Real-time Text Queries with Tunable Term Pair Indexes

Andreas Broschart Ralf Schenkel

MPI–I–2010–5-006 October 2010

Authors' Addresses

Andreas Broschart
Max-Planck-Institut für Informatik und
Universität des Saarlandes
Campus E 1 7

D-66123 Saarbrücken

Ralf Schenkel
Max-Planck-Institut für Informatik und
Universität des Saarlandes
Campus E 1 7

D-66123 Saarbrücken

Abstract

Term proximity scoring is an established means in information retrieval for improving result quality of full-text queries. Integrating such proximity scores into efficient query processing, however, has not been equally well studied. Existing methods make use of precomputed lists of documents where tuples of terms, usually pairs, occur together, usually incurring a huge index size compared to term-only indexes. This paper introduces a joint framework for trading off index size and result quality, and provides optimization techniques for tuning precomputed indexes towards either maximal result quality or maximal query processing performance under controlled result quality, given an upper bound for the index size. The framework allows to selectively materialize lists for pairs based on a query log to further reduce index size. Extensive experiments with two large text collections demonstrate runtime improvements of several orders of magnitude over existing text-based processing techniques with reasonable index sizes.

Contents

1 Introduction

1.1 Motivation

The proximity of query terms is often used to improve result quality for keyword-based text retrieval over scores that simply consider the frequency of terms in documents; Section 2 gives an extensive overview of proximity-aware scores. In addition to standard content-based scores, proximity scores reward occurrences of query terms within short distance in the same document with higher scores. As an example, consider the query 'efficient query processing with term pairs'. With standard content scores, a document that talks about 'efficient query processing' in one section and about 'term pairs' in another section could be falsely identified as good result. On the other hand, phrase queries are often too strict and rule out many relevant results (that talk, for example, about 'pair-based efficient processing of term queries'). The soft and automatic phrase queries provided by proximity scores can help in this situation.

The integration of such proximity scores into efficient query processing algorithms for quickly computing the best k results, however, has not been equally well studied. Existing methods like [9, 19, 22] make use of precomputed lists of documents where tuples of terms, usually pairs, occur together, usually incurring a huge index size compared to term-only indexes, or focusing on conjunctive queries only. Unlike and orthogonally to existing techniques for lossy index compression in this scenario that materialize only a subset of all term pair lists, this paper aims at limiting the size of each pair list by limiting the maximal list size and imposing a minimal proximity score per list. At the same time, the choice of term pair index lists to be materialized can be based on frequent queries in a query log. Our method can be tuned towards either guaranteeing maximal result quality or maximal query performance at controlled result quality within a given index size constraint. For both optimization goals, the result of the method are pruned index lists of a fixed maximal length, which means that the worst-case cost for evaluating a query with this index can be tightly bound as well. In our

experiments with the GOV2 collection (reported in Section 8), we show that 310 entries per list can be enough to give the same result quality as a standard score taking only term frequencies into account, which results in an average cold-cache retrieval time of less than 110 ms (warm cache less than 1 ms) for a standard query load, with approximately 12KB of data being read per query. In this configuration, the size of the compressed index is 95GB, only slightly larger than the compressed collection. Similar query processing costs can be achieved for much larger collections, such as the recent ClueWeb collection.

1.2 Contributions

This paper makes the following important contributions:

- It introduces a tunable indexing framework for terms and term pairs for optimizing index parameters towards either maximal result quality or maximal query processing performance under result quality control, given a maximal index size.
- It allows for a selective materialization of term pair index lists based on information from a query log.
- It transparently supports index compression, and presents a proof-of-concept implementation of a compressed index.
- The resulting indexes provide dependable query execution times while providing result quality comparable to or even better than unpruned text indexes.
- It experimentally demonstrates that the resulting index configurations allow for query processing that is several orders of magnitudes cheaper than existing text-based techniques in terms of execution cost and runtime while yielding results of at least comparable quality.

1.3 Outline of the paper

The remainder of the paper is structured as follows. Section 2 reviews related work, Section 3 introduces the proximity score used in this paper, Section 4 elaborates on the index organization and the employed index compression techniques, Section 5 presents the index tuning framework, Section 6 details the simple yet efficient query processing algorithm, Section 7 shows how the size of the index

can be reduced further using a query log, and Section 8 experimentally evaluates our methods with two large text collections from TREC, namely GOV2 and ClueWeb.

2 Related Work

Term proximity has been increasingly used recently to improve result quality for term queries without phrases, for example in [4, 5, 11, 17, 18, 20, 22, 24, 25, 32]. While some of these techniques demonstrate significant improvements in result quality, they do not consider the problem how these scores can be efficiently implemented in a search engine. Usually, implementations therefore resort to enriching term index lists with position information (e.g., [30]) and compute proximity scores after having determined an initial set of documents with 'good' text scores (e.g., [20]). One of the first approaches to integrate proximity scores as an integral part of query processing has been introduced by [22] which showed that proximity scores can not only improve result quality, but also efficiency. Our paper builds on results from there, but extends it towards a configurable indexing framework which can be tuned either for maximal and dependable query performance under result quality control or for maximal result quality.

On the other hand, there has been a noticeable amount of work using precomputed lists for documents containing two or more terms to speed up processing of conjunctive queries, for example [9, 15, 16] for centralized search engines and [19] for distributed search engines. None of these approaches includes proximity scores, so they can only improve processing performance, not result quality. Another bunch of papers deals with efficiently precomputing indexes for phrase queries [2, 8, 29], but again they do not include proximity scores. Some of these consider the problem of reducing the index size while providing decent performance for most queries, usually by restricting to phrases or term pairs in frequently occurring queries.

Processing of non-conjunctive queries with ranking has been dominated so far by highly efficient *top-k* or *dynamic pruning* algorithms [1, 13, 14, 33]. They access precomputed index lists where index entries are sorted in descending order of score. They incrementally read entries from these lists, maintaining partial scores for documents found during this process and a current estimate for the top-k results, consisting of the k documents which currently have the highest partial score. The algorithms additionally maintain upper bounds for the maximal score

any non-top-k item can get, including documents that have not yet been seen, by combining partial scores with current high score bounds in lists where a document has not yet been encountered. The algorithms can safely stop when no non-top-k result can get a final score that is above of the score of any document currently in the top-k. There are numerous extensions of this baseline algorithm, like adding random accesses to the score of selected items [7], probabilistic result pruning [26], clever access scheduling [3], execution with limited budget [23], or list organization based on term impacts [1].

A very recent approach to use term pair indexes for improving bounds in top-k text retrieval was presented in [31], which however focuses on distributed evaluation of queries in a cluster of machines.

3 Proximity Scoring

This section gives a short introduction of the proximity score used throughout the paper, which has been introduced in [22] as a modified version of the score developed by Büttcher et al. [4, 5]. We consider a fixed collection C of text documents. For a document $d \in C$, we denote by the term frequency $tf_d(t)$ the number of times term t occurs in d, and the length $l(d) = \sum tf_d(t)$ of document d is the sum of the term frequencies of all terms it contains. The most established content or term-only score in text retrieval is the BM25 score [21], which is computed for a query $q = \{t_1, \ldots, t_n\}$ of terms as

$$score_{\text{BM25}}(d, q) \quad = \quad \sum_{t \in q} \frac{tf_d(t) \cdot (k_1 + 1)}{tf_d(t) + k_1 \cdot (1 - b + b\frac{l(d)}{avgdl})} \cdot idf(t)$$

where k_1 and b are tunable parameters, $avgdl$ is the average length of all documents in the collection, and $idf(t)$ is the inverse document frequency of t in the collection: Denoting by $df(t)$ the number of documents in which t occurs and by N the number of documents in C, $idf(t)$ is defined as

$$idf(t) = \log \frac{N}{df(t)} \tag{3.1}$$

We denote by $p_i(d)$ the term at position i of d, omitting d when the document is clear from the context. For a term t, $P_d(t)$ denotes the positions in d where t occurs. For a query $q = \{t_1, \ldots, t_n\}$, $P_d(q) := \cup_{t_i \in q} P_d(t_i)$ denotes the positions of those terms in d, and

$$Q_d(q) := \{(i, j) \in P_d(q) \times P_d(q) \mid i < j \wedge p_i \neq p_j\}$$

denotes the position pairs of distinct terms from q in d. The score for a query $q = \{t_1, \ldots, t_n\}$ is then a linear combination of a standard BM25 content score and a BM25-style proximity score where term frequencies are replaced by per-

term accumulators acc':

$$\begin{aligned} score_{\text{Büttcher}}(d,q) &= score_{\text{BM25}}(d,q) \\ &\quad + \sum_{t \in q} min\{1, idf(t)\} \frac{acc'_d(t) \cdot (k_1+1)}{acc'_d(t)+1} \end{aligned}$$

Here, the accumulator for term $t_k \in q$ is defined as

$$\begin{aligned} acc'_d(t_k) &= \sum_{(i,j) \in Q_d(q): p_i = t_k} \frac{idf(p_j)}{(i-j)^2} \\ &\quad + \sum_{(i,j) \in Q_d(q): p_j = t_k} \frac{idf(p_i)}{(i-j)^2} \end{aligned}$$

This score shows two major differences from the original score developed by Büttcher et al.: (1) it does not include the document length in the proximity score, and (2) accumulators combine not only adjacent query term occurrences. It has been shown in [22] that these modifications do not have an impact on result quality, but allow for efficient precomputation and indexing. A simple reformulation of the definition of $acc'_d(t_k)$ yields

$$\begin{aligned} acc'_d(t_k) &= \sum_{t \in q} idf(t) \cdot \underbrace{\sum_{\substack{(i,j) \in Q_d(q): \\ (p_i = t_k, p_j = t) \\ \vee (p_i = t, p_j = t_k)}} \frac{1}{(i-j)^2}}_{:= acc_d(t_k, t)} \\ &= \sum_{t \in q} idf(t) \cdot acc_d(t_k, t) \end{aligned}$$

Now $acc'_d(t_k)$ is represented as a monotonous combination of per-pair scores $acc_d(t_k, t)$, which can be precomputed for all possible term pairs and stored in an inverted index.

In an initial set of experiments with the 100 topics from the TREC Terabyte tracks 2004 and 2005 on the GOV2 collection (see Section 8.1 for details on the collection), we evaluated the effect of Büttcher's score over standard BM25 for 60 combinations of values for k_1 and b, for precision at different cutoffs and MAP. For all experiments, the results with Büttcher's score were always at least as good as the results with BM25, significantly better (with $p \leq 0.05$ for a signed t-test) for 42 configurations in precision at 10 results, for 59 configurations in precision at 100 results, and always for MAP. We use the parameter setting from [4, 5] ($k_1 = 1.2, b = 0.5$), which was among the best configurations in our experiments as well.

4 Indexes

4.1 Index Organization

Following [22], we maintain two kinds of index lists:

- **text index lists** (short: text lists) where each list stores, for a single term t, an entry of the form $(d.docid, score_{\text{BM25}}(d,t))$ for each document d where this term occurs ($d.docid$ is a unique numerical id for document d),
- **combined index lists** (short: combined lists) where each list contains, for a single term pair (t_1, t_2), an entry of the form $(d.docid, acc_d(t_1,t_2), score_{\text{BM25}}(d,t_1), score_{\text{BM25}}(d,t_2))$ for each document where this term pair occurs within a certain window W (we'll discuss the window size in Subsection 5.1).

We denote the index consisting of all text index lists for collection C by $T(C)$, and the index consisting of all text and combined index lists for C by $I(C)$. We will use the term *inverted lists* synonymously for index lists.

4.2 Index Compression

Index compression is an established technique for reducing the size of an inverted index. The index tuning framework described in this paper transparently supports all kinds of index compression. This section introduces our proof-of-concept implementation of index compression which applies delta and v-byte encoding [10, 33]; we did not perform any specific optimization for the parameters, for example the number of bits to represent a score, but we think that the values we chose are reasonable.

Our inverted lists are usually sorted by docid, but may be also sorted by descending score ($score_{\text{BM25}}$ for text lists, acc_d for combined lists). We store all index lists in a single index file, sorted by descending key (term or term pair).

