Information Retrieval (CSD510)

Complete System

February 7, 2024

How do we compute the top K in ranking?

- In many applications, we don't need a complete ranking.
- We just need the top k for a small k (e.g., k $= 100$).
- If we don't need a complete ranking, is there an efficient way of computing just the top k?
- Naive:
  - Compute scores for all N documents
  - Sort
  - Return the top k
- What's bad about this?
- Alternative?

Efficient scoring and ranking

- For a query q, ranking involves generating relative (rather than absolute) scores of the documents in the collection w.r.t. the query.
- For a given query q=*gossip jealous*, we observe that
  - The unit vector $v(q)$ has only two non-zero components
  - In the absence of any weighting for query terms, these non-zero components are equal (i.e. 0.707)
- Use $V(q)$ instead of $v(q)$, where in $V(q)$ all non-zero components of the query vector are set to 1

Fast query processing

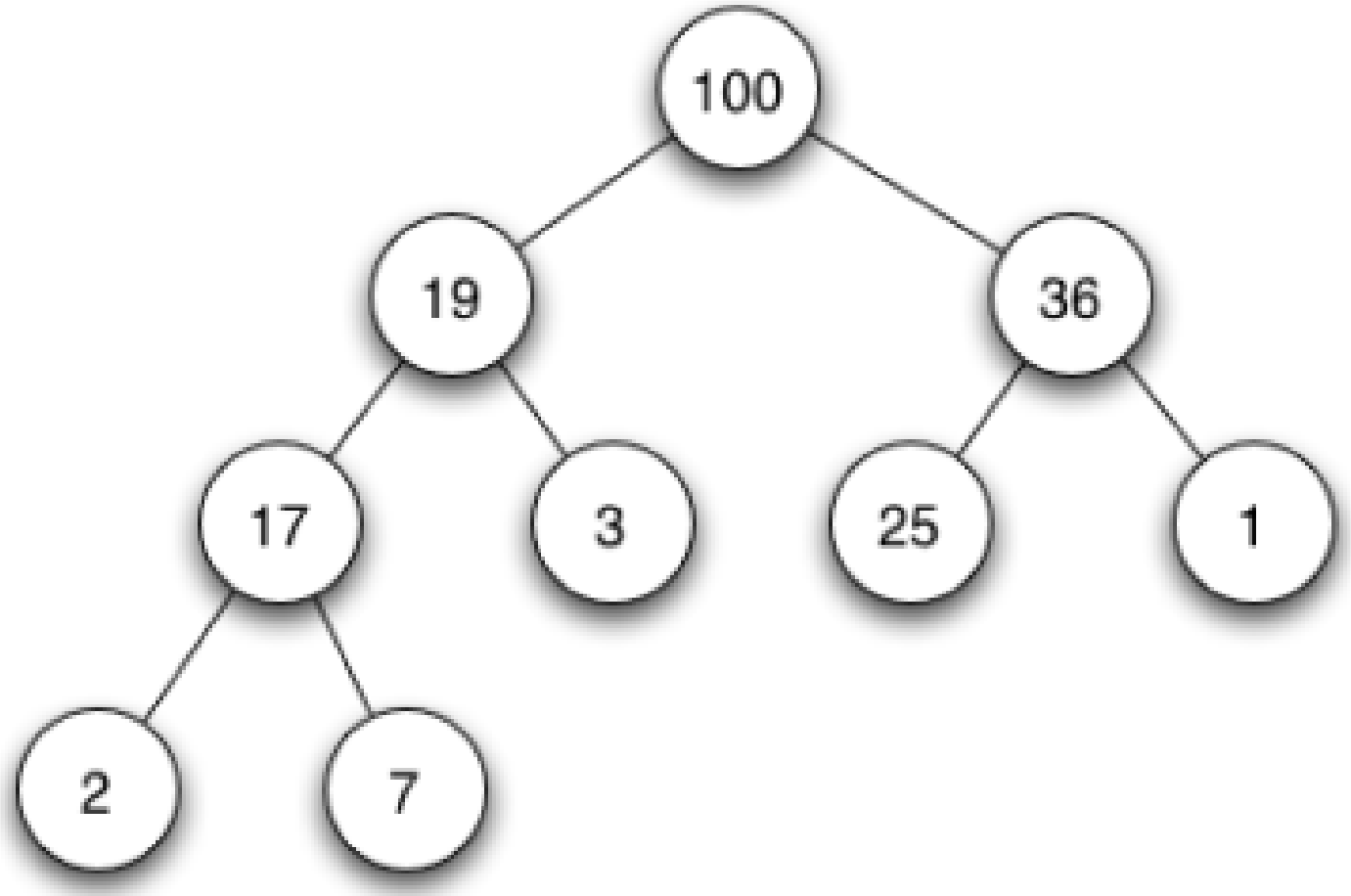
```
FastCosineScore(q)
 1  float Scores[N] = 0
 2  for each d
 3  do Initialize Length[d] to the length of doc d
 4  for each query term t
 5  do calculate w_{t,q} and fetch postings list for t
 6      for each pair(d, tf_{t,d}) in postings list
 7      do add wf_{t,d} to Scores[d]
 8  Read the array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

- For two documents, d_1 and d_2
  - $V(q).v(d_1) > V(q).v(d_2) \iff v(q).v(d_1) > v(q).v(d_2)$

Use heap for selecting the top k

- A heap efficiently implements a priority queue.
- Binary tree in which each node's value is greater than the values of its children
- Takes O(N) operations to construct (where N is the number of documents)...
- ...then each of k winners read off in $O(klogk)$ steps
