Efficient Text Proximity Search

Ralf Schenkel[1], Andreas Broschart[1], Seungwon Hwang[2], Martin Theobald[3], and Gerhard Weikum[1]

[1] Max-Planck-Institut für Informatik, Saarbrücken, Germany
{abrosch,schenkel,weikum}@mpi-inf.mpg.de
[2] POSTECH, Korea
swhwang@postech.ac.kr
[3] Stanford University
theobald@stanford.edu

Abstract. In addition to purely occurrence-based relevance models, term proximity has been frequently used to enhance retrieval quality of keyword-oriented retrieval systems. While there have been approaches on effective scoring functions that incorporate proximity, there has not been much work on algorithms or access methods for their efficient evaluation. This paper presents an efficient evaluation framework including a proximity scoring function integrated within a top-k query engine for text retrieval. We propose precomputed and materialized index structures that boost performance. The increased retrieval effectiveness and efficiency of our framework are demonstrated through extensive experiments on a very large text benchmark collection. In combination with static index pruning for the proximity lists, our algorithm achieves an improvement of two orders of magnitude compared to a term-based top-k evaluation, with a significantly improved result quality.

1 Introduction

Techniques for ranked retrieval of text documents have been intensively studied including relevance scoring models such as tf*idf, Okapi BM25, and statistical language models [13]. Most of the models in these families are based on the (multinomial) bag-of-words representation of documents, with consideration of term frequencies (tf) and inverse document frequencies (idf) but without considering term proximity. However, there are many queries where the best results contain the query terms in a single phrase, or at least in close proximity.

To illustrate the importance of proximity, let us consider the query "*surface area* of *rectangular pyramids*". Schemes that do not take proximity into account return general mathematical documents in which all the four terms *surface*, *area*, *rectangular* and *pyramid* are individually important, but the document does not necessarily contain information about the surface area of rectangular pyramids (for example, it may discuss the volume of pyramids and the area of rectangular prisms. On the other hand, an exact phrase match "*surface area* of *rectangular pyramids*" would most certainly ensure that the document retrieved is of

N. Ziviani and R. Baeza-Yates (Eds.): SPIRE 2007, LNCS 4726, pp. 287–299, 2007.

the desired type, but strictly enforcing such phrase matchings in a boolean way would exclude many relevant results. A good proximity-aware scoring scheme should give perfect phrase matches a high score, but reward also high proximity matches such as "*surface area* of a *rectangular*-based *pyramid*" with good scores. There has been a number of proposals in the literature for such proximity-aware scoring schemes [5,6,9,10,16,18,20]; however, none of these proposals considered efficiently finding the best results to queries in a top-k style with dynamic pruning techniques. This paper shows that integrating proximity in the scoring model can not only improve retrieval effectiveness, but also improve retrieval efficiency by up to two orders of magnitude compared to state-of-the-art processing algorithms for purely occurrence-based scoring models.

2 Related Work

Using phrases is a common means in term queries to restrict the results to those that exactly contain the phrase and is often useful for effective query evaluation [7]. A simple way to efficiently evaluate phrases are *word-level indexes*, inverted files that maintain positional information [24]. There have been some proposals for specialized index structures for efficient phrase evaluation that utilize term pair indexes and/or phrase caching, but only in the context of boolean retrieval and hence not optimized for top-k style retrieval with ranked results [8,22,23]. There are proposals to extend phrases to window queries, where users can specify the size of a window that must include the query terms to favor documents containing all terms within such a window [15,17,4]. However, this line of works has treated term proximity only as an afterthought after ranking, i.e., proximity conditions are formulated as a simplistic Boolean condition and optimized as separate post-pruning step after rank evaluation.

More recently, some scoring models were proposed that integrate content and proximity scores for ranking results [5,6,9,10,16,18,20]. These scoring models can be categorized into the following two classes. First, linear combination approaches attempt to reflect proximity in the scoring by linearly combining a proximity score with a text-based content score [5,6,16,18]. Monz quantified the proximity score based on the size of the minimum window containing all query keywords occurring in the document [16]. Rasolofo et al. consider term pairs that occur together in a small window in a document, and use a distance-based proximity score for these term pairs [18]. Büttcher et al. extend on this work by considering adjacent query term occurrences without a limit on the window size and use a proximity score similar to BM25 for text [5,6]. Second, holistic approaches have more tightly integrated proximity metrics and content scoring [9,10,20]. De Kretser and Moffat [10] and Clarke et al. [9] proposed scoring methods that reward the density of query terms in documents, and Song et al. [20] integrate a similar term density score within a BM25-based scoring model. However, none of the proximity proposals we are aware of has been designed to be used within a top-k style evaluation.