Every compressed index list starts with its key and the offset to the next index list in the index file (which is encoded as v-bytes). For each entry in the list, all scores are first normalized into the interval $[0, 2^{14} - 1]$ by first dividing them by the maximal score, multiplying them by $2^{14} - 1$ and rounding to the next integer; for combined lists, this is done separately for each of the three scores. The header of the list stores the maximal score(s). If the list is sorted in docid-order, docids are first delta-encoded and then stored as v-bytes, and the score(s) of the entries are encoded as v-bytes with at most 2 bytes per score. For score-order, the score after which the lists are sorted is first delta-encoded, then it (and, for combined lists, the two content scores) and the docid are encoded as v-bytes.

The access structure to find the inverted list for a given key is implemented analogously to that of MapFiles in Hadoop [28]; again, this is just a proof-of-concept implementation, we could alternatively have implemented the access structure with B+-trees, for example. An in-memory index keeps every 128th key and a pointer to the offset of its inverted list in the index file. To locate the inverted list for a key, the key or its closest neighbor key (in sort order) are determined in the in-memory index, then the index file is searched linearly from the offset of that key until either the right list is found or a larger key is encountered; in the latter case, there is no list for that key in the index.

5 Parameter Tuning

5.1 Tuning as optimization problem

It has been demonstrated in [22] that using text and combined index lists together for query processing can reduce processing cost by an order of magnitude compared to using only text index lists and a state-of-the-art top-k algorithm. At the same time, the proximity component of the score helps to additionally improve result quality. However, these great properties come at a big price: An index that maintains complete information for all combined lists will be several orders of magnitude larger than the original collection of documents and is therefore infeasible even for medium-sized collections. [22] proposed to keep only prefixes of fixed length of each list, and demonstrated that this improved both result quality and query performance while greatly reducing index size. It also included experiments indicating that term pair occurrences that are more than approximately 10 positions apart hardly play a role for result quality and can therefore usually be ignored. We take over this finding for this paper, so whenever we talk about term pair occurrences, we mean occurences of different terms within a window of at most 10 positions in the same document. Note, however, that all our methods are still valid when this constraint is relaxed.

However, [22] did not provide any means for selecting the list length cutoff, which usually depends on the document collection and on the required result quality. There is a tradeoff between index size and quality: Longer lists usually mean better results, but also a bigger index, while setting the length cutoff too low will greatly reduce index size, but at the same time also hurt result quality.

This section introduces an automated method to tune index parameters such that both the size of the resulting index and the quality of results generated using this index meet predefined requirements. (Note that for the moment, our approach keeps all combined lists, but limits the information stored in each list. We will discuss in Section 7 how a subset of all combined lists can be selected based on the occurrence of the pairs in a query log). We will proceed as follows: We first

define several parameters for tuning the index size, then we show how to estimate the size of an index given the tuning parameters. Next, we define measures for the quality of a pruned index, and finally, we formally define index tuning as an optimization problem and show how to solve it.

Parameters

We start with defining two parameters to tune the selection of index entries stored in each text or combined index list:

- *minimal score cutoff*: We keep only index entries with a score that is not below a certain lower limit m.
- *list length cutoff*: We keep at most the l entries from each list that have the highest scores.

These two parameters allow us to systematically reduce the size of the resulting index with a controllable influence on result quality. Figure 5.1 shows how the index size for GOV2, relative to an unpruned index, changes with varying l and m.

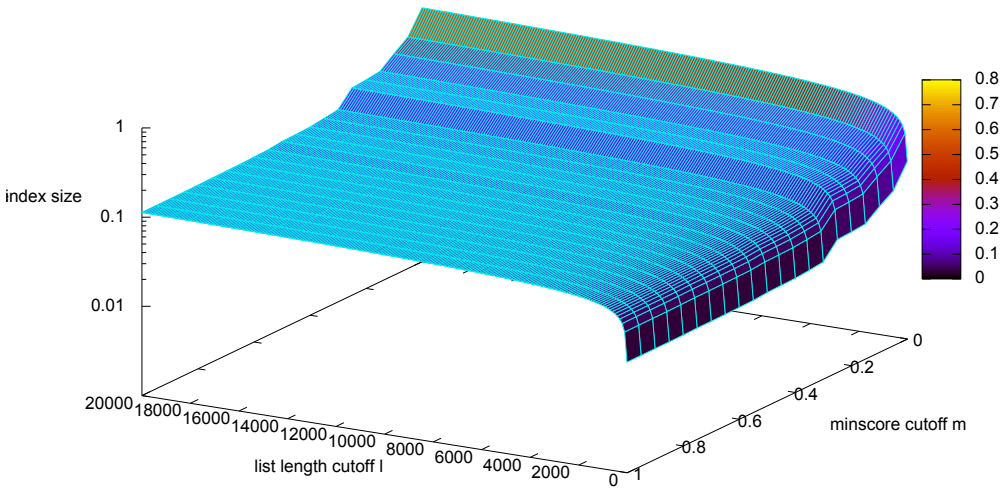

Figure 5.1: Relative index size with varying list length and minscore cutoffs

We write $I(C, l, m)$ for the index for document collection C that consists of text and combined index lists, where each list is limited to the l entries with highest

score and the combined lists contain only entries with an acc_d-score of at least m. We use the similar notation $T(C, l)$ for an index consisting of only text lists where each list contains only the l entries with highest score. Note that we currently do not perform score-based pruning on text lists. We omit C when the collection is clear from the context.

Index Size

An important constraint in our optimization process is the maximal storage space that the final pruned index is allowed to occupy. We will denote the *size* of an index I in bytes by $|I|$. The size of an uncompressed index depends on (1) the aggregated number $N(I)$ of index entries in each list, (2) the size s of each index entry in bytes, (3) the number of different keys $K(I)$ (i.e., terms and/or term pairs) in the index, and (4) the per-key overhead a of the access structure to associate a key with an offset in the inverted file; for a compressed index, s depends on the entry and the previous entry (due to delta encoding). We can therefore formally define the size of the uncompressed index I as

$$|I| := s \cdot N(I) + a \cdot K(I)$$

This simple definition is only valid when all index lists are of the same type. In our application, we may have two different index lists, text lists and combined lists, which may differ in number of entries, number of keys and entry size. We therefore write $N_t(I)$ for the number of text list entries in index I and $N_c(I)$ for the number of combined list entries in I, with $N(I) = N_t(I) + N_c(I)$, and use a similar notation for s and $K(I)$. The more accurate size of an uncompressed index I is then

$$|I| := s_t \cdot N_t(I) + s_c \cdot N_c(I) + a \cdot (K_t(I) + K_c(I))$$

Assuming that integers and floats need 4 bytes to store, we can set $s_t := 4+4 = 8$ (document id and content score) and $s_c := 4 + 4 + 4 + 4 = 16$ (document id, proximity score, and content scores for both terms). We can estimate a similarly (for example, by assuming that a corresponds to the average key length plus the space for a pointer into the inverted file).

We are typically interested in estimating the size of a pruned index $I(l, m)$ or $T(l)$ without actually materializing it (because materializing it takes a lot of time and the index may be too large to be completely materialized anyway). In the following we discuss how to estimate $|I(l, m)|$, the adaption to $|T(l)|$ is straightforward. We consider only a sample P of all possible keys (i.e., terms and term pairs) and use it to approximate the distribution of list lengths, given a list length cutoff l and minscore cutoff m. Formally, we denote by $X(l, m)$ a random variable for

the length of an index list in index $I(l, m)$, and want to estimate the distribution $F(l, m)$ of that random variable, i.e., estimate $F(l, m; x) = P[X(l, m) \leq x]$. We sample the index lists for a subset P of n keys chosen independently from all keys; each sample yields a value $X_i(l, m)$ for the length of that list in $I(l, m)$. Using the empirical distribution function [27], we can estimate the cdf of this distribution as

$$\hat{F}_n(l, m; x) := \frac{\sum_{i=1}^{n} J(X_i(l, m) \leq x)}{n}$$

where

$$J(X_i(l, m) \leq x) = \begin{cases} 1 & \text{if } X_i(l, m) \leq x \\ 0 & \text{else} \end{cases}$$

All we actually need is the expected length $E[F(l, m)]$, which can again be estimated from the sample as $\overline{X_i(l, m)}$ [27]. Assuming that there are $K(P)$ keys in the sample, the expected number of entries in the index for the sample is therefore $K(P) \cdot \overline{X_i(l, m)}$. To extend this estimate to the complete collection, we make sure that the size of P relative to the size of the collection is known, for example by sampling $p\%$ of all keys (this can be easily implemented using hash values of keys). The expected number of keys in the index is therefore $\frac{100 \cdot K(P)}{p}$, and the expected number of entries in the index is

$$N(l, p) := \frac{100 \cdot K(P)}{p} \cdot \overline{X_i(l, m)}$$

The size estimator for a compressed index is built similarly, but instead of computing just the length $X_i(l, m)$, we materialize and compress the list, and use its actual size, avoiding the need to multiply by s.

As the space of feasible values for the parameters l and m is in principle infinitely large, we cannot compute the estimate for all combinations. Instead, our implementation considers only selected step sizes for l and m, computes estimates for those values, and interpolates sizes for other value combinations. We currently consider a step size of 100 for l and 0.05 for m.

Index Quality

Intuitively, the fewer entries we keep in each list, the more will reduce the quality of query results, since the probability that relevant documents are dropped from the pruned lists increases. The goal is to find values for $\overline{m}$ and $\overline{l}$ that *maximize index quality* while generating an index that fits into a predefined amount of memory. We now define different notions of *index quality* measures $M(C, l, m, k)$ for index $I(C, l, m)$ and a fixed number k of results.

In the best case, a set of predefined *reference* or *training topics* Λ is available that include human assessments of the relevance of documents in the collection.

Such a set of topics can be build, for example, by first selecting a set of representative topics from a query log, then computing top-k results for different parameters settings, pooling those results per topic, and have human assessors determine the relevance of each result. Topic sets of this kind are frequently available for test collections such as TREC .GOV or .GOV2, but they cannot be reused for different document collections. Given such a set Λ of reference topics, we denote by $p_\Lambda[k; I]$ the average quality of the top-k results (e.g., precision or NDCG) computed using index I; our implementation currently uses precision. We can now define *effectiveness-oriented* and *efficiency-oriented absolute index quality*:

- *Effectiveness-oriented absolute index quality*: this is quantified as the ratio of the quality of the first k results with the pruned index to the quality of the first k results with the unpruned index or, formally, $\frac{p_\Lambda[k;I(C,l,m)]}{p_\Lambda[k;I(C)]}$.
- *Efficiency-oriented absolute index quality*: this is quantified as the reciprocal of the maximal query processing cost per query term (i.e., $\frac{1}{l}$) when the result quality of the pruned index is not worse than that of an unpruned text-only index without proximity lists (formally, when $\frac{p_\Lambda[k;I(C,l,m)]}{p_\Lambda[k;T(C)]} \geq 1$), and 0 otherwise.

Here, the effectiveness-oriented index quality measure aims at finding the best possible results by including as much proximity information in the index as possible. The efficiency-oriented quality measure, on the other hand, assumes that the quality of a text-only index is already sufficient and tries to minimize the length of index lists (assuming that query processing efforts are directly proportional to the lengths of index lists).

For most applications, such a set of reference topics does not exist or would be too expensive to generate. In this case, we fix a set Γ of *queries* (e.g., representative samples from a query log) and use *relative quality* to estimate how good results with the pruned index are, compared to results with the unpruned index. We define, for each query $\gamma_i \in \Gamma$, the set of relevant results to be the top-k documents with some index configuration I' and use this to compute result quality of index configuration I. When the quality measure is precision, this boils down to computing the overlap of the top-k results with index configurations I and I'. We formally denote the resulting quality of index I as $p_\Gamma[k; I|I']$.

