



# Efficient dynamic pruning on largest scores first (LSF) retrieval

Kun JIANG<sup>†</sup>, Yue-xiang YANG

(College of Computer, National University of Defense Technology, Changsha 410073, China)

<sup>†</sup>E-mail: jk\_365@126.com

Received June 6, 2015; Revision accepted Oct. 14, 2015; Crosschecked Dec. 24, 2015

**Abstract:** Inverted index traversal techniques have been studied in addressing the query processing performance challenges of web search engines, but still leave much room for improvement. In this paper, we focus on the inverted index traversal on document-sorted indexes and the optimization technique called dynamic pruning, which can efficiently reduce the hardware computational resources required. We propose another novel exhaustive index traversal scheme called largest scores first (LSF) retrieval, in which the candidates are first selected in the posting list of important query terms with the largest upper bound scores and then fully scored with the contribution of the remaining query terms. The scheme can effectively reduce the memory consumption of existing term-at-a-time (TAAT) and the candidate selection cost of existing document-at-a-time (DAAT) retrieval at the expense of revisiting the posting lists of the remaining query terms. Preliminary analysis and implementation show comparable performance between LSF and the two well-known baselines. To further reduce the number of postings that need to be revisited, we present efficient rank safe dynamic pruning techniques based on LSF, including two important optimizations called list omitting (LSF\_LO) and partial scoring (LSF\_PS) that make full use of query term importance. Finally, experimental results with the TREC GOV2 collection show that our new index traversal approaches reduce the query latency by almost 27% over the WAND baseline and produce slightly better results compared with the MaxScore baseline, while returning the same results as exhaustive evaluation.

**Key words:** Inverted index, Index traversal, Query latency, Largest scores first (LSF) retrieval, Dynamic pruning  
<http://dx.doi.org/10.1631/FITEE.1500190>

**CLC number:** TP393

## 1 Introduction

One major problem in the query processing of search engines is that the length of the posting list can easily grow to hundreds of MBs or even GBs for common terms (Dean, 2009). Given that search engines need to answer queries within fractions of a second, naively traversing the basic index structure, which could take hundreds of milliseconds, is not acceptable. This problem has long been recognized by researchers and has motivated much work on optimization techniques, including inverted index partitioning, distribution, index compression, and index

traversal (Turtle and Flood, 1995; Badue *et al.*, 2001; Melink *et al.*, 2001; Anh and Moffat, 2010; Puppini *et al.*, 2010). In this paper, we focus on exhaustive index traversal techniques, which naively access all the postings that are relevant to the query terms, and optimization techniques called dynamic pruning (Moffat and Zobel, 1996; Chakrabarti *et al.*, 2011), which return the best  $k$  results without an exhaustive traversal of the entire relevant posting lists.

Ingenious index traversal techniques can efficiently quicken the query processing procedure and reduce the hardware costs of search engines significantly (Croft *et al.*, 2010). The scheme for traversing posting lists within a document-sorted index for a query falls into two main classes: term-at-a-time

(TAAT) and document-at-a-time (DAAT) (Turtle and Flood, 1995). The main disadvantage of TAAT techniques is that they require large amounts of memory to store partial scores when traversing each posting list, especially for large index sizes. However, the iterative candidate selection of DAAT techniques has high costs associated with the random access of all the posting lists, which requires jumping back and forth between different posting lists. To address the shortcomings of the existing traversal techniques, we propose a novel exhaustive index traversal technique called largest scores first (LSF) retrieval, which can reduce the memory consumption from that of TAAT and avoid the costly candidate selection of DAAT at the expense of revisiting some of the remaining posting lists. In our previous study, the prototype of LSF retrieval was proposed, but without enough consideration of computational complexity analysis (Jiang and Yang, 2015). In addition, we describe two novel rank safe dynamic pruning heuristics, list omitting and partial scoring on LSF, i.e., LSF\_LO and LSF\_PS, to reduce the number of postings that need to be processed. Experimental results show that our LSF retrieval is faster than the TAAT retrieval and comparable to the DAAT retrieval, and also has potential advantages over dynamic pruning techniques. Further experiments show that our LSF-based pruning techniques reduce disjunctive query processing time by almost 27% over the WAND baseline and are slightly better than the MaxScore baseline, while returning the same results as exhaustive evaluation. With more query terms or more returned results, the LSF\_LO and LSF\_PS schemes show more improvements compared with the two baselines.

## 2 Related work

### 2.1 Inverted index

The inverted index plays a vital role in the efficient processing of ranked and Boolean queries (Zobel and Moffat, 2006). Given a collection of  $N$  documents, we assume that each document is identified by a unique document identifier (docid) between 0 and  $N-1$ . An inverted index can be seen as an array of lists or postings, where each entry of the array corresponds to a different term or word in the collection. The set of terms is called the lexicon of the collection. For each term  $t$  in the lexicon, the index

contains a posting list  $I_t$  consisting of a number of postings describing all places where term  $t$  occurs in the collection. More precisely, each posting in  $I_t$  contains the docid  $d$  and the number of occurrences of  $t$  in the document (called term frequency, denoted as  $f_{t,d}$ ), and possibly other information about the locations of the occurrences and their contexts. We assume postings have docids and frequencies, but do not consider other data such as positions or contexts in this paper; thus, each posting is of the form  $(d : f_{t,d})$ . However, we believe that our techniques can also apply to cases where other information is stored for future work. The tuple  $(d : f_{t,d})$  denotes the docid  $d$  and score  $f_{t,d}$  in document  $d$ .

The lexicon is comparatively small in most cases. However, the posting lists may consist of many millions of postings, which could be approximately linear with the size of the collection (Manning et al., 2008). The posting lists can be organized in either document-sorted indexes or impact-sorted indexes. The document-sorted index is the standard approach for basic exhaustive query processing, where the postings in each posting list are sorted by ascending docids. The impact-sorted index (commonly the impact is computed by term frequency) is organized in such a way that postings in each list are sorted by their impact. A problem with impact-sorted indexes is that compression could suffer as  $d$ -gaps in the posting lists may be very large (Ding and Suel, 2011; Dimopoulos et al., 2013). In this study, we focus on the document-sorted index. To allow faster access and limit the amount of memory needed, search engines use various compression techniques that significantly reduce the size of the posting lists within document-sorted indexes (Zukowski et al., 2006; Büttcher and Clarke, 2007; Silvestri and Venturini, 2010).

Researchers have presented additional synchronization structure into chunks of the posting lists and accelerated the query processing by skipping over compressed chunks (Moffat and Zobel, 1996; Strohman and Croft, 2007; Jonassen and Bratsberg, 2011). This facilitates searching for a particular docid within a posting list by skipping over a large number of compressed postings. Additional skipping pointers have to be embedded in the posting list, or a skipping table describing the front of a few selected chunks has to be stored together with the lexicon, both describing a potential skip when

decoding. This allows entire chunks that would be evaluated by an exhaustive evaluation to be skipped when searching for a given docid.

### 2.2 Index traversal

Most search engines first use a simple ranking function to compute a score for each document that passes Boolean query filters, i.e., conjunctive (AND) and disjunctive (OR) query filters. Conjunctive queries make a compulsory requirement that documents matching all of the query terms can be returned as a result. Disjunctive queries relax the requirement such that documents matched by any of the query terms may be reported as a result. For a large number of documents that pass the Boolean query filter, users are commonly interested in the top- $k$  results. Thus, search engines need only to return the  $k$  results with the highest similarity that match the query, and send the  $k$  results to a complex ranking process with hundreds of features. In this study, we use the bag-of-words similarity ranking model for the first simple ranking phase, in which the final score can be decomposed into a sum or simple combination of per-term scores. Thus, the scoring function  $S(q, d)$  can be defined as  $S(q, d) = \sum_{t \in q} s(t, d)$ , where  $q$  is the set of all terms in the query and  $s(t, d)$  is the similarity score of query term  $t$  and current docid  $d$ .

