Efficiency vs. Effectiveness in Terabyte-Scale Information Retrieval

Stefan Büttcher Charles L. A. Clarke

University of Waterloo, Canada

November 17, 2005

The Wumpus Search Engine

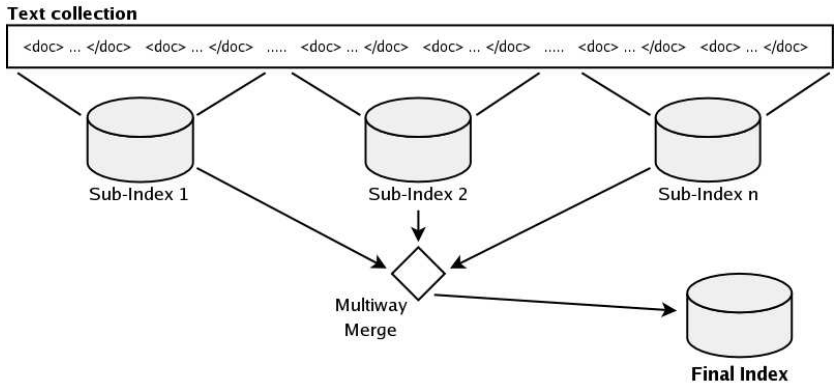
What is Wumpus?

- Multi-user file system search engine.
- Multi-purpose information retrieval system.

What are the main features?

- Multi-user support based on the UNIX security model: Each user can only search files for which she has read permission.
- Fully dynamic: Document insertions and deletions are immediately reflected by the index.
- Publicly available under the terms of the GPL: http://www.wumpus-search.org/

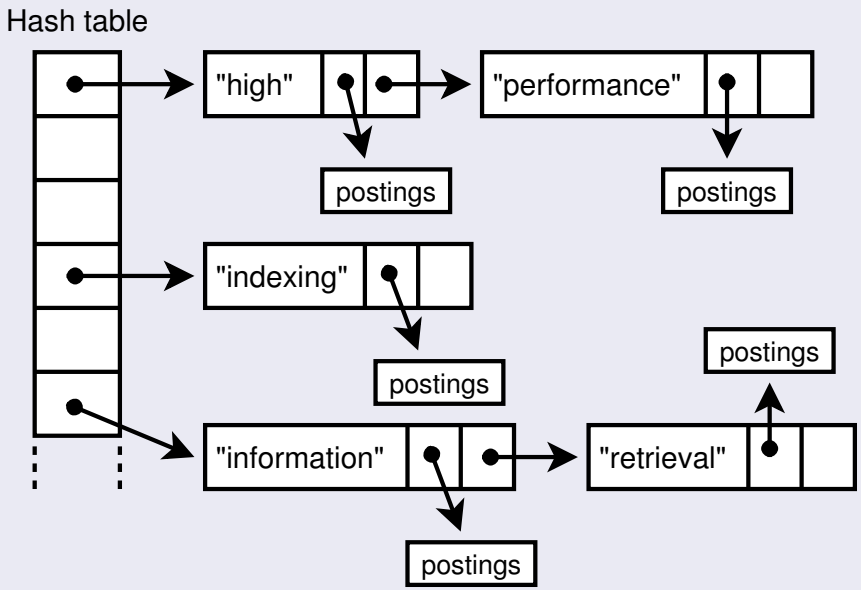
Single-Pass Indexing with Dynamic Partitioning



- Start indexing the text collection; accumulate postings in an in-memory index.
- When memory is exhausted, create on-disk inverted file from in-memory data. Clean memory and continue indexing.
- After the whole collection has been processed, construct final index by merging all on-disk indices created so far.

Hash-Based In-Memory Inversion

Vocabulary terms in the in-memory index are organized in a hash table with linked lists and move-to-front heuristic. Zipf-like term distribution makes hash table reorganizations unnecessary.