We can now define relative index quality measures in an analogous way to the absolute measures defined before. However, we then would always favor index configurations that produce exactly the results of the corresponding unpruned index, as we assume that any results not in the top-k results with the unpruned index are non-relevant. This is often overly conservative in practice, as many of the new results will be relevant to the user as well. It is therefore often sufficient to provide a "high" overlap, not a perfect one. We therefore introduce

another application-specific tuning parameter α that denotes the threshold for relative quality above which we accept an index configuration. This is especially important for efficiency-oriented index quality: We cannot expect that we will get the same results with the pruned index with text and combined lists than with just the unpruned text lists, so achieving an overlap of 1 there would be impossible. Instead, we use $I(C)$ also in that case and set α to a value below 1.

- *Effectiveness-oriented relative index quality*: this is the relative result quality $p_\Gamma[k; I(C, l, m)|I(C)]$ of the pruned index.
- *Efficiency-oriented relative index quality*: this is the reciprocal of the maximal query processing cost per query term (i.e., $\frac{1}{l}$) when the relative result quality $p_\Gamma[k; I(C, l, m)|I(C)]$ of the pruned index is at least α and 0 otherwise.

Index Tuning

We can now formally specify the index tuning problem:

Problem 1 *Given a collection C of documents, an upper limit S for the index size, a target number of results k, and an index quality measure M, estimate parameters $\overline{m}$ and $\overline{l}$ such that $M(C, \overline{l}, \overline{m}, k)$ is maximized, under the constraint that $|I(C, \overline{l}, \overline{m})| \leq S$. When there is more than one combination of m and l that maximize the quality measure and satisfy the size constraint, pick one of them where the index size is minimal.*

Note that even though the index is tuned for a specific number k of results, it can be still used to retrieve any other number of results. We will experimentally validate in Section 8.2 that result quality does not degrade much in these cases.

5.2 Implementation of the Tuning Framework

We implemented our tuning framework within the MapReduce paradigm [12], dividing the tuning process into several map-reduce operations. As stated before, the input to the tuning process is the collection C, a target index size S, a target number of results k, and an index quality measure M that includes a set of training topics T. Additionally, we fix the fraction p of index keys (terms, term pairs) to be sampled. The tuning process then proceeds in the following order, where each step is implemented as a map-reduce operation:

1. **Compute index for sample and training topics.** The *map* phase considers each document in the collection, parses it and creates index entries for

terms and term pairs that are either part of the sample or the training topics. These entries are still incomplete, because the final BM25 scores can be computed only when global properties of the collection are known, so they contain only term frequencies and document lengths (but already complete $acc_d(t_1, t_2)$ values for term pairs); their key is the term or term pair. The *reduce* phase then combines items with the same key into an index list, completing their scores as all global parameters of the score (average document length, number of documents and document frequency of each term) are now known.[1] The output of this phase are two indexes, one for the sample, the other for the set of training topics.

2. **Prepare the estimator for the index size.** In an initial map-reduce operation, we compute the baseline precisions. The *map* phase then considers each key in the sample and computes, for each combination (l, m) it considers, the size s of the corresponding index list when pruned according to the l and m cutoffs (or the size of its compressed representation for compressed indexes), which is then written out with key (l, m). The algorithm starts with $l = k$ and increases it by the step size for l, and considers all values for m, starting at 0 and increasing it by the step size for m. The *reduce* phase combines all values for a single pair of (l, m) cutoffs and computes the average index list size for this cutoff. This value is then stored in an on-disk data structure as size estimate for (l, m). This phase also counts the overall number of keys in the sample.

3. **Prepare solving the optimization problem.** The *map* phase considers each topic with its corresponding assessments and computes, for each (l, m) pair provided by the size estimator, the quality of the index for this topic. This can be efficiently implemented by a stepwise incremental join algorithm.

   In the first step, it reads the first k entries from each list and incrementally computes results for (k, m), starting at the highest value for m and decreasing it by the step size. This yields, for each m, a temporary set of results with (partial) scores, from which the k documents with highest partial score are considered as result. The index quality for this result is computed and written out with key (k, m). Note that for the efficiency-oriented quality measures, not $1/l$ is written, but the actual precision of the results; the reduce phase will transfer this to the 'real' quality measure later. If the score of the entry at position k is less than m (i.e., the list would be cut before it), the value m is marked as completed and will not be considered later. As

[1] At least Hadoop 0.20 does not directly provide these global parameters to the reduce phase, so we need to store them in files and aggregate them in each reducer. The alternative would be to combine the initial map with a do-nothing reducer, include additional map-reduce operations to compute the global values, and then have a map-reduce with a do-nothing mapper and the reducer we just described.

soon as m exceeds the score of the last read entry, all smaller values for m will get the same index quality.

In the following steps, the process reads more entries from each list corresponding to the step size for l. Assume that it read up to l entries from each list. It continues with the temporary set of partial results from the previous step and the highest value for m not yet marked as completed and repeats the above process. This phase ends when either all values for l have been considered or all lists have been completely read. It is evident that each entry of the lists is read at most once, so the complexity is linear in the aggregated number of entries in the index lists for this topic.

The *reduce* phase averages, for each combination of (l, m), the per-topic index quality values computed by the map phase, and computes the final index quality for this combination. For the efficiency-oriented measures, this means that it compares the average precision with the result quality of the text-only index and uses $1/l$ as final index quality when the average precision is high enough. If the (l, m) combination has a non-zero index quality, the reducer estimates its size using the size estimator. For each (l, m) combination with a non-zero index quality that matches the size constraint *S*, the reduce phase outputs an (l, m, q, s) tuple, where q is the index quality and s is the index size.

4. **Compute an approximate solution of the optimization problem.** The following centralized phase scans all output tuples from the previous step and determines the tuple $(\bar{l}, \overline{m}, \overline{q}, \overline{s})$ with highest quality. Optionally, it can further explore the solution space around $(\bar{l}, \overline{m})$ for better solutions. The output of this step is an approximate solution to Problem 1.

5. **Materialize the final index.** Analogously to phase 1, the final index is materialized in a single map-reduce operation. Note that each mapper can already restrict the index entries it generates: For term pair entries, it does not emit any entries whose score is below $\overline{m}$, and for term entries, it emits only the $\bar{l}$ entries with highest scores (which can be achieved using an additional combiner). An additional optimization for this step would be to generate only an approximation of the final index: If there are M mappers used to parse the collection, each mapper needs to emit at most $\frac{\beta}{M} \cdot \bar{l}$ entries, where $\beta \geq 1$ is a tuning parameter that steers the expected number of entries missing in the final index.

6 Query Processing

The highly efficient *top-k* or *dynamic pruning* algorithms [1, 13] that are frequently applied for efficient query processing incur a non-negligible processing overhead for maintaining candidate lists and candidate score bounds, for mapping newly read index entries to a possibly existing partially read document using hash joins, and for regularly checking if the algorithm can stop. In our scenario with index lists that are pruned to a maximal length, this processing overhead is not necessary. Instead, it is sufficient to evaluate queries in document-at-a-time evaluation. Our merge-based processing architecture for this consists of the following components:

1. After pruning index lists to a fixed maximal size (and, possibly, using a minimal score cutoff for combined lists), we resort each list in ascending order of document ids, and optionally compress it.

2. At query time, the n text and combined lists for the query are combined using an n-way merge join that combines entries for the same document and computes its score. If that score is higher than the current kth best score, the document is kept in a heap of candidate results, otherwise it is dropped.

3. Once all index entries have been read, the content of the heap is returned.

Instead of maintaining a heap with the currently best k results, an even simpler implementation could keep all results as result candidates and sort them at the end; however, this would increase the memory footprint of the execution as not k, but all encountered documents and their scores need to be stored.

Independent of the actual algorithm, processing a query with our pruned index lists has a guaranteed maximal abstract execution cost (i.e., the number of index entries read from disk during processing a query), so worst- and best-case runtime are very similar and basically depend only on the number of lists involved in the execution and the cutoff for list lengths. This is a great advantage over using non-pruned text lists with algorithms for dynamic pruning and early stopping, which

can read large and uncontrollable fractions of the index lists to compute the results, and may give arbitrarily bad results when stopped earlier [23].

7 Log-based Term Pair Pruning

Even with relatively short list length cutoffs $\bar{l}$, the overall space consumption of the pruned combined lists can still be pretty huge, because there are a lot more combined lists than text lists. On the other hand, the majority of combined lists are unlikely to ever occur in any query. A possible solution can be to selectively materialize only combined lists for term pairs that occur at least t times in a query log, which can drastically reduce the number of lists. However, when one of these unlikely queries occurs for which not all or even no combined lists are available, answering it using the pruned text lists and the available subset of combined lists only may affect result quality for this query. Figure 7.1 demonstrates this effect, using the AOL query log and our training topics on .GOV2 (see Section 8.1), with $\bar{l} = 4310$ and $\bar{m} = 0.00$. The x-axis of this chart shows different values for the threshold t of query pairs in the AOL log, and the y-axis shows the precision at 10 results. The blue line (with diamonds) depicts the result of running our algorithm from Section 6 with the available index lists only. It is evident that the higher the threshold, the lower result quality gets, which can be explained by fewer and fewer combined lists being materialized. For very high thresholds (not depicted in the chart), the precision drops to 0.396, compared to 0.617 when using all lists.

To overcome this negative effect, we propose to keep the unpruned text index lists when log-based pruning is applied. As soon as one combined list or all combined lists for a query term are missing, we read the available combined lists and the unpruned text list for that term. This improves result quality to at least the quality of an unpruned text index, but at the same time incurs an increased cost for query evaluation as longer text lists have to be read. Figure 7.1 also depicts the effect of these approaches on result quality (pink line with squares: read full text lists when at least one pair is missing; green line with triangles: read full text lists when all combined lists are missing), it is evident that this combined execution helps to keep precision close to the level of the precision with the unpruned $T(C)$ index only (which is 0.585). Our tuning framework can be extended to consider only combined lists where the corresponding term pair occurs at least t times in a query log, and tunes the parameters to reach the optimization goals even with this

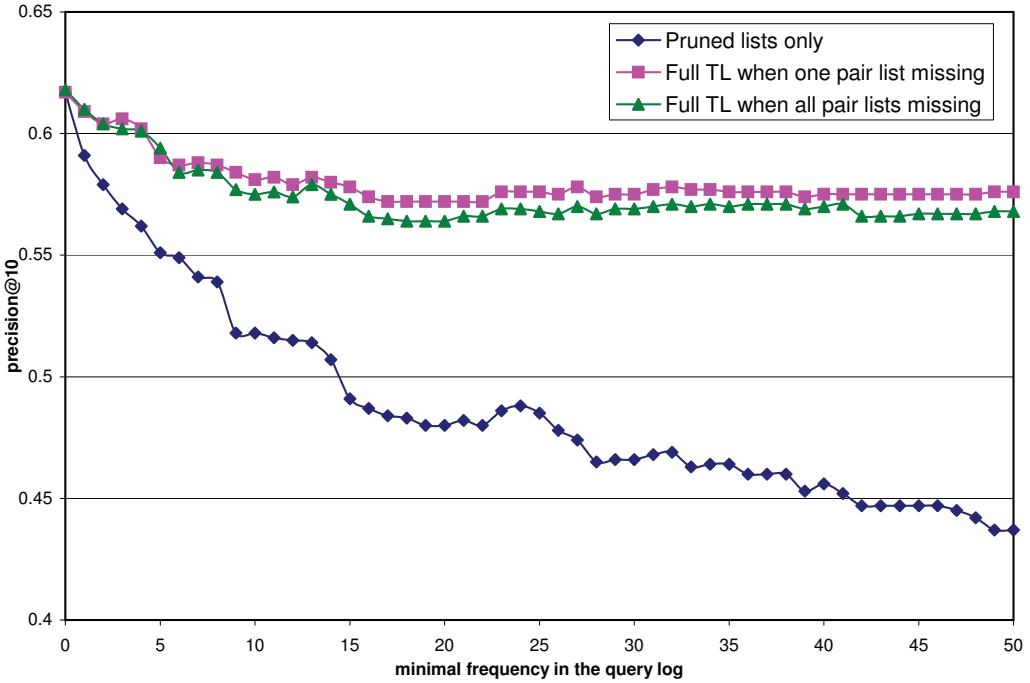

Figure 7.1: Effect of log-based pruning on query performance (on training topics)

limited selection of combined lists.