As mentioned, the schemes to exhaustively traverse posting lists within a document-sorted index for a query fall into mainly two categories: term-centric TAAT and document-centric DAAT (Turtle and Flood, 1995). With TAAT techniques, the query results are obtained by sequentially traversing one posting list at a time. As documents will occur in different posting lists, intermediate scores should be kept in accumulators until all of the occurrences of the document are considered. A simple example of TAAT traversal is shown in Fig. 1. With DAAT techniques, the posting lists that relate to a query are processed in parallel. As a consequence, the final scores of documents can be obtained and inserted into a heap structure before moving to the next document. The system may always store the top- $k$  results considered thus far and also the threshold of the result heap. Then if the score of a new candidate surpasses the threshold, it will be inserted into the result heap. At the end of the procedure, the  $k$  results in the heap will be required by the user, so

the memory requirements are fairly small. A simple example of DAAT traversal is shown in Fig. 2.

The DAAT and TAAT techniques can be enhanced into four algorithms with different query filters, i.e., AND\_DAAT, OR\_DAAT, AND\_TAAT, and OR\_TAAT. The evaluation of these index traversal techniques can be found in our previous work (Jiang et al., 2014). The result sets returned by conjunctive queries are potentially smaller than those returned by disjunctive queries. This might be a problem, especially when one of the terms is missing or mistyped. However, one of the main challenges associated with the two query filters is that disjunctive queries tend to be significantly more expensive than conjunctive queries, as they have to evaluate many more documents. The shortest posting list can be used to efficiently skip through the longer posting lists and thus reduce the amount of data to be processed. For this reason, the optimization

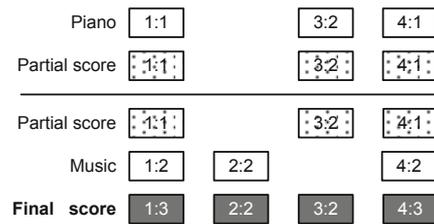


Fig. 1 An example showing the index traversal procedure of TAAT with two posting lists of the query terms ‘piano’ and ‘music’. The solid line separates different iterations of the procedure, and the scoring operation scans from left to right in each iteration to accumulate the partial scores of all the term frequencies  $f_{t,d}$ . The full result scores of the documents can be obtained when both ‘piano’ and ‘music’ are considered. Reprinted from Jiang and Yang (2015), Copyright 2015, with permission from Springer



Fig. 2 An example showing the index traversal procedure of DAAT with two posting lists of the query terms ‘piano’ and ‘music’. The solid line separates different iterations of the procedure from left to right, and the scoring operation scans from top to bottom in each iteration to sum up all of the term frequencies  $f_{t,d}$ . The full result score of a document can be obtained when both ‘piano’ and ‘music’ are considered in parallel. Reprinted from Jiang and Yang (2015), Copyright 2015, with permission from Springer

of disjunctive queries has become an important research topic in index traversal (Büttcher *et al.*, 2010; Jonassen and Bratsberg, 2011).

Impact-sorted indexes can be processed in either term-at-a-time or score-at-a-time (Anh and Moffat, 2005), in which all of the posting lists are simultaneously open, with the merge by decreasing contribution, so that pointers that make big contributions to document scores can be processed first. The score-at-a-time scheme has been proposed mainly for dynamic pruning that can obtain the top scored results as soon as possible (Strohman and Croft, 2007). A problem with impact-sorted indexes is that compression could suffer as  $d$ -gaps in the posting lists may be very large. In contrast, document-sorted indexes tend to be much smaller with different compression algorithms (Ding and Suel, 2011; Dimopoulos *et al.*, 2013). Dynamic pruning techniques on impact-sorted indexes are useful when the number of distinct impact layers is small, or frequencies are used instead of impacts (Ding and Suel, 2011).

### 2.3 Dynamic pruning

Dynamic pruning techniques return the top- $k$  scored candidates, which avoids fully evaluating all documents that satisfy the disjunctive or conjunctive condition (Ding and Suel, 2011). A min-heap (usually used in DAAT) and sets of accumulators (usually used in TAAT) are used to store the top- $k$  documents during evaluation, and we normally estimate and store the maximum achievable score of a posting list together with its lexicon term. The dynamic pruning techniques can be done in terms of the maximum score of the posting list and the threshold of the result heap or accumulator score. We define the resulting safe dynamic pruning techniques as the ones that can return the same top- $k$  results as the exhaustive evaluation, and the rank safe techniques are more strictly defined as those that can return all of the results in the same order as the exhaustive evaluation.

One of the earliest optimizations in TAAT processing comes from Buckley and Lewit (1985). In this approach, terms are sorted and processed from the least to the most frequent term, so that the important terms come first to obtain a larger accumulated score as early as possible. After processing a query term, it may be possible to show that it is not necessary to consider any more terms if they do not much

affect the final results. Thus, one or more lists for the query terms are completely omitted. As the document at rank  $k + 1$  cannot surpass the score of the document at rank  $k$ , the processing terminates for a set of accumulators with top- $k$  partial scores. This approach belongs to the result safe scheme for which only the identities of the top- $k$  results are fixed.

The TAAT MaxScore proposed by Turtle and Flood (1995) is somewhat similar to the method presented by Buckley and Lewit (1985). The improvement lies in a rank safe technique in which terms evaluated until the top- $k$  documents are guaranteed to be in the correct order. To ensure the rank safe results, the remaining posting lists should also be accessed for adding scores of the postings that already exist in the accumulator tables. Another improvement lies in that the approach removes the accumulators that cannot be part of the top- $k$  results after processing each term.

Moffat and Zobel (1996) presented quit and continue optimization for TAAT processing. The quit heuristic dynamically adds accumulators until the number of accumulators meets the fixed threshold. At this point, documents are returned to the user with the partial scores in the accumulators. The continue heuristic continues evaluation when the accumulator threshold is reached but considers only those documents that already have accumulators allocated. Lester *et al.* (2005) further introduced an efficient space-limited adaption of quit and continue optimization, which provides a trade-off between the number of accumulators and the result quality.

The DAAT MaxScore (Turtle and Flood, 1995; Lacour *et al.*, 2008; Fontoura *et al.*, 2011) takes advantage of only the partial scoring scheme for optimization. When a document is scored, we add to the partial score of the document the sum of the accumulated maximum score of the remaining terms to be scored. If this sum is less than the threshold of the min-heap, then we do not calculate the term score of the document, as the document could not be in the top- $k$  documents. The disadvantage of this scheme commonly applied in the DAAT index traversal technique is that the candidate is selected not in the posting list of the most important term but always the lowest current docid of all the posting lists. For the candidate that will not be inserted into the result heap, if the terms with larger maximum scores are evaluated first, then the partial score plus the

maximum score of the remaining terms tends to be smaller than the threshold much earlier, which avoids accessing the posting lists with less contribution.

