Scoring & result assembly

CE-324: Modern Information Retrieval

Sharif University of Technology

M. Soleymani

Fall 2015

Most slides have been adapted from: Profs. Manning, Nayak & Raghavan (CS-276, Stanford)

Outline

- Speeding up vector space ranking
- Putting together a complete search system
  - Will require learning about a number of miscellaneous topics and heuristics

Computing cosine scores

```
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[]
```

Term frequencies in the inverted index

- In each posting, store $tf_{t,d}$ in addition to docID
  - As an integer frequency, not as a (log-)weighted real number
    - because real numbers are difficult to compress.
    - overall, additional space requirements are small: a byte per posting or less

Efficient ranking

- Usually we don't need a complete ranking.
  - We just need the top k for a small k (e.g., k = 100).
- Find K docs in the collection "nearest" to query
  - $\Rightarrow K$ largest query-doc scores.
- Efficient ranking:
  - Computing a single score efficiently.
  - Choosing the K largest scores 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

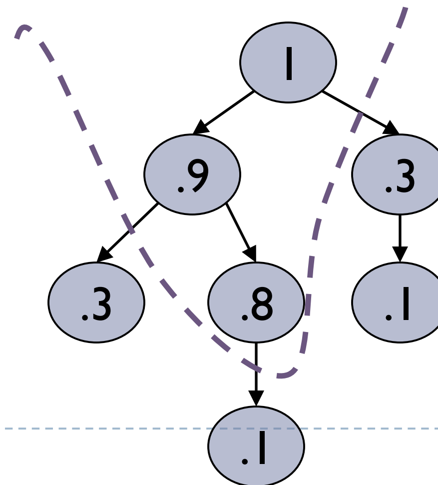
Computing the K largest cosines: selection vs. sorting

- Retrieve the top K docs
  - 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

- Construction: $2J$ operations
- K "winners": $2K \log J$ operations

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



Cosine similarity is only a proxy

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

More efficient computation of top k: Heuristics

- Idea 1: Reorder postings lists
  - Instead of ordering according to docID, order according to some measure of "expected relevance", "authority", etc.

- Idea 2: Heuristics to prune the search space
  - Not guaranteed to be correct but fails rarely.
  - In practice, close to constant time.

Generic idea of inexact top k search

- 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
- Same approach is also used for other scoring functions
  - Will look at several schemes following this approach

Ideas for more efficient computation of top k

- Index elimination
- Champion lists
- Static quality scores
- Impact ordering
- Cluster pruning

Index elimination for cosine computation

- Basic algorithm: considers docs containing at least one query term

- Extend this basic algorithm to:
  - Only consider docs containing many (or all) query terms
  - Only consider high-idf query terms

