Available online at www.sciencedirect.com

SCIENCE DIRECT®

Information Processing and Management 42 (2006) 875–898

INFORMATION PROCESSING & MANAGEMENT

www.elsevier.com/locate/infoproman

Performance of query processing implementations in ranking-based text retrieval systems using inverted indices

B. Barla Cambazoglu, Cevdet Aykanat *

Computer Engineering Department, Bilkent University, TR 06800 Bilkent, Ankara, Turkey

Received 22 May 2005; accepted 16 June 2005
Available online 10 August 2005

Abstract

Similarity calculations and document ranking form the computationally expensive parts of query processing in ranking-based text retrieval. In this work, for these calculations, 11 alternative implementation techniques are presented under four different categories, and their asymptotic time and space complexities are investigated. To our knowledge, six of these techniques are not discussed in any other publication before. Furthermore, analytical experiments are carried out on a 30 GB document collection to evaluate the practical performance of different implementations in terms of query processing time and space consumption. Advantages and disadvantages of each technique are illustrated under different querying scenarios, and several experiments that investigate the scalability of the implementations are presented.
© 2005 Elsevier Ltd. All rights reserved.

Keywords: Text retrieval; Query processing; Inverted index; Similarity calculations; Document ranking; Complexity; Scalability

1. Introduction

In the last decade, a shift has been observed from the Boolean model of query processing to the more effective ranking-based model. In text retrieval systems employing the ranking-based model, similarity calculations are performed between a user query and the documents in a collection. As a result of these calculations, the user is presented a set of relevant documents, ranked in decreasing order of relevance to the query. The similarity calculations and document ranking, which form the major source of overhead in

* Corresponding author. Tel.: +90 312 290 1625; fax: +90 312 266 4047.
E-mail addresses: berkant@cs.bilkent.edu.tr (B.B. Cambazoglu), aykanat@cs.bilkent.edu.tr (C. Aykanat).

0306-4573/$ - see front matter © 2005 Elsevier Ltd. All rights reserved.
doi:10.1016/j.ipm.2005.06.004

query processing, can be implemented in many ways, using different data structures and algorithms. The main focus of this work is on advantages and disadvantages of these data structures and algorithms.

Although other strategies may also be employed (Croft & Savino, 1988), a document collection is usually represented by an inverted index (Tomasic, Garcia-Molina, & Shoens, 1994; Zobel, Moffat, & Sacks-Davis, 1992). An inverted index is composed of two parts: a set of inverted lists and an index into these lists. The set of inverted lists $\mathscr{L} = \{\mathcal{I}_1, \mathcal{I}_2, \ldots, \mathcal{I}_T\}$ of size T, where T is the number of distinct terms in the collection, contains a list $\mathcal{I}_i$ for each term t_i in the collection. The index part contains a pointer to each term's inverted list. Each inverted list $\mathcal{I}_i$ keeps entries, called postings, about the documents in which term t_i appears. A posting $p \in \mathcal{I}_i$ includes a document id field $p.d = j$ and a weight field $p.w = w(t_i, d_j)$ for a document d_j containing term t_i, where $w(t_i, d_j)$ is a weight (Harman, 1986) which indicates the degree of relevance between t_i and d_j.

In construction of the inverted index, usually, the tf-idf (term frequency-inverse document frequency) weighting scheme (Salton & McGill, 1983) is used to compute $w(t_i, d_j)$. In this work, we use the following tf-idf variant

$$w(t_i, d_j) = \frac{f(t_i, d_j)}{\sqrt{|d_j|}} \times \ln \frac{D}{f(t_i)}, \tag{1}$$

where $f(t_i, d_j)$ is the number of times term t_i appears in document d_j, $|d_j|$ is the total number of terms in d_j, $f(t_i)$ is the number of documents containing t_i, and D is the number of documents.

