Efficiency vs. Effectiveness in Terabyte-Scale Information Retrieval

Stefan Büttcher and Charles L. A. Clarke
School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
{sbuettch,claclark}@plg.uwaterloo.ca

Abstract

We describe indexing and retrieval techniques that are suited to perform terabyte-scale information retrieval tasks on a standard desktop PC. Starting from an Okapi-BM25-based default baseline retrieval function, we explore both sides of the effectiveness spectrum. On one side, we show how term proximity can be integrated into the scoring function in order to improve the search results. On the other side, we show how index pruning can be employed to increase retrieval efficiency – at the cost of reduced retrieval effectiveness.

We show that, although index pruning can harm the quality of the search results considerably, according to standard evaluation measures, the actual loss of precision, according to other measures that are more realistic for the given task, is rather small and is in most cases outweighed by the immense efficiency gains that come along with it.

1 Introduction

This paper describes experiments conducted for this year's TREC Terabyte track by members of the information retrieval group at the University of Waterloo. This year, the Terabyte track had 3 different subtasks: ad-hoc retrieval, efficiency, and named-page finding. We participated in all 3 tasks. In this paper, however, we focus exclusively on ad-hoc retrieval and efficiency, as we did not develop any special techniques for named-page finding, but only took our existing ad-hoc retrieval methods and applied them to the named-page finding task.

The ad-hoc and efficiency tasks of the Terabyte track were not really independent tasks, but rather two different aspects of the same ad-hoc retrieval task, measuring both efficiency and effectiveness of the participating search engines. Our goal for this task was to adjust our existing search system so that it was able to

- index half a terabyte of text in less than 10 hours and
- use the resulting index to search the collection with sub-second response times

on a standard desktop PC. We were able to meet both goals.

This paper describes the techniques we employed to achieve these goals. All experiments presented here were conducted using the Wumpus[1] information retrieval system developed at the University of Waterloo. Except where stated otherwise, the system was run on a single PC based on an AMD Athlon64 3500+ processor (2.2 GHz) with 2 GB of RAM and 7,200-rpm SATA harddrives. All efficiency figures are given with respect to the 50 topics from the ad-hoc retrieval task, not the 50,000 topics used in the efficiency task. For all experiments, we used stemmed title-only queries with an average length of 2.9 terms (after stopword removal).

Since the experiments we describe in this paper were conducted after the official runs had been submitted, and our retrieval system experienced major changes after the submission, the experimental results reported here only partially correspond to the official runs. For the exact performance and precision of our official runs, see section 7.

[1] http://www.wumpus-search.org/

2 The Indexing Subsystem

For all experiments, the index was constructed using a single-pass procedure with dynamic corpus partitioning. The general strategy is to accumulate postings in an in-memory index until the available main memory is exhausted. At that point, an on-disk index is created from the data in memory, a fresh in-memory index is created, and the system continues to index the text collection. This process is repeated until the whole collection has been indexed. Eventually, all on-disk sub-indices that have been created during this process, are brought together through a multiway merge process, resulting in the final index. The sub-indices are then deleted.

The inversion strategy we employed for creating the individual sub-indices can be described as hash-based in-memory inversion: For every input token (<*term, position*> pair) that is read from the text collection, the corresponding term descriptor is looked up in a hash table (inserted if it is not present yet), and the position is added to the term's list of postings. Term descriptors inside the hash table are organized in linked lists, using a move-to-front heuristic [ZHW01].

Because the total number of occurrences of a term within the text corpus is not known beforehand, an efficient, dynamic data structure is needed to maintain the postings for a term. Usually, linked lists are used for this purpose. The disadvantage of linked lists is that a large amount of space is wasted by pointers in the list. Another approach is to use relocatable bit vectors. This avoids the space overhead introduced by the `next` pointers in the linked lists, but necessitates frequent reallocations, which can become a performance problem.

Our approach is to use linked lists, but to link groups of postings instead of individual postings. This strategy does not only improve memory efficiency by reducing the relative number of pointers needed, but it also increases locality by keeping postings for the same term together. This drastically decreases the number of CPU cache misses. In addition, all postings are stored in compressed form in the in-memory index, allowing for even better memory efficiency and thus for a smaller number of sub-indices that have to be combined in the final merge process.

A detailed description of our indexing strategy is given by Büttcher and Clarke [BC05]. Compared to the relocatable-bitvector approach, indexing can be performed about 10% faster, since no relocations are necessary. Using this method, our system is able to create an index of the 426-GB GOV2 collection, containing full positional information for all index terms, in 356 minutes on a standard desktop PC. This represents an indexing throughput of 72 GB per hour. At this speed, transferring data from/to hard disk is becoming a major bottleneck.

