Content

- Vector space scoring
- Speeding up vector space ranking
- Putting together a complete search system

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 (approximate) cosine efficiently
  - Choosing the K largest cosine values efficiently
    - Can we do this without computing all N cosines?
    - Can we find approximate solutions?

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

- Assume each query term occurs only once
- idf scores are considered in the document terms
- Then for ranking, don't need to consider the query vector weights
  - Slight simplification of algorithm from Chapter 6 IIR

Faster cosine: unweighted query

They are all 1

```
FastCosineScore(q)
    float Scores[N] = 0
    for each d
    do Initialize Length[d] to the length of doc d
    for each query term t
    do ~~calculate w_{t,q} and~~ fetch postings list for t
        for each pair(d, tf_{t,d}) in postings list
        do add wf_{t,d} to Scores[d]
    Read the array Length[d]
    for each d
    do Divide Scores[d] by Length[d]
    return Top K components of Scores[]
```

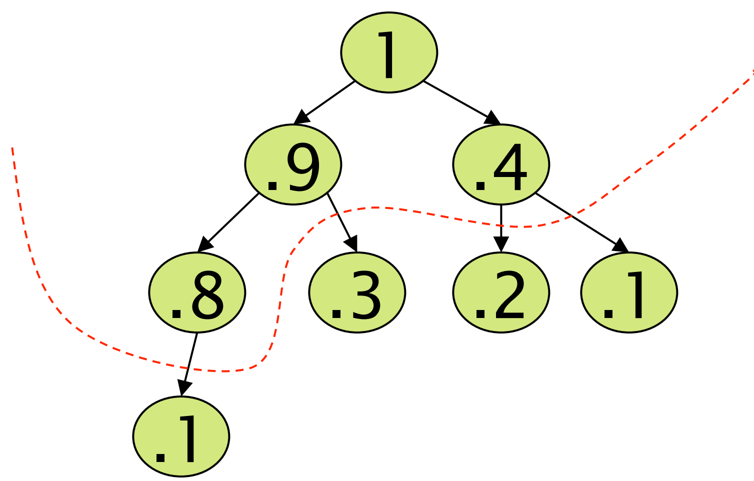
Figure 7.1 A faster algorithm for vector space scores.

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 *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 (assume that there are J nodes)
- Takes $2J$ operations to construct, then each of K "winners" read off in $2\log J$ steps.
- For J=1M, K=100, this is about 5% of the cost of sorting ($2J\log J$).



Cosine similarity is only a proxy

- User has a task and an will formulate a query
- The system computes 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
- *Remember, our final goal is to build effective and efficient systems, not to compute correctly our formulas.*

Generic approach

- Find a set *A* of ***contenders***, with $K < |A| << N$ (*N* is the total number of docs)
  - *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 <u>pruning</u> non-contenders
- The same approach is also used for other (non-cosine) scoring functions (remember spelling correction and the Levenshtein distance)
- 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 – obvious !
- Take this idea further:
  - Only consider **high-idf <u>query</u> terms**
  - Only consider **docs** containing **many <u>query</u> terms.**

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

for q, d length-normalized

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 <u>don't alter rank-ordering much</u>
  - *They are present in most of the documents and their idf weight is low*
- Benefit:
  - Postings of low-idf terms have many docs – then these docs (many) 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

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.

Champion lists (documents)

- 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? (i.e., eliminating query terms with low idf – compute the score only if a certain number of query terms appear in the document)
- 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

Modeling authority

- Assign to **each document** *d* a *query-independent* quality score in [0,1]
  - 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
- 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
  - Shortcut 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)* + tf-idf$_{td}$
- Order the postings by g(d)
- Seek top-*K* results from only the docs in these champion lists.

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 $wf_{t,d}$
  - *Hence, while considering the postings and computing the scores for documents not yet considered we have a bound on the final score for these documents*
- 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
  - Documents 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 (*if there are many*)
  - High idf terms likely to contribute most to score
- As we update score contribution from each query term
  - Stop if doc scores relatively unchanged
  - This will happen for **popular** query terms (low idf)
- Can apply to cosine or some other net scores.