3 Processing Model

We consider a document d_i $(i = 1, \ldots, m)$, with which we associate n scores $s_{i1}, \ldots, s_{in}$, each quantifying the relevance of d_i over n different dimensions like terms or term pairs (e.g., tf*idf or BM25-based scores for query terms or, as we will later introduce, proximity scores for term pairs). The scores are aggregated using a *monotonous* function; we will focus on weighted summation as aggregation function for ease of presentation.

Our processing uses algorithms from the family of *Threshold Algorithms* [12], similar to *dynamic pruning* approaches in the IR community [1,2,14]. These algorithms assume that the scores for each dimension j have been precomputed and stored in an inverted list L_j which is sorted by descending score (or, in IR terms, in *frequency* or *impact order*). The algorithms then sequentially scan each list involved in the query execution in an interleaved, round robin manner. As documents are discovered in this process, they are maintained as candidates in an in-memory pool, where each candidate has a current score (aggregated from the scores in dimensions where the document has been encountered so far). Additionally, each candidate d_i has an upper score bound that is computed by setting all unknown scores to the highest possible score high_j corresponding to the score at the current scan positions of the lists:

$$\text{bestscore}(d_i) = \sum_{j=1}^{n} \begin{pmatrix} s_{ij} & \text{if } d_i \text{ seen in } L_j \\ \text{high}_j & \text{otherwise} \end{pmatrix} \tag{1}$$

For a top-k query, the algorithms maintain a list of the k candidates with the highest current scores. Other candidates whose best score is below the lowest current score of a top-k candidate can be safely pruned from the evaluation. The execution can stop if all but the top-k candidates have been eliminated; this is typically the case long before the lists have been completely read.

To further speed up the execution, some algorithms additionally make random lookups for the scores of promising candidates in dimensions where they have not yet been encountered; as such a random access (RA) is a lot more expensive than a sequential scan (in the order of 100 to 1,000 times for real systems), an intelligent schedule for these RAs has a great influence on efficiency. The different variants within the family of threshold algorithms primarily differ in their RA schedules; the currently most efficient variant [3] schedules all RAs only at the end of the partial scans of inverted lists, namely, when the expected cost for RA is below the cost for all sequential accesses so far.

4 Proximity Scoring

4.1 Proximity Scoring Models

We focus on proximity scoring models that use a linear combination of a content-based score with a proximity score in the form of

$$score(d, q) = \alpha \cdot score_{content}(d, q) + (1 - \alpha) score_{proximity}(d, q)$$

There are several proximity metrics in this category [5,6,16,18]. In preliminary experiments on the TREC Terabyte collection, the scoring model proposed by Büttcher et al. [5,6] (labelled *Büttcher's scoring model* from now on) was the only one to yield significant improvements in result quality over BM25-based scoring, hence we use this model in our proximity-aware evaluation framework.

4.2 Büttcher's Scoring Model

For a document d with length l, we denote the term occurring at position i of d by $p_i(d)$, or p_i when the document is uniquely given by the context. For a term t, we denote by $P_d(t) \subseteq \{1, \ldots, l\}$ the positions in document d where t occurs; or we write $P(t)$. Given a query $q = \{t_1 \ldots t_n\}$, we write $P_d(q) := \cup_{t_i \in q} P_d(t_i)$ for the positions of query terms in document d, or $P(q)$ when d is given by the context. We denote pairs of positions of distinct query terms in document d by

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

and pairs of adjacent occurrences of distinct query terms, possibly with non-query terms in between, by

$$A_d(q) := \{(i,j) \in Q_d(q) \mid \forall k \in \{i+1, \ldots, j-1\} : k \notin P_d(q)\}$$

Büttcher's scoring model linearly combines the BM25 scoring function with a proximity score for each query term into a proximity-aware document-level score. Given a query $q = \{t_1, \ldots, t_n\}$ and a document d, they first compute an accumulated interim score *acc* for each query term that depends on the distance of this term's occurrences to other, adjacent query term occurrences. Formally,

$$acc_d(t_k) = \sum_{(i,j) \in A_d(q) : p_i = t_k} \frac{idf(p_j)}{(i-j)^2} + \sum_{(i,j) \in A_d(q) : p_j = t_k} \frac{idf(p_i)}{(i-j)^2} \qquad (2)$$

where *idf* is the inverse document frequency. The accumulated proximity score increases the more, the less distant the occurrences of two adjacent terms are and the less frequent the neighboring term is in the collection. The score for a document d is then computed by a linear combination of a standard BM25-based score and a proximity score, which is itself computed by plugging the accumulated proximity scores into a BM25-like scoring function:

$$score_{\text{Büttcher}}(d,q) = score_{\text{BM25}}(d,q) + \sum_{t \in q} min\{1, idf(t)\} \frac{acc_d(t) \cdot (k_1 + 1)}{acc_d(t) + K}$$

where, analogously to the BM25 formula,

$$K = k \cdot [(1-b) + b \cdot \frac{|d|}{avgdl}]$$

and b, k_1, and k are configurable parameters that are set to $b = 0.5$ and $k = k_1 = 1.2$, respectively [5].