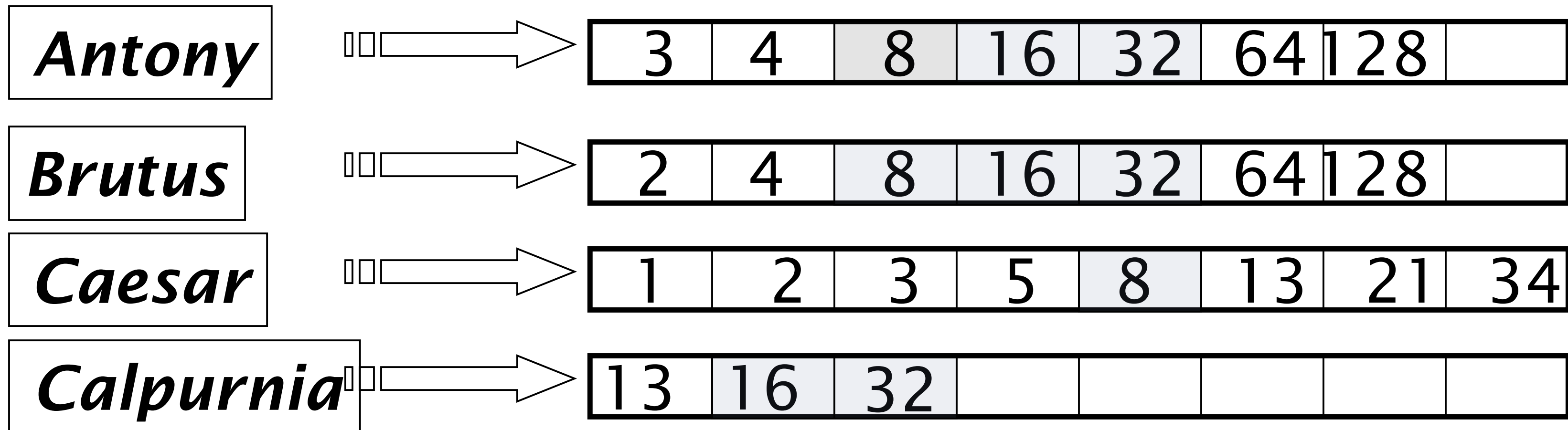
Docs containing many query terms

- 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)
  - May find fewer than k candidates

- Easy to implement in postings traversal

3 of 4 query terms



Scores only computed for docs 8, 16 and 32.

High-idf query terms only

- Query: ***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
    - $\rightarrow$ many docs are eliminated from set *A* of contenders

Champion lists

- r docs of highest weight in the posting list of each dictionary term
  - Call this the champion list for t
    - aka fancy list or top docs for t

- At query time, only compute scores for docs in the champion list of some (or all of) query terms
  - Pick the K top-scoring docs from amongst these

- Note that r has to be chosen at index build time
  - Thus, it's possible that obtained list of docs contains less than K docs

Static quality scores

- Top-ranking docs needs to be both *relevant* and *authoritative*
  - *Relevance:* modeled by cosine scores
  - *Authority:* typically a query-independent property of a doc

- Examples of authority signals
  - Wikipedia among websites
  - Articles in certain newspapers
  - A paper with many citations
  - Pagerank

Quantitative

Modeling authority

- Assign to each doc d a ***query-independent*** quality score in [0,1] (called $g(d)$)
  - A quantity like the number of citations scaled into [0,1]

Net score

- Simple total score: combining cosine relevance and authority

$$\text{NetScore}(q,d) = g(d) + cosine(q,d)$$

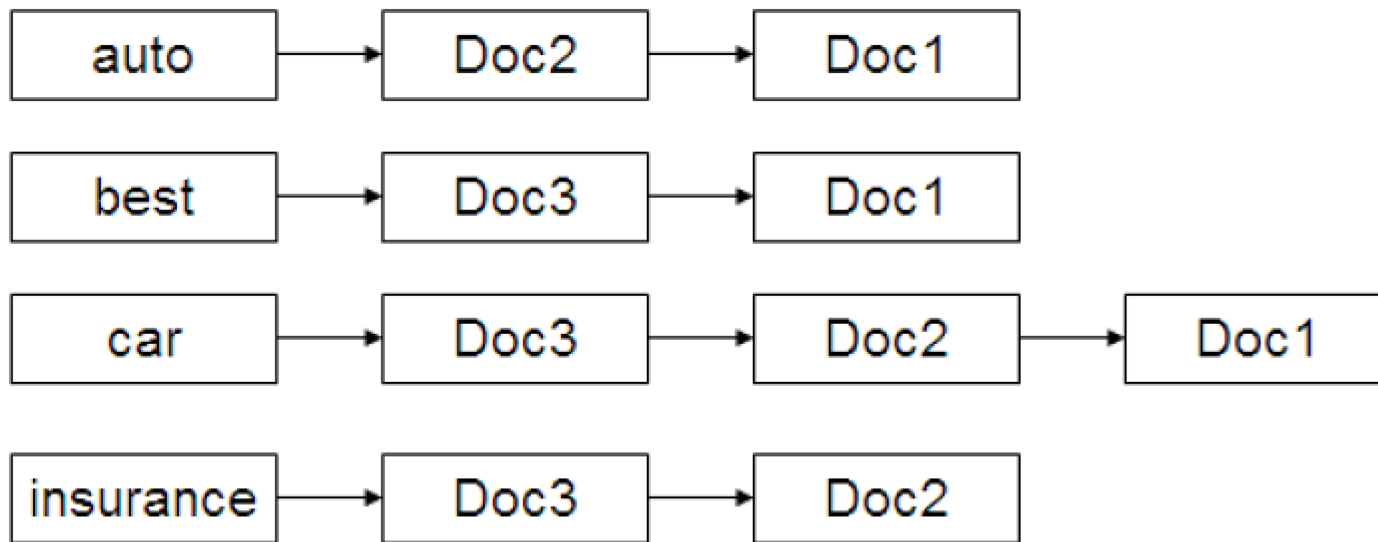
  - Can use some other linear combination
  - Indeed, any function of the two "signals" of user happiness

