Introduction to Informatics

Lecture 7: Scoring and Assembly

Recap: tf-idf weigh

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

$$w_{t,d} = (1 \qquad N/\mathrm{df}_t)$$

- Best known weighting sche mation retrieval

- Increases with the number of occurrences within a document
- Increases with the rarity of the term in the collection

Recap: Queries as

- 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 t
- proximity = similarity of ve

Recap: cosine(quer ment)

Dot product

Unit vectors

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

$\cos(\vec{q},\vec{d})$ 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 number of miscellaneous topic ristics

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

Term-at-a-time

Computing cosines

Efficient cosine ran

- Find the *K* docs in the collection "nearest" to the query $\Rightarrow$ *K* largest query-doc cosines.
- Efficient ran
  - Computing tly.
  - Choosing the *K* largest c s efficiently.
    - Can we do this without computing all *N* cosines?

Efficient cosine ran

- 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-dime
- But it is solvable for short q d standard indexes support this well

Special case – unw queries

- No weighting on query terms
  - Assume each query term occurs only once
- Then for rank malize query vector
  - Slight simplification of al om Lecture 6

Faster cosine: unw query

Computing the K largest cosines: selection vs. sorting

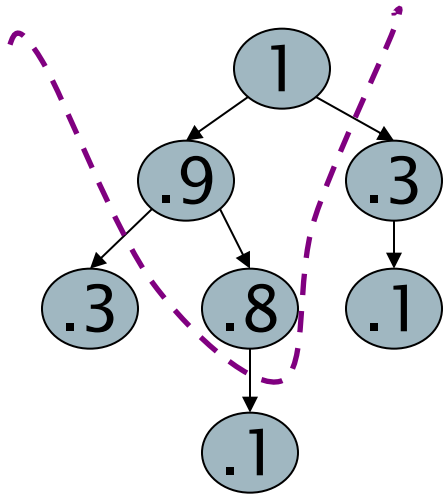
- Typically we want to retrieve the top *K* docs (in the cosine ranking for the query)
  - not to totally order all docs in the collection
- Can we pick o
- Let *n* of docs with nonzero
  - We seek the *K* best of th

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

Use heap for selec 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 2log

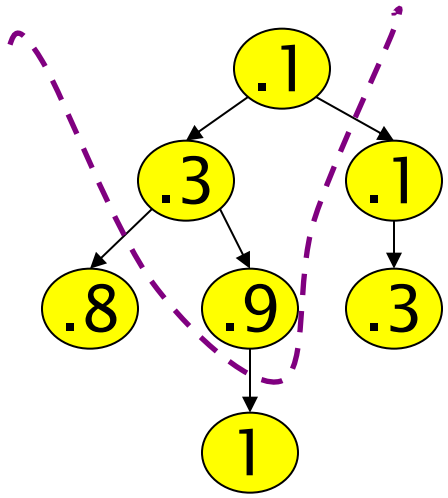
- Total time is O(n + k*log(n)) xity is O(n)
- For *n*=1M, *K*=100, this is about 1 ost of sorting.



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

Use heap for selec 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, do the following:
  - H.push (s)
  - H.pop()
- Total time is O(n*log(k) + k*log(k)); space complexity is O(k)



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

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 takes O(K*log(K)) time
- Total time is O(n + K*log(K))

[CMS09].begin

Query Processing

- Document-at-a-time
  - Calculates complete scores for documents by processing all term lists, one document at a time
- Term-at-a-tim
  - Accumulates 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 Ti

Document At A Ti

Term-At-A-Time

Term-At-A-Time


```
// accumulators
```



```
// A contains partial score
```

Optimization Tech

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

Conjunctive Proces

- Requires the result document containing all the query terms (i.e., conjunctive Boolean queries)
  - More efficient
  - Can also be eries
  - Default for
- Can be combined with both TAAT