4.3 Modified Büttcher Scoring Model

To include Büttcher's proximity score into query processing, it would be intriguing to use a standard word-level inverted list and compute proximity scores on the fly as a document is encountered. However, this is not feasible in a top-k style processing as the proximity score is not upper bounded, and hence it is not possible to compute tight score bounds for candidates which in turn disables pruning. For an efficient computation of the top-k results, we need to precompute proximity information into index lists that can be sequentially scanned and compute tight score bounds for early pruning. The main problem with Büttcher's scoring function in this respect is that $acc_d(t)$ is computed as a sum over adjacent *query term occurrences*, which is inherently query dependent, and we cannot precompute query-independent information. An additional, minor issue is that the scoring function includes the document length which cannot be easily factorized into a precomputed score contribution.

To solve this, we slightly modify Büttcher's original scoring function; this does not have much influence on result quality (as can be shown experimentally), but allows for precomputation. In addition to dropping the document length by setting $b = 0$ in the formula, we consider *every* query term occurrence, not only adjacent occurrences. The modified accumulation function acc' is defined as

$$acc'_d(t_k) = \sum_{(i,j)\in Q_d(q):p_i=t_k} \frac{idf(p_j)}{(i-j)^2} + \sum_{(i,j)\in Q_d(q):p_j=t_k} \frac{idf(p_i)}{(i-j)^2} \tag{3}$$

As the value of $acc'_d(t_k)$ does not depend only on d and t_k, but also on the other query terms, we still cannot precompute this value independently of the query. However, we can reformulate the definition of $acc'_d(t_k)$ as follows:

$$acc'_d(t_k) = \sum_{t\in q} idf(t) \underbrace{\left(\sum_{\substack{(i,\,j)\,\in\, Q_d(q)\,:\\ p_i\,=\,t_k,\,p_j\,=\,t}} \frac{1}{(i-j)^2} + \sum_{\substack{(i,\,j)\,\in\, Q_d(q)\,:\\ p_i\,=\,t,\,p_j\,=\,t_k}} \frac{1}{(i-j)^2} \right)}_{:=acc_d(t_k,t)} \tag{4}$$

$$= \sum_{t\in q} idf(t) \cdot acc_d(t_k, t) \tag{5}$$

We have now represented $acc'_d(t_k)$ as a monotonous combination of *query term pair scores* $acc_d(t_k, t)$. Note that term order does not play a role, i.e., $acc_d(t_k, t) = acc_d(t, t_k)$. We can precompute these pair scores for all term pairs occurring in documents and arrange them in index lists that are sorted in descending score order. Including these lists in the sequential accesses of our processing algorithm, we can easily compute upper bounds for $acc'_d(t_k)$ analogously to query term dimensions by plugging in the score at the current scan position in the lists where d has not yet been encountered. The current score of a document is then computed by evaluating our modified Büttcher score with the current value of

acc'_d, and the upper bound is computed using the upper bound for acc'_d; this is correct as the modified Büttcher score is monotonous in acc'_d.

5 Indexing and Evaluation Framework

5.1 Precomputed Index Lists and Evaluation Strategies

Our indexing framework consists of the following precomputed and materialized index structures, each primarily used for sequential access, but with an additional option for random access:

- `TextIndexList` (TL): for each term t_i, a list of the form $(d_k, score_{\text{BM25}}(d_k,t_i))$, sorted by descending score.
- `ProxIndexList` (PXL): for each unordered pair $\{t_i, t_j\}$ of terms with $t_i < t_j$, a list of the form $(d_k, acc_{d_k}(t_i, t_j))$, sorted by descending *acc*.
- `CombinedIndexList` (CL): for each unordered pair $\{t_i, t_j\}$ of terms with $t_i < t_j$, a list of the form $(d_k, acc_{d_k}(t_i, t_j), score_{\text{BM25}}(d_k, t_i), score_{\text{BM25}}(d_k, t_j)))$, sorted by descending *acc*.