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.

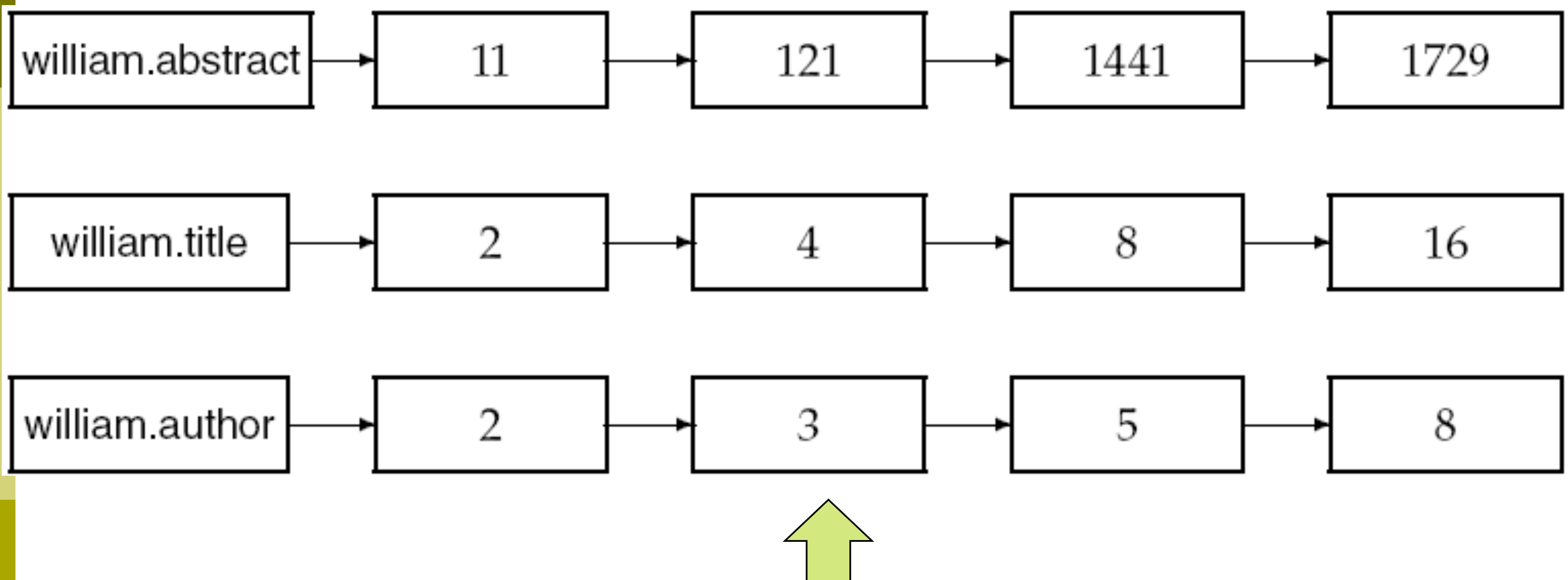
Fields

- 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 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)

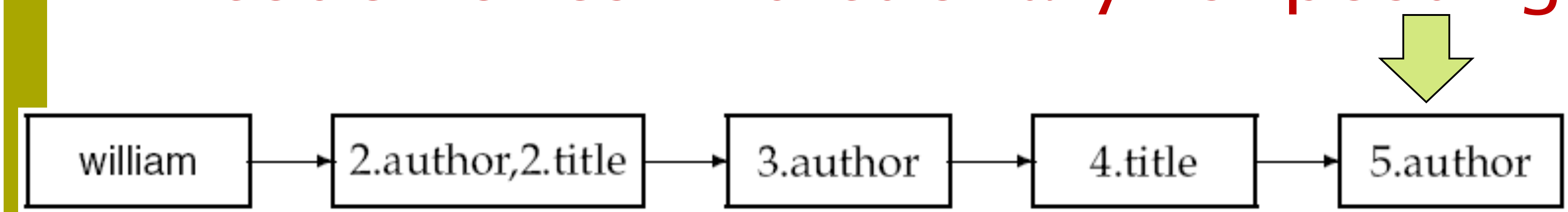
Zone

- 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.