In processing a query, only the inverted lists associated with the query terms are used. Specifically, if we have a query $\mathcal{Q} = \{t_{q_1}, t_{q_2}, \ldots, t_{q_Q}\}$ of Q distinct query terms, we work on a partial inverted index $\mathscr{L}_Q \subset \mathscr{L}$ of Q inverted lists, in which each list $\mathcal{I}_{q_i} \in \mathscr{L}_Q$ is associated with query term $t_{q_i} \in \mathcal{Q}$. The similarity $\mathrm{sim}(\mathcal{Q}, d_j)$ of query $\mathcal{Q}$ to a document d_j can be calculated using the cosine rule (Salton & McGill, 1983). Since, in Eq. (1), we already approximated cosine normalization by the $\sqrt{|d_j|}$ factor (Lee, Chuang, & Seamons, 1997), the cosine similarity metric can be simplified as

$$\mathrm{sim}(\mathcal{Q}, d_j) = \sum_{t_{q_i} \in \mathcal{Q}} w(t_{q_i}, d_j) \tag{2}$$

assuming that all query terms have equal importance. That is, to calculate the similarity between query $\mathcal{Q}$ and document d_j, we need to accumulate the weights $w(t_{q_i}, d_j)$ for each query term $t_{q_i} \in \mathcal{Q}$ in a memory location dedicated to document d_j. These memory locations are called accumulators. An accumulator a typically keeps an integer document id field $a.d$ and a floating point score field $a.s$, which contains the accumulated similarity value for document $a.d$. After all accumulator updates are completed, sorting them in decreasing order of finalized $a.s$ values gives a ranking of documents.

Both time and space are critical in ranking-based text retrieval. Especially, in cases where the inverted index is completely stored in volatile memory (a common practice for Web search engines) and disk accesses are avoided, similarity calculations and document ranking directly determine the query processing times. Considering the existence of search engines which indexed more than four billion pages, it is easily seen that space consumption is also a critical issue. In this work, we present 11 alternative implementations under four different categories for query processing in ranking-based text retrieval, taking time and space needs into consideration. To our knowledge, six of these implementations are not discussed in any publication before.

The rest of the paper is organized as follows. In Section 2, we give pointers to the related work on efficient query processing. In Section 3, we describe the implementation techniques and present an analysis of their asymptotic time and space complexities. In Section 4, we evaluate the practical performance of each technique on a large (30 GB) document collection. In Section 5, we present a discussion on advantages and disadvantages of the techniques and conclude.

2. Related work

In the literature, ranking-based text retrieval is well-studied in terms of both effectiveness (Can, Altingovde, & Demir, 2004; Clarke, Cormack, & Tudhope, 2000; Wilkinson, Zobel, & Sacks-Davis, 1995) and efficiency (Can et al., 2004; Long & Suel, 2003). Some of the basic query processing techniques are described in classical information retrieval books (Baeza-Yates & Ribeiro-Neto, 1999; Frakes & Baeza-Yates, 1992; Salton & McGill, 1983; Witten, Moffat, & Bell, 1999). Many optimizations are proposed for decreasing query processing times and efficiently using the memory (Buckley & Lewit, 1985; Harper, 1980; Lucarella, 1988; Moffat, Zobel, & Sacks-Davis, 1994; Persin, 1994; Smeaton & van Rijsbergen, 1981; Turtle & Flood, 1995; Wong & Lee, 1993). These optimizations are based on limiting the number of processed query terms and postings (short-circuit evaluation) or limiting the memory allocated to accumulators. They mainly differ in their choice for the processing order of postings and when to stop processing them.

Buckley and Lewit (1985) proposed an algorithm which traverses query terms in decreasing order of frequencies and limits the number of processed query terms by not evaluating the inverted lists for high-frequency terms whose postings cannot affect the final ranking. Harman and Candela (1990) used an insertion threshold on query terms, and the terms whose score contribution are below this threshold are not allowed to allocate new accumulators. Moffat et al. (1994) proposed two heuristics which place a hard limit on the memory allocated to accumulators. Turtle and Flood (1995) presented simulation results for the performance analysis of two optimization techniques, which employ term-ordered and document-ordered inverted list traversal. Wong and Lee (1993) proposed two optimization heuristics which traverse postings in decreasing magnitude of weights. For a similar strategy, Persin (1994) used thresholds for allocation and update of accumulators.