These index structures can be combined into several processing strategies:

- `TL`: This corresponds to standard, text-based retrieval without proximity.
- `PXL`: This scans only the proximity lists and uses the proximity part of our modified Büttcher scoring function for ranking.
- `TL+PXL`: This scans proximity and content lists (which would be the straightforward implementation of our scoring model with a Threshold algorithm).
- `TL+CL`: This stragety, which is the main contribution of this paper, exploits the additional content scores in the pair lists to reduce the uncertainty about the score of documents with high proximity scores early in the process, which often allows early termination of the algorithm. We can additionally tighten the bounds when a proximity list for a pair (t_1, t_2) runs empty: If a document was seen in the dimension for t_1, but not in the proximity list, it is certain that it won't appear in the list for t_2 any more.

5.2 Index List Pruning

For large collections, the size of the inverted lists may be too large to completely store them, especially when the index includes proximity lists. As we do not consider only adjacent terms, but *any* terms occurring in the same document, a complete set of proximity lists will be much larger than the original text collection. Lossless index compression techniques (see, e.g., [11]) are one way to solve this problem, but the compression ratio will not be sufficient for really huge collections. We therefore apply *index pruning* (which is a lossy index compression technique) to reduce the size of the index, while at the same time sacrificing as little result quality as possible. Following the literature on inverted lists for text processing, a common way is pruning lists *horizontally*, i.e., dropping entries towards the end of the lists. These entries have low scores and hence will not

play a big role when retrieving the best results for queries. Unlike text lists, pair lists contain many entries with very low scores (as the score depends on the distance of term occurrences), so the pruning effect on pair lists should be a lot higher than on text lists.

Our indexing framework provides three different pruning methods, mainly geared towards proximity lists. First, we heuristically limit the distance of term occurrences within a document, as occurrences within a large distance have only a marginal contribution to the proximity score. Second, we heuristically limit the list size to a constant, usually in the order of a few thousand entries. Third, we leverage the seminal work by Soffer et al. [19] for proximity lists. They introduced list pruning with quality guarantees for the scores of query results, assuming top-k style queries with a fixed (or at least bounded) k. For each list l, they consider the score $s_k(l)$ at position k of the list, and drop each entry from that list whose score is below $\epsilon \cdot s_k(l)$, where $0 < \epsilon < 1$ is a tuning parameter.

6 Evaluation

6.1 Setup

We evaluated our algorithms with the Java-based, open-source TopX search engine[1] [21]. Our experiments were run using the TREC Terabyte collection with roughly 25 million documents, corresponding to about 426GB of data. We evaluated our methods with the 100 adhoc topics from the 2004 and 2005 TREC Terabyte tracks. As we are focusing on top-k retrieval, we measured precision at several cutoffs. To evaluate efficiency, we measured the number of sequential (SA) and random (RA) accesses to the index lists and the number of bytes transferred from disk, assuming sizes of 8 bytes for scores and document ids. As random accesses are usually much more expensive than sequential accesses, we additionally compute a byte-based abstract cost $Cost(\gamma) = \#bytesSA + \gamma \cdot \#bytesRA$ for each run, based on the cost ratio $\gamma := c_{RA}/c_{SA}$ of random to sequential accesses.We indexed the documents with the indexer included in the TopX system with stopword removal enabled and computed the proximity lists needed for the queries with an additional tool. For the Okapi BM25 model, we used the parameters $k = k_1 = 1.2$ and $b = 0.5$. We ran the results with TopX configured in `RR-LAST` mode and a batch size of 5,000, i.e., round-robin sequential accesses in batches of 5,000 items to the index lists and postponing random accesses to the end. Index lists were stored in an Oracle database.

6.2 Results with Unpruned Index Lists

Table 1 shows our experimental results for top-10 retrieval with unpruned index lists and stemming enabled. It is evident that the configuration `TL+CL` improves precision@10 to 0.6 over the original BM25 setting (which corresponds to `TL` with a precision of 0.56), with a t-test and a Wilcoxon signed-rank confirming

[1] `http://topx.sourceforge.net`

Table 1. Experimental results for top-10 retrieval of 100 topics on Terabyte

Configuration	P@10	#SA	#RA	bytes SA	bytes RA	Cost(100)	Cost(1000)
TL	0.56	24,175,115	196,174	386,801,840	1,569,392	543,741,040	1,956,193,840
TL+PXL	0.60	24,743,914	149,166	395,902,624	1,193,328	515,235,424	1,589,230,624
TL+CL	0.60	4,362,509	8,663	108,743,568	79,256	116,669,168	187,999,568
PXL	0.40	867,095	2,925	13,873,520	23,400	16,213,520	37,273,520