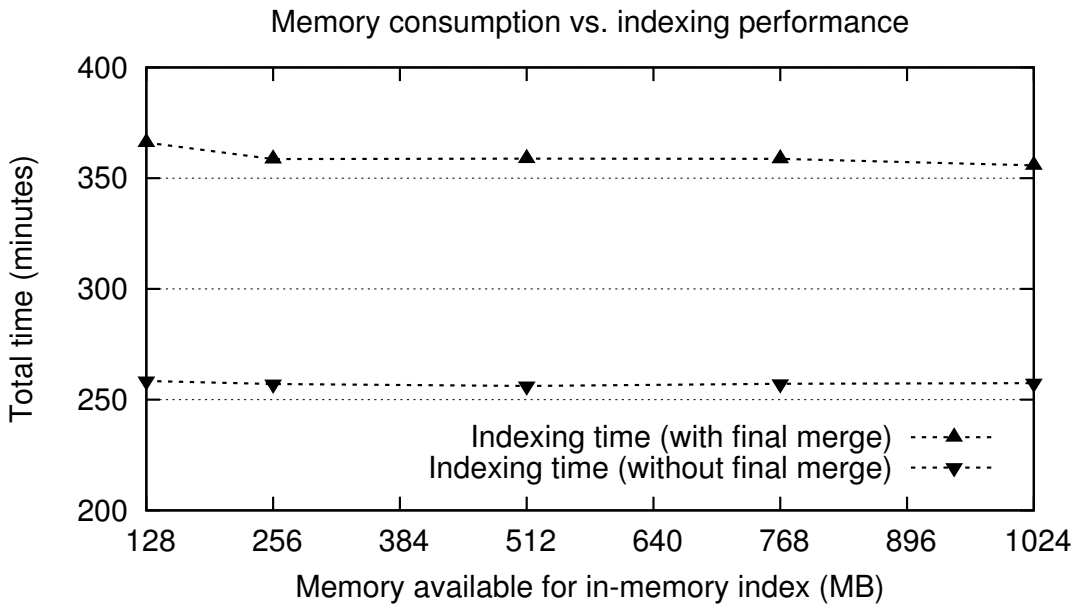


Managing In-Memory Posting Lists

How are the individual posting lists organized?

- List implementation has to be extensible, fast, and space-efficient.
- Solutions in the literature:
  - Fixed-size array (requires two-pass indexing: slow!).
  - Augmentable bit-vector (uses `realloc`: slow!);
  - Linked list (`next` pointers: space overhead!).
- Our solution:
  - Linked list, but linking groups of (compressed) postings instead of individual postings.
  - Adaptive group size, grows exponentially.
  - Very fast and minimal pointer overhead. ≈10% performance gain over `realloc` approach.

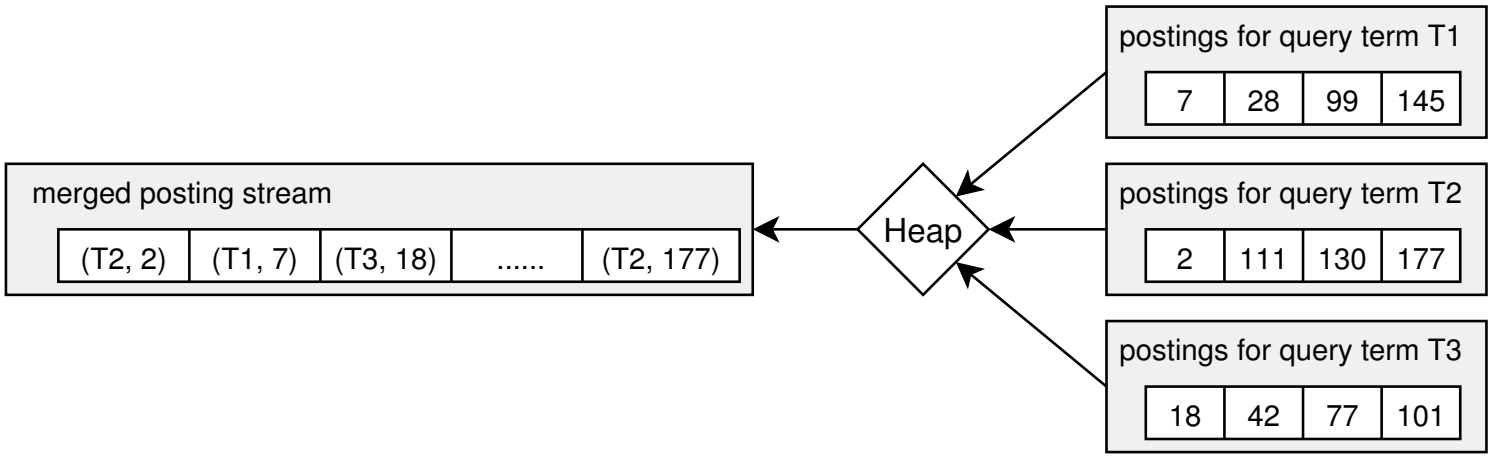
Indexing Performance



Indexing throughput with 1024 MB for the in-memory index:
73.5 GB per hour.

Query Processing: Document-at-a-time

- Posting lists are arranged in a priority queue and merged into a single stream of (term, position) tuples.