Conjunctive Term-at-a-Time

Conjunctive Document-at-a-Time

Threshold Method

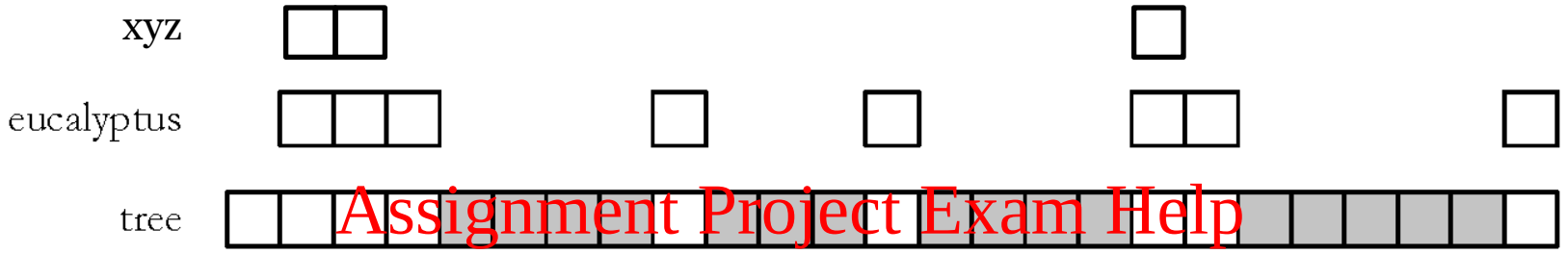
- Threshold methods use number of top-ranked documents needed (k) to optimize query processing
  - for most applications, k is small
- For any query document ne core that each can be shown to the user
  - score of the kth-highest scoring document
  - gives *threshold* τ
  - optimization methods estimate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

Threshold Method

- 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
- *MaxScore* me ximum score that remaining documents to τ'
  - *safe* optimization in that ranking will be the same without optimization

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

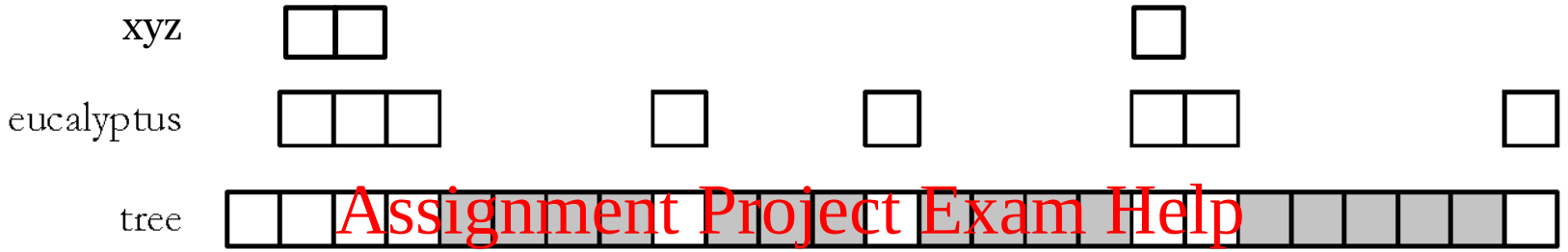
MaxScore Example



- Compute ma ch list and sort them in decr ing 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 contains *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* optim
- List ordering
  - order inverted lists by qualit ., 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

  - a doc *not* in the top *K* m to the list of *K* output docs
  - Is this such a bad thing?

Cosine similarity is a proxy

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

Generic approach

- Find a set *A* of *contenders*, with $K < |A| << N$
  - *A* does not necessarily contain the top *K*, but has many docs from among the top *K*
  - Return the t
- Think of *A* as pruning non-c
- The same approach is also her (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 conside
  - Only consider docs containin ry terms

High-idf query ter

- For a query such as *catcher in the rye*
- Only accumulate scores from *catcher* and *rye*

- Intuition: *in* a to the scores and so don't ch

- Benefit:

  - Postings of low-idf terms hav s → 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

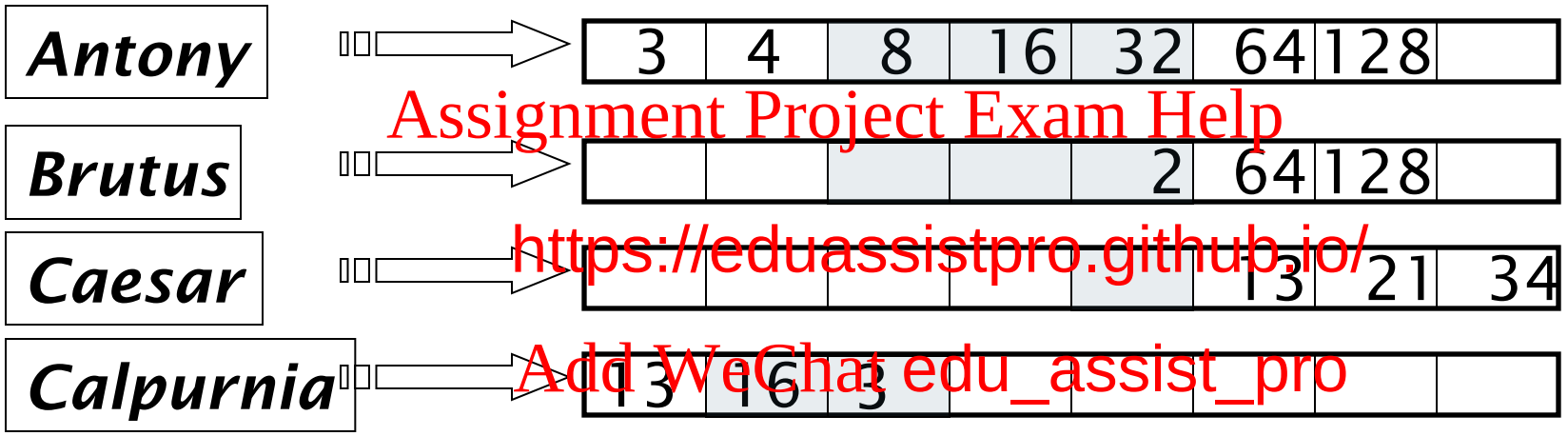
- For multi-term queries, only compute scores for docs containing several