Table 2. Index sizes (million items) with different length limits, with and without window limit

index/limit	500	1000	1500	2000	2500	3000	unpruned
TL	295	355	402	442	472	496	3,191
PXL/CL (est.)	368,761	435,326	481,949	515,079	542,611	566,277	1,410,238
PXL/CL, window$\leq$ 10 (est.)	23,050	28,855	34,023	38,985	42,085	45,186	87,049

statistical significance. At the same time, it dramatically reduces the number of accesses, bytes transferred, and abstract costs by a factor of 5 to 15 over the BM25 baseline, due to the additional text scores available in `CL` and the better bounds. The configuration `TL+PXL` with simple proximity lists achieves the same improvement in precision as it uses the same scoring function, but needs to run longer until it can safely stop. Scanning only the proximity lists exhibits poor result precision, even though it is much faster. We verified by additional experiments (not shown here) that the retrieval quality of our modified Büttcher scoring model was as good as the original Büttcher model.

6.3 Results with Pruned Index Lists

We first study the size of our indexes at different levels of pruning for an index (without stemming as this is an upper bound for the index size with stemming). As the complete set of proximity lists is too large to completely materialize it, we randomly sampled 1,500,000 term pairs with a frequency of at least 10, of which about 1.2% had a nonempty proximity list. Table 2 shows the index sizes (number of list entries) for text (exact) and proximity lists (estimated), for different length limits. They are calculated/estimated according to the kind of data stored in the lists as described in Subsection 5.1. We assume that document identifiers and scores a size of 8 bytes each. Therefore one TL entry or PXL entry (consisting of document identifier and BM25 score or accumulated score

Table 3. Index sizes (disk space) with different length limits, with and without window limit

index/limit	500	1000	1500	2000	2500	3000	unpruned
TL	4.4 GB	5.3 GB	6.0 GB	6.6 GB	7.0 GB	7.4 GB	47.5 GB
PXL (est.)	5.4 TB	6.3 TB	7.0 TB	7.5 TB	7.9 TB	8.2 TB	20.5 TB
PXL, window$\leq$ 10 (est.)	343.5 GB	430 GB	507 GB	580.9 GB	627.1 GB	673.3 GB	1.3 TB
CL (est.)	10.7 TB	12.7 TB	14.0 TB	15.0 TB	15.8 TB	16.5 TB	41.0 TB
CL, window$\leq$ 10 (est.)	686.9 GB	860 GB	1.0 TB	1.1 TB	1.2 TB	1.3 TB	2.5 TB

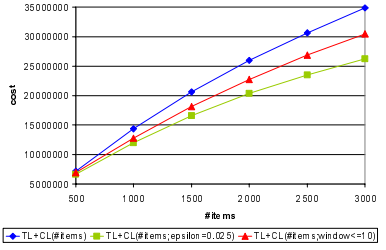

Fig. 1. TL+CL approaches: cost

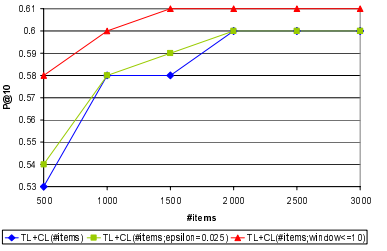

Fig. 2. TL+CL approaches: P@10

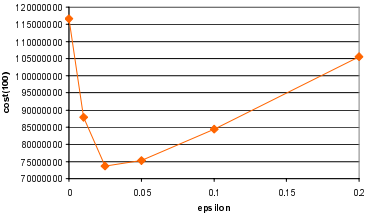

Fig. 3. TL+CL(ϵ varied): cost

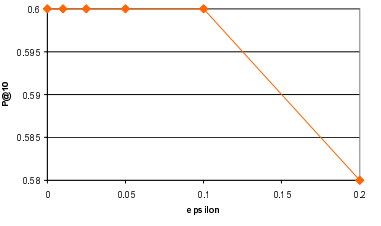

Fig. 4. TL+CL(ϵ varied): P@10

respectively) takes a size of 16 bytes whereas one CL entry takes a size of 32 bytes as it stores the document identifier, the accumulated score and two BM25 scores. It is evident that keeping all proximity lists, even with a length limit, is infeasible. However, limiting the window to 10 reduces the size of the index noticably to at most a factor of 8-15 over the unpruned text index, which may be tolerated given the cheap disk space available today. Table 3 shows the index sizes (required disk space) for the very same lists. The sze of TLs is not that big issue as the unpruned TLs only amount to 47.5 GB, and can be further downsized using maximum list lengths. The far more critical indexes are PXLs and CLs that exhibit the prohibitive estimated size of 20.5 TB and 41.0 TB respectively. Limiting the list size helps, although the lists remain too large. Additionally restricting PXLs and CLs by a window size of ten finally leads to tolerable sizes between 343.5 GB and 673.3 GB for PXLs and 686.9GB and 1.3 TB for CLs. As we show later (Table 4), excellent results can be achieved when limiting the index size to 1,500. Hence, we need about 1 TB of disk space to execute TL+CL(1500;window$\leq$10) on a document collection with 426 GB data. Additional lossless compression may further reduce the index sizes.