- Essentially linear in N for small k and large N.

Binary max heap



Inexact top K document retrieval

- Retrieving precisely the K highest-scoring documents for a query may be extremely costly for large collection.
- Retrieve the K documents that are *likely* to be among the top-K.
  - This significantly lowers the cost of computing.
  - Does not materially alter the user's perceived relevance of the top K results.
- Retrieve K documents whose scores are very close to those of the K best.
- This potentially avoids computing scores all N documents in the collection.

Inexact top K document retrieval

- The delay arises from the
  1. Cost of computation of the cosine similarity scores.
  2. Maintaining and selecting the top K documents from data structures (e.g. heaps) when there are several contending documents for the top K positions.
- **Idea:** Eliminate a large number of documents without computing their cosine scores.
- The heuristics have the following two-step scheme:
  1. Find a set A of documents that are contenders, where $K < A < N$.
  2. Return the K top-scoring documents in A

Index elimination

- For a multi-term query q, consider the documents containing at least one of the query terms.
- Additional heuristics for inexact search
  1. We only consider documents containing terms whose idf exceeds a preset threshold.
  2. We only consider documents that contain many (and as a special case, all) of the query terms.

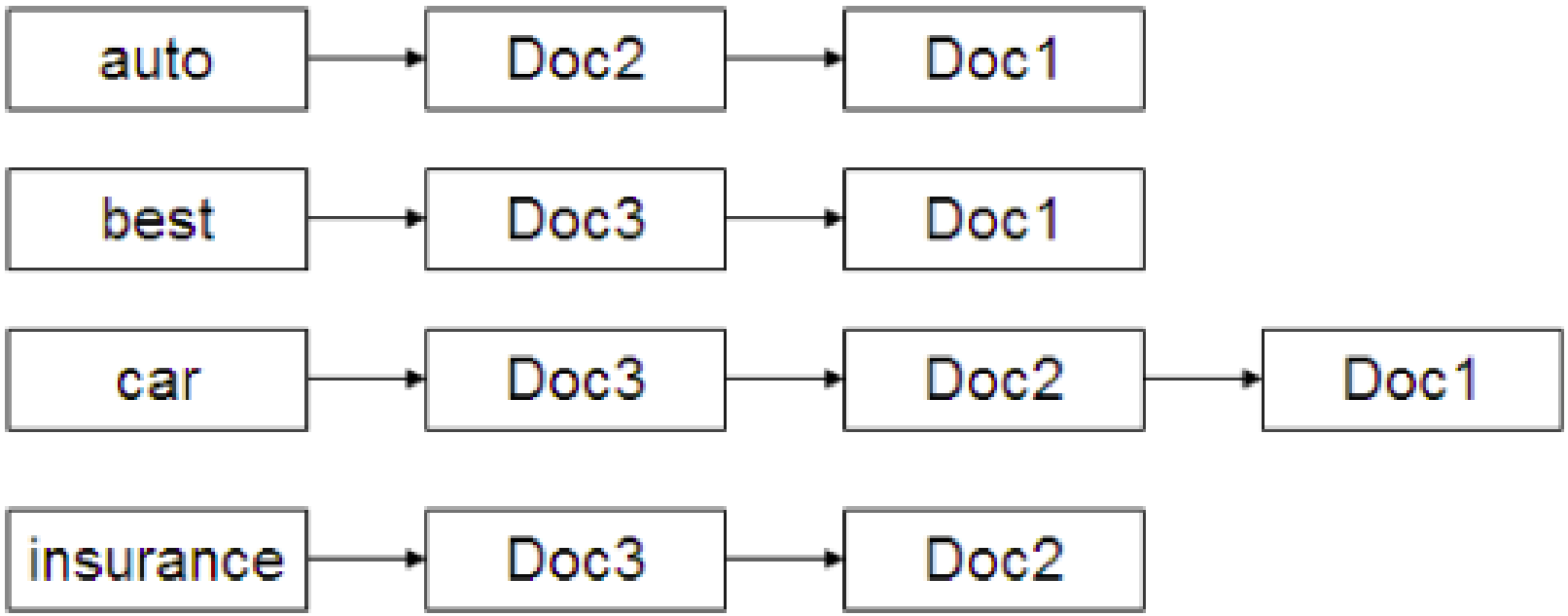
Champion lists

- **Champions list:** For a term t, the list of r documents for which the weights of t are greater than a given threshold.
  - The value of r is chosen in advance.
  - For tf-idf weighting, these would be the r documents with the highest tf values for term t.
