Background

- Score computation is a large (10s of %) fraction of the CPU work on a query
  - Generally, we have a tight budget on latency (say, 250ms)
  - CPU provisioning doesn’t permit exhaustively scoring every document on every query
- Today we’ll look at ways of cutting CPU usage for scoring, without compromising the quality of results (much)
- Basic idea: avoid scoring docs that won’t make it into the top K

Safe vs non-safe ranking

- The terminology “safe ranking” is used for methods that guarantee that the K docs returned are the K absolute highest scoring documents
- Is it ok to be non-safe?

Ranking function is only a proxy

- User has a task and a query formulation
- Ranking function matches docs to query
- Thus the ranking function is anyway a proxy for user happiness
- If we get a list of K docs “close” to the top K by the ranking function measure, should be ok

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, measured by cosine similarity

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?
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
- 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 10% of the cost of sorting.

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
  - As noted earlier, this may not be a bad thing

SPEEDING COSINE COMPUTATION BY PRUNING

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 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 cosine computation algorithm 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

3 of 4 query terms

<table>
<thead>
<tr>
<th></th>
<th>3</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>Antony</td>
<td>10</td>
<td>3</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>Brutus</td>
<td>10</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>Caesar</td>
<td>10</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>13</td>
</tr>
<tr>
<td>Calpurnia</td>
<td>10</td>
<td>13</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td></td>
</tr>
</tbody>
</table>

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 the champion list of some query term
  - Pick the \( K \) top-scoring docs from amongst these

Exercises

- How can Champion Lists be implemented in an inverted index?
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 bitlys, likes, or bookmarks
  - Pagerank

Quantitative

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
  - \[ \text{net-score}(q,d) = g(d) + \text{cosine}(q,d) \]
  - 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
  - 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 \(g(d) + \text{tf-idf}_d\)
- Seek top-\(K\) results from only the docs in these champion lists
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.

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 recurse on leader/follower construction.
**TIERED INDEXES**

**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 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

**Impact-ordered postings**
- We only want to compute scores for docs for which \( w_{f,t,d} \) is high enough
- We sort each postings list by \( w_{f,t,d} \)
- Now: not all postings in a common order!
- How do we compute scores in order to pick off top \( K \)?
  - Two ideas follow

**Example tiered index**

**1. Early termination**
- When traversing \( t \)’s postings, stop early after either
  - a fixed number of \( r \) docs
  - \( w_{f,t,d} \) 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

SAFE RANKING

Safe vs non-safe ranking

- The terminology “safe ranking” is used for methods that guarantee that the $K$ docs returned are the $K$ absolute highest scoring documents
  - (Not necessarily just under cosine similarity)

Safe ranking

- When we output the top $K$ docs, we have a proof that these are indeed the top $K$
- Does this imply we always have to compute all $N$ cosines?
  - We’ll look at pruning methods
  - So we only fully score some $J$ documents

WAND scoring

- An instance of DAAT scoring
- Basic idea reminiscent of branch and bound
  - We maintain a running threshold score – e.g., the $K^{th}$ highest score computed so far
  - We prune away all docs whose cosine scores are guaranteed to be below the threshold
  - We compute exact cosine scores for only the un-pruned docs

Index structure for WAND

- Postings ordered by docID
- Assume a special iterator on the postings of the form “go to the first docID greater than or equal to X”
- Typical state: we have a “finger” at some docID in the postings of each query term
  - Each finger moves only to the right, to larger docIDs
- Invariant – all docIDs lower than any finger have already been processed, meaning
  - These docIDs are either pruned away or
  - Their cosine scores have been computed

Upper bounds

- At all times for each query term $t$, we maintain an upper bound $UB_t$ on the score contribution of any doc to the right of the finger
  - Max (over docs remaining in $t$’s postings) of $w_t(doc)$

$$UB_t = w_t(38)$$

As finger moves right, $UB$ drops

Pivoting

- Query: catcher in the rye
- Let’s say the current finger positions are as below

<table>
<thead>
<tr>
<th>Term</th>
<th>Position</th>
<th>$UB_t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>catcher</td>
<td>273</td>
<td>2.3</td>
</tr>
<tr>
<td>rye</td>
<td>304</td>
<td>1.8</td>
</tr>
<tr>
<td>in</td>
<td></td>
<td>3.3</td>
</tr>
<tr>
<td>the</td>
<td></td>
<td>4.3</td>
</tr>
</tbody>
</table>

Threshold = 6.8

Prune docs that have no hope

- Terms sorted in order of finger positions
- Move fingers to 589 or right

<table>
<thead>
<tr>
<th>Term</th>
<th>Position</th>
<th>$UB_t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>catcher</td>
<td>273</td>
<td>2.3</td>
</tr>
<tr>
<td>rye</td>
<td>304</td>
<td>1.8</td>
</tr>
<tr>
<td>in</td>
<td>589</td>
<td>3.3</td>
</tr>
<tr>
<td>the</td>
<td>762</td>
<td>4.3</td>
</tr>
</tbody>
</table>

Update $UB$’s

Compute 589’s score if need be

- If 589 is present in enough postings, compute its full cosine score – else some fingers to right of 589
- Pivot again ...

<table>
<thead>
<tr>
<th>Term</th>
<th>Position</th>
<th>$UB_t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>catcher</td>
<td></td>
<td>589</td>
</tr>
<tr>
<td>rye</td>
<td></td>
<td>589</td>
</tr>
<tr>
<td>in</td>
<td></td>
<td>589</td>
</tr>
<tr>
<td>the</td>
<td></td>
<td>762</td>
</tr>
</tbody>
</table>

WAND summary

- In tests, WAND leads to a 90+% reduction in score computation
  - Better gains on longer queries
- Nothing we did was specific to cosine ranking
  - We need scoring to be additive by term
- WAND and variants give us safe ranking
  - Possible to devise “careless” variants that are a bit faster but not safe (see summary in Ding+Suel 2011)
  - Ideas combine some of the non-safe scoring we considered

FINISHING TOUCHES FOR A COMPLETE SCORING SYSTEM
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

Putting it all together