To limit storage overhead, we split unpruned text lists in two pieces: the $\bar{l}$ entries with highest scores are stored in docid-order and the remaining entries in score-order. When processing a query where some combined lists are missing, we first process the first piece of the text lists and the available combined lists with the algorithm from Section 6, keeping all documents and their scores in memory. After that, a standard top-k algorithm (in our case NRA [13]) consumes the second piece of the text lists, using the already read documents as candidates. The acc_d contribution for non-available combined lists is 0 in both steps. This algorithm will terminate more quickly than running it on the unpruned text lists alone, and will usually give better results due to the proximity score from the combined lists.

8 Experimental Evaluation

8.1 Setup

We evaluated our methods with two standard collections from TREC, the GOV2 collection and the ClueWeb collection. The TREC GOV2 collection consists of approximately 25 million documents from U.S. governmental Web sites with an aggregated size of approx. 426GB. Here, we used the 100 adhoc topics from the TREC 2004 and 2005 Terabyte tracks as training topics for tuning index parameters, and the TREC 2006 Terabyte topics for testing the quality of results computed from the resulting indexes. We used the AOL query log[1] for the log-based technique. We measure result quality as precision@k, i.e., the average number of relevant results among the first k results.

The ClueWeb collection[2] consists of approximately 1 billion Web documents crawled in January and February 2009. Following standards at the TREC Web Track, we restricted the collection first to the approximately 500 million English documents, from which we chose the 50% documents with the smallest probabilities to be spam according to the Waterloo Fusion spam ranking[3]. As only 50 topics from the TREC Web track 2009 are available with relevance assessments, we can run only a limited set of experiments on this collection as it would be hardly possible to separate test and training topics with such a small set. We therefore report detailed tuning results for GOV2 only.

Whenever we report times for parameter tuning or index construction, they were measured on a cluster of 10 servers in the same network, where each server had 8 CPU cores plus 8 virtual cores through hyperthreading, 32GB of memory, and four local hard drives of 1TB each. The cluster was running Hadoop 0.20 on Linux, with replication level set to two for space reasons.

[1] `http://gregsadetsky.com/aol-data/`
[2] `http://boston.lti.cs.cmu.edu/Data/clueweb09/`
[3] `http://durum0.uwaterloo.ca/clueweb09spam/`

8.2 Index Tuning on GOV2

We evaluated our index tuning techniques from Section 5 for different maximal index sizes and result counts (10 and 100), with and without index compression. The effect of additional log-based combined list pruning will be evaluated in Section 8.4. For each setting, we first estimated index parameters using the training topics, built an index with these parameters, and then evaluated result quality on the test topics.

Absolute Index Quality

Table 8.1 shows the results of index tuning on the training topics with selected size limits below the collection size, for uncompressed indexes. In this table, each row shows results for a given index size constraint and number of query results, namely the resulting index parameters, the estimated and real index size for these parameters, and the result quality on the training and test topics with this index. The rows with size limit ∞ denote the corresponding unpruned indexes with text+combined lists or text lists, respectively. Estimating one set of parameters took approximately 5 hours, where about 3.5 hours were required for the first map-reduce phase to build the index for the sample and the test topics. The time for building the final index strongly depends on the chosen parameters; for an index with up to 310 entries per list and a score threshold of 0.05, this took less than five hours on our cluster.

Opt. goal	k	size limit	$\bar{l}$	$\overline{m}$	size[GB] est.	size[GB] real	prec@k on train	prec@k on test
effectiveness oriented index quality	10	100GB	19010	0.40	96.4	96.9	0.596	0.572
		200GB	19010	0.15	170.5	170.8	0.610	0.578
		400GB	4310	0.00	396.4	396.4	0.617	0.592
		∞	$I(C)$			757.0	0.614	0.578
	100	100GB	10200	0.35	97.4	97.6	0.3899	0.3146
		200GB	17900	0.15	169.1	169.4	0.3975	0.3244
		400GB	4200	0.00	394.3	394.3	0.4035	0.3176
		∞	$I(C)$			757.0	0.4108	0.3338
efficiency-oriented index quality	10	100GB	5010	0.30	87.2	87.0	0.586	0.578
		200GB	310	0.05	128.1	127.9	0.588	0.534
		∞	$T(C)$			22.9	0.585	0.538
	100	400GB	800	0.00	270.6	270.6	0.3848	0.2850
		∞	$T(C)$			22.9	0.3847	0.3002

Table 8.1: GOV2: Index tuning results for absolute index quality, without index compression

It is evident that all indexes with the estimated parameters meet the index size constraint. For the effectiveness-oriented quality goal, all precision results (for the training and, more importantly, also for the test topics) are better than the precision with an unpruned text-only index (significantly better under a paired t-test

with $p < 0.05$ when the size limit is at least 200GB), so the additional combined index lists help to improve precision even when they are pruned. For the efficiency-oriented quality goal, it turns out that already very short list prefixes (310 entries for top-10, 800 entries for top-100 results) are enough to yield results with a quality comparable to standard text indexes, given a sufficiently large index size constraint. If this constraint is too tight, short lists cannot guarantee the quality target.

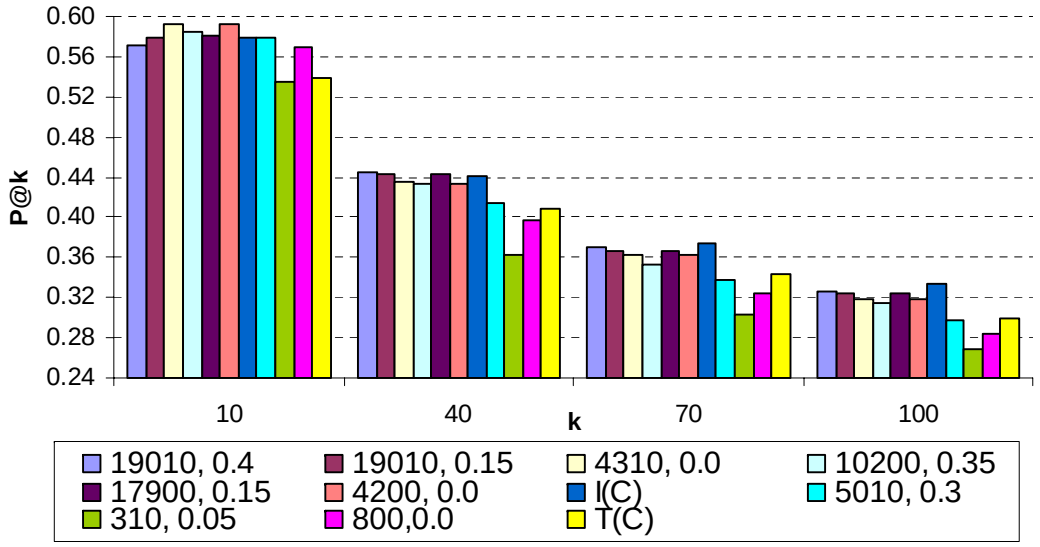

Figure 8.1: Precision@k on test topics for effectiveness- and efficiency-oriented absolute index quality

Figure 8.1 shows precision values for the test topics with all index configurations from Table 8.1 for varying numbers of retrieved results. It is evident that result quality with indexes tuned for 10 results does not degrade much when returning longer result lists.

Tuning results with index compression are depicted in Table 8.2, which has a similar layout as the table before; the size of a compressed $I(C)$ (in score-order for use with top-k algorithms) is 468.9GB, the size of a compressed $T(C)$ is 14.5GB. It is evident that index compression helps to achieve better result quality for a given index size constraint; alternatively, smaller indexes are sufficient to reach a quality goal. Our index compression scheme is effective: an index in configuration (4310, 0.00) requires 396.4GB uncompressed, but only 248.8GB compressed; the size estimator for compressed indexes is effective as well, with usually only minor overestimation.

Compared to the original runs from the TREC 2006 Terabyte Track [6], our tuned indexes do well in terms of precision. The best P@20 we get for the effec-

Opt. goal	k	size limit	$\bar{l}$	$\overline{m}$	size[GB] est.	size[GB] real	prec@k on train	prec@k on test
effective-ness oriented index quality	10	40GB	510	0.30	39.3	39.2	0.570	0.524
		50GB	6810	0.75	48.3	48.1	0.589	0.586
		70GB	19010	0.30	64.6	64.5	0.599	0.574
		100GB	19010	0.20	98.5	98.0	0.608	0.574
		200GB	19010	0.05	173.8	172.8	0.615	0.584
		400GB	4310	0.00	249.7	248.8	0.617	0.592
		∞	$I(C)$			468.9	0.614	0.578
	100	40GB	900	0.40	39.3	39.2	0.3585	0.2624
		50GB	10400	0.85	49.9	49.7	0.3795	0.3124
		70GB	19800	0.30	64.9	64.7	0.3926	0.3250
		100GB	20000	0.20	99.0	98.5	0.3970	0.3248
		200GB	19700	0.05	174.4	173.4	0.4008	0.3264
		400GB	14400	0.00	295.1	293.5	0.4067	0.3272
		∞	$I(C)$			468.9	0.4108	0.3338
efficiency-oriented index quality	10	40GB	510	0.30	39.3	39.2	0.570	0.524
		50GB	6310	0.75	47.9	47.7	0.585	0.588
		70GB	5010	0.30	55.6	55.4	0.586	0.578
		100GB	310	0.05	94.9	94.9	0.588	0.534
		200GB	310	0.05	94.9	94.9	0.588	0.534
		400GB	310	0.05	94.9	94.9	0.588	0.534
		∞	$T(C)$			14.5	0.585	0.538
	100	40GB	900	0.40	39.9	39.7	0.3585	0.2624
		50GB	10400	0.85	49.9	49.7	0.3795	0.3124
		70GB	6000	0.30	56.9	56.7	0.3847	0.2988
		100GB	3600	0.15	84.6	84.2	0.3849	0.2946
		200GB	900	0.00	193.8	193.5	0.3861	0.2868
		400GB	900	0.00	193.8	193.5	0.3861	0.2868
		∞	$T(C)$			14.5	0.3847	0.3002

Table 8.2: GOV2: Index tuning results for absolute index quality, with index compression

tiveness-oriented goal is 0.5310 (for (10200, 0.35)), none of the P@20 values underscores 0.5210. Our best indexes outperform 14 of 20 competitors in P@20. Note that our index tuning was not carried out with the TREC 2006 topics but with the training topics and for retrieval of the top-10 or top-100 results instead of the top-20, which clearly imposes a penalty on us. For the efficiency-oriented goal the best index (5010, 0.30) reaches a P@20 of 0.5210 while it is natural that very short list lengths deteriorate in later precision values, at 0.4520 for (310, 0.05).

Relative Index Quality

Here, we first performed an experiment to estimate good values for α: We computed, for a selection of possible values for α, optimal index parameters for the training topics under relative index quality, then instantiated the corresponding pruned indexes and compared the resulting absolute precisions (using the assessments from TREC) to the precision of the same topics with $I(C)$ and $T(C)$. The results of this experiment are displayed in Table 8.3. This allows to estimate values for α that are sufficient to yield similar precision as the unpruned text-only

index for the efficiency-oriented measure; a good choice is $\alpha = 0.75$.

α	0.7	0.75	0.8	0.85	0.9	0.95
$\frac{p[100;I(C,l,m)]}{p[100;I(C)]}$	0.9343	0.9471	0.9626	0.9759	0.9914	1.0010
$\frac{p[100;I(C,l,m)]}{p[100;T(C)]}$	0.9945	1.0081	1.0246	1.0388	1.0553	1.0655

Table 8.3: Relative result quality for different values of α

Table 8.4 gives tuning results for relative index quality with uncompressed indexes. We can get close to the result quality for top-10 results of an unpruned index with the effectiveness-oriented techniques (we even get better quality for some scenarios), for both the test and the training topics. For top-100 results, the situation is slightly worse, there is a small gap to the quality of an unpruned index (which, however, may be tolerable). For the efficiency-oriented indexes, we achieve comparable or even better precisions than the unpruned text indexes, at a reasonable index size of less than 100GB. Figure 8.2 depicts precision values for efficiency-oriented and effectiveness-oriented index quality goals on all (l, m) combinations from Table 8.4, for varying numbers of retrieved results. It is evident that the relative index quality approach ensures retrieval quality on test topics even without relevance assessments.

