Efficient query processing

Efficient scoring, distributed query processing

Web Search

Ranking functions

- In general, document scoring functions are of the form

$$\mathrm{score}(q,d) \;=\; \mathrm{quality}(d) + \sum_{t\in q} \mathrm{score}(t,d)$$

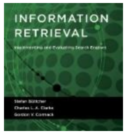
- The BM25 function, is one of the best performing:

$$\begin{aligned}\mathrm{Score}_{\mathrm{BM25}}(q,d) &= \sum_{t\in q} \log\left(\frac{N}{N_t}\right)\cdot \mathrm{TF}_{\mathrm{BM25}}(t,d),\\ \mathrm{TF}_{\mathrm{BM25}}(t,d) &= \frac{f_{t,d}\cdot(k_1+1)}{f_{t,d}+k_1\cdot((1-b)+b\cdot(l_d/l_{avg}))}\end{aligned}$$

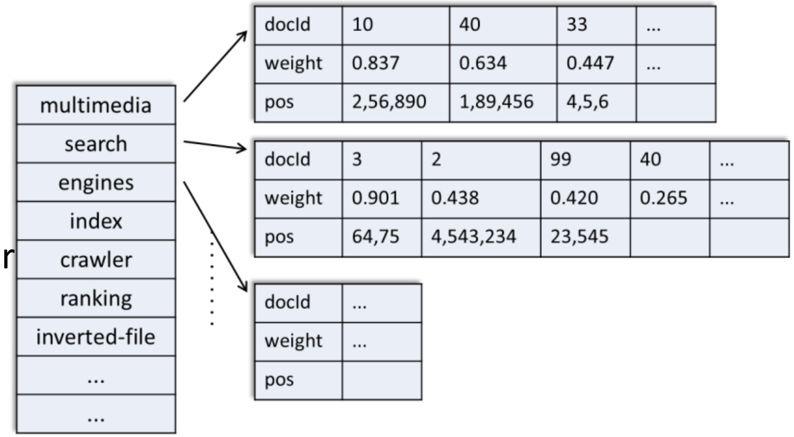
- The term frequency is upper bounded: $\lim_{f_{t,d}\to\infty} \mathrm{TF}_{\mathrm{BM25}}(t,d) \;=\; k_1+1$

Efficient query processing

Section 5.1



- 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 centr
- Other approaches



multimedia
search
engines
index
crawler
ranking
inverted-file
...
...

docId	10	40	33	...
weight	0.837	0.634	0.447	...
pos	2,56,890	1,89,456	4,5,6	

docId	3	2	99	40	...
weight	0.901	0.438	0.420	0.265	...
pos	64,75	4,543,234	23,545		

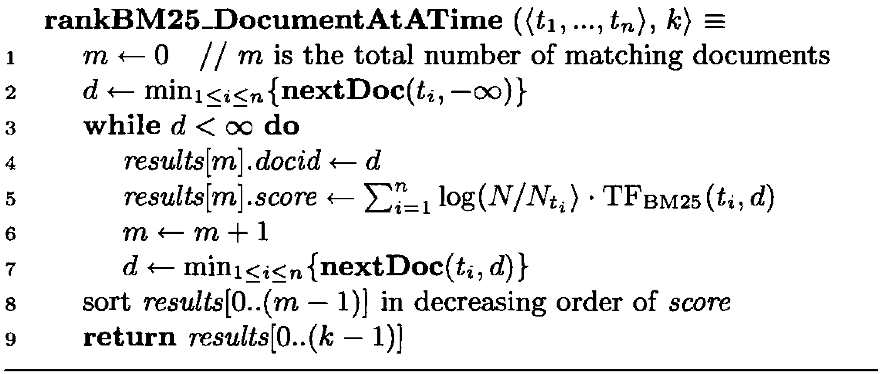
docId	...
weight	...
pos	

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 computationaly too complex.