We then evaluated retrieval quality with pruned (text and proximity) index lists, where we used combinations of window-based pruning with a maximal size of 10, fixed-length index lists and the pruning technique by Soffer et al. [19] for $k = 10$. All measurements were done without random accesses, hence we report only a single cost value based on the number of bytes transferred by sequential accesses. Additional experiments without this constraint showed that TopX only rarely attempts to make RAs in this setting as the pruned lists are often very short.

Table 4. Experimental results for top-10 retrieval with pruned lists

Configuration	P@10	#SA	bytes SA	cost
TL+CL (window≤10)	0.60	5,268,727	111,119,408	111,119,408
TL (500 items)	0.27	148,332	2,373,312	2,373,312
TL (1000 items)	0.30	294,402	4,710,432	4,710,432
TL (1500 items)	0.32	439,470	7,031,520	7,031,520
TL (2000 items)	0.34	581,488	9,303,808	9,303,808
TL (2500 items)	0.36	721,208	11,539,328	11,539,328
TL (3000 items)	0.37	850,708	13,611,328	13,611,328
TL+CL (500 items)	0.53	295,933	7,178,960	7,178,960
TL+CL (1000 items)	0.58	591,402	14,387,904	14,387,904
TL+CL (1500 items)	0.58	847,730	20,605,312	20,605,312
TL+CL (2000 items)	0.60	1,065,913	25,971,904	25,971,904
TL+CL (2500 tuples)	0.60	1,253,681	30,648,064	30,648,064
TL+CL (3000 tuples)	0.60	1,424,363	34,904,576	34,904,576
TL+CL ($\epsilon = 0.01$)	0.60	4,498,890	87,877,520	87,877,520
TL+CL ($\epsilon = 0.025$)	0.60	3,984,801	73,744,304	73,744,304
TL+CL ($\epsilon = 0.05$)	0.60	4,337,853	75,312,336	75,312,336
TL+CL ($\epsilon = 0.1$)	0.60	5,103,970	84,484,976	84,484,976
TL+CL ($\epsilon = 0.2$)	0.58	6,529,397	105,584,992	105,584,992
TL+CL (500;$\epsilon = 0.025$)	0.54	281,305	6,628,528	6,628,528
TL+CL (1000;$\epsilon = 0.025$)	0.58	521,519	12,034,320	12,034,320
TL+CL (1500;$\epsilon = 0.025$)	0.59	732,919	16,606,064	16,606,064
TL+CL (2000;$\epsilon = 0.025$)	0.60	910,721	20,377,904	20,377,904
TL+CL (2500;$\epsilon = 0.025$)	0.60	1,060,994	23,519,296	23,519,296
TL+CL (3000;$\epsilon = 0.025$)	0.60	1,191,956	26,211,376	26,211,376
TL+CL (500;window≤10)	0.58	290,788	6,931,904	6,931,904
TL+CL (1000;window≤10)	0.60	543,805	12,763,376	12,763,376
TL+CL (1500;window≤10)	0.61	780,157	18,117,552	18,117,552
TL+CL (2000;window≤10)	0.61	984,182	22,734,544	22,734,544
TL+CL (2500;window≤10)	0.61	1,166,144	26,854,608	26,854,608
TL+CL (3000;window≤10)	0.61	1,325,250	30,466,512	30,466,512

Table 5. Experimental results for top-100 retrieval with unpruned and pruned lists