- 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
  - All postings are ordered by a single common ordering and the merge is then performed by a single pass through the postings
- Can concurrently traverse query terms' postings for
  - Postings intersection
  - Cosine score computation (doc-at-a-time)

Static quality-ordered index



$g(1) = 0.25$
$g(2) = 0.5$
$g(3) = 1$

Why order postings by $g(d)$?

- $g(d)$-ordering: top-scoring docs likely to appear early in postings traversal
- In time-bound applications:
  - It allows us to stop postings traversal early
    - E.g., we have to return search results in 50 ms

Global champion lists

- Can combine champion lists with $g(d)$-ordering?
- Maintain for each term a champion list of r docs with highest $g(d) + \mathrm{tf.idf}_{td}$
  - Sorted by a common order $g(d)$
- Seek top-K results from only the docs in these champion lists

High and low lists

- For each term, two postings lists *high* and *low*
  - High: like the champion list
  - Low: all other docs containing t
- Only traverse *high* lists first
  - If we get more than K docs, select top K and stop
  - Else proceed to get docs from *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 sort each postings list according to weight $wf_{t,d}$
  - Simplest case: normalized tf-idf weight
  - Docs in the top k are likely to occur early in these ordered lists.
    - ⇒ Early termination while processing postings lists is unlikely to change the top k.

- How do we compute scores in order to pick off inexact top K?
  - Early termination
  - idf-ordered terms

Impact-ordered postings

- Now: not all postings in a common order!
  - We no longer have a consistent ordering of docs in postings lists.
  - no longer can employ document-at-a-time processing

- Term-at-a-time processing
  - Create an accumulator for each docID you encounter
  - completely process the postings list of the first query term, then completely process the postings list of the second query term and so forth

Term-at-a-time processing

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 $Scores[]$

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

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 we can stop when doc scores are relatively unchanged
    - As we get to query terms with lower idf, we can determine whether to proceed based on the changes in doc scores.
      - If these changes are minimal, we may omit accumulation from the remaining query terms
      - or alternatively process shorter prefixes of their postings lists.