- A list of the n top-scoring documents is computed following a document-at-a-time approach with MaxScore heuristic and partial score evaluation (use upper bounds to prune search).

Query Processing: Okapi BM25

Actual document scoring: Okapi BM25

$$S_{\text{BM25}}(D) = \sum_{T \in Q} w_T \cdot \frac{f_{D,T} \cdot (k_1 + 1)}{f_{D,T} + k_1 \cdot (1 - b + b \cdot \frac{dl}{avgdl})}$$

- $w_T = \log(\text{rel. number of docs containing } T$
- $f_{D,T} =$ number of occurrences of T in D
- $dl =$ length of document D (number of tokens)
- $avgdl =$ average document length in the collection

Experimental Results

2005 Ad-hoc Topics

Measure	Value
Precision @ 5	0.6560
Precision @ 10	0.6040
Precision @ 20	0.5470
Average time per query	2.133 sec

Okapi BM25 parameters: $k_1 = 1.2$, $k_3 = \infty$, $b = 0.5$.

Proximity Score Accumulators

Remember that during query processing the posting lists of all query terms are combined through a multiway merge process, giving a stream of (term, position) pairs.

During this process, all pairs of neighboring postings are examined and proximity score accumulators associated with the corresponding query terms are increased.

Updating the Proximity Score Accumulators

$$acc(T_j) := acc(T_j) + w_{T_k} \cdot \frac{1}{(dist(T_j+T_k))^2}$$
$$acc(T_k) := acc(T_k) + w_{T_j} \cdot \frac{1}{(dist(T_j+T_k))^2}$$

Adjusting the Final Document Score

Original document score (BM25):

$$S_{\text{BM25}}(D) = \sum_{T \in \mathcal{Q}} w_T \cdot \frac{f_{D,T} \cdot (k_1 + 1)}{f_{D,T} + K}$$

New Document Score (BM25 + Term Proximity)

$$S_{\text{BM25TP}}(D) = S_{\text{BM25}}(D) + \sum_{T \in \mathcal{Q}} \min\{1.0, w_T\} \cdot \frac{acc(T) \cdot (k_1 + 1)}{acc(T) + K}$$

Limiting the term weight to a maximum of 1.0 keeps the impact of term proximity low and ensures that the original BM25 score still is the dominant component.

Experimental Results

2004 Ad-hoc Topics

Run description	P@5	P@10	P@20	MAP
Pure BM25	0.5184	0.5061	0.4857	0.2477
BM25 + Term proximity	0.5551	0.5612	0.5347	0.2767

2005 Ad-hoc Topics

Run description	P@5	P@10	P@20	MAP
Pure BM25	0.6560	0.6040	0.5470	0.3220
BM25 + Term proximity	0.6200	0.5900	0.5730	0.3371

Document-Level Indexing

Basic idea: For pure BM25, we don't need full positional information in the index.

$\Rightarrow$ During index construction, only keep one posting per (term, document) pair. Encode TF values in the lowest 6 bits of each posting.

Efficiency Gains

Run description	Indexing time	Index size	Query time	MAP
BM25, positional	355 minutes	59.7 GB	2.133 sec	0.3220
BM25, doc. level	245 minutes	19.3 GB	0.326 sec	0.3213

Simple Index Pruning (top-k)

Once we have an index with document-level postings, we can prune the posting lists (using the score associated with each posting).

Pruning in general: Only keep the top-k postings (documents) for each term.

Pruning here: Construct a pruned index containing the top-k postings (documents) for the n most frequent terms.

Goal: To get a pruned index that is small enough to fit into main memory. Then use both indices (unpruned on-disk index; pruned in-memory index) in parallel, fetching posting lists for the n most frequent terms from the in-mem index, everything else from disk.

Threshold-Based Index Pruning (top-k-ε)

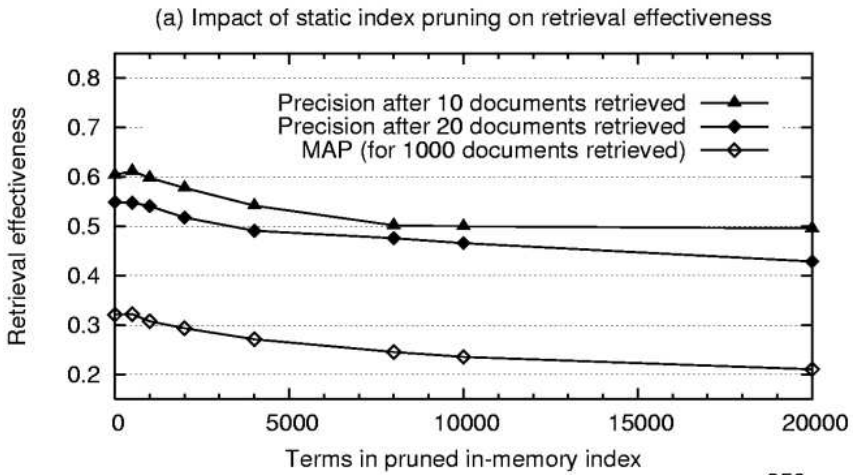
A little more sophisticated:

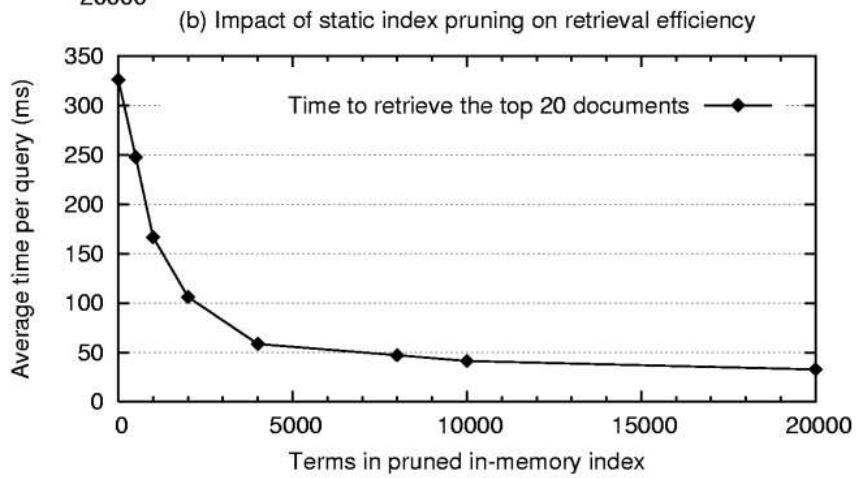
- For every term, pick the k-th best posting (impact: $\mathcal{I}_k$).
- Then throw away all postings whose impact is less than $\varepsilon \cdot \mathcal{I}_k$.

(for details see Carmel et al. at TREC 2001)

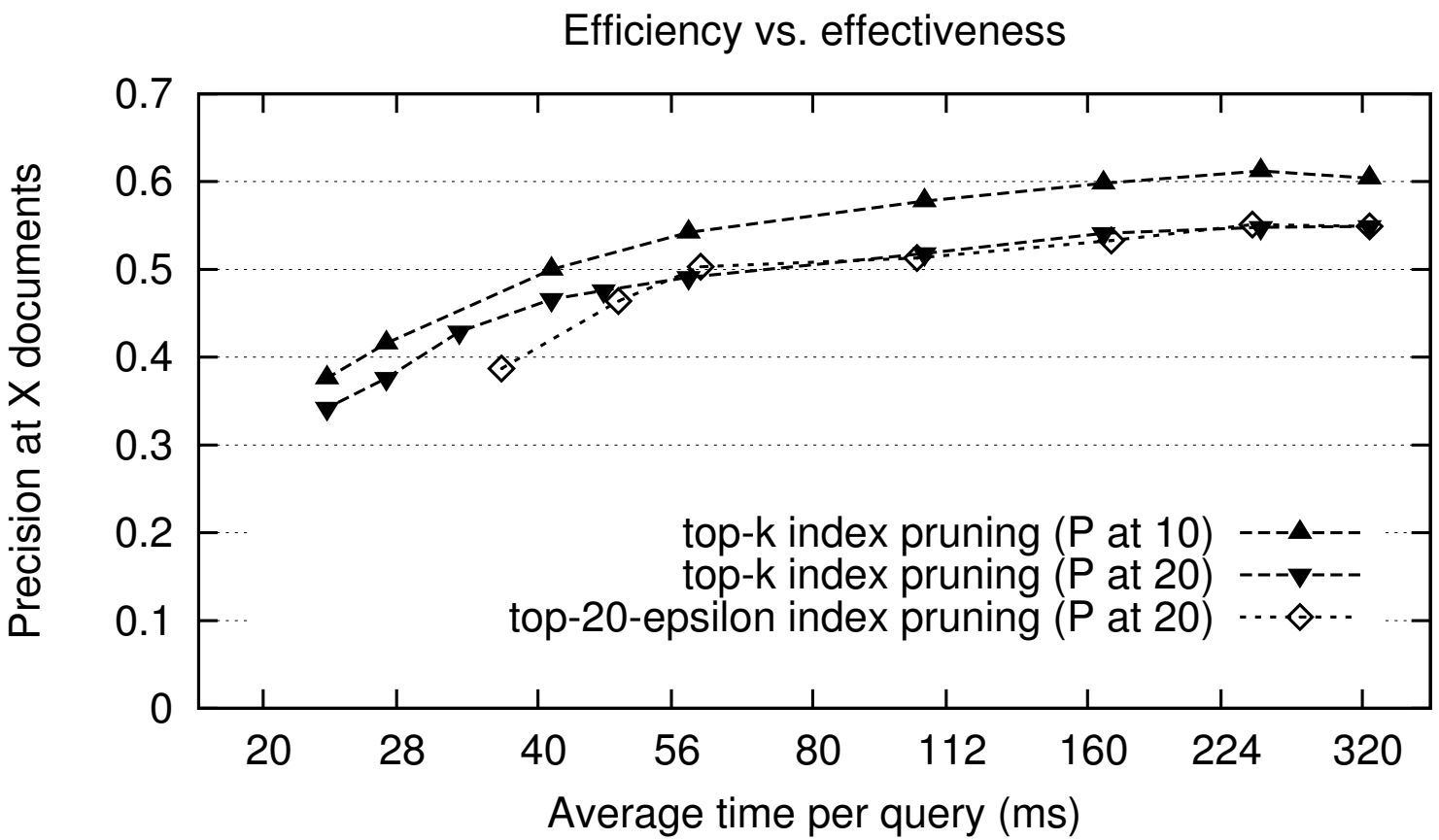
For various values of n, we built indices containing pruned posting lists for the n most frequent terms. The pruning parameters k and ε were chosen in such a way that the size of of the resulting index was $\approx$ 1.2 GB in all cases.