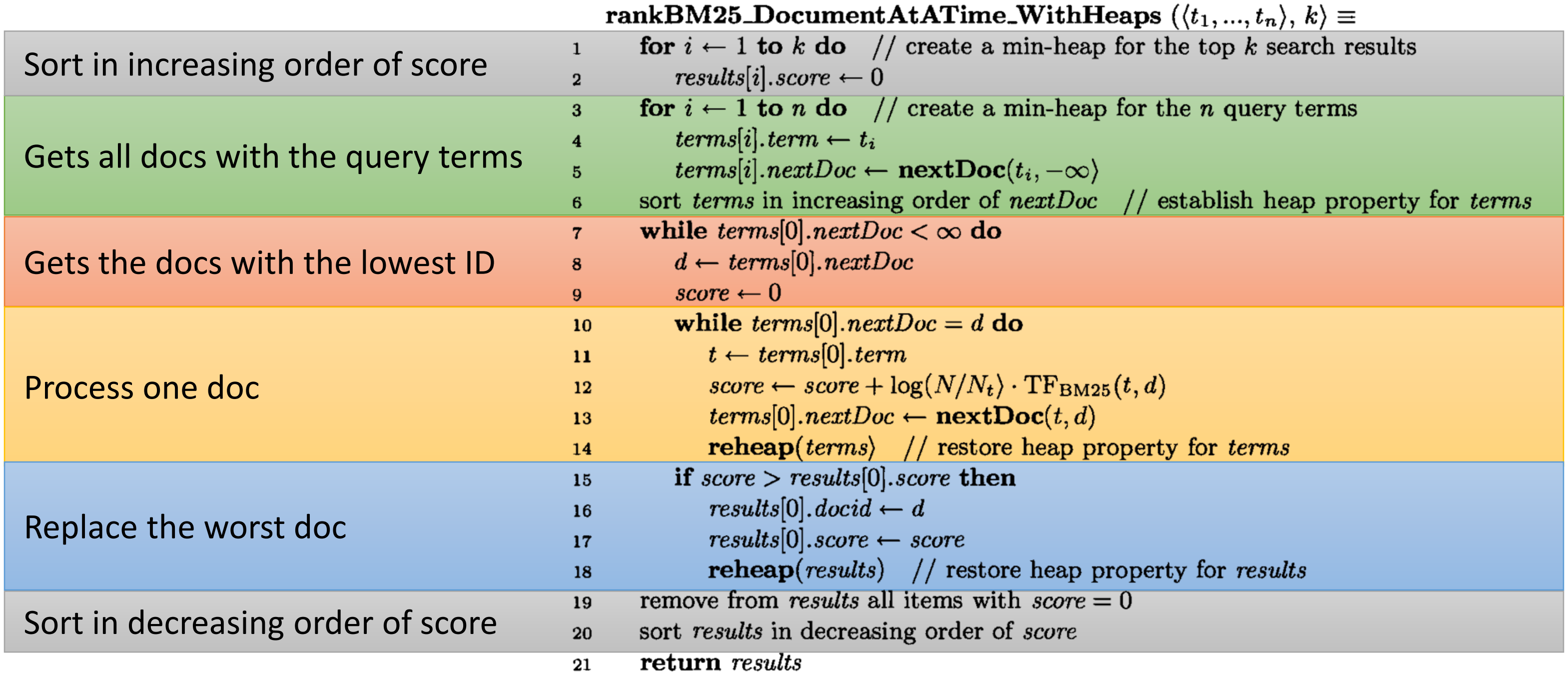
- Using a heap to process queries is faster



```
**rankBM25_DocumentAtATime** ($\langle t_1, ..., t_n \rangle$, $k$) $\equiv$
    $m \leftarrow 0$   // $m$ is the total number of matching documents
    $d \leftarrow \min_{1 \le i \le n}\{\textbf{nextDoc}(t_i, -\infty)\}$
    **while** $d < \infty$ **do**
        $results[m].docid \leftarrow d$
        $results[m].score \leftarrow \sum_{i=1}^{n} \log(N/N_{t_i}) \cdot \text{TF}_{\text{BM25}}(t_i, d)$
        $m \leftarrow m + 1$
        $d \leftarrow \min_{1 \le i \le n}\{\textbf{nextDoc}(t_i, d)\}$
    sort $results[0..(m-1)]$ in decreasing order of $score$
    **return** $results[0..(k-1)]$
```

Figure 5.1 Document-at-a-time query processing with BM25.

Scoring document-at-a-time: Algorithm



Sort in increasing order of score

Gets all docs with the query terms

Gets the docs with the lowest ID

Process one doc

Replace the worst doc

Sort in decreasing order of score

```
rankBM25_DocumentAtATime_WithHeaps ($\langle t_1, ..., t_n \rangle$, $k$) ≡
for $i \leftarrow 1$ to $k$ do   // create a min-heap for the top $k$ search results
    $results[i].score \leftarrow 0$
for $i \leftarrow 1$ to $n$ do   // create a min-heap for the $n$ query terms
    $terms[i].term \leftarrow t_i$
    $terms[i].nextDoc \leftarrow$ nextDoc($t_i, -\infty$)
sort $terms$ in increasing order of $nextDoc$   // establish heap property for $terms$
while $terms[0].nextDoc < \infty$ do
    $d \leftarrow terms[0].nextDoc$
    $score \leftarrow 0$
    while $terms[0].nextDoc = d$ do
        $t \leftarrow terms[0].term$
        $score \leftarrow score + \log(N/N_t) \cdot \mathrm{TF}_{\mathrm{BM25}}(t, d)$
        $terms[0].nextDoc \leftarrow$ nextDoc($t, d$)
        reheap($terms$)   // restore heap property for $terms$
    if $score > results[0].score$ then
        $results[0].docid \leftarrow d$
        $results[0].score \leftarrow score$
        reheap($results$)   // restore heap property for $results$
remove from $results$ all items with $score = 0$
sort $results$ in decreasing order of $score$
return $results$
```