  - Say, at least 3 out of 4

  - Imposes a "soft conjunction" 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
- Note that *r* ha x build time
  - Thus, it's possible that *r* < *K*
- At query time, only compute scores for docs in A = $\cup_{t \in Q}$ 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 inde
  - Note that the champion list to do with small docIDs

Static quality score

- 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 signa
  - Wikipedia among websites
  - Articles in certain newspapers
  - A paper with many citations
  - Many diggs, Y!buzzes or del.icio.us marks
  - (Pagerank)

Quantitative

Modeling authorit

- Assign to each document a *query-independent* quality score in [0,1] to each document *d*
  - Denote this by *g*(*d*)
- Thus, a quanti citations is scaled into [0,1]
  - Exercise: suggest a formula f

Net score

- Consider a simple total score combining cosine relevance and authority
- 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 – methods

- First idea: Order all postings by *g(d)*
- Key: this is a common ordering for all postings

- Thus, can con ry 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 sea

0 ms), this allows us to stop pos

  - Short of computing scores fo postings

Champion lists in g -ering

- 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* re 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 the *high* lists first, only traverse all the *high* lists f

  - If we get more than *K* docs, s p *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 post

- 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 p order!
- How do we compute score o 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
  - One from the postings of ea m
- Compute only the scores fo is union

2. idf-ordered term

- 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 https://eduassistpro.github.io/ in each query term
  - Stop if doc scores relatively
- Can apply to cosine or some other net scores

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

Cluster pruning: pr sing

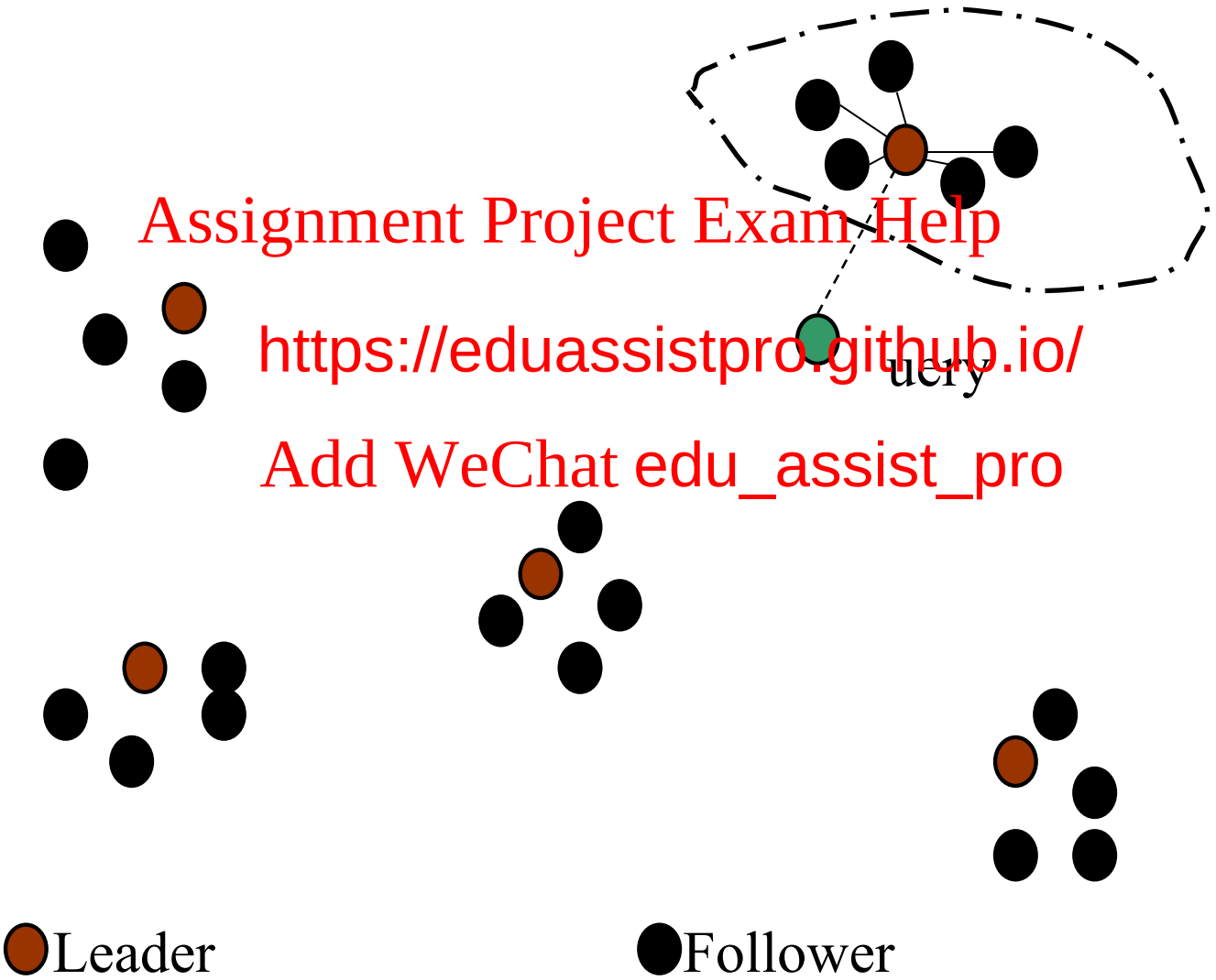
- Pick √N *docs* at random: call these *leaders*
- For every other doc, pre-compute nearest leader