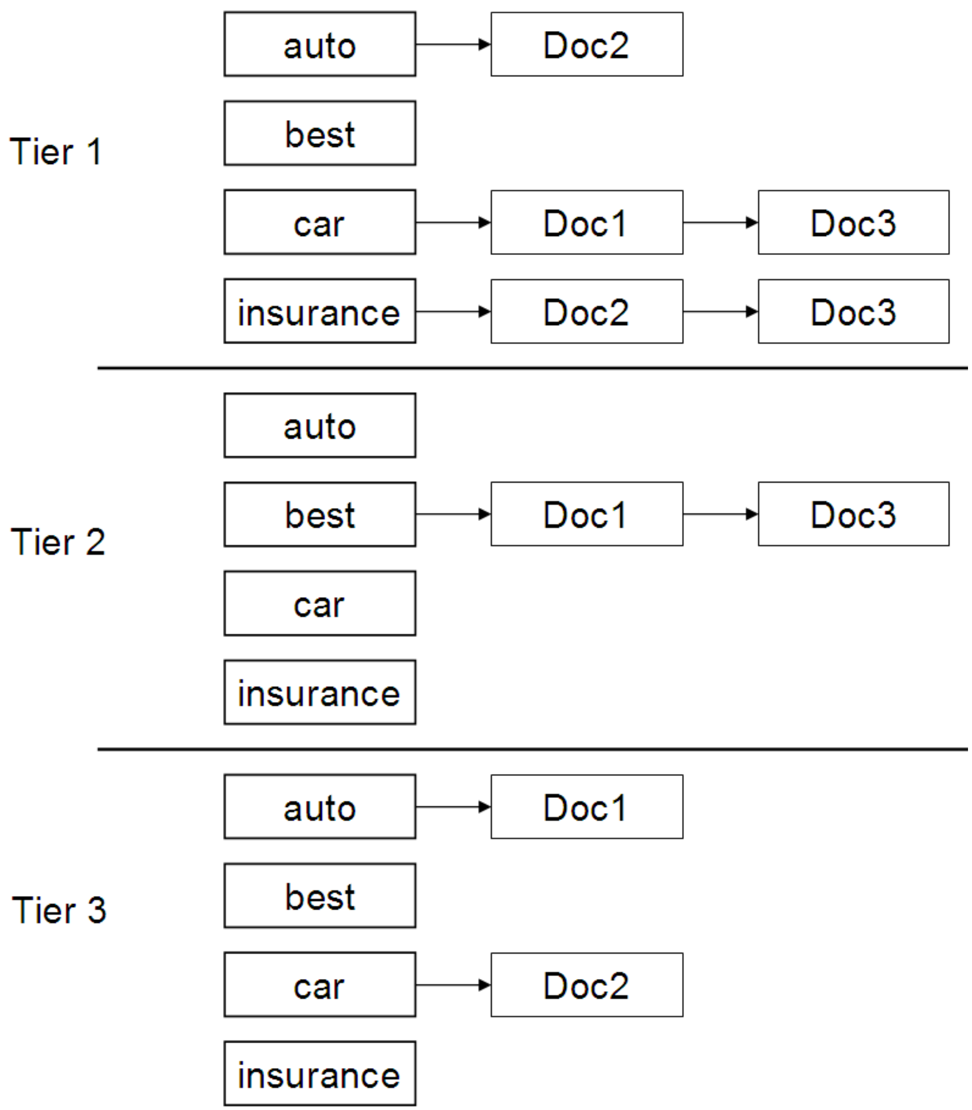
High and low lists

- 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** (*not documents*) 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

- **One** 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 a forthcoming lecture.

Putting it all together