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

$$\lim_{f_{t,d} \to \infty} \mathrm{TF}_{\mathrm{BM25}}(t, d) = k_1 + 1$$

- 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

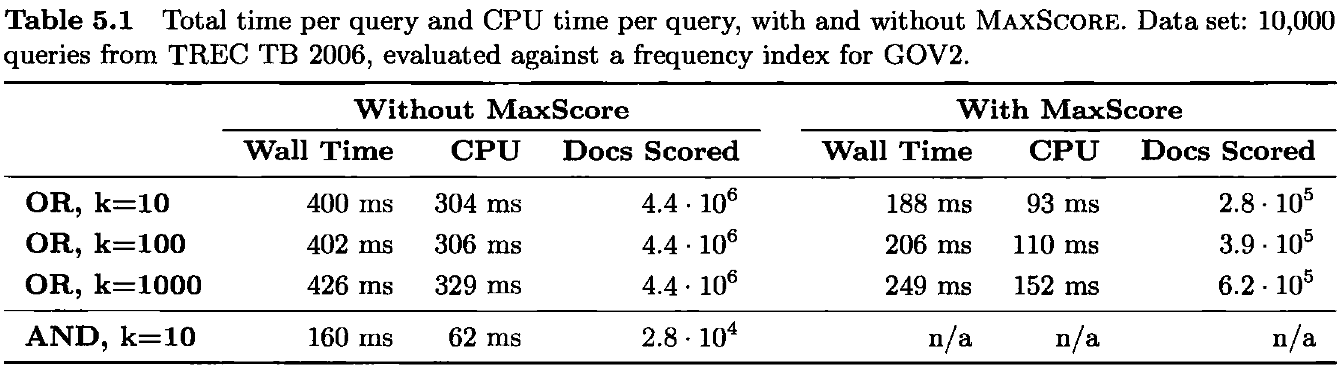


Table 5.1 Total time per query and CPU time per query, with and without MaxScore. Data set: 10,000 queries from TREC TB 2006, evaluated against a frequency index for GOV2.

	Without MaxScore			**With MaxScore**		
	Wall Time	**CPU**	**Docs Scored**	**Wall Time**	**CPU**	**Docs Scored**
OR, k=10	400 ms	304 ms	$4.4 \cdot 10^6$	188 ms	93 ms	$2.8 \cdot 10^5$
OR, k=100	402 ms	306 ms	$4.4 \cdot 10^6$	206 ms	110 ms	$3.9 \cdot 10^5$
OR, k=1000	426 ms	329 ms	$4.4 \cdot 10^6$	249 ms	152 ms	$6.2 \cdot 10^5$
AND, k=10	160 ms	62 ms	$2.8 \cdot 10^4$	n/a	n/a	n/a

Both methods are exact!

Approximating the K largest scores

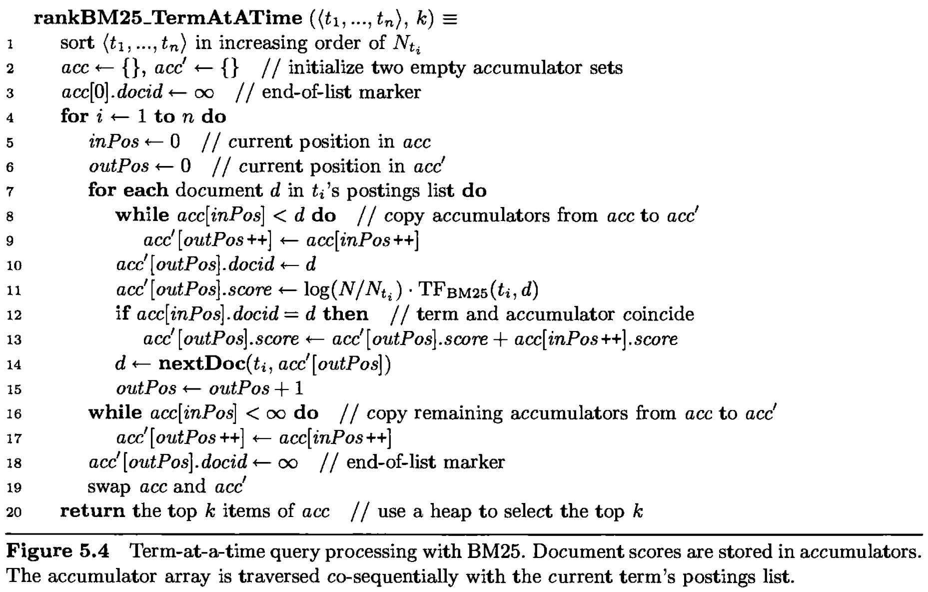
- Typically we want to retrieve the top *K* docs
    - not to totally order all docs in the collection
- Can we approximate the *K* highest scoring documents?
- Let J = number of docs with nonzero scores
    - We seek the K best of these J