Configuration	P@100	MAP@100	#SA	#RA	bytes SA	bytes RA
TL	0.37	0.13	42,584,605	434,233	681,353,680	3,473,864
TL+PXL	0.39	0.14	44,450,513	394,498	711,208,208	3,155,984
TL+CL	0.39	0.14	12,175,316	32,357	302,386,896	380,552
PXL	0.27	0.09	867,095	2,925	13,873,520	23,400
TL+CL (window≤ 10)	0.39	0.14	17,714,952	0	346,997,712	0
TL+CL (500 items)	0.34	0.11	310,469	0	7,558,816	0
TL+CL (1000 items)	0.37	0.13	610,983	0	14,838,144	0
TL+CL (1500 items)	0.38	0.13	904,910	0	21,911,520	0
TL+CL (2000 items)	0.38	0.14	1,184,658	0	28,615,776	0
TL+CL (2500 items)	0.39	0.14	1,457,093	0	35,138,176	0
TL+CL (3000 items)	0.39	0.14	1,723,204	0	41,493,728	0
TL+CL (500;$\epsilon = 0.025$)	0.33	0.11	281,485	0	6,631,408	0
TL+CL (1000;$\epsilon = 0.025$)	0.36	0.12	527,171	0	12,156,256	0
TL+CL (1500;$\epsilon = 0.025$)	0.37	0.13	753,012	0	17,054,112	0
TL+CL (2000;$\epsilon = 0.025$)	0.37	0.13	957,593	0	21,371,376	0
TL+CL (500;window≤10)	0.34	0.12	290,968	0	6,934,784	0
TL+CL (1000;window≤ 10)	0.37	0.13	551,684	0	12,940,576	0
TL+CL (1500;window≤ 10)	0.38	0.13	802,538	0	18,638,752	0
TL+CL (2000;window≤ 10)	0.38	0.13	1,039,466	0	23,969,632	0
TL+CL (2500;window≤ 10)	0.38	0.13	1,261,124	0	28,907,200	0
TL+CL (3000;window≤ 10)	0.38	0.13	1,483,154	0	33,856,144	0

Table 4 shows the experimental results for top-10 queries in this setup, again with stemming enabled. It is evident that `TL+CL` with length-limited lists and a lim-

Table 6. Costs for top-100 retrieval with unpruned and pruned lists

Configuration	Cost(100)	Cost(1000)
TL	1,028,740,080	4,155,217,680
TL+PXL	1,026,806,608	3,867,192,208
TL+CL	340,442,096	682,938,896
PXL	16,213,520	37,273,520
TL+CL (window$\leq$ 10)	346,997,712	346,997,712
TL+CL (500 items)	7,558,816	7,558,816
TL+CL (1000 items)	14,838,144	14,838,144
TL+CL (1500 items)	21,911,520	21,911,520
TL+CL (2000 items)	28,615,776	28,615,776
TL+CL (2500 items)	35,138,176	35,138,176
TL+CL (3000 items)	41,493,728	41,493,728
TL+CL (500;$\epsilon = 0.025$)	6,631,408	6,631,408
TL+CL (1000;$\epsilon = 0.025$)	12,156,256	12,156,256
TL+CL (1500;$\epsilon = 0.025$)	17,054,112	17,054,112
TL+CL (2000;$\epsilon = 0.025$)	21,371,376	21,371,377
TL+CL (2500;$\epsilon = 0.025$)	25,288,646	25,288,646
TL+CL (3000;$\epsilon = 0.025$)	28,924,720	26,211,376
TL+CL (500;window$\leq$10)	6,934,784	6,934,784
TL+CL (1000;window$\leq$ 10)	12,940,576	12,940,576
TL+CL (1500;window$\leq$ 10)	18,638,752	18,638,752
TL+CL (2000;window$\leq$ 10)	23,969,632	23,969,632
TL+CL (2500;window$\leq$ 10)	28,907,200	28,907,200
TL+CL (3000;window$\leq$ 10)	33,856,144	33,856,144

ited window size gives a factor of 50-150 over the unpruned `TL` baseline in terms of saved cost, while yielding the same result quality (`TL+CL (1000;window`$\leq$`10)`). Using `TL` with text lists of limited length is a lot worse in effectiveness. Pruning with ϵ is not as efficient, and large values for ϵ in fact *increase* cost: Many entries from the proximity lists are pruned away, but at the same time the additional content scores available from these entries are not available any more. In combination with length limiting, results are comparable to our best configuration, but with slightly longer lists. Figures 1 to 4 illustrate some of these experimental results. We obtain the best precision values when for limiting the list size to 1,500 or more elements. Out of the approaches depicted in Figures 1 and 2, `TL+CL(#items)` is the approach with the worst precision values at the highest cost. `TL+CL(#items, window`$\leq$`10)` provides the best precision values at a medium cost, whereas `TL+CL(#items,` $\epsilon = 0.025$`)` only comes up with a slightly better precision than `TL+CL(#items)`, however at the best costs. For mere static index list pruning, precision values are most favorable for choices of ϵ below 0.1.

As especially pruning along the lines of Soffer et al. [19] is done for a specific value of k, it is interesting to see how good results using the index pruned with $k = 10$ are for larger values of k. For space reasons, we limit the presentation to $k = 100$; Tables 5 and 6 shows the results for pruned and unpruned lists. Even though proximity awareness cannot improve much on result quality, most runs with pruning are at least as effective as the unpruned runs, while saving one or two orders of magnitude in accesses, bytes transferred, and cost. The combination of length-limited lists and limited window size is again best, with a peak factor of 350 over the unpruned `TL` baseline at the same quality (`TL+CL (1000;window`$\leq$`10)`).