The description of MaxScore by Strohman *et al.* (2005) differs from the original one, since the term importance is considered. The improvement is that with the increase of the number of results in the heap, the threshold becomes larger and larger. At some point, no document can make it into the top- $k$  results just using candidates selected in query terms with the maximum score that is lower than the threshold. Thus, the candidate can be selected only from important terms with larger maximum scores. More recently, Jonassen and Bratsberg (2011) presented a unified description of the above two versions with efficient skipping operations. In the latest MaxScore, posting lists are sorted from top to bottom by their maximum scores and divided into essential lists and non-essential lists in terms of the accumulated maximum score sum up to less than the threshold or not. The essential lists with larger maximum scores are possibly more important, and the candidates are selected and scored first in such lists. The partial scoring and skipping are applied in the non-essential lists with less contribution.

Broder *et al.* (2003) proposed an ingenious posting lists sorting and skipping movement metric named WAND based on DAAT. The candidate term is selected by summing up the maximum scores of the query terms in ascending order of the list pointer docids, until the sum becomes larger than the threshold of the result heap. The current docid of the term is selected as the candidate. The posting list with its docid smaller than the pivot should be processed in AND mode, and then skipping and scoring are conducted. Analysis and experimental results proposed by Dimopoulos *et al.* (2013) show that WAND is not as efficient as MaxScore due to the frequent posting lists sorting and the lack of partial scoring.

Anh and Moffat (2005; 2006) described efficient dynamic pruning methods on an impact-sorted index with score-at-a-time traversal. The methods trim the accumulator table every time meeting with a new impact score when the top- $k$  results have been determined. Strohman and Croft (2007) further improved the pruning technique by continuous accumulator pruning and the combination of skipping and skipping length. The candidate documents are selected by the smallest docid in each segment of the

query terms with equal term importance. The idea of continuous trimming can be seen as immediate partial scoring with different impact layers.

### 3 Largest scores first traversal

In this section, we present a novel exhaustive index traversal technique called largest scores first (LSF) and give a detailed implementation description of its main methods.

#### 3.1 Largest scores first

We describe a novel index traversal technique called LSF retrieval which makes full use of query term importance. With LSF traversal, it first iterates the posting list for every query term by sorting the posting lists in ascending order of document frequency so that we evaluate the shortest list first, but the postings are accessed in an order that is neither strictly term-centric nor strictly document-centric. Not surprisingly, the posting list with the highest document frequency is almost always the one with the largest upper bound score in common ranking functions, such as BM25. This means that the usual estimates of term importance (e.g.,  $\text{idf}_t = \log(N/N_t)$ , where  $\text{idf}$  represents the inverse document frequency) will decrease for lists encountered later in the evaluation. The iteration from the most significant term to the lesser makes the most promising documents appear early for scoring. Thus, the threshold of the result heap will rise quickly, which can avoid lower scored documents from being inserted into the heap. Then we choose one of the query terms with the shortest posting list length as the candidate selection term (CST) and the remaining terms as scoring adding terms (SATs). Next, we select the postings of the candidate selection term as the candidate document and access all posting lists of the query terms simultaneously for scoring when it appears. We should check if the candidate has been considered in previous candidate selection terms except for the first candidate selection term. Finally, we insert the result document and its full score into the result heap. Every time we reach the end of the candidate selection term, we remove the term from the query term set and select another term with the most significant term as the candidate selection term; additionally, we select those remaining as the SAT if the set is not empty. An example showing the

procedure of LSF is depicted in Fig. 3, in which the score of each docid can be obtained by adding all of the term frequencies in the tuple of the posting with that docid for simplicity.



**Fig. 3** An example showing the index traversal procedure of LSF with the query terms ‘piano’ and ‘music’. The dashed line separates different candidate document scorings of a given posting list from left to right to obtain partial scores. The solid line separates different query term iterations to scan another posting list for candidate documents. The cross denotes that the posting has been considered in previous posting lists

### 3.2 Algorithm and analysis

Given the inverted index and the result heap, LSF returns the  $k$  documents that have the highest scores according to the scoring function  $S(q, d)$ . The details of the technique are depicted in Algorithm 1. As shown, LSF iterates the posting list for every query term by first sorting the posting lists by ascending length. For a given posting list, it selects a candidate document  $d$  and checks if it has been considered before (lines 5–7). We have used the bitmap data structure to store the documents that were already considered. This can reduce the revisiting of postings. Then every scoring posting list should first skip to the candidate document  $d$  (line 12), and, if the candidate does not exist in the list, it will point to the first document larger than  $d$ . If it points to the candidate, just add its contribution to the accumulated score using a simple ranking function (lines 13–15). When all of the posting lists are considered for  $d$ , we insert the candidate  $d$  into the result heap if its score is larger than the threshold (line 17). When the posting list of the candidate selection term reaches the end, the scoring posting lists should be reset to make the cursor point to its first document. Finally, we return the top  $k$  scoring documents of the results in the heap (line 24).

For  $n$  query terms and  $N_t$  matching documents

---

#### Algorithm 1 Exhaustive OR\_LSF retrieval

---

**Input:** inverted index iterators  $\{I_{t_1}, I_{t_2}, \dots, I_{t_n}\}$   
 sorted by ascending list length

**Output:** top- $k$  query results sorted by descending score

- 1:  $\text{resHeap}(k) \leftarrow \emptyset, \text{Bitmap}() \leftarrow \emptyset$
- 2: **for** all posting lists  $I_{t_i} (1 \leq i \leq n)$  **do**
- 3:   **while**  $I_{t_i}$  is not finished **do**
- 4:      $d \leftarrow I_{t_i}.\text{getdid}()$
- 5:     **if**  $i > 1$  &&  $\text{Bitmap}.\text{contains}(d)$  **then**
- 6:        $I_{t_i}.\text{next}()$ , proceed to line 3
- 7:     **else**
- 8:        $\text{Bitmap}.\text{insert}(d)$
- 9:        $\text{score} \leftarrow s(t_i, d)$
- 10:     **end if**
- 11:     **for** all posting lists  $I_{t_j} (i < j \leq n)$  **do**
- 12:        $I_{t_j}.\text{skipto}(d)$
- 13:       **if**  $I_{t_j}$  points to  $d$  **then**
- 14:           $\text{score} \leftarrow \text{score} + s(t_j, d)$
- 15:       **end if**
- 16:     **end for**
- 17:      $\text{resHeap}.\text{insert}(d, \text{score})$
- 18:      $I_{t_i}.\text{next}()$
- 19:   **end while**
- 20: **for** all posting lists  $I_{t_j} (i < j \leq n)$  **do**
- 21:    $I_{t_j}.\text{reset}()$
- 22: **end for**
- 23: **end for**
- 24: **return** the top- $k$  results