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 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?



```
rankBM25_TermAtATime ($\langle t_1, ..., t_n \rangle$, $k$) ≡
  sort $\langle t_1, ..., t_n \rangle$ in increasing order of $N_{t_i}$
  $acc \leftarrow \{\}$, $acc' \leftarrow \{\}$   // initialize two empty accumulator sets
  $acc[0].docid \leftarrow \infty$   // end-of-list marker
  for $i \leftarrow 1$ to $n$ do
    $inPos \leftarrow 0$   // current position in $acc$
    $outPos \leftarrow 0$   // current position in $acc'$
    for each document $d$ in $t_i$'s postings list do
      while $acc[inPos] < d$ do   // copy accumulators from $acc$ to $acc'$
        $acc'[outPos{+}{+}] \leftarrow acc[inPos{+}{+}]$
      $acc'[outPos].docid \leftarrow d$
      $acc'[outPos].score \leftarrow \log(N/N_{t_i}) \cdot \mathrm{TF}_{\mathrm{BM25}}(t_i, d)$
      if $acc[inPos].docid = d$ then   // term and accumulator coincide
        $acc'[outPos].score \leftarrow acc'[outPos].score + acc[inPos{+}{+}].score$
      $d \leftarrow$ nextDoc($t_i$, $acc'[outPos]$)
      $outPos \leftarrow outPos + 1$
    while $acc[inPos] < \infty$ do   // copy remaining accumulators from $acc$ to $acc'$
      $acc'[outPos{+}{+}] \leftarrow acc[inPos{+}{+}]$