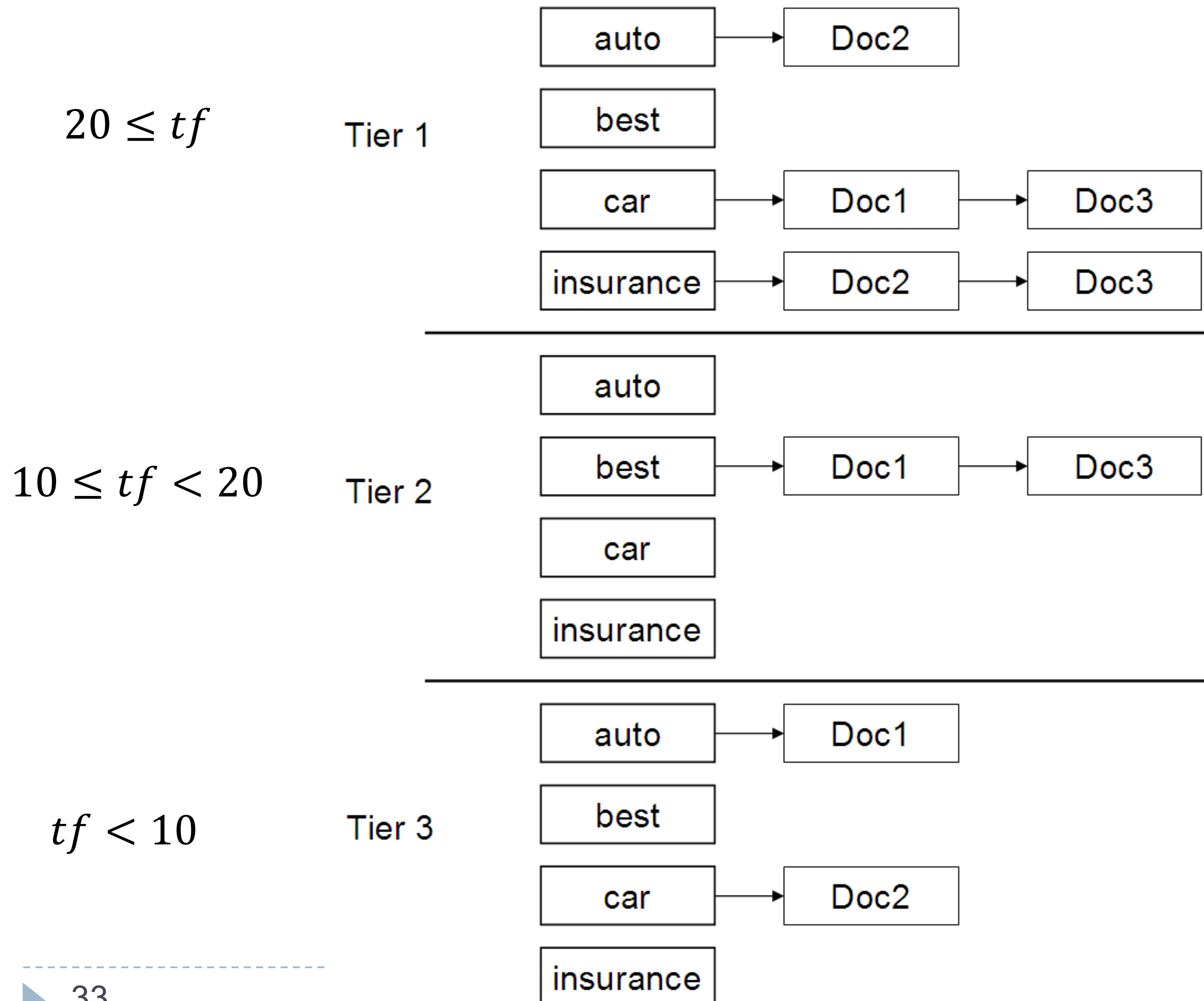
Tiered indexes

- Basic idea:
  - Create several tiers of indexes
  - During query processing, start with highest-tier index
  - If highest-tier index returns at least k (e.g., k = 100) results:
    - stop and return results to user
  - If we've only found < k hits: repeat for next index in tier cascade

Tiered indexes

- Break postings up into a hierarchy of lists
  - Most important to least important
  - Can be done by $g(d)$ or another measure
- Inverted index ⇒ tiers of decreasing importance
- At query time use top tier unless it fails to yield K docs
  - If so drop to lower tiers
- Tiered indexes as one of the reasons for the success of early Google (2000/01)
  - along with PageRank, use of anchor text and proximity constraints