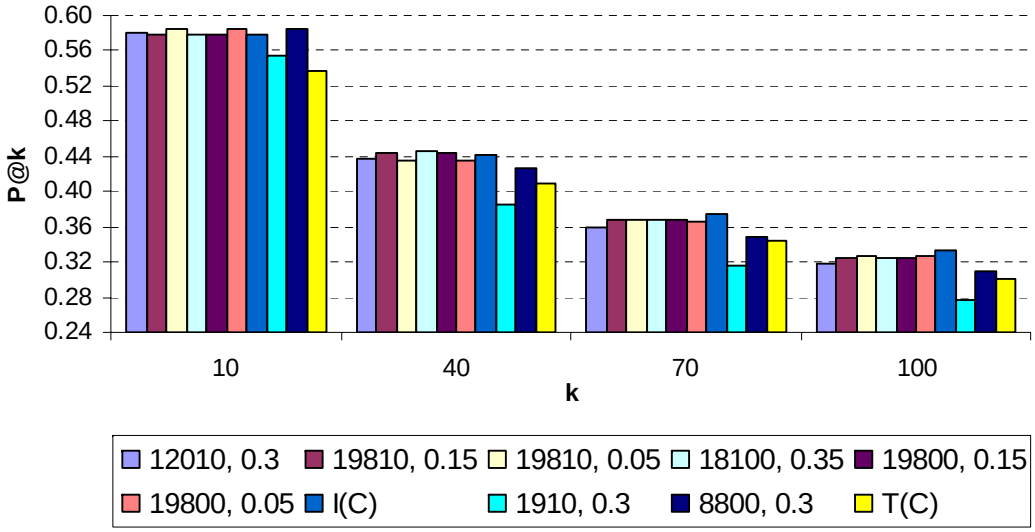

Figure 8.2: Precision@k on test topics for effectiveness- and efficiency-oriented relative index quality

Our tuning experiments for compressed indexes yielded similar findings; Table 8.5 shows the detailed results. Index compression helps to achieve a higher

Opt. goal	k	size limit	$\bar{l}$	$\overline{m}$	size[GB] est.	size[GB] real	overlap on train	prec@k on train	prec@k on test
effectiveness oriented index quality	10	100GB	12010	0.30	99.7	100.1	0.837	0.591	0.580
		200GB	19810	0.15	171.4	171.8	0.893	0.609	0.578
		400GB	19810	0.05	293.3	293.4	0.924	0.615	0.584
		∞	$I(C)$			757.0	-	0.614	0.578
	100	100GB	18100	0.35	100.0	100.4	0.773	0.3899	0.3252
		200GB	19800	0.15	171.4	171.8	0.829	0.3983	0.3248
		400GB	19800	0.05	293.3	293.4	0.868	0.4008	0.3266
		∞	$I(C)$			757.0	-	0.4108	0.3338
efficiency-oriented index quality	10	100GB	1910	0.30	73.1	72.7	0.750	0.574	0.554
		∞	$T(C)$			22.9	-	0.585	0.538
	100	100GB	8800	0.30	95.3	95.4	0.750	0.3903	0.3104
		∞	$T(C)$			22.9	-	0.3847	0.3002

Table 8.4: GOV2: Index tuning results for relative index quality, without index compression

Opt. goal	k	size limit	$\bar{l}$	$\overline{m}$	size[GB] est.	size[GB] real	overlap on train	prec@k on train	prec@k on test
effectiveness oriented index quality	10	50GB	7010	0.55	49.9	49.7	0.776	0.584	0.588
		70GB	19810	0.30	64.9	64.7	0.854	0.596	0.576
		100GB	19810	0.20	98.9	98.4	0.882	0.606	0.580
		200GB	19810	0.05	174.5	173.5	0.924	0.614	0.584
		400GB	19810	0.00	306.8	304.9	0.946	0.613	0.586
	100	70GB	20000	0.30	64.9	64.8	0.786	0.3923	0.3252
		100GB	20000	0.20	99.0	98.5	0.819	0.3968	0.3250
		200GB	19800	0.05	174.5	173.5	0.867	0.4007	0.3266
		400GB	19800	0.00	306.7	304.8	0.903	0.4062	0.3282
efficiency-oriented index quality	10	50GB	1910	0.30	48.6	48.4	0.750	0.572	0.556
		70GB	1910	0.30	48.6	48.4	0.750	0.572	0.556
		100GB	1010	0.10	91.3	91.1	0.755	0.585	0.568
		200GB	510	0.00	176.1	176.0	0.752	0.614	0.556
		400GB	510	0.00	176.1	176.0	0.752	0.614	0.556
	100	70GB	8900	0.30	59.6	59.4	0.750	0.3903	0.3102
		100GB	4100	0.15	86.1	85.7	0.751	0.3874	0.2978
		200GB	2100	0.05	130.1	129.7	0.752	0.3844	0.2940
		400GB	1200	0.00	203.5	203.2	0.750	0.3893	0.2900

Table 8.5: GOV2: Index tuning results for relative index quality, with index compression

overlap on the training topics which is usually equivalent to a higher result quality for a given index size constraint. Alternatively, smaller indexes are sufficient to reach a quality goal. An index in configuration (1910, 0.30) requires 72.7GB uncompressed, but only 48.4GB compressed.

8.3 Query Processing with GOV2

We compared the query processing performance using pruned indexes and our merge-based technique from Section 6 with the performance of a state-of-the-art top-k processing algorithm that uses sequential accesses to the index lists only

(coined NRA in [13]) as a representative of the class of dynamic pruning algorithms; a similar algorithm was used in [22]. We also present a qualitative comparison with the recent methods from [1] and [3]. Note that NRA (and also the other top-k algorithms) would show similar performance as our algorithm when run on the pruned lists; the goal of this section is to compare query performance with pruned indexes to dynamic pruning on unpruned indexes.

To assess processing performance, we use both abstract cost measures and query processing times. Abstract cost in the form of the average number of entries or bytes read from disk to process the training or test topics is not influenced by transient effects like caching or other processes running on the same machine, and it masks out the quality of the actual implementation. We additionally provide processing times of a single-threaded, Java-based implementation running on a single cluster node. These measurements were taken by running the complete batch of queries five times and taking the average; for the results with cold cache, the filesystem cache was emptied before running each query (not just each batch); this is a very conservative setting.

Opt. goal	k	size limit	$\bar{l}$	$\overline{m}$	size[GB] est.	size[GB] real	ø reads·10^{-5} train	ø reads·10^{-5} test	ø bytes·10^{-5} train	ø bytes·10^{-5} test	øt_{warm}[ms] train	øt_{warm}[ms] test	øt_{cold}[ms] train	øt_{cold}[ms] test
effectiveness oriented index quality	10	100GB	19010	0.40	96.4	96.9	0.63	0.61	6.01	5.81	28.39	31.61	226.89	243.57
		200GB	19010	0.15	170.5	170.8	0.66	0.64	6.56	6.24	28.21	27.78	232.91	245.80
		400GB	4310	0.00	396.4	396.4	0.21	0.19	2.32	2.07	9.53	8.61	177.43	182.61
		∞	$I(C)$			757.0	8.43	3.37	71.73	29.70	898.29	429.83	1368.07	1020.75
	100	100GB	10200	0.35	97.4	97.6	0.38	0.36	3.76	3.53	16.43	15.43	199.59	203.76
		200GB	17900	0.15	169.1	169.4	0.63	0.61	6.27	5.96	27.52	25.96	236.69	243.03
		400GB	4200	0.00	394.3	394.3	0.20	0.19	2.27	2.03	9.34	8.47	174.68	175.37
		∞	$I(C)$			757.0	16.81	12.91	71.73	29.70	1978.76	1628.06	2276.06	2068.93
efficiency-oriented index quality	10	100GB	5010	0.30	87.2	87.0	0.21	0.19	2.14	2.00	8.87	8.44	172.76	181.61
		200GB	310	0.05	128.1	127.9	0.02	0.02	0.21	0.20	1.19	1.11	135.03	131.86
		∞	$T(C)$			22.9	14.05	9.45	112.40	75.60	1550.40	926.13	1764.85	1311.10
	100	400GB	800	0.00	270.6	270.6	0.04	0.04	0.52	0.49	2.30	2.22	145.27	150.60
		∞	$T(C)$			22.9	20.33	15.09	112.40	75.60	3453.74	2159.41	4078.60	2669.32

Table 8.6: GOV2: Query performance for absolute index quality, without index compression

Results with uncompressed indexes are depicted in Tables 8.6 and 8.7 for training and test topics that show the number of read index entries as well as the number of bytes and runtimes with cold and warm caches, averaged over all topics. Results with the top-k algorithm on the unpruned indexes are included in the rows for $I(C)$ and $T(C)$, respectively. For the efficiency-oriented indexes, these results clearly demonstrate that query processing on the pruned indexes is up to two orders of magnitude (o.o.m.) more efficient than on the unpruned indexes. For top-10 results, we require less than 1,800 reads per topic on average with an index of 128GB, which is less than one disk block per index list. For the effectiveness-

Opt. goal	k	size limit	$\bar{l}$	$\overline{m}$	size[GB] est.	size[GB] real	⌀reads·10^{-5} train	⌀reads·10^{-5} test	⌀bytes·10^{-5} train	⌀bytes·10^{-5} test	$\varnothing t_{warm}$[ms] train	$\varnothing t_{warm}$[ms] test	$\varnothing t_{cold}$[ms] train	$\varnothing t_{cold}$[ms] test
effectiveness oriented index quality	10	100GB	12010	0.30	99.7	100.1	0.44	0.41	4.31	4.06	20.46	22.00	203.65	214.74
		200GB	19810	0.15	171.4	171.8	0.69	0.66	6.77	6.44	29.81	29.77	232.32	243.00
		400GB	19810	0.05	293.3	293.4	0.72	0.68	7.34	6.86	31.95	29.81	240.28	257.10
		∞	$I(C)$			757.0	8.43	3.37	71.73	29.70	898.29	429.83	1368.07	1020.75
	100	100GB	18100	0.35	100.0	100.4	0.61	0.59	5.83	5.61	26.35	24.85	219.07	232.50
		200GB	19800	0.15	171.4	171.8	0.69	0.66	6.76	6.44	30.07	27.99	245.75	259.52
		400GB	19800	0.05	293.3	293.4	0.72	0.68	7.34	6.86	32.07	29.47	247.53	258.77
		∞	$I(C)$			757.0	16.81	12.91	71.73	29.70	1978.76	1628.06	2276.06	2068.93
efficiency-oriented index quality	10	100GB	1910	0.30	73.1	72.7	0.09	0.08	0.96	0.87	4.05	3.77	146.11	151.57
		∞	$T(C)$			22.9	14.05	9.45	112.40	75.60	1550.40	926.13	1764.85	1311.10
	100	100GB	8800	0.30	95.3	95.4	0.33	0.32	3.37	3.15	14.67	13.53	187.33	191.77
		∞	$T(C)$			22.9	20.33	15.09	112.40	75.60	3453.74	2159.41	4078.60	2669.32

Table 8.7: GOV2: Query performance for relative index quality, without index compression

oriented indexes, the pruned index requires up to one o.o.m. less reads than the unpruned index. For absolute index quality tuning, query performance for larger indexes is actually better, because the smaller indexes need to use long list length cutoffs, but high minscore cutoffs to meet the index size constraint, which makes query processing expensive. For relative index quality tuning, query performance for larger indexes slightly deteriorates, because the larger indexes use longer list length cutoffs but also provide higher precision values. The runtimes reported in these tables demonstrate that the theoretical cost advantage of our approach is very beneficial in practice for warm cache as well as cold cache scenarios, with average warm cache times of about 1 ms for top-10 retrieval with the best efficiency-oriented index. This corresponds to two to three o.o.m. performance advantage over standard top-k algorithm evaluation on unpruned text index lists. Unlike that, the number of read items and the runtime of our technique does not increase when retrieving more than 10 results by the nature of the merge join.