    $acc'[outPos].docid \leftarrow \infty$   // end-of-list marker
    swap $acc$ and $acc'$
  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

Sec. 7.1.5

- 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

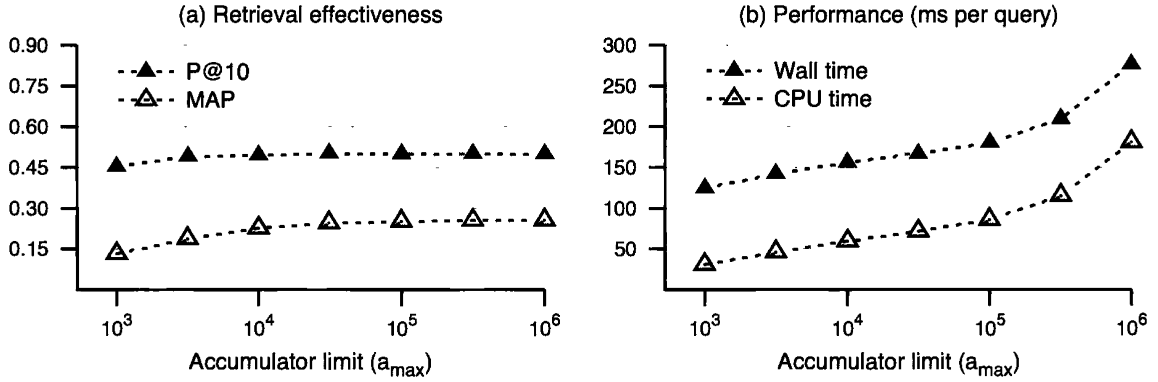
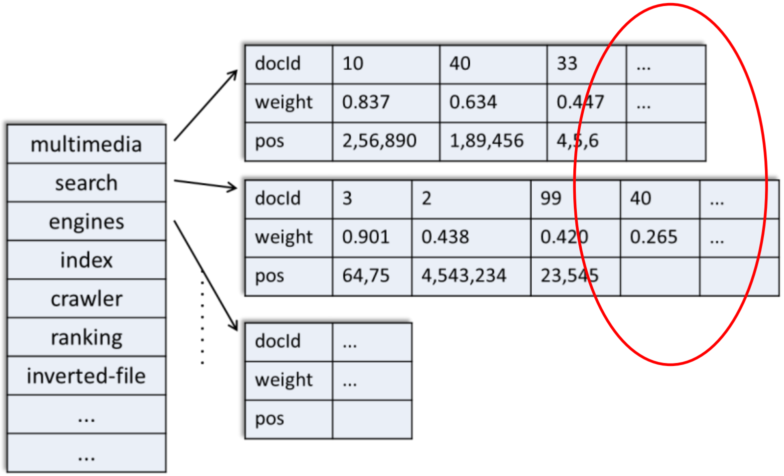


Figure 5.6 Retrieval effectiveness and query processing performance for term-at-a-time query evaluation with accumulator pruning. Data set: 10,000 queries from TREC TB 2006, run against a frequency index for GOV2.

- 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.



multimedia
search
engines
index
crawler
ranking
inverted-file
...
...

docId	10	40	33	...
weight	0.837	0.634	0.447	...
pos	2,56,890	1,89,456	4,5,6	

docId	3	2	99	40	...
weight	0.901	0.438	0.420	0.265	...
pos	64,75	4,543,234	23,545		

docId	...
weight	...
pos	

Term-centric index prunning

- **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 $0.0 < \epsilon < 1.0$

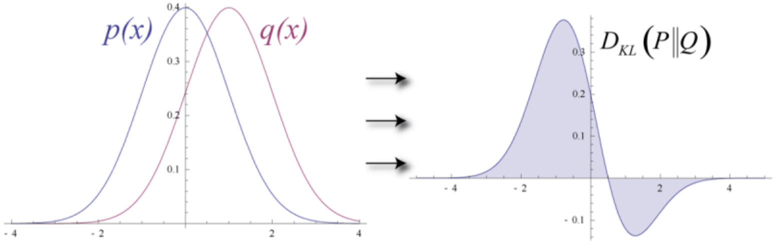
Document-centric index prunning

- **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 <u>considered representative of the document</u>

- 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

$$p_d(t) \cdot \log\left(\frac{p_d(t)}{p_C(t)}\right)$$



Head-to-head comparison

Document-centric

Term-centric

