Efficient query processing
Efficient scoring, distributed query processing
Web Search
Ranking functions

• In general, document scoring functions are of the form

\[
\text{score}(q, d) = \text{quality}(d) + \sum_{t \in q} \text{score}(t, d)
\]

• The BM25 function, is one of the best performing:

\[
\begin{align*}
\text{Score}_{BM25}(q, d) &= \sum_{t \in q} \log \left( \frac{N}{N_t} \right) \cdot \text{TF}_{BM25}(t, d), \\
\text{TF}_{BM25}(t, d) &= \frac{f_{t,d} \cdot (k_1 + 1)}{f_{t,d} + k_1 \cdot \left( (1 - b) + b \cdot \frac{l_d}{l_{avg}} \right)}
\end{align*}
\]

• The term frequency is upper bounded:

\[
\lim_{f_{t,d} \to \infty} \text{TF}_{BM25}(t, d) = k_1 + 1
\]
Efficient query processing

• Accurate retrieval of top k documents
  • Document at-a-time
  • MaxScore

• Approximate retrieval of top k documents
  • At query time: term-at-a-time
  • At indexing time: term centric and document centric

• Other approaches
Scoring document-at-a-time

- All documents containing at least one term is scored
- Each document is scored sequentially
  - A naïve method to score all documents is computationally too complex.
- Using a heap to process queries is faster

```plaintext
rankBM25_DocumentAtATime ((t1, ..., tn), k) ≡
1   m ← 0  // m is the total number of matching documents
2   d ← min1≤i≤n{nextDoc(ti, −∞)}
3   while d < ∞ do
4       results[m].docid ← d
5       results[m].score ← \[ \sum_{i=1}^{n} \log(N/N_{t_i}) \cdot TF_{BM25}(t_i, d) \]
6       m ← m + 1
7       d ← min1≤i≤n{nextDoc(ti, d)}
8   sort results[0..(m − 1)] in decreasing order of score
9   return results[0..(k − 1)]
```

Figure 5.1  Document-at-a-time query processing with BM25.
### Scoring document-at-a-time: Algorithm

**rankBM25** Document AtATime WithHeaps \((t_1, \ldots, t_n), k\) \(\equiv\)

<table>
<thead>
<tr>
<th>Sort in increasing order of score</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. for (i \leftarrow 1) to (k) do // create a min-heap for the top (k) search results</td>
<td></td>
</tr>
<tr>
<td>2. (\text{results}[i].\text{score} \leftarrow 0)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Gets all docs with the query terms</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3. for (i \leftarrow 1) to (n) do // create a min-heap for the (n) query terms</td>
<td></td>
</tr>
<tr>
<td>4. (\text{terms}[i].\text{term} \leftarrow t_i)</td>
<td></td>
</tr>
<tr>
<td>5. (\text{terms}[i].\text{nextDoc} \leftarrow \text{nextDoc}(t_i, -\infty))</td>
<td></td>
</tr>
<tr>
<td>6. sort (\text{terms}) in increasing order of (\text{nextDoc}) // establish heap property for (\text{terms})</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Gets the docs with the lowest ID</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7. while (\text{terms}[0].\text{nextDoc} &lt; \infty) do</td>
<td></td>
</tr>
<tr>
<td>8. (d \leftarrow \text{terms}[0].\text{nextDoc})</td>
<td></td>
</tr>
<tr>
<td>9. (\text{score} \leftarrow 0)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process one doc</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10. while (\text{terms}[0].\text{nextDoc} = d) do</td>
<td></td>
</tr>
<tr>
<td>11. (t \leftarrow \text{terms}[0].\text{term})</td>
<td></td>
</tr>
<tr>
<td>12. (\text{score} \leftarrow \text{score} + \log(N/N_t) \cdot \text{TF}_{BM25}(t, d))</td>
<td></td>
</tr>
<tr>
<td>13. (\text{terms}[0].\text{nextDoc} \leftarrow \text{nextDoc}(t, d))</td>
<td></td>
</tr>
<tr>
<td>14. (\text{reheap}(\text{terms})) // restore heap property for (\text{terms})</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Replace the worst doc</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>15. if (\text{score} &gt; \text{results}[0].\text{score}) then</td>
<td></td>
</tr>
<tr>
<td>16. (\text{results}[0].\text{docid} \leftarrow d)</td>
<td></td>
</tr>
<tr>
<td>17. (\text{results}[0].\text{score} \leftarrow \text{score})</td>
<td></td>
</tr>
<tr>
<td>18. (\text{reheap}(\text{results})) // restore heap property for (\text{results})</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sort in decreasing order of score</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>19. remove from (\text{results}) all items with (\text{score} = 0)</td>
<td></td>
</tr>
<tr>
<td>20. sort (\text{results}) in decreasing order of (\text{score})</td>
<td></td>
</tr>
<tr>
<td>21. return (\text{results})</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 5.3** Document-at-a-time query processing with BM25, using binary heaps for managing the set of terms and managing the set of top-\(k\) documents.
MaxScore