- For a query q, the steps to create set A is as follows;
  - Take the union of the champion lists for each of the terms comprising q
  - Restrict cosine computation to only the documents in A
- The value of r is term frequency dependent.

Static quality scores and ordering

- **Static quality scores:** A measure of quality $g(d)$ for each document d that is query-independent.
- Examples
  - For news stories, $g(d)$ may be derived from the number of favorable reviews of the story.
  - PageRank for web page that measures relevance in terms of link structure of the web.
- The net score for a document d is some combination of $g(d)$ together with the *query-dependent score*.
- The contributions of $g(d)$ and the query-dependent scores needs to be decided by heuristics or learning method.
- For intersection in single pass, docs sorted on $g(d)$.

Static quality scores and ordering



Impact ordering

- Both docID-ordering and PageRank-ordering impose a consistent ordering on documents in postings lists.
- Computing cosines in this scheme is document-at-a-time.
- We complete computation of the cosine score of document d_i before starting to compute the cosine score of d_{i+1}.
- Alternative: term-at-a-time processing

Impact ordering

- Idea: don't process postings that contribute little to final score
- Order documents in inverted list according to weight
- Simplest case: normalized tf-idf weight
- Documents in the top k are likely to occur early in these ordered lists
- Early termination while processing inverted lists is unlikely to change top k
- We no longer have a consistent ordering of documents in postings lists.
- We no longer can employ document-at-a-time processing.