Example tiered index



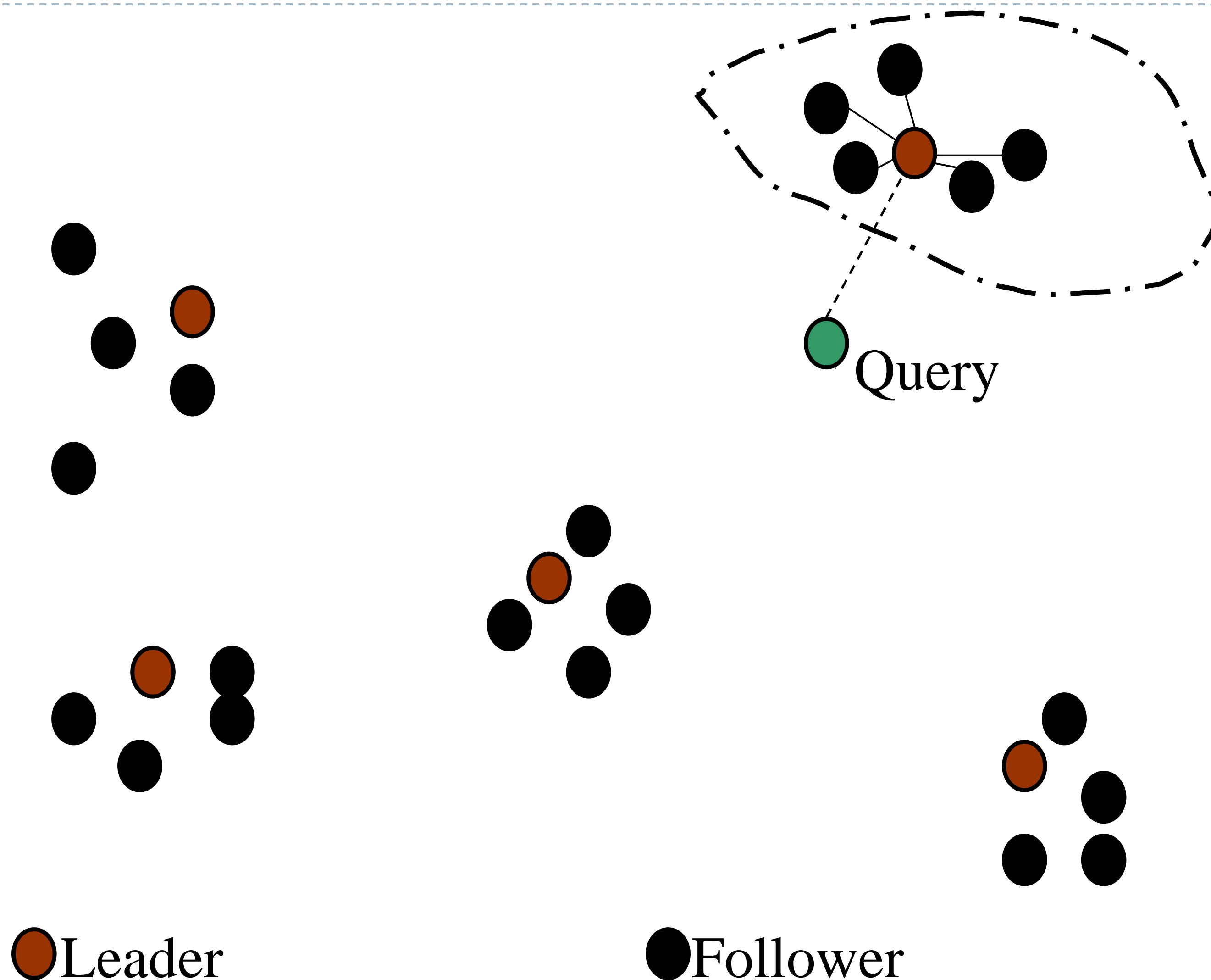
Cluster pruning: preprocessing

- *Leaders:* $\sqrt{N}$ *docs* at random
- For every other doc, pre-compute nearest leader
  - *Followers:* Docs attached to a leader
  - Likely: each leader has ~ $\sqrt{N}$ followers.
- Why random sampling for finding leaders:
  - Fast approach
  - Leaders reflect data distribution

Cluster pruning: query processing

- Given query Q, find its nearest *leader* L.
- Seek K nearest docs from among *L*'s followers.

