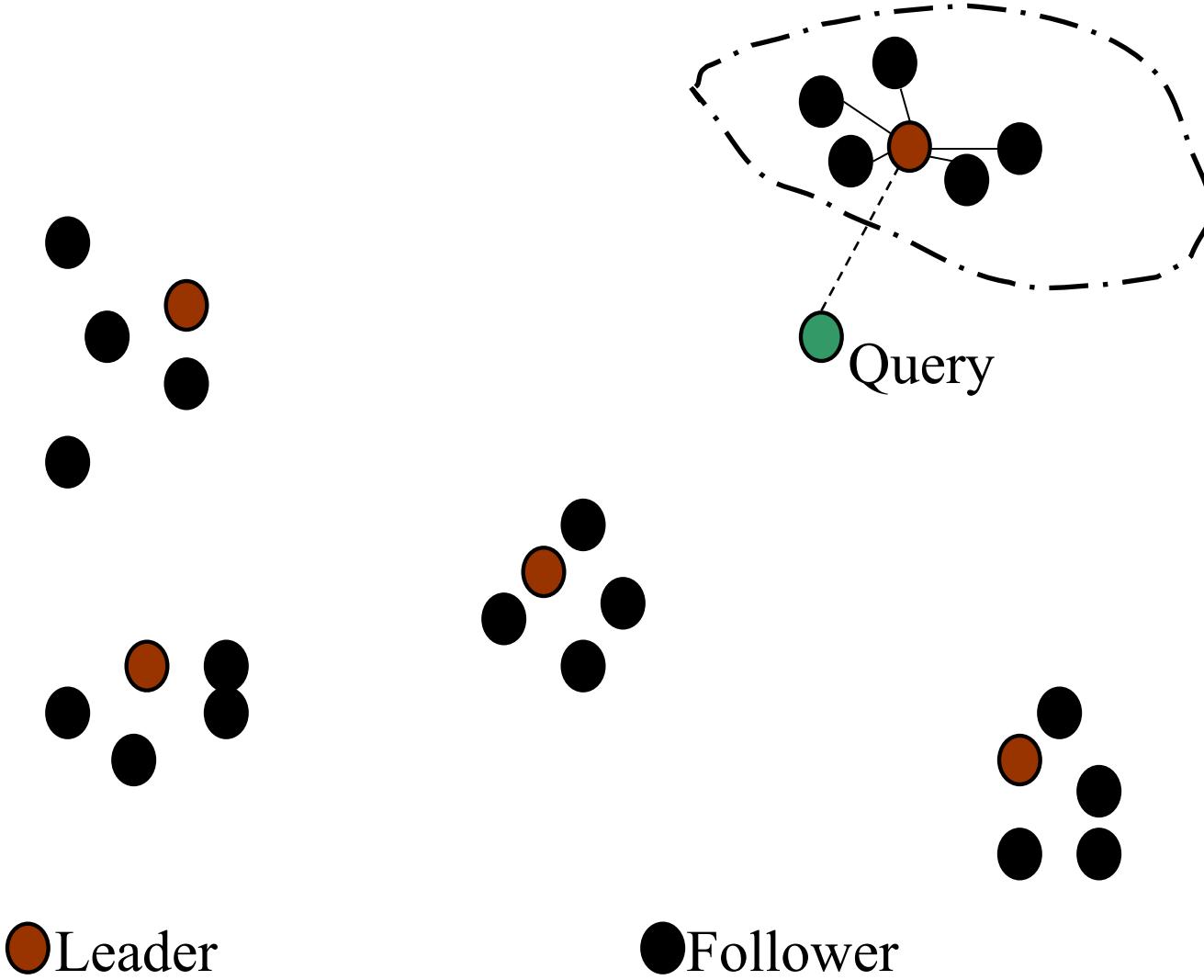
Cluster pruning: preprocessing

- 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 $b1=3$ (say) nearest leaders.
- From query, find $b2=4$ (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?
 - Hint: write down the algorithm, model its cost, and minimize the cost.
- What is the effect of the constants b_1, b_2 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