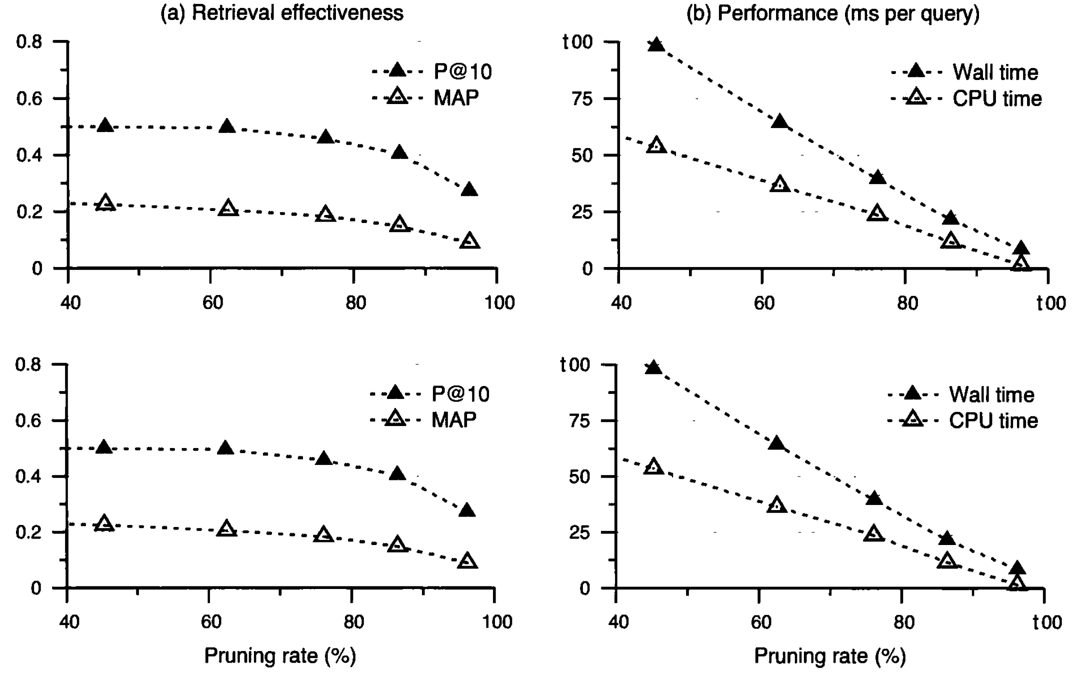
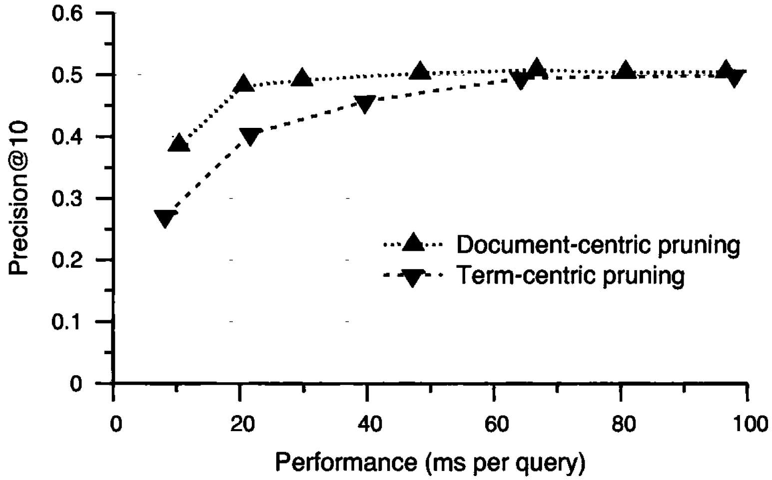


Figure 5.8 Term-centric index pruning with $K = 1{,}000$ and ε between 0.5 and 1. Data set for efficiency evaluation: 10,000 queries from TREC TB 2006. Data set for effectiveness evaluation: TREC topics 701–800.

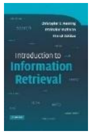
Head-to-head comparison



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

Other approaches

Chapter 7



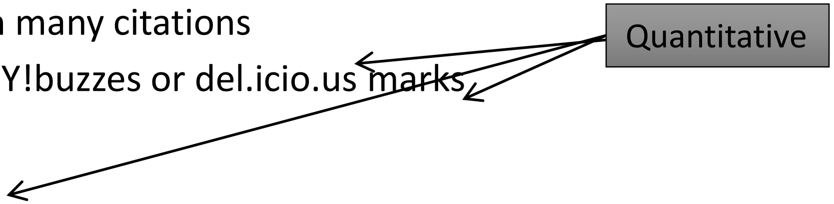
Introduction to
Information
Retrieval

- Static scores
- Cluster pruning
- Number of query terms
- Impact ordering
  - Champion lists
  - Tiered indexes

Static quality scores

Sec. 7.1.4

- 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)



Quantitative

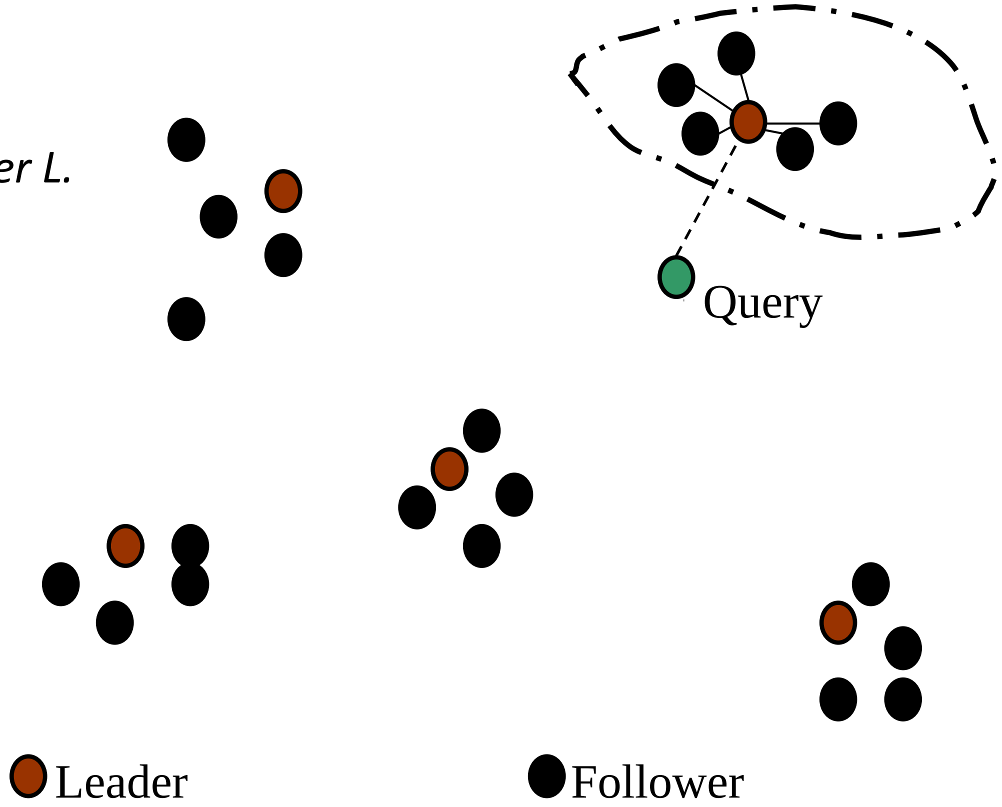
Cluster pruning: preprocessing

Sec. 7.1.6

- Pick √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 ~ √*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

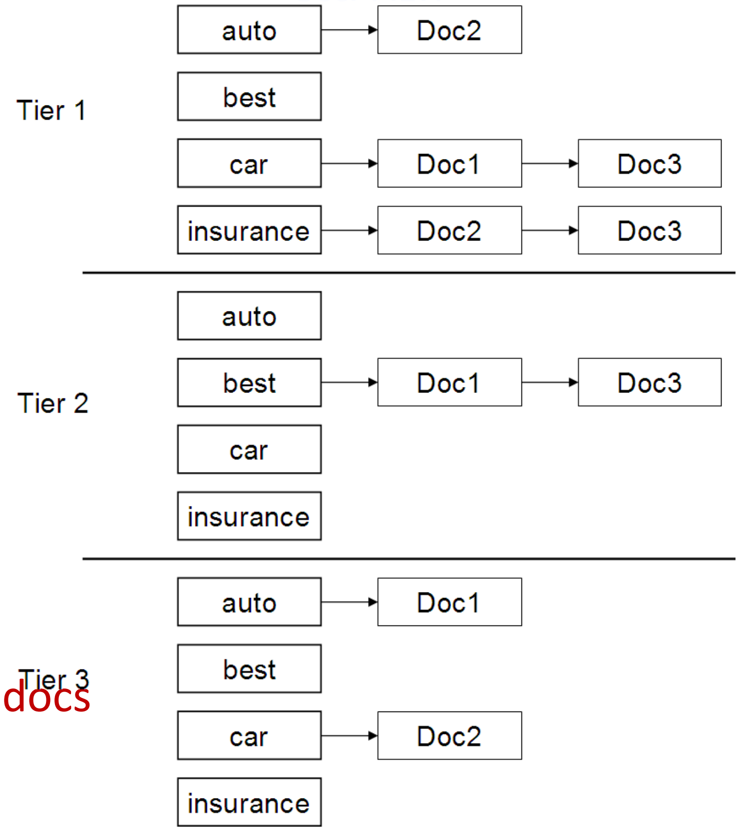
- Any doc with at least one query term is a candidate for the top *K* output list
- For multi-term queries, only compute scores for docs containing several of the query terms
  - Say, at least 3 out of 4
  - Imposes a “soft conjunction” on queries seen on web search engines (early Google)