Results with compressed indexes are depicted in Tables 8.8 and 8.9, which include the average number of entries and bytes read per query; the general findings for abstract costs are similar to those for uncompressed indexes. Figure 8.3 shows runtimes on the test topics for a selection of configurations from Table 8.2 for top-10 results; times for top-100 results with these indexes are very similar (the run time depends mostly on the list length, not on the number of results). From that figure, it is evident that decompressing the index is not expensive, delivering running times often beating the time required to process queries over similar uncompressed settings. This means that both in compressed and uncompressed settings, processing performance is excellent compared to NRA, with the additional bonus of lower space requirements for compressed settings which potentially allows for small list lengths resulting in fast query processing at low index size limits.

In a second line of experiments, we ran the 50,000 queries from the TREC

Opt. goal	k	size limit	$\bar{l}$	$\overline{m}$	size[GB] est.	size[GB] real	⌀reads·10^{-5} train	⌀reads·10^{-5} test	⌀bytes·10^{-5} train	⌀bytes·10^{-5} test	⌀t_{warm}[ms] train	⌀t_{warm}[ms] test	⌀t_{cold}[ms] train	⌀t_{cold}[ms] test
effectiveness oriented index quality	10	40GB	510	0.30	39.3	39.2	0.03	0.02	0.17	0.16	1.88	2.20	111.52	109.83
		50GB	6810	0.75	48.3	48.1	0.26	0.25	1.33	1.26	8.09	8.21	190.74	193.28
		70GB	19010	0.30	64.6	64.5	0.64	0.61	2.93	2.82	17.70	17.98	270.14	254.78
		100GB	19010	0.20	98.5	98.0	0.66	0.63	3.11	2.96	17.58	18.52	245.71	245.52
		200GB	19010	0.05	173.8	172.8	0.70	0.66	3.37	3.16	18.54	18.16	238.10	228.23
		400GB	4310	0.00	249.7	248.8	0.21	0.19	1.18	1.06	6.31	5.90	177.82	175.61
	100	40GB	900	0.40	39.3	39.2	0.04	0.04	0.27	0.25	3.08	2.92	143.14	143.06
		50GB	10400	0.85	49.9	49.7	0.38	0.36	1.83	1.72	11.43	11.43	186.81	201.13
		70GB	19800	0.30	64.9	64.7	0.66	0.63	3.01	2.91	17.66	18.43	298.94	290.48
		100GB	20000	0.20	99.0	98.5	0.69	0.66	3.22	3.08	18.28	19.77	324.06	320.26
		200GB	19700	0.05	174.4	173.4	0.72	0.68	3.46	3.25	19.36	18.74	281.63	289.48
		400GB	14400	0.00	295.1	293.5	0.59	0.54	2.98	2.71	16.44	15.58	232.98	266.87
efficiency-oriented index quality	10	40GB	510	0.30	39.3	39.2	0.03	0.02	0.17	0.16	1.88	2.20	111.52	109.83
		50GB	6310	0.75	47.9	47.7	0.24	0.23	1.26	1.19	6.92	6.83	232.75	232.40
		70GB	5010	0.30	55.6	55.4	0.21	0.19	1.10	1.03	6.06	5.90	183.71	187.79
		100GB	310	0.05	94.9	94.9	0.02	0.02	0.12	0.11	0.81	0.88	107.30	98.84
		200GB	310	0.05	94.9	94.9	0.02	0.02	0.12	0.11	0.81	0.88	107.30	98.84
		400GB	310	0.05	94.9	94.9	0.02	0.02	0.12	0.11	0.81	0.88	107.30	98.84
	100	40GB	900	0.40	39.9	39.7	0.04	0.04	0.27	0.25	3.08	2.92	143.14	143.06
		50GB	10400	0.85	49.9	49.7	0.38	0.36	1.83	1.72	11.43	11.43	186.81	201.13
		70GB	6000	0.30	56.9	56.7	0.24	0.23	1.26	1.18	7.00	6.82	214.27	210.03
		100GB	3600	0.15	84.6	84.2	0.16	0.15	0.91	0.83	5.04	4.90	140.28	136.67
		200GB	900	0.00	193.8	193.5	0.05	0.05	0.32	0.30	1.89	1.86	127.25	117.92
		400GB	900	0.00	193.8	193.5	0.05	0.05	0.32	0.30	1.89	1.86	127.25	117.92

Table 8.8: GOV2: Query performance for absolute index quality, with index compression

Terabyte Efficiency Track 2005 with compressed indexes from Table 8.1 and measured the number of read index entries and runtimes. While we cannot report detailed results here for space reasons, trends from our earlier experiments are validated.

As a showcase we present performance values for the (310, 0.05) compressed index setting where average runtimes are 1.4 ms for warm caches and 127 ms for cold caches, respectively. For a more realistic simulation of a running system, we also performed an experiment where we emptied the file system cache only before the first query and then ran all queries sequentially without further cache invalidation. In this scenario that corresponds to a steady-state execution in a running search engine, processing takes 24 ms on average. Figure 8.4 depicts the average running times and the standard deviations of the running times depending on the number of keywords in the query. As expected, the average running time is monotonous in the number of query terms as more query terms potentially lead to more fetched index lists at processing time. However the variance of running times for a given query length is low such that the average running time is usually a good approximation for the expected running time of a query.

Here, we can make an interesting comparison to results by Anh and Moffat [1] for their technique based on impact order: They report an average number

Opt. goal	k	size limit	$\bar{l}$	$\overline{m}$	size[GB] est.	size[GB] real	ø reads·10^{-5} train	ø reads·10^{-5} test	ø bytes·10^{-5} train	ø bytes·10^{-5} test
effectiveness oriented index quality	10	30GB	210	0.55	29.7	29.7	0.01	0.01	0.08	0.07
		40GB	1310	0.55	40.0	39.8	0.06	0.06	0.37	0.34
		50GB	7010	0.55	49.9	49.7	0.27	0.26	1.37	1.29
		70GB	19810	0.30	64.9	64.7	0.66	0.63	3.01	2.91
		100GB	19810	0.20	98.9	98.4	0.68	0.65	3.20	3.05
		200GB	19810	0.05	174.5	173.5	0.72	0.68	3.47	3.26
		400GB	19810	0.00	306.8	304.9	0.76	0.71	3.75	3.46
	100	30GB	200	0.55	29.4	29.4	0.01	0.01	0.07	0.06
		40GB	1400	0.60	39.9	39.7	0.07	0.06	0.39	0.35
		50GB	10300	0.80	50.0	49.9	0.37	0.35	1.82	1.71
		70GB	20000	0.30	64.9	64.8	0.66	0.64	3.03	2.93
		100GB	20000	0.20	99.0	98.5	0.69	0.66	3.22	3.07
		200GB	19800	0.05	174.5	173.5	0.72	0.68	3.47	3.26
		400GB	19800	0.00	306.7	304.8	0.76	0.71	3.75	3.46
efficiency-oriented index quality	10	30GB	210	0.55	29.7	29.7	0.01	0.01	0.08	0.07
		40GB	1310	0.55	40.0	39.8	0.06	0.06	0.37	0.34
		50GB	1910	0.30	48.6	48.4	0.09	0.08	0.52	0.47
		70GB	1910	0.30	48.6	48.4	0.09	0.08	0.52	0.47
		100GB	1010	0.10	91.3	91.1	0.05	0.05	0.33	0.31
		200GB	510	0.00	176.1	176.0	0.03	0.03	0.19	0.18
		400GB	510	0.00	176.1	176.0	0.03	0.03	0.19	0.18
	100	30GB	200	0.55	29.4	29.4	0.01	0.01	0.07	0.07
		40GB	1400	0.60	39.9	39.7	0.07	0.06	0.39	0.35
		50GB	10300	0.80	50.0	49.9	0.37	0.35	1.82	1.71
		70GB	8900	0.30	59.6	59.4	0.34	0.32	1.69	1.59
		100GB	4100	0.15	86.1	85.7	0.18	0.17	1.01	0.92
		200GB	2100	0.05	130.1	129.7	0.10	0.10	0.62	0.57
		400GB	1200	0.00	203.5	203.2	0.06	0.06	0.41	0.38

Table 8.9: GOV2: Query performance for relative index quality, with index compression

of 110,000 index entries read per query (for the top-20 results) for 49,990 queries from the TREC Terabyte Efficiency track (Table 2 in [1]) on the GOV2 collection, using a standard text score. We read only 1,119 index entries per query (7,106 bytes) to retrieve the top-10 results. So even though we cannot directly compare these two experiments, we think it is fair to say that their method is at least one order of magnitude more expensive than our best method, while providing a comparable result quality.

In addition, our method comes with a much smaller memory footprint during execution as we keep only the best k results plus the current document in memory, compared to on average 104,000 accumulators in [1]. We achieve this at the cost of a larger disk-based index, which is 94.9GB for our method compared to 6.1GB for the method in [1]; however, given the availability of cheap, huge hard disks, this seems to be tolerable, and our method allows to trade some performance off for smaller indexes.

We can also compare the abstract execution cost with our pruned indexes to that reported in [3], which currently seems to be the state-of-the-art method for top-k evaluation making use of random accesses. Experiments in that paper are

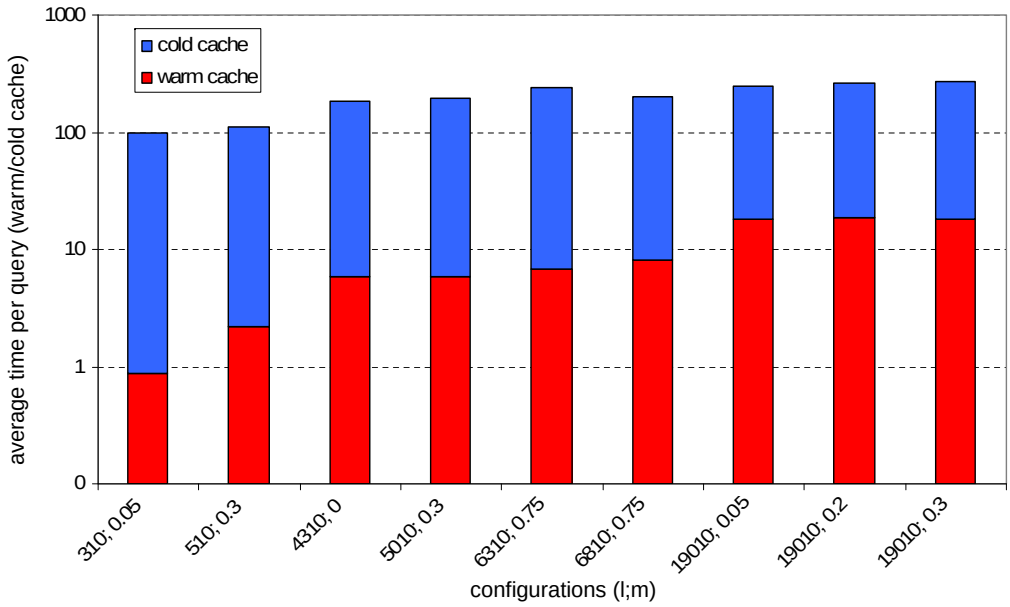

Figure 8.3: Average cold/warm cache times on test with compressed indexes

done with the GOV2 collection on the 50 topics from the TREC Terabyte Track 2005, using the same BM25 score that we use (even though there may be small differences due to different parsing, tokenization, or stopword lists). Their cost measure is the weighted sum of sorted and random accesses, where sorted accesses have weight 1 and the weight of random accesses can vary; it therefore corresponds directly to our execution cost (which is also the number of sorted accesses to our pruned indexes). They report average costs of 2,890,768 index accesses for the full merge (about 144.5M accesses for all 50 queries) compared to average costs of 788,511 for retrieving the top-10 results with their NRA implementation (total 39.4M). The best result reported in [3]—for a random access weight of 1,000—has average costs of 386,847 which is about two orders of magnitude more expensive than our method: for the same query load on the efficiency-oriented (310, 0.05) compressed index, our average cost is just 1,633 sequentially accessed tuples per topic. Additionally, we do not need to provide specific indexes for supporting random access. So we can again make the conservative statement that, even though we cannot directly compare absolute costs, our method is about two orders of magnitude faster than the best method from [3]. We do not think that it is valid to compare runtimes of our implementation to that of [3], since they use a heavily optimized C-based implementation, whereas our method is pure Java, and they don't state if and when file system caches were cleared.