7 Conclusion

This paper presented novel algorithms and implementation techniques for efficient evaluation of top-k queries on text data with proximity-aware scoring. We have shown that our techniques can speed up evaluation by one or two orders of magnitude, trading in runtime for cheap disk space and maintaining the very high result quality (effectiveness) of proximity-aware scoring models. Our future work will focus on smarter data structures for indexes and applying index compression techniques.

References

1. Anh, V.N., de Kretser, O., Moffat, A.: Vector-space ranking with effective early termination. In: SIGIR, pp. 35–42 (2001)
2. Anh, V.N., Moffat, A.: Pruned query evaluation using pre-computed impacts. In: SIGIR, pp. 372–379 (2006)
3. Bast, H., et al.: Io-top-k: Index-access optimized top-k query processing. In: VLDB, pp. 475–486 (2006)
4. Botev, C., et al.: Expressiveness and performance of full-text search languages. In: Ioannidis, Y., Scholl, M.H., Schmidt, J.W., Matthes, F., Hatzopoulos, M., Boehm, K., Kemper, A., Grust, T., Boehm, C. (eds.) EDBT 2006. LNCS, vol. 3896, pp. 349–367. Springer, Heidelberg (2006)
5. Büttcher, S., Clarke, C.L.A.: Efficiency vs. effectiveness in terabyte-scale information retrieval. In: TREC (2005)
6. Büttcher, S., Clarke, C.L.A., Lushman, B.: Term proximity scoring for ad-hoc retrieval on very large text collections. In: SIGIR, pp. 621–622 (2006)
7. Callan, J.P., Croft, W.B., Broglio, J.: Trec and tipster experiments with inquery. Inf. Process. Manage. 31(3), 327–343 (1995)
8. Chang, M., Poon, C.K.: Efficient phrase querying with common phrase index. In: Lalmas, M., MacFarlane, A., Rüger, S., Tombros, A., Tsikrika, T., Yavlinsky, A. (eds.) ECIR 2006. LNCS, vol. 3936, pp. 61–71. Springer, Heidelberg (2006)
9. Clarke, C.L.A., Cormack, G.V., Tudhope, E.A.: Relevance ranking for one to three term queries. Inf. Process. Manage. 36(2), 291–311 (2000)
10. de Kretser, O., Moffat, A.: Effective document presentation with a locality-based similarity heuristic. In: SIGIR, pp. 113–120 (1999)
11. de Moura, E.S., et al.: Fast and flexible word searching on compressed text. ACM Trans. Inf. Syst. 18(2), 113–139 (2000)
12. Fagin, R., Lotem, A., Naor, M.: Optimal aggregation algorithms for middleware. J. Comput. Syst. Sci. 66(4), 614–656 (2003)
13. Grossman, D.A., Frieder, O.: Information Retrieval. Springer, Heidelberg (2005)
14. Hawking, D.: Efficiency/effectiveness trade-offs in query processing. SIGIR Forum 32(2), 16–22 (1998)
15. Metzler, D., et al.: Indri at TREC 2004: Terabyte track. In: TREC (2004)
16. Monz, C.: Minimal span weighting retrieval for question answering. In: IR4QA (2004)
17. Papka, R., Allan, J.: Why bigger windows are better than small ones. Technical report, CIIR (1997)

18. Rasolofo, Y., Savoy, J.: Term proximity scoring for keyword-based retrieval systems. In: Sebastiani, F. (ed.) ECIR 2003. LNCS, vol. 2633, pp. 207–218. Springer, Heidelberg (2003)
19. Soffer, A., et al.: Static index pruning for information retrieval systems. In: SIGIR, pp. 43–50 (2001)
20. Song, R., et al.: Viewing term proximity from a different perspective. Technical Report MSR-TR-2005-69, Microsoft Research Asia (May 2005)
21. Theobald, M., Schenkel, R., Weikum, G.: An efficient and versatile query engine for TopX search. In: VLDB, pp. 625–636 (2005)
22. Williams, H.E., et al.: What's next? index structures for efficient phrase querying. In: Australasian Database Conference, pp. 141–152 (1999)
23. Williams, H.E., et al.: Fast phrase querying with combined indexes. ACM Trans. Inf. Syst. 22(4), 573–594 (2004)
24. Witten, I.H., Moffat, A., Bell, T.: Managing Gigabytes. Morgan Kaufman, San Francisco (1999)