- Easy to implement in postings traversal

Tiered indexes

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

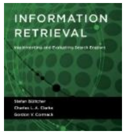
- Inverted index thus broken up into <u>tiers </u>of decreasing importance

- 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

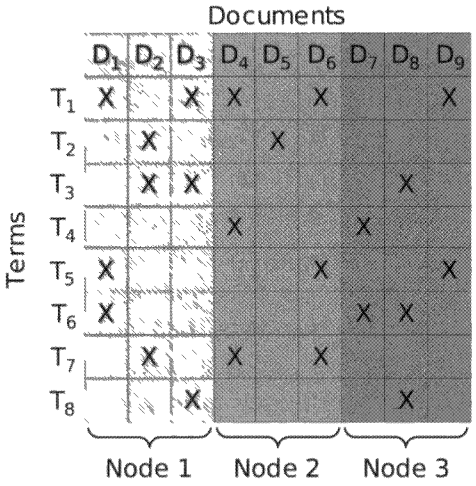


Scalability: Index partitioning

Section 14.1

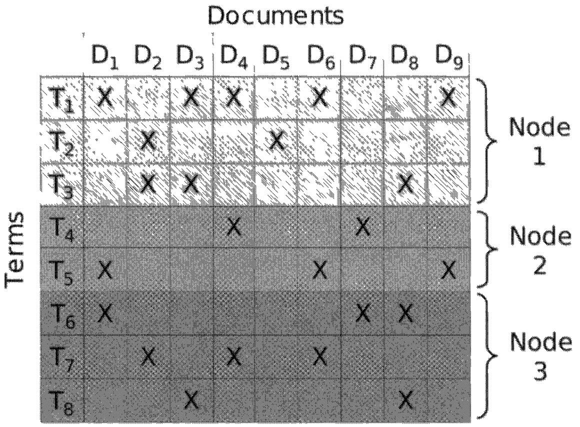


Document partitioning



	D1	D2	D3	D4	D5	D6	D7	D8	D9
T1	X		X	X		X			X
T2		X			X				
T3		X	X					X	
T4				X			X		
T5	X					X			X
T6	X						X	X	
T7		X		X		X			
T8			X					X	

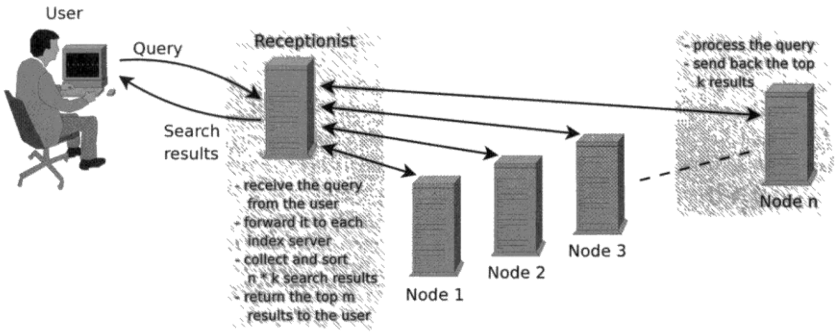
Term partitioning



	D1	D2	D3	D4	D5	D6	D7	D8	D9
T1	X		X	X		X			X
T2		X			X				
T3		X	X					X	
T4				X			X		
T5	X					X			X
T6	X						X	X	
T7		X		X		X			
T8			X					X	

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.

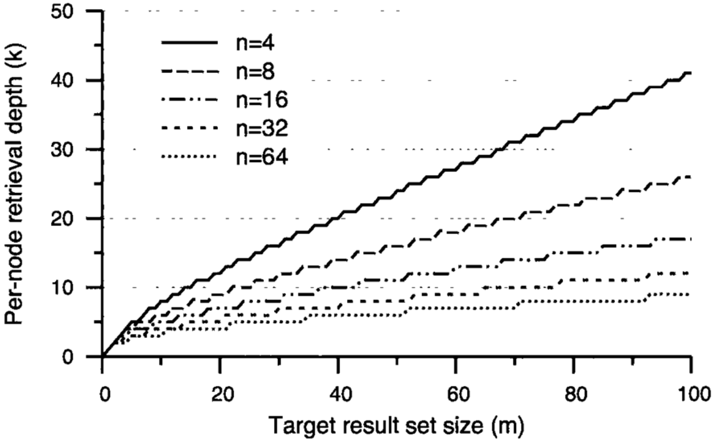
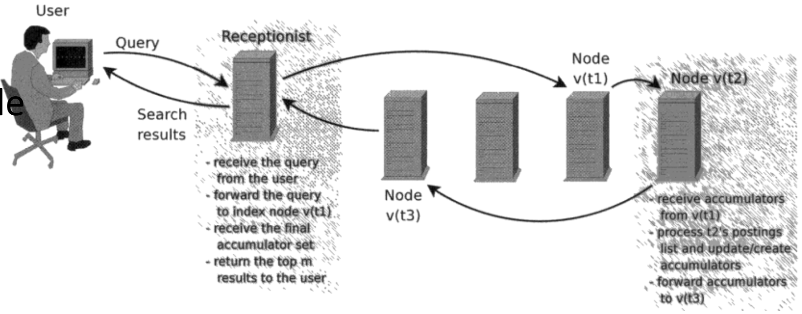


Figure 14.3 Choosing the minimum retrieval depth k that returns the top m results with probability $p(n, m, k) > 99.9\%$, where n is the number of nodes in the document-partitioned index.

Term-partitioning indexes

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

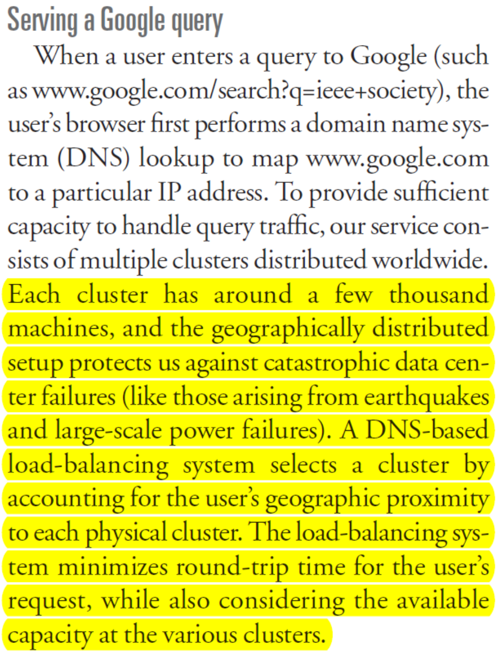
- A query is sent simultaneously to the term’s corresponding nodes.