---

(containing at least one query term), the first pass of the query costs  $\Theta(N_{t_1} \cdot n + N_{t_1} \cdot \log_2 k)$ . Here,  $N_t$  is the number of documents containing  $t$ ,  $N_{t_1} \cdot n$  corresponds to the times of scorings in line 14, and  $N_{t_1} \cdot \log_2 k$  corresponds to the times of sortings in line 17. For the  $i$ th query pass, corresponding to the  $i$ th loop in line 14, the cost is  $\Theta(N_{t_1} \cdot (n - (i - 1)) + N_{t_1} \cdot \log_2 k)$ . Depending on how often the query terms co-occur in a document, the posting can be scored  $n - (i - 1)$  times for the  $i$ th loop (line 14) in the worst case. The time complexity of the duplicated checking for the  $i$ th ( $i > 1$ ) loop is  $\Theta(\sum_{k=1}^{i-1} N_{t_k})$ . Thus, for all  $n$  loop iterations, the overall time complexity of the algorithm is

$$\Theta\left(\sum_{i=1}^n N_{t_i} (n - (i - 1)) + N_q \log_2 k + \sum_{i=1}^n \sum_{k=1}^{i-1} N_{t_k}\right), \quad (1)$$

where  $N_q = N_{t_1} + N_{t_2} + \dots + N_{t_n}$ . Since the inverted index iterators are sorted by ascending list length, we can deduce that  $\Theta(\sum_{k=1}^{i-1} N_{t_k}) < \Theta(N_{t_i} \cdot (i - 1))$ . Then we find that

$$\Theta\left(\sum_{i=1}^n N_{t_i} (n - (i - 1)) + N_q \log_2 k + \sum_{i=1}^n \sum_{k=1}^{i-1} N_{t_k}\right)$$

$$\begin{aligned}
 &< \Theta \left( \sum_{i=1}^n N_{t_i} (n - (i - 1)) + N_q \log_2 k + \sum_{i=1}^n N_{t_i} (i - 1) \right) \\
 &= \Theta \left( \sum_{i=1}^n N_{t_i} n + N_q \log_2 k \right) = \Theta(N_q n + N_q \log_2 k).
 \end{aligned} \tag{2}$$

It was reported in Büttcher *et al.* (2010) that the time complexities of DAAT and TAAT with normal implementation are both  $\Theta(N_q n + N_q \log_2 k)$ . Although the complexity of the three techniques is the same, the run time cost is quite different. The superiority of LSF is that it avoids having a large memory footprint and the generation of the smallest docid in candidate selection. The disadvantage of LSF processing is the revisiting of the remaining posting lists, although it can be reduced by the dynamic pruning techniques, and the time consuming duplication checking due to the large number of if-else branch predictions in confirming the posting visitings and bit operations in setting and obtaining the exact position of the bitmap.

The memory consumption of LSF lies mainly on the priority queue with  $k$  results and the bitmap data structure. That is almost the same as DAAT traversal, which needs only a priority queue with  $k$  results. However, the TAAT traversal keeps all the intermediate results in memory, which can be linear with the documents in the data sets. Thus, we omit the consideration of exhaustive TAAT and its dynamic pruning techniques in the following sections, due to our preliminary analysis and other descriptions in Lacour *et al.* (2008) and Fontoura *et al.* (2011).

### 3.3 Posting list iterator

We now describe an efficient posting list iterator that directly operates on the physical index data and provide methods for various index traversal techniques. For a given posting, the iterator contains mainly two important attributes: ‘currentptr’ is the address of a pointer of the posting list, and ‘currentdid’ is the document identifier of a given posting. There are also other attributes that can be used to score the document, but we do not need them when traversing the posting list. Regardless of the underlying index structure, we define five basic methods on the posting list iterator as an abstract data type:

1. `getdid()` returns the docid of the current position in the posting list where the pointer address is ‘currentptr’. The iterator is initialized by ‘current-

did’ to be considered as the docid of the first posting element in its posting list.

2. `getscore()` computes the partial score of ‘currentdid’ in the posting list considered by  $s(t_i, d)$ , using the data stored in the posting lists (e.g., frequencies or the number of postings) and other statistics stored outside the inverted index (e.g., document lengths).

3. `reset()` sets the current address of the posting list to zero and initializes its current posting to be the first posting element in the posting list, i.e., `currentptr ← 0`, `currentposting ← t.iterator[currentptr]`.

4. `next()` increments the current address to return the next posting of term  $t$ ’s posting list, i.e., `currentptr ++`, `currentposting ← t.iterator[currentptr]` after initialization. If the iterator reaches the end of the posting list, it calls the close function to release the memory source.

5. `skipto(d)` randomly accesses the position with docid equal to  $d$  or the one next to docid  $d$  in the posting list. The details of the skipping procedure depend on the implementation of the skipping structure.

The skipping structure we use is similar to that proposed by Jonassen and Bratsberg (2011), in which the postings are separated into data chunks and the skipping pointers are built upon to generate hierarchical skipping chunks. All the data chunks and skipping chunks are of the same length  $L$ . The last docid and the end-offset of each chunk are recursively stored in a chunk at the level above. The `skipto(d)` operation can be performed by comparing  $d$  with the last docid of the active chunk. When  $d$  is greater, the higher level chunk is considered as the active chunk. Conversely, the chunk referred to by the offset of the docid is decoded. The search goes down iteratively until the data chunk is reached. The binary search is used within each chunk for the element indicating the lower level chunk or the exact target  $d$ .

For compressed posting lists, the worst-case number of decompressed chunks and fetched blocks in a random `skipto(d)` operation is equal to the number of skip-levels, i.e.,  $L_t = \log_L N_t$ , where  $N_t$  is the collection frequency of the term  $t$  and  $L$  is the equal skipping chunk length ( $L = 128$  in this study). The skiplist structure is integrated into the posting lists. For a posting list with length  $N_t$  and skipping chunk

length  $L$ , we can deduce the total number of the extra skiplists  $E_t = \sum_{i=1}^{L_t} \lceil N_t/L^i \rceil$ , which sums the number of skipping pointers expected at each skipping level. The search complexity of the skiplists is defined by the number of exact steps necessary to find the target posting. In the worst case, the number of steps at each level is at most  $L/2$  with at most  $L_t$  levels. Thus, the search complexity of the skiplists is  $\Theta(L_t \cdot L/2) = \Theta(\log N_t)$ .

## 4 Dynamic pruning on LSF retrieval

In this section, we present the checking of the candidate for pruning postings of our rank safe dynamic pruning techniques based on LSF retrieval.

### 4.1 Ignoring postings

The basic idea of the proposed pruning technique can be achieved by monitoring two basic quantities: a threshold value  $\theta$  and a remainder function  $\rho$ . The threshold value  $\theta$  is a lower bound on the score of the last document that will be returned to the user. Here, we assume that  $k$  represents the number of documents requested. We can compute a reasonable  $\theta$  by using the  $k$ th largest score in the current result heap (or accumulator table). If  $k$  accumulators do not yet exist,  $\theta = 0$ . The second monitored quantity is the score remainder function  $\rho$ . This function computes an upper bound on the total additional score that the posting could possibly achieve through further processing of the remaining terms' posting lists. The upper bound of the posting list can be estimated with an on-the-fly scheme in query processing (Macdonald *et al.*, 2011) or be exactly computed in the index construction phase that we adopt here (Ding and Suel, 2011; Jonassen and Bratsberg, 2011).

We define the function  $\rho$  as follows:  $\rho(t_i)$  is the largest score that the posting could achieve in the posting list for term  $t_i$ ;  $\rho(q) = \sum_{t \in q} \rho(t)$ , where  $q$  is the set of terms in the query. We also maintain the threshold score  $\theta$  as the  $k$ th largest score in the result heap. We define  $q_r$  as the set of the remaining terms thus far unconsidered. Every time we finish one posting list of a given term, we check if  $\theta > \rho(q_r)$ , where  $\rho(q_r)$  is the largest achievable score of new postings in the remaining terms. When  $\theta > \rho(q_r)$ , we know that no more posting lists of the remaining terms need to be considered. This is true because no more post-

ings will achieve a score greater than  $\rho(q_r)$ . Thus, we terminate the processing and return the best  $k$  scoring documents in the heap. We call this method LSF index traversal with list omitting (LSF\_LO). Performance gains are achieved when the termination condition is met as no more posting lists of term  $t$  ( $t \in q_r$ ) need to be processed. However, the performance cost lies in the revisiting of posting lists of term  $t$ . The lists for the query terms are sorted in ascending order by their upper bounds, and the lists with larger upper bounds should be considered first to make the high scoring postings appear as early as possible.