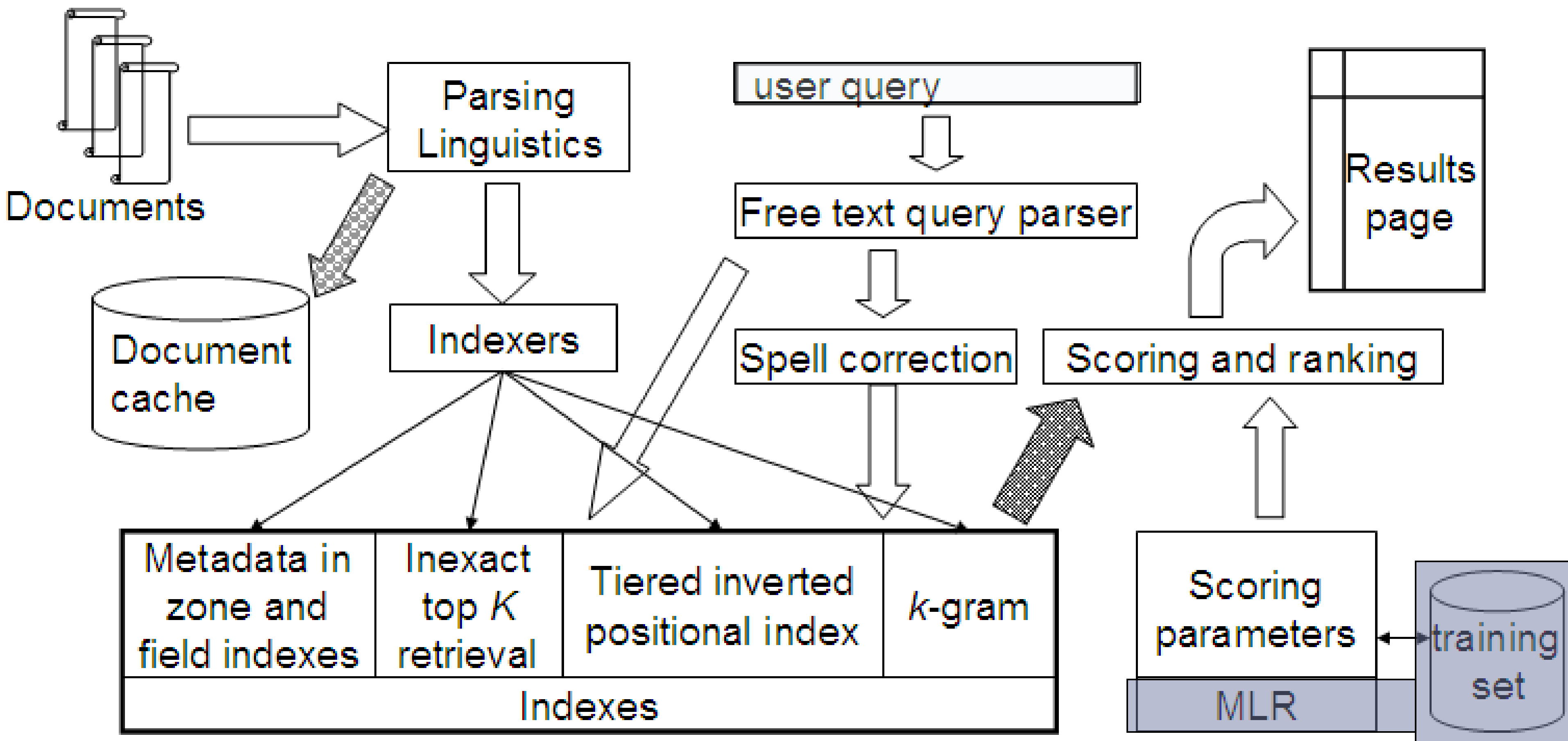
Visualization



General variants

- Have each follower attached to b_1 nearest leaders.
- From query, find b_2 nearest leaders and their followers.
- Can recurse on leader/follower construction.

Complete search system



Components we have introduced thus far

- Document preprocessing (linguistic and otherwise)
- Positional indexes
- Spelling correction
  - k-gram indexes for wildcard queries and spelling correction
- Document scoring
- Tiered indexes

Components we haven't covered yet

- Proximity ranking
  - rank docs in which the query terms occur in the same local window higher
- Query parser
- Zone indexes:
  - the body of the doc, all highlighted text in the doc, anchor text, text in metadata fields etc
- Document cache: we need this for generating snippets
- Machine-learned ranking functions

Parametric and zone indexes

- Thus far, a doc has been a sequence of terms
- In fact docs have multiple parts, some with special semantics:
  - Author
  - Title
  - Date of publication
  - Language
  - Format
  - etc.
- These constitute the <u>metadata</u> 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
    - Author last name = shakespeare
- 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

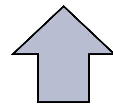
- A zone is a region of the doc that can contain an arbitrary amount of text, e.g.,
  - Title
  - Abstract
  - References …
- Example: “find docs with *merchant* in the title zone and matching the query *gentle rain*”
- Build inverted indexes on zones (to permit querying)

Example zone indexes

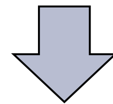
william.abstract → 11 → 121 → 1441 → 1729

william.title → 2 → 4 → 8 → 16

william.author → 2 → 3 → 5 → 8



Encode zones in dictionary vs. postings.



Query term proximity

- Free text queries: just a set of terms
  - Users may prefer docs in which query terms occur within close proximity of each other