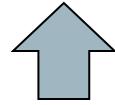
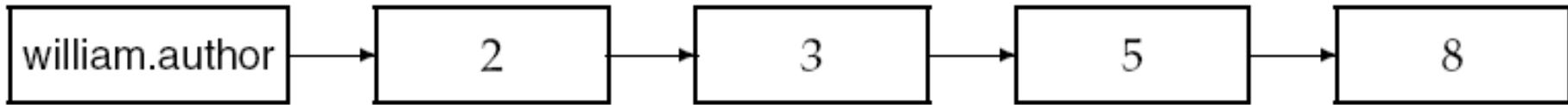
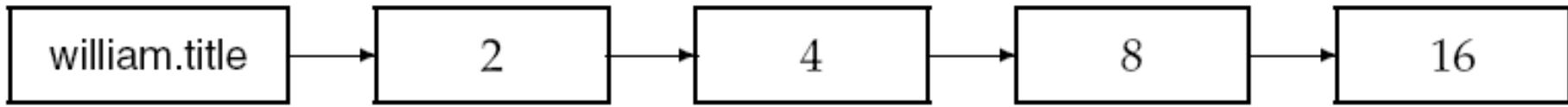
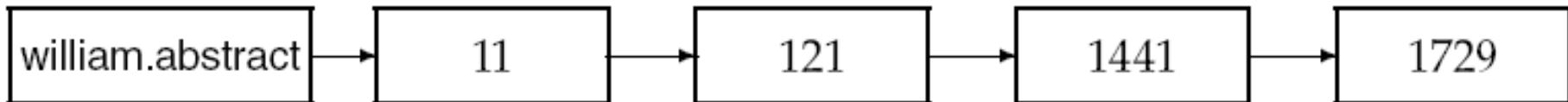
Fields

- 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)

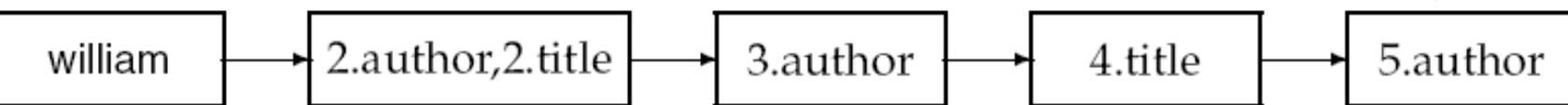
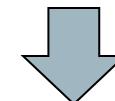
Zone

- 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



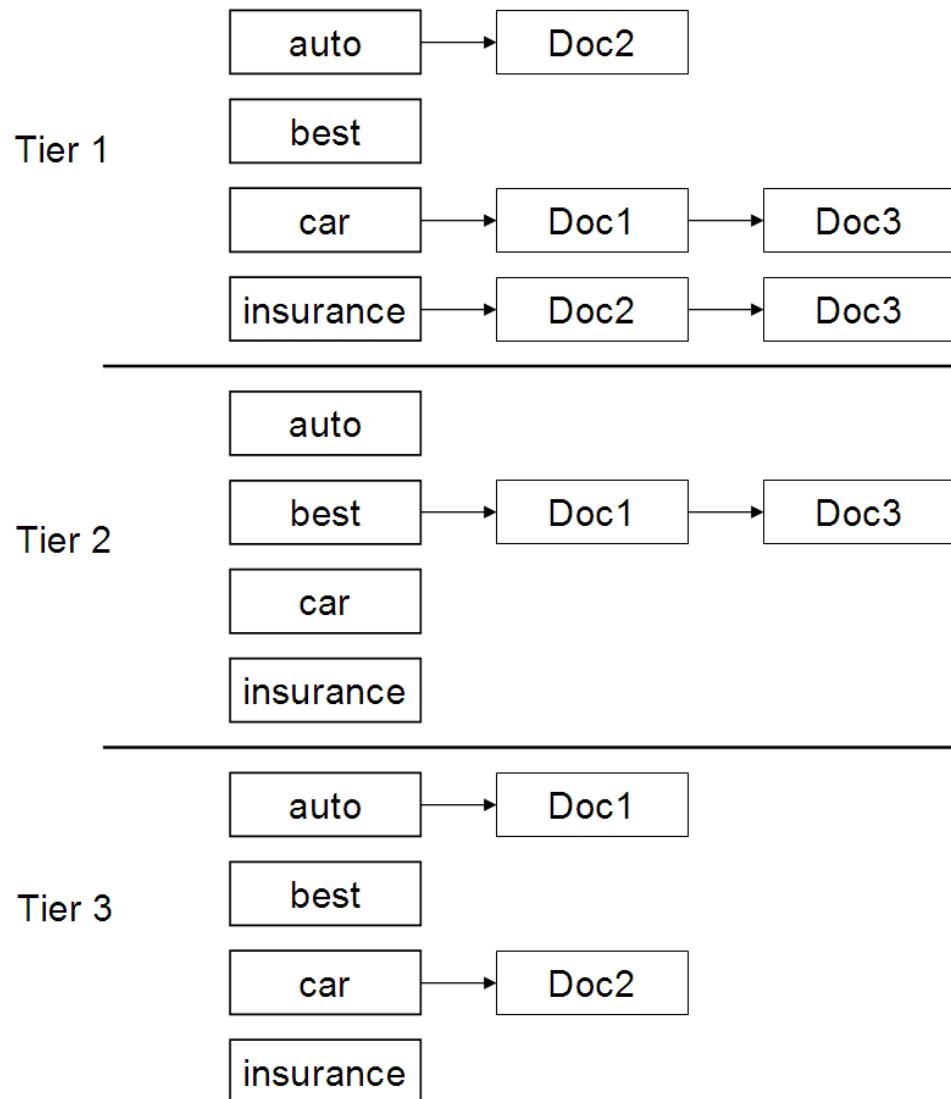
Encode zones in dictionary vs. postings.