P@10	P@20	MAP	bpref	MRR
0.6100	0.5460	0.3227	0.3390	0.7764

Table 1: Effectiveness of the baseline retrieval method, Okapi BM25 ($k_1 = 1.2$, $b = 0.5$).

3 Baseline Retrieval Method

The baseline retrieval method for all our experiments is the Okapi BM25 formula [RWJ+94] [RWB98]. Given a query $\mathcal{Q} = \{T_1, \ldots, T_n\}$, BM25 assigns to a document D the relevance score

$$S^{(D)}_{\text{BM25}} = \sum_{i=1}^{n} w_{T_i} \cdot \frac{(k_1 + 1) \cdot f_{D,T_i}}{f_{D,T_i} + k_1 \cdot ((1 - b) + b \cdot \frac{|D|}{avgdl})}, \tag{1}$$

where f_{D,T_i} is the number of occurrences of T_i within D, $|D|$ is the length of the document D (number of tokens), and *avgdl* is the average document length in the text collection. w_{T_i} is T_i's inverse document frequency:

$$w_{T_i} = \log \left(\frac{\#\text{documents}}{\#\text{documents containing } T_i} \right). \tag{2}$$

k_1 and b are free parameters and were chosen to be $k_1 = 1.2$ and $b = 0.5$. These values are the results of preliminary experiments using the 50 ad-hoc topics from the 2004 TREC Terabyte track. They are in line with results reported by Plachouras et al. [PHO04] for the same collection.

Even though all of our runs differ from this version of the BM25 formula, the differences are not material, and the retrieval methods used in our experiments can be thought of as minor modifications to the original BM25 algorithm.

Query processing is performed following a document-at-a time approach that arranges the posting lists for all terms in a priority queue and traverses the merged list of postings in ascending order, computing document scores on-the-fly. A pruning strategy similar to Turtle and Flood's *MaxScore* optimization [TF95] is employed to reduce the computational cost by allowing partial document score evaluations when searching for the top-k documents to be returned.

HTML Field(s)	Boosting factor
<title>	6
<h1>	4
<h2>, <b>, , and <u>	3
<i> and	2

Table 2: Exploiting document structure – boosting factors for terms appearing in certain HTML fields.

Run description	P@10	P@20	MAP
Pure BM25	0.6100	0.5460	0.3227
BM25+doc.structure	0.5880	0.5480	0.3117

Table 3: Impact of using document structure on search precision. No improvements are achieved by giving special treatment to certain HTML fields.

4 Exploiting the Structure of Documents

Many documents, such as HTML documents, provide a rich structure that can be exploited in order to obtain better search results. Cutler et al. [CSM97] showed that using the document structure in order to assign higher weights to terms that appear in the document title, for example, increases the effectiveness of the search system. We followed their approach and boosted all terms that appeared within certain fields by pretending there were n occurrences instead of the one that actually was there. This is essentially BM25 extension proposed by Robertson et al. [RZT04] (without their adjustment of the k_1 parameter). The exact values of n, for different HTML fields, are shown in Table 2.

The evaluation, in terms of P@10, P@20, and MAP, shows that in our experiments the use of document structure did not improve the search results at all (cf. Table 3). This is somewhat surprising, since the boosting values were determined through a training process using the 50 ad-hoc topics from the 2004 Terabyte track and produced a 5-10% improvement on the 2004 data.

5 Document Retrieval Using Term Proximity

Rasolofo and Savoy [RS03] showed that integrating term proximity into the BM25 scoring function can improve overall retrieval effectiveness of the search system. Our implementation of term proximity scoring is similar to that proposed in their paper.

Suppose a user submits a query $\mathcal{Q} = \{T_1, \ldots, T_n\}$ to the search engine. Then our implementation of BM25 fetches the posting lists for all query terms from the index and arranges them in a priority queue. It then starts consuming postings from all posting lists, one posting at a time, in ascending order, to find matching documents and simultaneously compute the relevance scores of all matching documents found.

Method	P@20	MAP	Avg. query time
BM25	0.5460	0.3227	2.133 sec
BM25TP	0.5730	0.3377	2.137 sec

Table 4: Search performance and retrieval effectiveness for pure BM25 and the proximity-enhanced BM25TP algorithm.