- We know that each term frequency is bounded by

\[
\lim_{f_{t,d} \to \infty} \text{TF}_{BM25}(t,d) = k_1 + 1
\]

- We call this score the MaxScore of a term

- If the score of the k’th document exceeds the MaxScore of a term X,
  - We can ignore documents containing only term X
  - When considering term Y, we still need to check the term X contribution
  - If the score of the k’th document exceeds the MaxScore of terms X and Y,
    - We can ignore documents containing terms Y
Scoring document-at-a-time: comparison

- Comparison between reheap with & w/out MaxScore

<table>
<thead>
<tr>
<th>Table 5.1</th>
<th>Total time per query and CPU time per query, with and without MAXSCORE. Data set: 10,000 queries from TREC TB 2006, evaluated against a frequency index for GOV2.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Without MaxScore</td>
</tr>
<tr>
<td></td>
<td>Wall Time</td>
</tr>
<tr>
<td>OR, k=10</td>
<td>400 ms</td>
</tr>
<tr>
<td>OR, k=100</td>
<td>402 ms</td>
</tr>
<tr>
<td>OR, k=1000</td>
<td>426 ms</td>
</tr>
<tr>
<td>AND, k=10</td>
<td>160 ms</td>
</tr>
</tbody>
</table>

Both methods are exact!
Approximating the K largest scores

• Typically we want to retrieve the top $K$ docs
  • not to totally order all docs in the collection

• Can we approximate the $K$ highest scoring documents?

• Let $J =$ number of docs with nonzero scores
  • We seek the $K$ best of these $J$
Scoring term-at-time

• The index is organized by postings-lists
  • Processing a query a document-at-a-time requires several disk seeks
  • Processing a query a term-at-a-time minimizes disk seeks

• In this method, a query is processed a term-at-a-time and an accumulator stores the score of each term in the query.

• When all terms are processed, the accumulator contains the scores of the documents.

• Do we need to have an accumulator the size of the collection or the largest posting list?
Scoring term-at-time

- A query is processed term-at-time and an accumulator stores the score of each term in the query.

- When all terms are processed, the accumulator contains the scores of the documents.

- Do we need to have an accumulator the size of the collection or the largest posting list?

```plaintext
rankBM25_TermAtATime((t₁, ..., tₙ), k) ≡
1: sort \{t₁, ..., tₙ\} in increasing order of N_i
2: acc ← \{\}, acc' ← \{\} // initialize two empty accumulator sets
3: acc[0].docid ← ∞ // end-of-list marker
4: for i ← 1 to n do
5:   inPos ← 0 // current position in acc
6:   outPos ← 0 // current position in acc'
7:   for each document d in tᵢ's postings list do
8:     while acc[inPos] < d do // copy accumulators from acc to acc'
9:       acc'[outPos++] ← acc[inPos++]
10:      acc'[outPos].docid ← d
11:     acc'[outPos].score ← log(N/N_i) · TFBM25(tᵢ, d)
12:     if acc[inPos].docid = d then // term and accumulator coincide
13:       acc'[outPos].score ← acc'[outPos].score + acc[inPos++].score
14:       d ← nextDoc(tᵢ, acc'[outPos])
15:     outPos ← outPos + 1
16:     while acc[inPos] < ∞ do // copy remaining accumulators from acc to acc'
17:       acc'[outPos++] ← acc[inPos++]
18:     acc'[outPos].docid ← ∞ // end-of-list marker
19:     swap acc and acc'
20: return the top k items of acc // use a heap to select the top k
```

Figure 5.4 Term-at-a-time query processing with BM25. Document scores are stored in accumulators. The accumulator array is traversed co-sequentially with the current term’s postings list.
Limited accumulator

• The accumulator may not fit in memory, so, we ought to limit the accumulator’s length

• When traversing t’s postings
  • Add the posting only if it is below a $v_{TF}$ threshold

• For each document in the postings list
  • accumulate the term score or use new positions in accumulator for that doc
High-idf query terms only

• When considering the postings of query terms

• Look at them in order of decreasing idf
  • High idf terms likely to contribute most to score

• For a query such as “catcher in the rye”
  • Only accumulate scores from “catcher” and “rye”
Scoring term-at-a-time

• Baseline:
  • Top 10  MaxScore  188ms, 93 ms, 2.8x10^5 docs
  • Top 1000  MaxScore  242ms, 152 ms, 6.2x10^5 docs
Index pruning

• The accumulator technique ignores several query term’s postings
  • This is done in query time.