Experimental Results: top-k index pruning

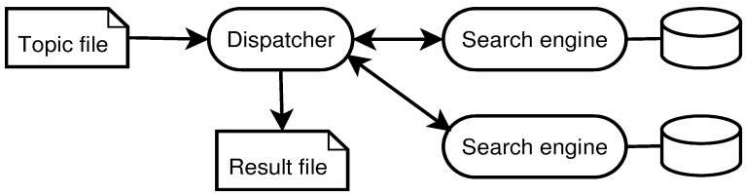




Experimental Results: top-k and top-k-ε index pruning



System Configuration for All Official Runs



System setup:

- 2 PCs running in parallel.
- Each PC: AMD Athlon64 3500+ (2.2 GHz), 2 GB RAM, 7,200-rpm HDD.
- First PC: 1 process (search engine). Second PC: 2 processes (search engine and dispatcher process).
- No term weight propagation (collection big enough).

Runs submitted for the Ad-Hoc Retrieval Task

Run	Avg. query time	P@10	P@20	MAP	bpref
uwmtEwtaP**tdn**	29.464 sec	0.6900	0.6160	0.3480	0.3568
uwmtEwtaP**t**	1.264 sec	0.6320	0.5760	0.3451	0.3526
uwmtEwtaD00**t**	**0.194 sec**	0.6040	0.5650	0.3173	**0.3352**
uwmtEwtaD02**t**	**0.059 sec**	0.5060	0.4490	0.2173	**0.2555**

A 228% performance increase at the cost of a 24% decrease of bpref.

Runs submitted for the Efficiency Retrieval Task

Run	Avg. query time	P@5	P@10	P@20	S@10
uwmtEwtePTP	**1.094 sec**	0.6760	**0.6380**	0.5780	0.9400
uwmtEwtaD00	0.137 sec	0.6000	0.6040	0.5570	0.9600
uwmtEwtaD02	**0.049 sec**	0.4960	**0.4980**	0.4450	0.9400
uwmtEwtaD10	0.027 sec	0.4920	0.4380	0.3900	0.8400

A 22% decrease in P@10 buys 22-fold increase in query processing performance.

Runs submitted for the Named Page Finding Task

Run	Avg. query time	MRR	Success@10	Success@INF
uwmtEwtnP	3.468 sec	**0.366**	**0.508**	**0.894**
uwmtEwtnPpr	3.717 sec	0.322	0.464	0.894
uwmtEwtnD	0.141 sec	**0.290**	**0.413**	**0.663**
uwmtEwtnDpr	0.167 sec	0.238	0.381	0.663

The integration of term proximity increases MRR by 26%, Success@10 by 23%, and Success@INF by 35%.

The End

Thank You!