For an in-depth posting ignoring heuristic, when  $\theta > \rho(q)$  (only occurring in the first posting list considered), we can conclude that the remaining postings in the current posting list need to exist in all other posting lists. The posting that does not meet the condition will be ignored. Moreover, instead of checking the termination condition at the end of every list traversal, we perform immediate checking of the achievable score of every posting. Thus, at every step when considering the next posting in the current list, instead of fully scoring each such candidate in all remaining posting lists, we perform partial scoring as follows. We first evaluate the score of this posting in the current list. If the sum of the score plus the upper bounds of the remaining lists is less than threshold  $\theta$ , then we can abort the evaluation of the posting, which avoids large amounts of skipping. Otherwise, we score the candidates in the remaining posting lists until either we have a complete score or the maximum possible score drops below the threshold  $\theta$ . We call this pruning method LSF index traversal with partial scoring (LSF\_PS).

### 4.2 Algorithm description

In this subsection, we present an in-depth description of the implementation of the algorithm. The details of the pseudo code are depicted in Algorithm 2. The algorithm combines the pruning technique of partial scoring and list omitting in LSF. Prior to query processing, we order the iterators by descending upper bounds and calculate their cumulative upper bounds from last to first. We maintain a heap to store  $k$  candidates.

Each iteration begins by finding and scoring the lowest docid of the posting list with the highest upper bound (line 2). We call the term of this

**Algorithm 2** Dynamic pruning on LSF retrieval

---

**Input:** inverted index iterators  $\{I_{t_1}, I_{t_2}, \dots, I_{t_n}\}$  sorted by descending upper bounds  $\hat{s}(I_{t_i})$

**Output:** top- $k$  query results sorted by descending scores

- 1:  $\text{resHeap}(k) \leftarrow \emptyset$ ,  $\text{Bitmap}() \leftarrow \emptyset$ , cumulative upper bounds  $\hat{a}(I_{t_i}) = \sum_{i \leq k \leq n} \hat{s}(I_{t_k})$
- 2: **for** all posting lists  $I_{t_i} (1 \leq i \leq n)$  **do**
- 3:   **while**  $I_{t_i}$  is not finished **do**
- 4:      $d \leftarrow I_{t_i}.\text{getdid}()$
- 5:     **if**  $i > 1$  &&  $\text{Bitmap.contains}(d)$  **then**
- 6:        $I_{t_i}.\text{next}()$ , proceed to line 3
- 7:     **else**
- 8:        $\text{Bitmap.insert}(d)$
- 9:        $\text{score} \leftarrow s(t_i, d)$
- 10:     **end if**
- 11:     **for** all posting lists  $I_{t_j} (i < j \leq n)$  **do**
- 12:       **if**  $\text{score} + \hat{a}(I_{t_j}) < \text{resHeap.minScore}$  **then**
- 13:         proceed to line 21
- 14:       **end if**
- 15:        $I_{t_j}.\text{skipto}(d)$
- 16:       **if**  $I_{t_j}$  points to  $d$  **then**
- 17:          $\text{score} \leftarrow \text{score} + s(t_j, d)$
- 18:       **end if**
- 19:     **end for**
- 20:      $\text{resHeap.insert}(d, \text{score})$
- 21:      $I_{t_i}.\text{next}()$
- 22:   **end while**
- 23: **if**  $\hat{a}(I_{t_{i+1}}) < \text{resHeap.minScore}$  **then**
- 24:   proceed to line 30
- 25: **end if**
- 26: **for** all posting lists  $I_{t_j} (i < j \leq n)$  **do**
- 27:    $I_{t_j}.\text{reset}()$
- 28: **end for**
- 29: **end for**
- 30: **return**  $\text{resHeap.decrSortResults}()$

---

posting list the candidate generating term and the other terms with smaller upper bounds score adding terms. Furthermore, we check every other iterator of the score adding terms in descending order, and, if the current score plus the cumulative upper bounds of the remaining terms is less than the threshold, the algorithm stops further evaluation for this candidate and proceeds to the next one (lines 12–14). This is the core idea of partial scoring in LSF\_PS. All terms that are used for scoring can be skipped efficiently (line 15). When a document is inserted to the full heap with  $k$  candidates, we set the threshold to the minimum score of the candidates in the heap (line 20). When the cumulative upper bound of the remaining terms is less than the threshold,

the algorithm terminates because no more candidate documents that have not appeared can be added to the result heap (lines 23–25). This is the core idea of list omitting in LSF\_LO. Otherwise, all of the iterators of the scoring terms are reset for another iteration (lines 26–28).

## 5 Experiments

In our experiments, we used the TREC GOV2 collection containing approximately 25.2 million documents and approximately 32.8 million terms in its vocabulary, with an uncompressed size of 426 GB. The index structure contains three parts, including the lexicon file, posting lists, and the statistical file. We built posting lists and the hierarchical skipping structure with 128 docids per chunk using the PForDelta compression algorithm (Zukowski *et al.*, 2006), removing standard English stopwords, and applying Porter’s English stemmer. We generated the maximum score of every term in the lexicon according to the contribution of its postings and stored them along with the lexicon file. The statistical file stored the total number of unique terms, documents, postings, etc. The final compressed index size was 6.72 GB. We also built another index without skipping, and found that the skiplist structure added about 82.1 MB to the index size, which is a 1.19% increase. We used the first 10 000 queries selected from the TREC 2005 Efficiency Track Queries using distinct numbers of terms with  $|q| \geq 2$ , and we separated the queries into those with length from 2 to 5 and those with length larger than 5.

Our experiments were performed on an Intel® Xeon® E5620 processor running at 2.40 GHz with 8 GB RAM and 12 288 KB cache. The default physical block size is 16 KB unless stated otherwise. All traversal solutions were implemented in JAVA on the Terrier retrieval platform, which is widely used in research on information retrieval (Ounis *et al.*, 2006). All of the code is openly available at <https://github.com/deeper2/LSF>. In every experiment where we report running time, the JVM was first executed a certain number of times the benchmark for warmup and the numbers were averaged over three independent runs when the JVM has reached a steady state performance (Delbru *et al.*, 2012). The default number of documents retrieved for each query equaled 10 on a results page. The

performance was always measured by average time per query in milliseconds. As we focused on rank safe query optimization techniques, the effectiveness measurement is ignored in the following experiments.

In this study, we used Okapi BM25 as the ranking function (Büttcher *et al.*, 2010). The standard Okapi BM25 bag-of-words similarity scoring function as a reference point for  $S(q, d)$  is based on their predicted relevance to the query, defined as

$$\begin{aligned} S(q, d) &= \sum_{t \in q} s(t, d) \\ &= \sum_{t \in q} \text{idf}_t \cdot \text{TF}_{\text{BM25}}(t, d) \\ &= \sum_{t \in q} \log(N/N_t) \cdot \text{TF}_{\text{BM25}}(t, d), \end{aligned} \quad (3)$$

where

$$\text{TF}_{\text{BM25}}(t, d) = \frac{f_{t,d} \cdot (k_1 + 1)}{f_{t,d} + k_1 \cdot [(1 - b) + (b \cdot l_d / l_{\text{avg}})]} \quad (4)$$

is the document frequency. Note that  $l_d$  is the number of symbols in document  $d$  and  $l_{\text{avg}}$  is the average of  $l_d$  over the collection. The formula has two free parameters,  $k_1 = 1.2$  and  $b = 0.75$ , as default.