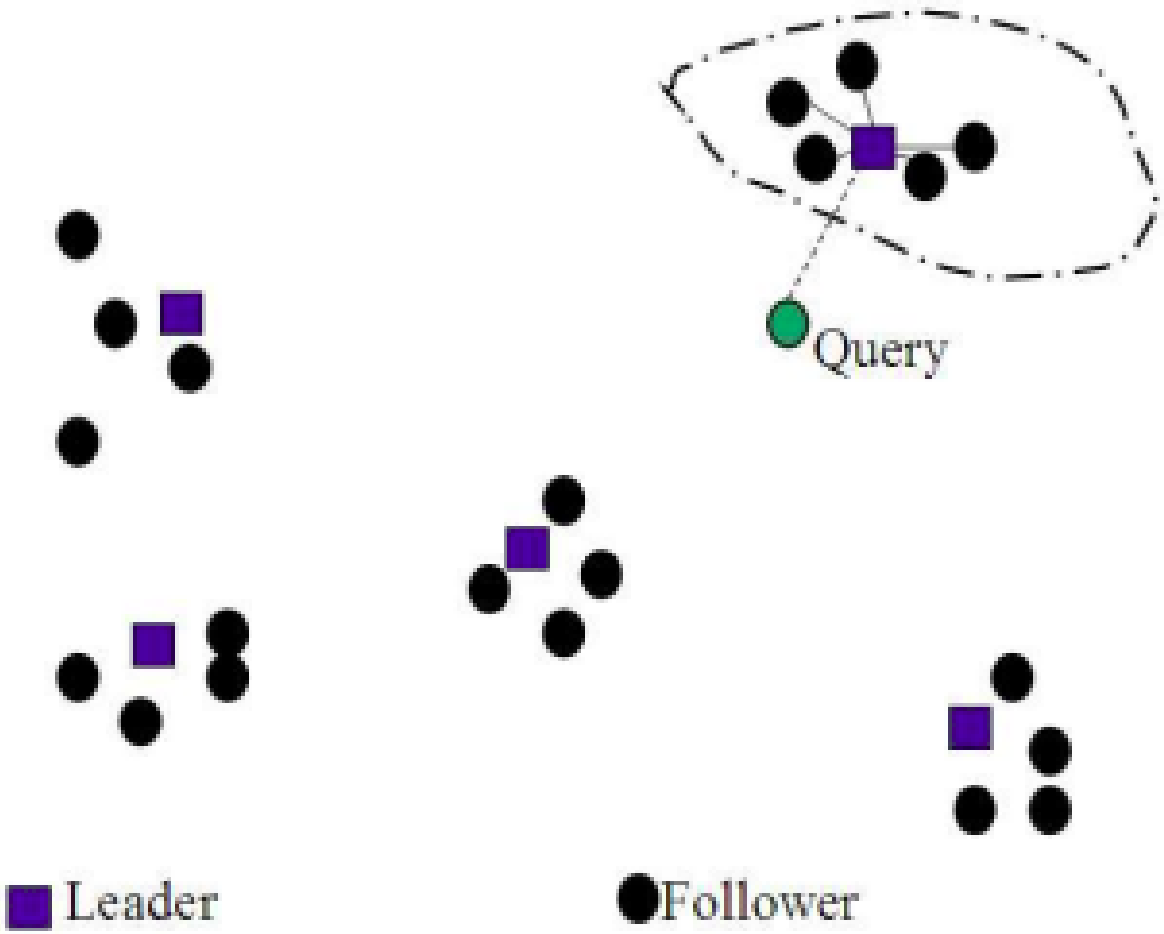
Term-at-a-time processing

- Simplest case: completely process the postings list of the first query term
- Create an accumulator for each docID you encounter
- Then completely process the postings list of the second query term
- Two ideas have been found to significantly lower the number of documents for which we accumulate scores:
  1. when traversing the postings list for a query term t, we stop after considering a prefix of the postings list
  2. Consider the query terms in decreasing order of idf
- Sorting may also be done on other scoring functions (e.g. global scores).

Cluster pruning

- **Preprocessing:** Cluster the document vectors.
- **Query time:** consider only documents in a small number of clusters as candidates for which we compute cosine scores.
- Preprocessing steps:
  1. Pick $\sqrt{N}$ documents at random from the collection. Call these *leaders*.
  2. For each document that is not a leader (*follower*), we compute its nearest leader
- Query processing step
  1. Given a query q, find the leader L that is closest to q. This entails computing cosine similarities from q to each of the $\sqrt{N}$ leaders.
  2. The candidate set A consists of L together with its followers.
  3. We compute the cosine scores for all documents in this candidate set.
- Variation
  - **Preprocessing:** Attach each follower to its b_1 closest leaders
  - **Query time:** Consider the b_2 leaders closest to the query q.