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?

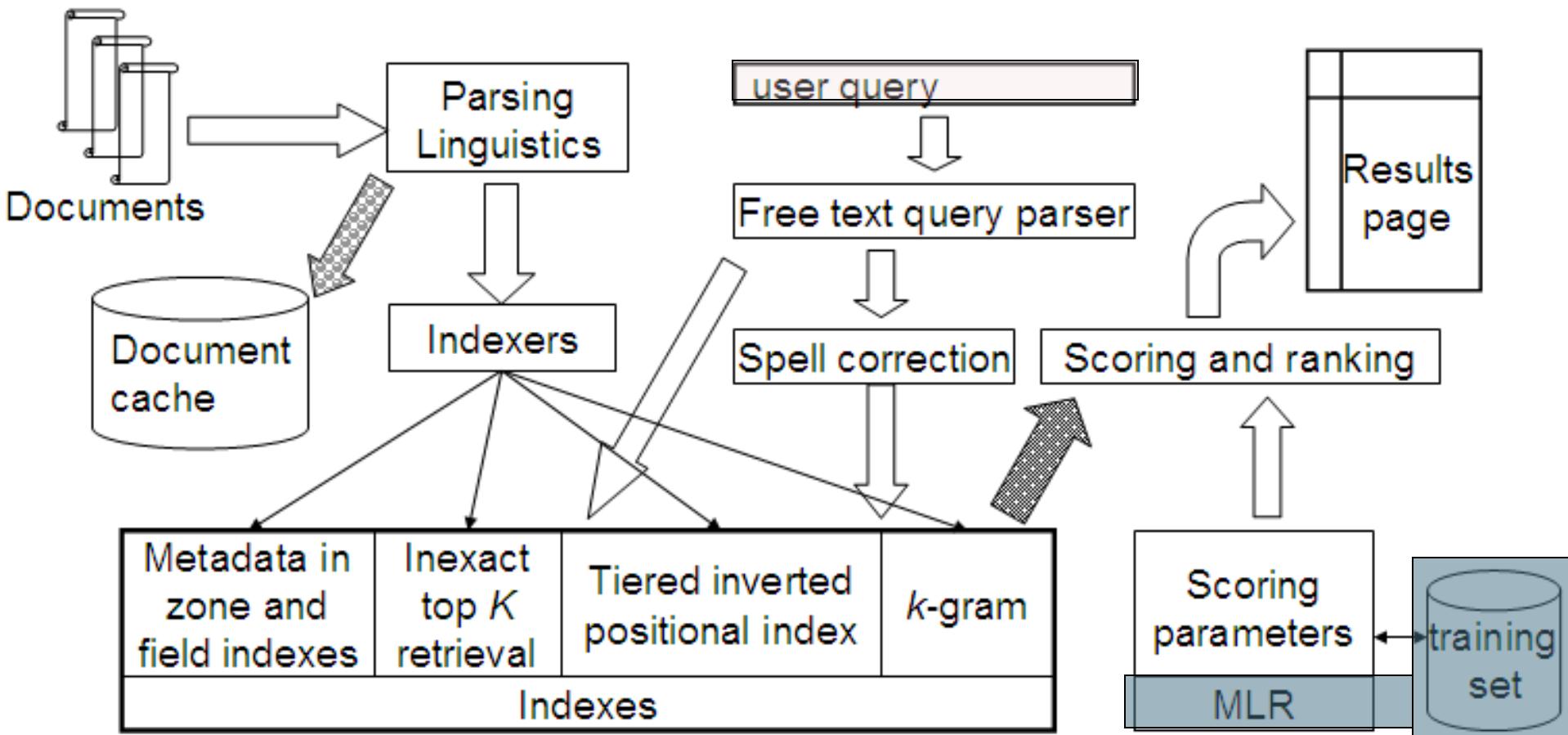
Query parsers

- 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 later lecture

Putting it all together



Resources

- IIR 7, 6.1