### 5.1 Exhaustive index traversal

As mentioned, one of the main challenges associated with the processing of disjunctive (OR) queries is that they tend to be much slower than conjunctive (AND) queries, as they have to evaluate more documents. The two types of queries can be applied to each of the index traversal strategies mentioned above. We implemented six types of exhaustive index traversal algorithms, namely disjunctive DAAT, disjunctive TAAT, disjunctive LSF, conjunctive DAAT, conjunctive TAAT, and conjunctive LSF based on the code of the Terrier retrieval platform. A preliminary comparison of exhaustive index traversal techniques, including LSF, DAAT, and TAAT, is depicted in Table 1.

From Table 1, we can see that disjunctive queries tend to be significantly more expensive (by approximately an order of magnitude) than conjunctive queries with all of the index traversal techniques. This finding suggests the necessity of reducing the huge performance gap between disjunctive queries and conjunctive queries. Thus, we focus on disjunctive queries of LSF in this study, i.e., those that do not disqualify one document when it misses some of the results. In addition, we can find that DAAT runs

**Table 1 Average query latency of the exhaustive index traversal techniques with different numbers of results**

Algorithm	Average query latency (ms)		
	$k=10$	$k=100$	$k=1000$
OR_TAAT	1290.1	1250.8	1275.1
OR_DAAT	231.6	232.6	237.9
OR_LSF	279.6	281.2	284.8
AND_TAAT	238.7	238.3	238.4
AND_DAAT	24.1	24.1	24.3
AND_LSF	34.2	34.3	34.5

much faster than TAAT in both disjunctive and conjunctive queries. The LSF traversal performs a little slower than DAAT but much better than TAAT in both disjunctive and conjunctive queries. This is mainly due to the shortage of the main memory used for storing all document information during TAAT processing with increasing data sets and the miss rate of the cache of the CPU caused by random access in accumulators used for storing intermediate results. Overall, the LSF exhaustive index traversal technique gives comparable performance results to those of the DAAT exhaustive index traversal technique, and we intend to improve the performance further via dynamic pruning techniques.

We also measure the performance by other criteria, including some processed elements, i.e., the total number of candidates inserted into the result heap, calls to the scoring function, docids evaluated, and decompressed chunks. These criteria provide in-depth descriptions of the above query latency measurement. These criteria have also been used in previous works (Lacour *et al.*, 2008; Jonassen and Bratsberg, 2011). Table 2 shows other criteria for the six index traversal techniques with the number of results  $k = 10$ .

As observed in Table 2, the disjunctive queries processed considerably more elements than did the conjunctive queries in all cases with different exhaustive traversing techniques. This is because skipping can be used in conjunctive queries to avoid processing a large number of postings by allowing the shortest posting list to skip through other posting lists to the pointer address with the same docid. With disjunctive queries, the numbers of scoring functions called, docids evaluated, and chunks decoded are the same for DAAT and TAAT. This is because all of the postings or chunks are considered, just at different periods in the schema. However, the numbers of docids evaluated and chunks decoded of LSF

increase significantly due to the revisiting of chunks and postings. The number of scorings called is the same as the disjunctive baseline, due to the duplication checking for posting scoring. With conjunctive queries, the three criteria of DAAT and LSF are significantly reduced compared with TAAT. This is because the largest docid of the current position can be selected as the candidate document in DAAT, and a lot of postings that appear only in part of the lists can be avoided for processing. For LSF, we selected the candidate document from the posting list of the most significant terms, which must be checked first to appear in all the other posting lists. Thus, the processed elements of the LSF are the same as those of the DAAT.

**Table 2 The average number of processed elements for different index traversal techniques with the number of results  $k = 10$**

Algorithm	$N_{hi}$	$N_{sc} (\times 10^3)$	$N_{de} (\times 10^3)$	$N_{cd}$
OR_TAAT	10.0	2591.6	2591.6	284.8
OR_DAAT	119.5	2591.6	2591.6	284.8
OR_LSF	83.4	2591.6	3256.1	650.0
AND_TAAT	10.0	207.6	2591.6	284.8
AND_DAAT	50.5	76.8	227.7	250.0
AND_LSF	50.5	76.8	227.7	250.0

$N_{hi}$ : number of heap inserts;  $N_{sc}$ : number of scorings called;  $N_{de}$ : number of docids evaluated;  $N_{cd}$ : number of chunks decoded

Here, we look at mainly the number of heap inserts of the three exhaustive traversal techniques. It is worth noting that the TAAT can just do a partial sort to extract the top- $k$  results and then a full ranking on the results, which can avoid a large number of heap inserts. With conjunctive queries, the criteria are the same for LSF and DAAT; this is because the posting lists of the two are read in sequential order, which obtain the same candidate and the same threshold changes. With disjunctive queries, the number of heap inserts of LSF is almost 30% less than that of DAAT. In essence, this is mainly because LSF sorts the terms in descending frequency order and processes postings in important terms first. Thus, the threshold is significantly promoted and greatly reduces the number of candidates that are inserted into the result heap. This allows us to consider whether the advantage of LSF retrieval can be used for further query processing improvements by dynamic pruning techniques.

## 5.2 LSF dynamic pruning

As well-known dynamic pruning techniques, MaxScore and WAND differ in list sorting and partial scoring, but both achieve the best performance in query processing of large-scale document-sorted indexes. To evaluate the performance of our proposed LSF-based dynamic pruning techniques on document-sorted indexes, we present a detailed comparison of WAND, MaxScore, LSF\_LO, and LSF\_PS. We re-implemented the WAND approach using JAVA referring to the code provided by Dimopoulos *et al.* (2013) and the MaxScore approach based on the code available at <https://github.com/s-j/laika>. In the following subsection, we compare mainly two non-partial scoring methods and two partial scoring methods. We measure the performance according to the following criteria: average time per query, the numbers of chunks decoded, docids evaluated, calls to the scoring function, and candidates inserted into the result heap.

Table 3 shows the average query processing time of the four dynamic pruning techniques with different query lengths. MaxScore and WAND both achieve significant performance gains compared with the OR\_DAAT technique, and MaxScore performs better due to the costly list sorting phase of WAND and its lack of consideration of term importance. LSF\_LO achieves significant performance gains compared with the exhaustive traversal, which shows the effect of consideration of term importance and list omitting. We find that LSF\_PS achieves better performance with term length larger than two compared with MaxScore. The reason is that the query terms of LSF-based techniques are sorted in descending order, and we select the candidate document in terms that are more important.

**Table 3 Average query latency of different dynamic pruning techniques for different numbers of query terms**

Algorithm	Average query latency (ms)					
	Average	2	3	4	5	> 5
WAND	47.3	30.6	44.6	58.4	68.3	103.7
MaxScore	35.6	26.7	33.3	38.8	47.7	63.1
LSF_LO	61.1	38.9	56.9	70.2	89.9	140.3
LSF_PS	34.4	28.3	33.2	37.8	45.9	58.6

Second, we note that the performance gap between LSF\_PS and LSF\_LO increases with the

increase of the number of query terms, and this also occurs between LSF\_PS and WAND. With the increase of term length, there is significant improvement compared with MaxScore, due to the consideration of the LSF scheme. This phenomenon proves the importance of partial scoring in the LSF inverted index traversal. With more query terms, it causes more mis-scoring parts of the query terms, which can slow down the candidate document checking. Overall, our LSF-based dynamic pruning techniques achieve comparable query processing performance to the best DAAT-based dynamic pruning technique.