  - Docs atta s *ollowers;*
  - Likely: each leader ha lowers.

Cluster pruning: q cessing

- Process a query as follows:
  - Given query *Q*, find its nearest *leader L.*
  - Seek *K* ne ong *L's* followers.

Visualization



Why use random s

- 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/follow tion.

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 d [illegible] its cost, and minimize the

- What is the effect of the co [illegible] $, b2$ 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 zo es

- 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

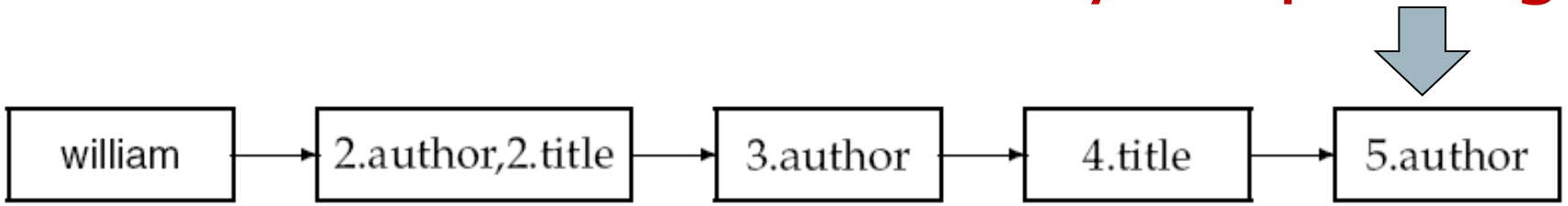
- We sometimes wish to search by these metadata
  - E.g., find docs authored by William Shakespeare in the year 1601, containing *alas poor Yorick*
- Year = 1601 is an exa
- Also, author l re, etc
- Field or parametric index: p r each field value
  - Sometimes build range trees (e.g., for dates)
- Field 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 to permit querying
- E.g., "find docs with *merchant* in the title zone and matching the query *gentle rain*"

Example zone inde

Encode zones in dictionary vs. postings.



Tiered indexes

- Break postings up into a hierarchy of lists
  - Most important
  - …
  - Least import

- Can be done sure
- Inverted index thus broken rs of decreasing importance

- At query time use top tier unless it fails to yield *K* docs
  - If so drop to lower tiers

Example tiered ind

Query term proxim

- 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 proximi
- Let *w* be the smallest wind c 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 quer

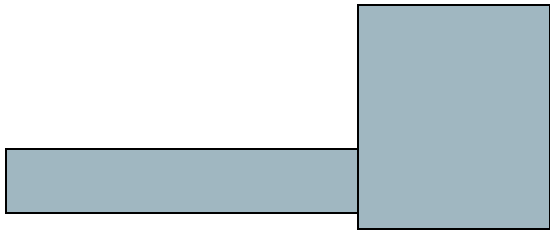
  - If <*K* docs co *erest ra* , run the two phrase queries *rising int* *erest rates*

  - 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 applica
- Increasingly common: mac
  - See later lecture

Putting it all toget



Resources

- IIR 7, 6.1