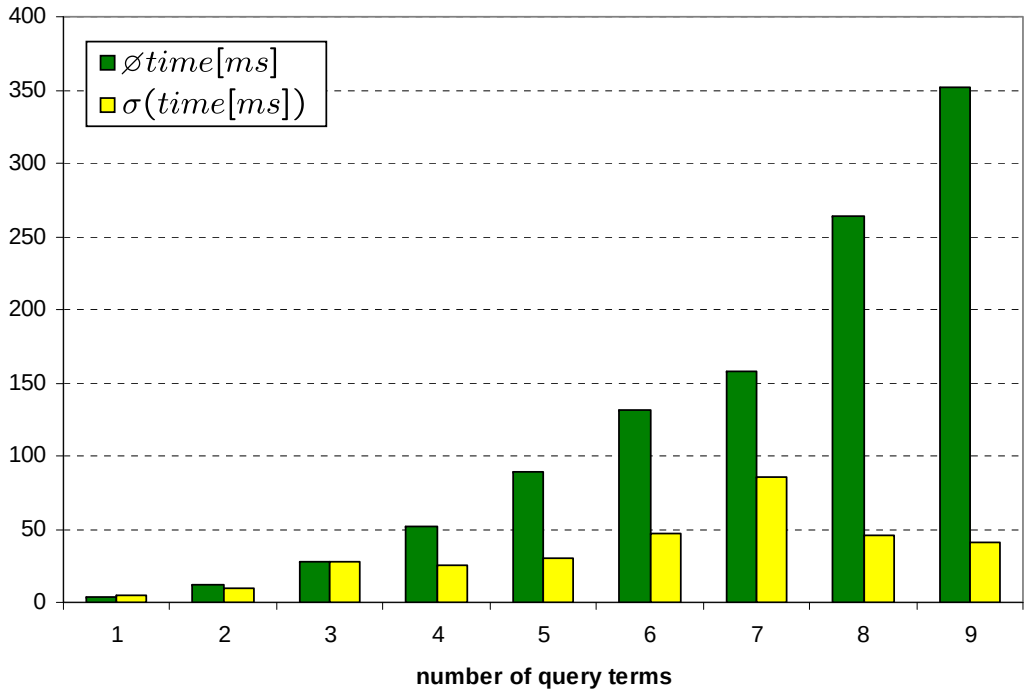

Figure 8.4: Efficiency Track: real system performance, query size

8.4 Log-based pruning with GOV2

Table 8.10 shows index tuning results for t=1, i.e., materializing combined lists for term pairs that occur at least once in the AOL query log, without index compression. It is evident that using log-based pruning helps to get smaller index sizes for the efficiency-based techniques (down to approximately twice the size of the unpruned text index) with similar result quality, but with much longer lists, which in turn affects run times (cp. Table 8.11): compared to the best results without log-based pruning, query processing takes an order of magnitude longer (which

Opt. goal	k	size limit	$\bar{l}$	$\overline{m}$	size[GB] est.	size[GB] real	prec@k on train	prec@k on test
effectiveness-oriented index quality	10	100GB	5010	0.00	96.0	96.0	0.610	0.586
	100	100GB	19800	0.10	93.6	93.6	0.3927	0.3192
		400GB	20000	0.00	293.4	293.4	0.3969	0.3196
efficiency-oriented index quality	10	50GB	2210	0.15	47.8	47.8	0.538	0.462
		100GB	1810	0.00	64.7	64.7	0.587	0.554
	100	50GB	7200	0.35	50.0	50.2	0.3771	0.3036
		100GB	3800	0.00	86.2	86.2	0.3847	0.2980

Table 8.10: Index tuning results with log-based pruning (t=1) for absolute index quality, without index compression

Opt. goal	k	$\bar{l}$	$\overline{m}$	prec@k train	prec@k test	⌀reads·10^{-5} train	⌀reads·10^{-5} test	$\varnothing t_{warm}$[ms] train	$\varnothing t_{warm}$[ms] test	$\varnothing t_{cold}$[ms] train	$\varnothing t_{cold}$[ms] test
efficiency-oriented index quality	10	2210	0.15	0.538	0.462	1.22	1.97	162.4	241.6	427.1	514.2
		1810	0.00	0.587	0.554	1.20	1.96	162.2	229.6	441.3	525.7
	100	7200	0.35	0.3771	0.3036	1.38	2.12	229.4	388.8	504.3	685.7
		3800	0.00	0.3847	0.2980	1.29	2.03	255.6	341.8	471.0	649.7

Table 8.11: Query performance with log-based pruning (t=1) for absolute index quality, without index compression

is due to the text lists that are read completely to preserve retrieval quality if all corresponding combined lists are missing as detailed in Section 7), but it is still a lot faster than NRA with unpruned $T(C)$. We could not achieve the quality goal for the effectiveness-based methods as there were not enough combined lists left to boost quality enough; we did not evaluate performance for these settings. Log-based pruning therefore mostly serves to reduce index size in situations with strong resource constraints, where it can still significantly improve execution cost, while result quality stays comparable to a text-only index.

8.5 Results with ClueWeb

Opt. goal	k	$\bar{l}$	$\overline{m}$	size[GB] est.	size[GB] real	prec@k	⌀reads ·10^{-5}	⌀bytes ·10^{-5}	$\varnothing t_{warm}$ [ms]	$\varnothing t_{cold}$ [ms]
effectiveness-o. index quality	10	1410	1.00	640.2	636.2	0.200	0.04	0.24	2.32	67.35
	100	4600	0.30	979.0	977.9	0.1344	0.14	0.75	5.35	83.69
efficiency-o. index quality	10	410	1.00	503.6	499.8	0.190	0.01	0.07	0.77	79.61
	100	400	1.00	464.6	462.8	0.1170	0.01	0.07	0.63	69.05

Table 8.12: ClueWeb: Index tuning results and evaluation of query performance for absolute index quality, with index compression, size limit set to 1TB

To demonstrate the scalability of our index tuning approach, we carried out experiments on the recent ClueWeb collection similar to the ones shown in Section 8.2 for GOV2. We considered the 50% least spammy English ClueWeb documents according to the Waterloo Fusion spam scores (approx. 6 TB uncompressed size) as described in Subsection 8.1. As currently only the 50 topics from the AdHoc Task of TREC Web Track 2009 are available with relevance assessments, we used them to train and optimize index parameters. Whenever we retrieve a document without assessment, we consider it non-relevant. We use the 50 AdHoc Task Web Track 2010 topics as test data to provide additional data on query processing speed and costs. Relevance assessments for the test topics are not yet available, so we can't report precision values for them. As the collection size of ClueWeb is large and it turned out that query processing on compressed indexes for GOV2

is often faster than for similar uncompressed settings, we restricted our tuning experiments to compressed indexes for ClueWeb. For all experiments we keep the index size limit of 1 TB fixed which corresponds to about 17% of the size of the uncompressed collection.

Table 8.12 shows the results of absolute index quality tuning on the training topics for compressed ClueWeb indexes. With index parameters tuned for efficiency and top-10 document retrieval, the most efficient resulting index (410, 1.00) requires on average less than 1 ms and 80 ms per query for warm and cold caches, respectively (1,302 reads on average), providing a result quality comparable to pure BM25 scores (where precision@10 is 0.180) for the training topics. Due to shorter index lists (1,045 reads on average) for the test topics, query processing is even a bit faster for them, showing better warm cache times and cold cache times of about 70 ms at an index size of less than 500 GB. Our effectiveness-oriented index (1410, 1.00) requires 640 GB and provides a retrieval quality comparable to using the proximity score without index pruning (with precision@10 of 0.198). Query execution takes about 2 ms for the training topics and slightly more than 1 ms for the test topics for warm caches.

Efficiency-oriented indexes for top-100 document retrieval require less than 500 GB disk space and provide running times of less than 1 ms for warm caches and around 70 ms for cold caches on the training topics at a result quality comparable to pure BM25 scores (with precision@100 of 0.1110). On the test topics query processing is again faster and takes about 55 ms on cold caches. Effectiveness-oriented indexes for top-100 document retrieval require less than 1 TB disk space and thus stay within our index size limit, providing a result quality comparable to the proximity score without index pruning (which yields a precision@100 of 0.1324). Query execution takes about 5 ms and 3 ms for warm caches, whereas cold cache times range below 85 and 65 ms for training and test topics respectively. The results show that the size estimator for compressed indexes also works effectively on the ClueWeb collection with only minor overestimation.

Table 8.13 shows the results for relative index quality tuning on ClueWeb with the training topics. While the effectiveness-oriented approaches result in indexes which deliver result quality comparable to using the proximity score without index pruning (at the price of longer lists compared to absolute index quality), result quality with the efficiency-oriented indexes falls shortly behind BM25 score quality, but the difference could still be tolerable in applications. We assume that this effect can at least partly be attributed to the fact that relevance assessments from TREC 2009 are very sparse compared to those from earlier years; unassessed documents contribute to the overlap with the groundtruth, but do not increase precision values if they are in the result list of a query, even though a user may consider them relevant.

Although the indexed part of the ClueWeb collection is one order of magnitude

larger in size than GOV2 (6 TB vs. 426 GB uncompressed), the required index space doesn't grow as fast as the collection (e.g., index size grows from 94.9 GB to 499.8 GB for the efficiency setting (310, 0.05) on GOV2 compared to (410, 1.00) on ClueWeb). For absolute index quality tuning, the indexes tend to have shorter list lengths on ClueWeb such that query processing is often even faster on ClueWeb indexes.

Opt. goal	k	$\bar{l}$	$\overline{m}$	size[GB] est.	size[GB] real	overlap	prec@k	⌀reads $\cdot 10^{-4}$	⌀bytes $\cdot 10^{-5}$	$\varnothing t_{warm}$ [ms]	$\varnothing t_{cold}$ [ms]
effect.-o.	10	9810	1.00	927.9	923.6	0.986	0.198	2.89	1.45	8.46	152.17
index qual.	100	19000	0.80	1008.3	1006.5	0.972	0.1330	5.52	2.72	13.77	135.93
effic.-o.	10	110	1.00	395.3	391.6	0.856	0.160	0.04	0.02	0.28	70.89
index qual.	100	200	1.00	408.1	407.6	0.781	0.1010	0.06	0.04	0.43	82.95

Table 8.13: ClueWeb: Index tuning results for relative index quality and evaluation of query performance, with index compression, size limit set to 1TB

9 Conclusion and Outlook

We clearly demonstrated that indexing terms and term pairs, together with tunable list pruning, is a viable method to improve either result quality or, providing a similar quality as pure text indexes, processing performance. Results with effectiveness-oriented indexes are comparable to the best results using unpruned indexes, and query performance with efficiency-oriented indexes is up to two orders of magnitude better than with top-k algorithms on unpruned text indexes.

Future work will consider improving the construction of the final index by reducing the number of temporary index entries, and improving the estimation stage which currently needs to parse the complete collection. We will further consider the impact of pruned indexes on cache effectiveness as well as index maintenance.

Bibliography

[1] V. N. Anh and A. Moffat. Pruned query evaluation using pre-computed impacts. In *SIGIR*, pages 372–379, 2006.

[2] D. Bahle, H. E. Williams, and J. Zobel. Efficient phrase querying with an auxiliary index. In *SIGIR*, pages 215–221, 2002.

[3] H. Bast, D. Majumdar, R. Schenkel, M. Theobald, and G. Weikum. Io-top-k: Index-access optimized top-k query processing. In *VLDB*, pages 475–486, 2006.

[4] S. Büttcher and C. L. A. Clarke. Efficiency vs. effectiveness in terabyte-scale information retrieval. In *TREC*, 2005.

[5] S. Büttcher, C. L. A. Clarke, and B. Lushman. Term proximity scoring for ad-hoc retrieval on very large text collections. In *SIGIR*, pages 621–622, 2006.

[6] S. Büttcher, C. L. A. Clarke, and I. Soboroff. The trec 2006 terabyte track. In *TREC*, 2006.