Table 4 shows other criteria for the four pruning techniques that provide an in-depth demonstration of the performance. The numbers of candidates inserted into the result heap are almost the same for the two DAAT-based techniques, and also for the two LSF-based techniques. The costly revisiting of posting lists makes LSF\_LO process a few more elements than the other three techniques, including scoring functions called, docids evaluated, and blocks decoded, although most of them can be ignored by list omitting. However, the number of candidate documents inserted into the result heap is sharply reduced for LSF-based techniques compared with that of the DAAT-based techniques. This is mainly due to the important term selection in LSF. The threshold of the result heap of LSF-based techniques grows faster than that of the DAAT-based techniques, which avoids a large number of mis-inserts into the result heap. This phenomenon proves the superiority of LSF to DAAT in dynamic pruning techniques.

**Table 4** The average number of processed elements of different dynamic pruning techniques with the number of results  $k = 10$

Algorithm	$N_{hi}$	$N_{sc} (\times 10^3)$	$N_{de} (\times 10^3)$	$N_{cd}$
WAND	119.4	114.6	379.5	280.0
MaxScore	119.4	215.4	238.9	223.3
LSF_LO	83.4	258.6	393.9	324.0
LSF_PS	83.4	188.5	219.0	240.6

$N_{hi}$ : number of heap inserts;  $N_{sc}$ : number of scorings called;  $N_{de}$ : number of docids evaluated;  $N_{cd}$ : number of chunks decoded

Furthermore, the best performance achieved by LSF\_PS is significantly better on the three criteria when compared with LSF\_LO using partial scoring techniques. The numbers of heap inserts, docids evaluated, and scoring functions called for LSF\_PS

are less than those of the best performance MaxScore, and the numbers of blocks decoded are almost the same. Although both LSF\_PS and MaxScore use partial scoring techniques, the candidates selected as more important terms in LSF\_PS make the termination condition come early. With these explicit results, we can conclude that the improvement for the techniques lies mainly in highly efficient LSF index traversal and partial scoring operations.

### 5.3 Extensions

Real search engines use ranking functions based on hundreds of features. However, such functions are quite expensive to evaluate. To achieve high efficiency, search engines commonly separate the ranking process into two or more phases (Wang *et al.*, 2011). In the first phase, a very simple and fast ranking function is used to score the full postings that match the query and then return the top- $k$  scored documents. In the following phases, more increasingly complicated ranking functions with more and more features are applied to documents that pass through the earlier phases. Thus, the later phases examine only a fairly small number of result candidates, and a significant amount of computation is still spent in the first phase. A simple and fast ranking function is adopted to obtain the top 100 or 1000 documents in the first phase. Then a complicated ranking function with hundreds of features is used to explicitly evaluate the results.

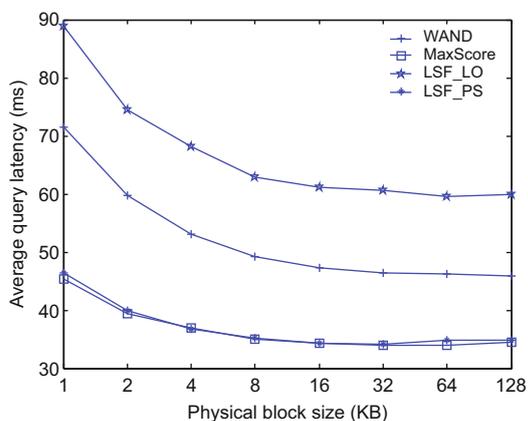
Table 5 shows the experimental results of the four dynamic pruning techniques as we increase the number of results for a first phase ranking. We find that the query latency of the four techniques increases with  $k$  and that LSF\_PS achieves a slightly better stable performance than MaxScore with different  $k$ . One reason is that we focus on techniques in which the temporal data structure that can be influenced by  $k$  is avoided. Thus, the size of the result set influences only the number of heap inserts and does not affect any other counts, e.g., memory cost. With the increase of the number of results, we find that the gap between LSF\_LO and MaxScore is reduced. This is because the threshold of the result heap keeps a lower value for a larger number of results, and the effect of partial scoring is reduced. However, the important term selection and list omitting make LSF-based dynamic pruning techniques stop early. In fact, we also note that the superiority

of LSF\_PS increases compared with MaxScore for a larger number of results.

Fig. 4 illustrates the average latency of the pruning techniques as we increase the physical size of the fetched blocks (varying from 1 KB to 128 KB). As the results show, the query efficiency improves gradually as the block size becomes larger. The decrease in the measured latency between 1 KB and 128 KB blocks is 32.6% for LSF\_LO and 25.0% for LSF\_PS. The query efficiency becomes roughly stable when the size of block is larger than 16 KB. This can be explained by a decrease in random disk-seeks for the larger physical block size and the fact that when compressed data is split between two smaller physical blocks, it requires two block fetches to fetch the data chunk.

**Table 5 Average query latency of different dynamic pruning techniques with different numbers of results**

Algorithm	Average query latency (ms)				
	$k = 10$	$k = 50$	$k = 100$	$k = 500$	$k = 1000$
WAND	47.3	54.4	62.4	76.6	86.3
MaxScore	35.6	45.5	51.5	66.6	79.0
LSF_LO	61.1	67.3	70.4	86.9	94.4
LSF_PS	34.4	41.5	45.1	63.7	71.1



**Fig. 4 Average query latency of different dynamic pruning techniques with various physical block sizes ( $k = 10$ )**

## 6 Conclusions

In this paper, we have proposed a novel exhaustive index traversal technique called largest scores first (LSF). It can reduce the memory consumption and candidate selection cost of existing methods at

the expense of revisiting the posting lists of the remaining query terms. Preliminary analysis and implementation showed comparable performance between LSF and existing methods. To further reduce the number of postings that need to be revisited, we have also presented efficient dynamic pruning techniques based on LSF, including list omitting (LSF\_LO) and partial scoring (LSF\_PS). The two dynamic pruning techniques take advantage of the term importance consideration of LSF, which gives prior concern to more promising candidates. Experimental results with the TREC GOV2 collection showed that our LSF retrieval is better than the TAAT retrieval and is comparable to the DAAT retrieval, but has potential advantages over dynamic pruning techniques. Further experiments showed that our LSF-based pruning techniques reduce disjunctive query processing time by almost 27% over the WAND baseline and lead to slightly better results compared with the MaxScore baseline, while returning the same results as the exhaustive evaluation. With more query terms or more returned results, our LSF-based pruning schemes (including list omitting and partial scoring) showed more improvements compared with the two baselines.

Further investigations will extend to the LSF-based dynamic pruning techniques on the block-max indexes (Chakrabarti *et al.*, 2011; Ding and Suel, 2011), an augmented structure that stores the piecewise maximum score of each block in a posting list, to obtain a more exact estimate of the upper bound score of candidates. We believe that the adoption of block-max indexes is orthogonal to our work and can further improve the query processing performance of the LSF-based dynamic pruning techniques.