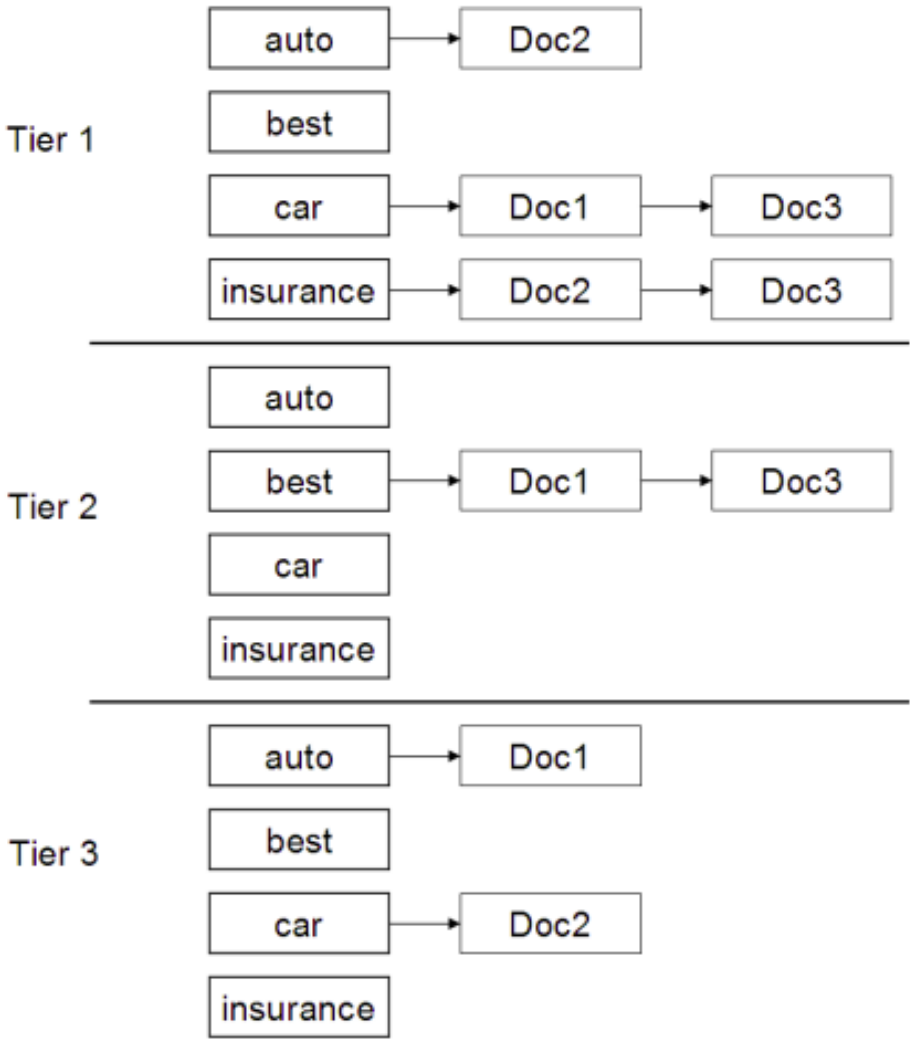
Cluster pruning



Tiered indexes

- A generalization of *champions list*.
- Basic idea:
  - Create several tiers of indexes, corresponding to importance of indexing terms
  - 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
- Example: two-tier system
  - Tier 1: Index of all titles
  - Tier 2: Index of the rest of documents
  - Pages containing the search words in the title are better hits than pages containing the search words in the body of the text.

Tired index


Tier 1
auto
Doc2
best
car
Doc1
Doc3
insurance
Doc2
Doc3
Tier 2
auto
best
Doc1
Doc3
car
insurance
Tier 3
auto
Doc1
best
car
Doc2
insurance

Tired index

- The use of tiered indexes is believed to be one of the reasons that Google search quality was significantly higher initially (2000/01) than that of competitors.
- (along with PageRank, use of anchor text and proximity constraints)

Query-term proximity

- **Idea:** If the relevance of a document is a function of the size of the window in which all (most of the) query terms occur.
- Consider a query with two or more query terms, $t_1, t_2, \cdots, t_k$
- Let ω be the width of the smallest window in a document d that contains all the query terms.
  - **Doc:** The quality of mercy is not strained
  - **Query:** strained mercy
  - ω=4
- Such proximity-weighted scoring functions are a departure from pure cosine similarity.

Complete search system