• How can we prune postings that we know in advance that they will be almost noise?

• The goal is to keep only the most informative postings in the index.
Term-centric index pruning

- **Examining only term postings**, we can decide if a given term is relevant in general (IDF) or relevant for the document (TF).

- If a term appears less than $K$ times in documents, store all of t’s postings in the index

- If the term t appears in more than $K$ documents
  - Compute the term score of the $K$'th document
  - Consider only the postings with scores $> score(d_k, t) \cdot \epsilon$ where $0.0 < \epsilon < 1.0$
**Document-centric index pruning**

- **Examining each document’s terms distribution** we can predict which terms are the most representative of that document.
  - The terms is added to the index only if it is considered representative of the document.

- Compute the Kullback-Leibler divergence between the terms distribution in the document and in the collection.

- For each document, select the top $\lambda \cdot n$ terms to be added to the index.
Head-to-head comparison

**Document-centric**

**Term-centric**

![Graphs showing retrieval effectiveness and performance](image)

**Figure 5.8** Term-centric index pruning with $K = 1,000$ and $\varepsilon$ between 0.5 and 1. Data set for efficiency evaluation: 10,000 queries from TREC TB 2006. Data set for effectiveness evaluation: TREC topics 701–800.
Head-to-head comparison

- **Baseline:**
  - Top 10  MaxScore 188ms, 93 ms, 2.8x10^5 docs
  - Top 1000 MaxScore 242ms, 152 ms, 6.2x10^5 docs
Other approaches

• Static scores

• Cluster pruning

• Number of query terms

• Impact ordering
  • Champion lists
  • Tiered indexes
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)
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

- Given query $Q$, find its nearest leader $L$.
- Seek $K$ nearest docs from among $L$’s followers.
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
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
Tiered indexes

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

- 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
  - Common practice in web search engines
Scalability: Index partitioning

**Document partitioning**

<table>
<thead>
<tr>
<th>Documents</th>
<th>D_1</th>
<th>D_2</th>
<th>D_3</th>
<th>D_4</th>
<th>D_5</th>
<th>D_6</th>
<th>D_7</th>
<th>D_8</th>
<th>D_9</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T_2</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T_3</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T_4</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T_5</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T_6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T_7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T_8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

**Term partitioning**

<table>
<thead>
<tr>
<th>Documents</th>
<th>D_1</th>
<th>D_2</th>
<th>D_3</th>
<th>D_4</th>
<th>D_5</th>
<th>D_6</th>
<th>D_7</th>
<th>D_8</th>
<th>D_9</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T_2</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T_3</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T_4</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T_5</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T_6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T_7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T_8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

**Node 1**

- D_1
- T_1

**Node 2**

- D_2
- T_2
- T_3

**Node 3**

- D_3
- T_4
- T_5
- T_6
- T_7
- T_8
Doc-partitioning indexes

• Each index server is responsible for a random sub-set of the documents

• All $n$ nodes return $k$ results to produce the final list with $m$ results

• Requires a background process to keep the $idf$ (and other general statistics) synchronized across index servers
How many documents to return per index-server?

- The choice of $k$ has impact on:
  - The network load to transfer partial search results from the index-servers to the server doing the rank fusion;
  - The precision of the final rank.

*Figure 14.3* Choosing the minimum retrieval depth $k$ that returns the top $m$ results with probability $p(n, m, k) > 99.9\%$, where $n$ is the number of nodes in the document-partitioned index.
Term-partitioning indexes

- Each index server receives a sub-set of the dictionary’s terms

- A query is sent simultaneously to the term’s corresponding nodes.
  - Each node passes its accumulator to the next node or to the central node to compute the final rank.

- Disadvantages:
  - This requires that each node loads the full posting list for each term.
  - Uneven load balance due to query drifts.
  - Unable to do support efficient document-at-a-time scoring.
Planet-scale load-balancing

• When a systems receives requests from the entire planet at every second...

• The best way to load-balance the queries is to use DNS to distributed queries across data-centers.

• Each query is assigned a different IP according to the data-center load and to the user’s geographic location.

Summary

- Relevance feedback
  - Pseudo-relevance feedback

- Query expansion
  - Dictionary based
  - Statistical analysis of words co-occurrences

- Efficient scoring
  - Per-term and per-doc pruning

- Distributed query processing
  - Per-term and per-doc pruning

Chapter 7 and 9

Section 5.1
Section 14.1