## References

- Anh, V.N., Moffat, A., 2005. Simplified similarity scoring using term ranks. Proc. 28th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval, p.226-233. <http://dx.doi.org/10.1145/1076034.1076075>
- Anh, V.N., Moffat, A., 2006. Pruned query evaluation using pre-computed impacts. Proc. 29th Annual ACM SIGIR Conf. on Research and Development in Information Retrieval, p.372-379. <http://dx.doi.org/10.1145/1148170.1148235>
- Anh, V.N., Moffat, A., 2010. Index compression using 64-bit words. *Softw. Pract. Exper.*, **40**(2):131-147. <http://dx.doi.org/10.1002/spe.948>
- Badue, C., Ribeiro-Neto, B., Baeza-Yates, R., *et al.*, 2001. Distributed query processing using partitioned inverted files. Proc. 8th Int. Symp. on String Processing and

- Information Retrieval, p.10-20.  
<http://dx.doi.org/10.1109/SPIRE.2001.989733>
- Broder, A.Z., Carmel, D., Herscovici, M., et al., 2003. Efficient query evaluation using a two-level retrieval process. Proc. 12th Int. Conf. on Information and Knowledge Management, p.426-434.  
<http://dx.doi.org/10.1145/956863.956944>
- Buckley, C., Lewit, A.F., 1985. Optimization of inverted vector searches. Proc. 8th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval, p.97-110. <http://dx.doi.org/10.1145/253495.253515>
- Büttcher, S., Clarke, C.L.A., 2007. Index compression is good, especially for random access. Proc. 16th ACM Conf. on Information and Knowledge Management, p.761-770. <http://dx.doi.org/10.1145/1321440.1321546>
- Büttcher, S., Clarke, C.L.A., Cormack, G.V., 2010. Information Retrieval: Implementing and Evaluating Search Engines. The MIT Press, USA.
- Chakrabarti, K., Chaudhuri, S., Ganti, V., 2011. Interval-based pruning for top- $k$  processing over compressed lists. Proc. 27th Int. Conf. on Data Engineering, p.709-720.  
<http://dx.doi.org/10.1109/ICDE.2011.5767855>
- Croft, B., Metzler, D., Strohman, T., 2010. Search Engines: Information Retrieval in Practice. Addison Wesley, USA.
- Dean, J., 2009. Challenges in building large-scale information retrieval systems: invited talk. Proc. 2nd ACM Int. Conf. on Web Search and Data Mining, p.1.  
<http://dx.doi.org/10.1145/1498759.1498761>
- Delbru, R., Campinas, S., Tummarello, G., 2012. Searching web data: an entity retrieval and high-performance indexing model. *Web Semant. Sci. Serv. Agents World Wide Web*, **10**:33-58.  
<http://dx.doi.org/10.1016/j.websem.2011.04.004>
- Dimopoulos, C., Nepomnyachiy, S., Suel, T., 2013. Optimizing top- $k$  document retrieval strategies for block-max indexes. Proc. 6th ACM Int. Conf. on Web Search and Data Mining, p.113-122.  
<http://dx.doi.org/10.1145/2433396.2433412>
- Ding, S., Suel, T., 2011. Faster top- $k$  document retrieval using block-max indexes. Proc. 34th Int. ACM SIGIR Conf. on Research and Development in Information Retrieval, p.993-1002.  
<http://dx.doi.org/10.1145/2009916.2010048>
- Fontoura, M., Josifovski, V., Liu, J.H., et al., 2011. Evaluation strategies for top- $k$  queries over memory-resident inverted indexes. *Proc. VLDB Endow.*, p.1213-1224.
- Jiang, K., Yang, Y.X., 2015. Exhaustive hybrid posting lists traversing technique. Proc. 5th Int. Conf. on Intelligence Science and Big Data Engineering, p.1-11.  
[http://dx.doi.org/10.1007/978-3-319-23862-3\\_1](http://dx.doi.org/10.1007/978-3-319-23862-3_1)
- Jiang, K., Song, X.S., Yang, Y.X., 2014. Performance evaluation of inverted index traversal techniques. Proc. 17th Int. Conf. on Computational Science and Engineering, p.1715-1720. <http://dx.doi.org/10.1109/CSE.2014.315>
- Jonassen, S., Bratsberg, S.E., 2011. Efficient compressed inverted index skipping for disjunctive text-queries. Proc. 33rd European Conf. on Advances in Information Retrieval, p.530-542.  
[http://dx.doi.org/10.1007/978-3-642-20161-5\\_53](http://dx.doi.org/10.1007/978-3-642-20161-5_53)
- Lacour, P., Macdonald, C., Ounis, I., 2008. Efficiency comparison of document matching techniques. Proc. European Conf. on Information Retrieval, p.37-46.
- Lester, N., Moffat, A., Webber, W., et al., 2005. Space-limited ranked query evaluation using adaptive pruning. Proc. 6th Int. Conf. on Web Information Systems Engineering, p.470-477.  
[http://dx.doi.org/10.1007/11581062\\_37](http://dx.doi.org/10.1007/11581062_37)
- Macdonald, C., Ounis, I., Tonellotto, N., 2011. Upper-bound approximations for dynamic pruning. *ACM Trans. Inform. Syst.*, **29**(4):17.1-17.28.  
<http://dx.doi.org/10.1145/2037661.2037662>
- Manning, C.D., Raghavan, P., Schütze, H., 2008. Introduction to Information Retrieval. Cambridge University Press, Cambridge, USA.
- Melink, S., Raghavan, S., Yang, B., et al., 2001. Building a distributed full-text index for the Web. Proc. 10th Int. Conf. on World Wide Web, p.396-406.  
<http://dx.doi.org/10.1145/371920.372095>
- Moffat, A., Zobel, J., 1996. Self-indexing inverted files for fast text retrieval. *ACM Trans. Inform. Syst.*, **14**(4):349-379. <http://dx.doi.org/10.1145/237496.237497>
- Ounis, I., Amati, G., Plachouras, V., et al., 2006. Terrier: a high performance and scalable information retrieval platform. Proc. OSIR Workshop, p.18-25.
- Puppini, D., Silvestri, F., Perego, R., et al., 2010. Tuning the capacity of search engines: load-driven routing and incremental caching to reduce and balance the load. *ACM Trans. Inform. Syst.*, **28**(2):5.1-5.36.  
<http://dx.doi.org/10.1145/1740592.1740593>
- Silvestri, F., Venturini, R., 2010. VSEncoding: efficient coding and fast decoding of integer lists via dynamic programming. Proc. 19th ACM Int. Conf. on Information and Knowledge Management, p.1219-1228.  
<http://dx.doi.org/10.1145/1871437.1871592>
- Strohman, T., Croft, W.B., 2007. Efficient document retrieval in main memory. Proc. 30th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval, p.175-182.  
<http://dx.doi.org/10.1145/1277741.1277774>
- Strohman, T., Turtle, H., Croft, W.B., 2005. Optimization strategies for complex queries. Proc. 28th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval, p.219-225.  
<http://dx.doi.org/10.1145/1076034.1076074>
- Turtle, H., Flood, J., 1995. Query evaluation: strategies and optimizations. *Inform. Process. Manag.*, **31**(6):831-850. [http://dx.doi.org/10.1016/0306-4573\(95\)00020-H](http://dx.doi.org/10.1016/0306-4573(95)00020-H)
- Wang, L.D., Lin, J., Metzler, D., 2011. A cascade ranking model for efficient ranked retrieval. Proc. 34th Int. ACM SIGIR Conf. on Research and Development in Information Retrieval, p.105-114.  
<http://dx.doi.org/10.1145/2009916.2009934>
- Zobel, J., Moffat, A., 2006. Inverted files for text search engines. *ACM Comput. Surv.*, **38**(2):6.1-6.56.  
<http://dx.doi.org/10.1145/1132956.1132959>
- Zukowski, M., Heman, S., Nes, N., et al., 2006. Super-scalar RAM-CPU cache compression. Proc. 22nd Int. Conf. on Data Engineering, p.59.  
<http://dx.doi.org/10.1109/ICDE.2006.150>