    - Each node passes its accumulator to the next node or to the central node to compute the final rank.



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



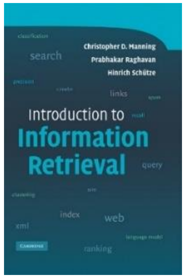
Serving a Google query

When a user enters a query to Google (such as www.google.com/search?q=ieee+society), the user’s browser first performs a domain name system (DNS) lookup to map www.google.com to a particular IP address. To provide sufficient capacity to handle query traffic, our service consists of multiple clusters distributed worldwide. Each cluster has around a few thousand machines, and the geographically distributed setup protects us against catastrophic data center failures (like those arising from earthquakes and large-scale power failures). A DNS-based load-balancing system selects a cluster by accounting for the user’s geographic proximity to each physical cluster. The load-balancing system minimizes round-trip time for the user’s request, while also considering the available capacity at the various clusters.

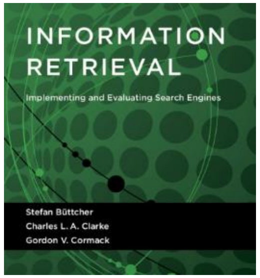
Barroso, Luiz André, Jeffrey Dean, and Urs Hölzle. "Web search for a planet: The Google cluster architecture." IEEE *Micro* (2003)

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