  - *w*: smallest window in a doc containing all query terms
    - Query: ***strained mercy***
    - Doc: "*The quality of mercy is not strained*"
    - w: 4 (words)
  - Would like scoring function to take this into account – how?

Query parsers

- Free text query from user may spawn one or more queries to the indexes
  - Run the query as a phrase query
  - If $<K$ docs contain the phrase run smaller phrase queries
  - If we still have $<K$ docs, run the vector space query
  - Rank matching docs by vector space scoring
- This sequence is issued by a query parser

Query parsers

- Example:
  - Query: ***rising interest rates***
  - If $< K$ docs contain
    "***rising interest rates***"
    run queries
    "***rising interest***" and "***interest rates***"
  - If we still have $< K$ docs, run the vector space query
    rising interest rates

- We need aggregate scoring function that *accumulates evidence* of a doc's relevance from multiple sources

Aggregate scores

- Score functions can combine cosine, static quality, proximity, etc.

- How do we know the best combination?
  - Some applications – expert-tuned
  - Increasingly common: machine-learned

Vector space retrieval: Interactions

- Combining **Boolean retrieval** with vector space retrieval?
  - no easy way of combining vector space and Boolean queries
  - Postfiltering is simple, but can be very inefficient – no easy answer.
- Combining **phrase retrieval** with vector space retrieval?
  - no way of demanding a vector space score for a phrase query
  - can in some cases be combined usefully (query parser)
- Combining **wild cards** with vector space retrieval?

Resources

- IIR 7, 6.1
- Resources at http://ifnlp.org/ir
  - How Google tweaks its ranking function
  - Interview with Google search guru Udi Manber
  - Amit Singhal on Google ranking
  - SEO perspective: ranking factors