[7] K.-C. Chang and S.-W. Hwang. Minimal probing: supporting expensive predicates for top-k queries. In *SIGMOD*, pages 346–357, 2002.

[8] M. Chang and C. K. Poon. Efficient phrase querying with common phrase index. In *ECIR*, pages 61–71, 2006.

[9] S. Chaudhuri, K. W. Church, A. C. König, and L. Sui. Heavy-tailed distributions and multi-keyword queries. In *SIGIR*, pages 663–670, 2007.

[10] B. Croft, D. Metzler, and T. Strohman. *Search Engines - Information Retrieval in Practice*. Addison Wesley, 2010.

[11] O. de Kretser and A. Moffat. Effective document presentation with a locality-based similarity heuristic. In *SIGIR*, pages 113–120, 1999.

[12] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *C. ACM*, 51(1):107–113, 2008.

[13] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.*, 66(4):614–656, 2003.

[14] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-*k* query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4), 2008.

[15] R. Kumar, K. Punera, T. Suel, and S. Vassilvitskii. Top-*k* aggregation using intersections of ranked inputs. In *WSDM*, pages 222–231, 2009.

[16] X. Long and T. Suel. Three-level caching for efficient query processing in large web search engines. In *WWW*, pages 257–266, 2005.

[17] Y. Lv and C. Zhai. Positional language models for information retrieval. In *SIGIR*, pages 299–306, 2009.

[18] C. Monz. Minimal span weighting retrieval for question answering. In *SIGIR Workshop on Information Retrieval for Question Answering (IR4QA)*, 2004.

[19] I. Podnar, M. Rajman, T. Luu, F. Klemm, and K. Aberer. Scalable peer-to-peer web retrieval with highly discriminative keys. In *ICDE*, pages 1096–1105. IEEE, 2007.

[20] Y. Rasolofo and J. Savoy. Term proximity scoring for keyword-based retrieval systems. In *ECIR*, pages 207–218, 2003.

[21] S. E. Robertson and H. Zaragoza. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends in Information Retrieval*, 3(4):333–389, 2009.

[22] R. Schenkel, A. Broschart, S. won Hwang, M. Theobald, and G. Weikum. Efficient text proximity search. In *SPIRE*, pages 287–299, 2007.

[23] M. Shmueli-Scheuer, C. Li, Y. Mass, H. Roitman, R. Schenkel, and G. Weikum. Best-effort top-k query processing under budgetary constraints. In *ICDE*, pages 928–939. IEEE, 2009.

[24] R. Song, M. J. Taylor, J.-R. Wen, H.-W. Hon, and Y. Yu. Viewing term proximity from a different perspective. In *ECIR*, pages 346–357, 2008.

[25] T. Tao and C. Zhai. An exploration of proximity measures in information retrieval. In *SIGIR*, pages 295–302, 2007.

[26] M. Theobald, G. Weikum, and R. Schenkel. Top-k query evaluation with probabilistic guarantees. In *VLDB*, pages 648–659, 2004.

[27] L. Wasserman. *All of Statistics*. Springer, 2005.

[28] T. White. *Hadoop - The definite guide*. O'Reilly, 2009.

[29] H. E. Williams, J. Zobel, and D. Bahle. Fast phrase querying with combined indexes. *ACM Trans. Inf. Syst.*, 22(4):573–594, 2004.

[30] H. Yan, S. Ding, and T. Suel. Compressing term positions in web indexes. In *SIGIR*, pages 147–154, 2009.

[31] H. Yan, S. Shi, F. Zhang, T. Suel, and J.-R. Wen. Efficient term proximity search with term-pair indexes. In *CIKM*, 2010.

[32] J. Zhao and Y. Yun. A proximity language model for information retrieval. In *SIGIR*, pages 291–298, 2009.

[33] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), 2006.

Below you find a list of the most recent research reports of the Max-Planck-Institut für Informatik. Most of them are accessible via WWW using the URL `http://www.mpi-inf.mpg.de/reports`. Paper copies (which are not necessarily free of charge) can be ordered either by regular mail or by e-mail at the address below.

Max-Planck-Institut für Informatik
– Library and Publications –
Campus E 1 4

D-66123 Saarbrücken

E-mail: `library@mpi-inf.mpg.de`

MPI-I-2010-RG1-001	M. Suda, C. Weidenbach, P. Wischnewski	On the saturation of YAGO
MPI-I-2010-5-002	M. Theobald, M. Sozio, F. Suchanek, N. Nakashole	URDF: Efficient Reasoning in Uncertain RDF Knowledge Bases with Soft and Hard Rules
MPI-I-2010-5-001	K. Berberich, S. Bedathur, O. Alonso, G. Weikum	A language modeling approach for temporal information needs
MPI-I-2009-RG1-005	M. Horbach, C. Weidenbach	Superposition for fixed domains
MPI-I-2009-RG1-004	M. Horbach, C. Weidenbach	Decidability results for saturation-based model building
MPI-I-2009-RG1-002	P. Wischnewski, C. Weidenbach	Contextual rewriting
MPI-I-2009-RG1-001	M. Horbach, C. Weidenbach	Deciding the inductive validity of $\forall\exists^*$ queries
MPI-I-2009-5-007	G. Kasneci, G. Weikum, S. Elbassuoni	MING: Mining Informative Entity-Relationship Subgraphs
MPI-I-2009-5-006	S. Bedathur, K. Berberich, J. Dittrich, N. Mamoulis, G. Weikum	Scalable phrase mining for ad-hoc text analytics
MPI-I-2009-5-005	G. de Melo, G. Weikum	Towards a Universal Wordnet by learning from combined evidenc
MPI-I-2009-5-004	N. Preda, F.M. Suchanek, G. Kasneci, T. Neumann, G. Weikum	Coupling knowledge bases and web services for active knowledge
MPI-I-2009-5-003	T. Neumann, G. Weikum	The RDF-3X engine for scalable management of RDF data
MPI-I-2009-5-002	M. Ramanath, K.S. Kumar, G. Ifrim	Generating concise and readable summaries of XML documents
MPI-I-2009-4-006	C. Stoll	Optical reconstruction of detailed animatable human body models
MPI-I-2009-4-005	A. Berner, M. Bokeloh, M. Wand, A. Schilling, H. Seidel	Generalized intrinsic symmetry detection
MPI-I-2009-4-004	V. Havran, J. Zajac, J. Drahokoupil, H. Seidel	MPI Informatics building model as data for your research
MPI-I-2009-4-003	M. Fuchs, T. Chen, O. Wang, R. Raskar, H.P.A. Lensch, H. Seidel	A shaped temporal filter camera
MPI-I-2009-4-002	A. Tevs, M. Wand, I. Ihrke, H. Seidel	A Bayesian approach to manifold topology reconstruction
MPI-I-2009-4-001	M.B. Hullin, B. Ajdin, J. Hanika, H. Seidel, J. Kautz, H.P.A. Lensch	Acquisition and analysis of bispectral bidirectional reflectance distribution functions
MPI-I-2008-RG1-001	A. Fietzke, C. Weidenbach	Labelled splitting
MPI-I-2008-5-004	F. Suchanek, M. Sozio, G. Weikum	SOFI: a self-organizing framework for information extraction
MPI-I-2008-5-003	G. de Melo, F.M. Suchanek, A. Pease	Integrating Yago into the suggested upper merged ontology
MPI-I-2008-5-002	T. Neumann, G. Moerkotte	Single phase construction of optimal DAG-structured QEPs
MPI-I-2008-5-001	G. Kasneci, M. Ramanath, M. Sozio, F.M. Suchanek, G. Weikum	STAR: Steiner tree approximation in relationship-graphs
MPI-I-2008-4-003	T. Schultz, H. Theisel, H. Seidel	Crease surfaces: from theory to extraction and application to diffusion tensor MRI
MPI-I-2008-4-002	D. Wang, A. Belyaev, W. Saleem, H. Seidel	Estimating complexity of 3D shapes using view similarity
MPI-I-2008-1-001	D. Ajwani, I. Malinger, U. Meyer, S. Toledo	Characterizing the performance of Flash memory storage devices and its impact on algorithm design
MPI-I-2007-RG1-002	T. Hillenbrand, C. Weidenbach	Superposition for finite domains
MPI-I-2007-5-003	F.M. Suchanek, G. Kasneci, G. Weikum	Yago : a large ontology from Wikipedia and WordNet

MPI-I-2007-5-002	K. Berberich, S. Bedathur, T. Neumann, G. Weikum	A time machine for text search
MPI-I-2007-5-001	G. Kasneci, F.M. Suchanek, G. Ifrim, M. Ramanath, G. Weikum	NAGA: searching and ranking knowledge
MPI-I-2007-4-008	J. Gall, T. Brox, B. Rosenhahn, H. Seidel	Global stochastic optimization for robust and accurate human motion capture
MPI-I-2007-4-007	R. Herzog, V. Havran, K. Myszkowski, H. Seidel	Global illumination using photon ray splatting
MPI-I-2007-4-006	C. Dyken, G. Ziegler, C. Theobalt, H. Seidel	GPU marching cubes on shader model 3.0 and 4.0
MPI-I-2007-4-005	T. Schultz, J. Weickert, H. Seidel	A higher-order structure tensor
MPI-I-2007-4-004	C. Stoll, E. de Aguiar, C. Theobalt, H. Seidel	A volumetric approach to interactive shape editing
MPI-I-2007-4-003	R. Bargmann, V. Blanz, H. Seidel	A nonlinear viseme model for triphone-based speech synthesis
MPI-I-2007-4-002	T. Langer, H. Seidel	Construction of smooth maps with mean value coordinates
MPI-I-2007-4-001	J. Gall, B. Rosenhahn, H. Seidel	Clustered stochastic optimization for object recognition and pose estimation
MPI-I-2007-2-001	A. Podelski, S. Wagner	A method and a tool for automatic veriication of region stability for hybrid systems
MPI-I-2007-1-003	A. Gidenstam, M. Papatriantafilou	LFthreads: a lock-free thread library
MPI-I-2007-1-002	E. Althaus, S. Canzar	A Lagrangian relaxation approach for the multiple sequence alignment problem
MPI-I-2007-1-001	E. Berberich, L. Kettner	Linear-time reordering in a sweep-line algorithm for algebraic curves intersecting in a common point
MPI-I-2006-5-006	G. Kasnec, F.M. Suchanek, G. Weikum	Yago - a core of semantic knowledge
MPI-I-2006-5-005	R. Angelova, S. Siersdorfer	A neighborhood-based approach for clustering of linked document collections
MPI-I-2006-5-004	F. Suchanek, G. Ifrim, G. Weikum	Combining linguistic and statistical analysis to extract relations from web documents
MPI-I-2006-5-003	V. Scholz, M. Magnor	Garment texture editing in monocular video sequences based on color-coded printing patterns
MPI-I-2006-5-002	H. Bast, D. Majumdar, R. Schenkel, M. Theobald, G. Weikum	IO-Top-k: index-access optimized top-k query processing
MPI-I-2006-5-001	M. Bender, S. Michel, G. Weikum, P. Triantafilou	Overlap-aware global df estimation in distributed information retrieval systems
MPI-I-2006-4-010	A. Belyaev, T. Langer, H. Seidel	Mean value coordinates for arbitrary spherical polygons and polyhedra in $\mathbb{R}^3$
MPI-I-2006-4-009	J. Gall, J. Potthoff, B. Rosenhahn, C. Schnoerr, H. Seidel	Interacting and annealing particle filters: mathematics and a recipe for applications
MPI-I-2006-4-008	I. Albrecht, M. Kipp, M. Neff, H. Seidel	Gesture modeling and animation by imitation
MPI-I-2006-4-007	O. Schall, A. Belyaev, H. Seidel	Feature-preserving non-local denoising of static and time-varying range data
MPI-I-2006-4-006	C. Theobalt, N. Ahmed, H. Lensch, M. Magnor, H. Seidel	Enhanced dynamic reflectometry for relightable free-viewpoint video
MPI-I-2006-4-005	A. Belyaev, H. Seidel, S. Yoshizawa	Skeleton-driven laplacian mesh deformations