If the underlying index contains full positional information, term proximity can be integrated into this process without much effort. With every query term, we associate an accumulator that contains that term's proximity score within the current document. Whenever the search system encounters a posting that belongs to the query term T_j, it looks at the previous posting, belonging to the query term T_k, and determines the distance (number of postings) between the current posting and the previous one. If $T_j \neq T_k$, then both terms' accumulators are incremented:

$$\begin{aligned} acc(T_j) &:= acc(T_j) + w_{T_k} \cdot \frac{1}{(dist(T_j + T_k))^2}, \\ acc(T_k) &:= acc(T_k) + w_{T_j} \cdot \frac{1}{(dist(T_j + T_k))^2}, \end{aligned}$$

where w_{T_i} is T_i's IDF weight (cf. equation 1). For $T_j = T_k$, the accumulator contents remain unchanged. When the end of the current document is reached, the document's score is computed, and all proximity accumulators are reset to zero. The score of a document D is:

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

where k_1 and K are the usual BM25 parameters. The main difference to the strategy followed by Rasolofo and Savoy is that in our approach only neighboring query terms can affect each other's accumulator, which allows for slightly faster query processing.

Query processing performance and average precision for both BM25 and BM25TP are given by Table 4. Since, during query processing, most time is spent examining documents that contain only a single query term, the slowdown caused by taking term

Indexing method	Index size	Total indexing time	Indexing throughput	Time per query
Positional indexing	59.7 GB	356 minutes	72 GB/h	2.133 seconds
Document-level indexing	19.3 GB	245 minutes	104 GB/h	0.326 seconds

Table 5: Performance comparison: Positional indexing vs. document-level indexing.

proximity into account is negligible. On the other hand, by integrating term proximity, both P@20 and MAP can be increased by about 5% over an already very high baseline.

6 Document-Level Indexing & Static Index Pruning

In its standard configuration, the Wumpus search engine creates an index containing full positional information. This is necessary to support certain query types, such as phrase queries or the proximity-based retrieval method described in the previous section. For document retrieval tasks like the ad-hoc retrieval task of this year's Terabyte TREC, however, positional information is not absolutely necessary. In fact, the BM25 retrieval function in its pure form does not even make use of positional information. All the information needed to rank documents according to their BM25 score is the number of times each query term appears in each document. By restricting the index so that it only contains this information, a great improvement in terms of both space and time complexity can be achieved.

Since our whole retrieval system is built around term positions, and documents are represented by their start and end position in the text collection, changing the system towards a document-based retrieval paradigm would have been too much work and would have been difficult to integrate into the existing framework. Instead, we chose to encode the number of times a term appears in a document in its position within that document. The within-document term frequency of a term is encoded in the 6 least significant bits of each posting. This caused some problems for documents shorter than 64 tokens. However, these documents tended to be framesets or redirections, and there were only 6,055 such documents in the collection anyway.

Whenever a term appeared less than 32 times in a document, the exact number of occurrences was encoded in that term's posting for the document in question. When it had more than 32 occurrences, the within-document frequency of the term was encoded approximately using a logarithmic encoding scheme (with base 1.1). Thus, the maximal TF value encodable by our system was $32 \cdot 1.1^{31} = 614$, which was sufficient for most documents.

The effects of document-level indexing on indexing performance, index size, and query processing performance are shown in Table 5. The total size of the on-disk data structures drops by 68%, the time needed to build the index decreases by 33%, and the average time per query is reduced by 85%. The gains achieved by omitting positional information come along with a small loss of precision, caused by the slight inaccuracies introduced by our term frequency encoding method and the inability to index certain short documents.

Index Pruning

In addition to using a document-level index instead of full positional information, we modified the final merge process that brings together all sub-indices created during index construction and joins them into one big index. Instead of creating one single index, two indices were created – a small one, holding postings for the most frequent terms in the collection, and a bigger one, containing the postings for all other terms. Furthermore, the small index, containing the n most frequent terms, was pruned in such a way that, for every term T, the index only contained postings for the k documents in which T's impact on the BM25 score of that document was greatest.

In other words, for each term T among the n most frequent terms in the collection and each document D it appeared in, D's score for the query $\mathcal{Q} = \{T\}$ was computed:

$$S^{(D)}_{\text{BM25}} = w_T \cdot \frac{(k_1 + 1) \cdot f_{D,T}}{f_{D,T} + k_1 \cdot ((1 - b) + b \cdot \frac{|D|}{avgdl})} \quad (3)$$

(cf. equation 1). For each term T, only the k documents with highest score S_D were kept in the index. The values of n and k were chosen in such a way that the size of the pruned index was about 1.2 GB ($n \cdot k \approx 5 \cdot 10^8$). This way, it was possible to keep the small index in memory and to use both the in-memory index and the bigger on-disk index in parallel during query processing.

This procedure is similar to the static index pruning method described by Carmel et al. [CCF+01] [CAH+01], but in contrast to their approach, it creates a pruned index that is much smaller than the

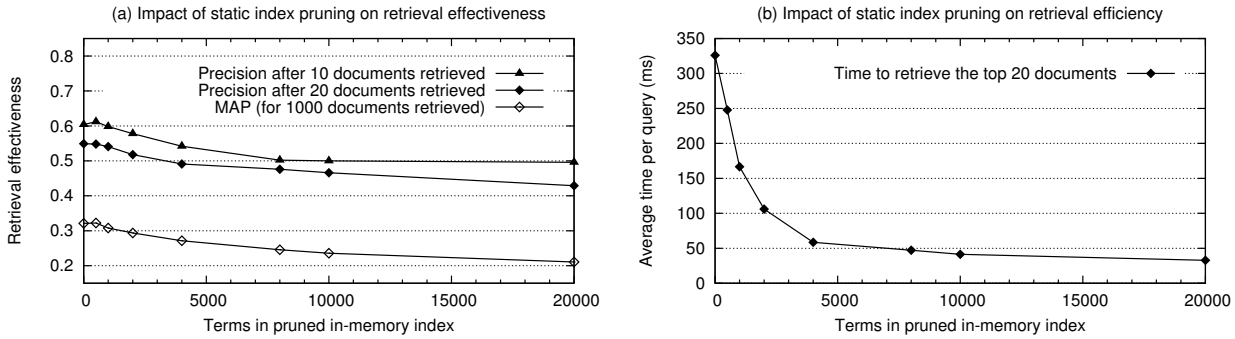

Figure 1: Impact of static index pruning on retrieval efficiency and effectiveness. The figures represent pruned in-memory indices containing 500,000,000 postings (roughly 1.2 GB). The number of distinct terms in the in-memory index is varied between 0 and 20,000. The performance gains that can be achieved are likely to outweigh the loss of precision they entail.

original index ($\approx 6\%$) and uses both indices in parallel when processing a search query.

Figure 1 presents the impact that different values for n and k have on query processing performance and retrieval effectiveness. It shows that tremendous performance gains can be achieved by pruning the index in the way described above. On the other hand, retrieval effectiveness drops as the number of terms n in the pruned in-memory index is increased. However, for early precision measures, such as P@10, the effect is not as strong as for other measures (e.g., MAP). For instance, using a pruned in-memory index with 2,000 terms, containing the top 250,000 documents for each term, decreases MAP by 8.6% (from 0.3213 to 0.2937), but P@10 only by 4.3% (from 0.6040 to 0.5780). At the same time, the average search time per query drops by 67.5% – from 326 ms down to 106 ms. This suggests the use of different retrieval methods, depending on whether the user only requested the top 10 or 20 documents or a deeper result set. For example, a pruned index could be used to produce the first page of search results, while the full index is used to generate all subsequent result pages.

7 Official Runs

We submitted 4 runs for the ad-hoc retrieval task of the Terabyte track and 4 runs for the efficiency task. In contrast to the experiments described in the previous sections, which were conducted on a single PC, all our official runs were performed on two PCs with AMD Athlon64 3500+ processors (2.2 GHz) running in parallel, with each machine having an index covering 50% of the whole collection. The subcollections were big enough to make term weight propagation between the two indices unnecessary.

For the ad-hoc retrieval tasks, we submitted:

uwmtEwtaPt Title-only run using a full positional index and term proximity scoring.

uwmtEwtaPtdn Title+description+narrative run using a full positional index and term proximity scoring. Relative term weights were 0.7 (title), 0.2 (description), and 0.1 (narrative).

uwmtEwtaD00t Title-only run using a document-level index without index pruning.

uwmtEwtaD02t Title-only run using a document-level index with index pruning. The pruned in-memory index contained the 2,000 most frequent terms ($n = 2000$, $k = 250000$).

For the efficiency task, we submitted:

uwmtEwtePTP Title-only run using a full positional index and term proximity scoring.

uwmtEwteD00 Title-only run using a document-level index without index pruning.

uwmtEwteD02 Title-only run using a document-level index with index pruning. The pruned in-memory index contained the 2,000 most frequent terms ($n = 2000$, $k = 250000$).

uwmtEwteD10 Title-only run using a document-level index with index pruning. The pruned in-memory index contained the 10,000 most frequent terms ($n = 10000$, $k = 50000$).

All 8 runs submitted are summarized in tables 6 and 7. The precision differences between the runs shown in the tables and the experiments described in the previous sections of this paper are due to changes made to our search engine:

Name	Total indexing time	Average time per query	P@10	P@20	MAP
uwmtEwtaP**t**	208 minutes	1.264 seconds	0.6320	0.5760	0.3451
uwmtEwtaP**tdn**	208 minutes	29.464 seconds	0.6900	0.6160	0.3480
uwmtEwtaD00**t**	132 minutes	0.194 seconds	0.6040	0.5650	0.3173
uwmtEwtaD02**t**	147 minutes	0.059 seconds	0.5060	0.4490	0.2173

Table 6: Runs submitted for the TREC Terabyte ad-hoc retrieval task. For all runs shown in this table, both indexing and query processing was performed on two PCs running in parallel.

Name	Total indexing time	Average time per query	P@5	P@10	P@20
uwmtEwtePTP	208 minutes	1.094 seconds	0.6760	0.6380	0.5780
uwmtEwteD00	132 minutes	0.137 seconds	0.6000	0.6040	0.5570
uwmtEwteD02	144 minutes	0.049 seconds	0.4960	0.4980	0.4450
uwmtEwteD10	147 minutes	0.027 seconds	0.4920	0.4380	0.3900

Table 7: Runs submitted for the TREC Terabyte efficiency task. For all runs shown in this table, both indexing and query processing was performed on two PCs running in parallel.

- We changed the document tokenizer. Apart from a few exceptions (e.g., `<meta>` tags), our new tokenizer ignores all attribute values inside HTML tags. While this improves indexing performance, it decreases retrieval effectiveness. We did not perform any additional experiments in order to find out the exact reason for this behavior, but we surmise that omitting `alt` attributes of `<img>` is not a good idea.
- Following the TREC tradition, we found a bug in our index pruning implementation the day after we had submitted the official runs. Fixing this bug increased the precision numbers for our pruned runs substantially and significantly.

References

[BC05] Stefan Büttcher and Charles L. A. Clarke. Memory Management Strategies for Single-Pass Index Construction in Text Retrieval Systems. University of Waterloo Technical Report CS-2005-32, October 2005.

[CAH+01] David Carmel, Einat Amitay, Michael Herscovici, Yoelle S. Maarek, Yael Petruschka, and Aya Soffer. Juru at TREC 10 - Experiments with Index Pruning. In *Proceedings of the 10th Text REtrieval Conference (TREC 2001)*, November 2001.

[CCF+01] D. Carmel, D. Cohen, R. Fagin, E. Farchi, M. Herscovici, Y. Maarek, and A. Soffer. Static Index Pruning for Information Retrieval Systems. In *Proceedings of the 24th ACM SIGIR Conference (SIGIR 2001)*, pages 43–50, 2001.

[CSM97] Michael Cutler, Yungming Shih, and Weiyi Meng. Using the Structure of HTML Documents to Improve Retrieval. In *USENIX Symposium on Internet Technologies and Systems*, 1997.

[PHO04] Vassilis Plachouras, Ben He, and Iadh Ounis. University of Glasgow at TREC 2004: Experiments in Web, Robust and Terabyte tracks with Terrier. In *Proceedings of the 13th Text REtrieval Conference (TREC 2004)*, November 2004.

[RS03] Yves Rasolofo and Jacques Savoy. Term Proximity Scoring for Keyword-Based Retrieval Systems. In *Proceedings of the 25th European Conf. on Information Retrieval (ECIR 2003)*, pages 207–218, April 2003.

[RWB98] S. Robertson, S. Walker, and M. Beaulieu. Okapi at TREC-7. In *Proceedings of the Seventh Text REtrieval Conference (TREC 1998)*, November 1998.

[RWJ+94] S. Robertson, S. Walker, S. Jones, M. Hancock-Beaulieu, and M. Gatford. Okapi at TREC-3. In *Proceedings of the Third Text REtrieval Conference (TREC 1994)*, November 1994.

[RZT04] Stephen Robertson, Hugo Zaragoza, and Michael Taylor. Simple BM25 Extension to Multiple Weighted Fields. In *Proceedings of the 13th ACM Conference on Information and Knowledge Management (CIKM 2004)*, pages 42–49, New York, USA, 2004.

[TF95] Howard Turtle and James Flood. Query Evaluation: Strategies and Optimization. *Information Processing & Management*, 31(1):831–850, November 1995.

[ZHW01] Justin Zobel, Steffen Heinz, and Hugh E. Williams. In-Memory Hash Tables for Accumulating Text Vocabularies. *Information Processing Letters*, 80(6), 2001.