These optimizations can be classified as safe or approximate (Turtle & Flood, 1995). Safe optimizations guarantee that best-matching documents are ranked correctly. Approximate optimizations may trade effectiveness for efficiency producing a partial ranking, which does not necessarily contain the best-matching documents, or may present them in an incorrect order. Our focus in this work is not on partial query evaluation or approximate optimizations. We investigate the complexities of implementations and data structures in total document ranking as well as their performance in practice.

Throughout the paper, we take an information retrieval point of view in analyzing various implementation techniques. However, there exists a significant amount of related work in the database literature. The interested reader may refer to prior works by Lehman and Carey (1986), Goldman, Shivakumar, Venkatasubramanian, and Garcia-Molina (1998), Bohannon, McIlroy, and Rastogi (2001), Hristidis, Gravano, and Papakonstantinou (2003), Elmasri and Navathe (2003) and Ilyas et al. (2004).

3. Query processing implementations

The analyses presented in this work are based on processing of a single query $\mathcal{Q} = \{t_{q_1}, t_{q_2}, \ldots, t_{q_Q}\}$ with Q distinct terms over a document collection with D documents. u denotes the total number of postings in the processed Q inverted lists $\mathcal{I}_{q_i} \in \mathscr{L}_Q$, all of which are stored in the volatile memory. The number of distinct document ids in these postings is denoted by e. The text retrieval system returns the most relevant (highly ranked) s documents to the user as the result of the query. Table 1 displays the notation used in the paper.

Although other orderings are possible, the postings in our inverted lists are ordered by increasing document id since this ordering is strictly required by some of the algorithms we implemented. Moreover, this ordering is necessary in case inverted index is compressed (Bell, Moffat, Nevill-Manning, Witten, & Zobel, 1993; Zobel & Moffat, 1995). In postings, we store normalized tf scores $(f(t_i, d_j)/\sqrt{|d_j|})$, thus eliminating the need to lookup the document lengths ($|d_j|$) and allocate a large array to store them. This way, the main

Table 1
The notation used in the paper

Symbol	Description
T	The number of distinct terms in the collection
D	The number of documents in the collection
t_i	A term in the collection
d_i	A document in the collection
$\|d_i\|$	The total number of terms in d_i
$\mathscr{L}$	The set of inverted lists
$\mathcal{I}_i$	The inverted list associated with t_i
$p.d$, $p.w$	Document id and weight fields of a posting p
$f(t_i, d_j)$	The number of times t_i appears in d_j
$f(t_i)$	The number of documents containing t_i
$\mathcal{Q}$	A user query
Q	The number of distinct terms in $\mathcal{Q}$
$\mathscr{L}_{\mathcal{Q}}$	The partial set of inverted lists processed in answering $\mathcal{Q}$
$a.d$, $a.s$	Document id and score fields of an accumulator a
u	The total number of postings in all $\mathcal{I}_i \in \mathscr{L}_{\mathcal{Q}}$
e	The number of postings with distinct document ids in all $\mathcal{I}_i \in \mathscr{L}_{\mathcal{Q}}$
s	The number of documents to be returned to the user
B	The number of buckets in the hashing implementation

space demand is for the accumulators and the postings in the inverted lists. The idf component ($\ln(D/f(t_i))$) is not pre-computed in postings but computed during query processing, allowing easy updates over the inverted index.

In a query processing implementation, depending on the operations on accumulators, we distinguish five phases which affect the processing time of a query: *creation*, *update*, *extraction*, *selection*, and *sorting*. Descriptions of these phases are given below.

- *Creation*: Each document d_i is associated with an accumulator a_i, initialized as $a_i.d = i$ and $a_i.s = 0$. Depending on the implementation, either previously allocated locations are used as accumulators or space is dynamically allocated for accumulators as needed. In this phase, some auxiliary data structures may also be allocated and initialized.
- *Update*: Once an accumulator a_i is created for a document d_i, the weight $p.w$ of each posting p where $p.d = i$ is simply added to the score of accumulator a_i, i.e., $a_i.s = a_i.s + p.w$. It is necessary and sufficient to perform u updates since each posting incurs a single update.
- *Extraction*: The accumulators with nonzero scores (i.e., $a_i.s > 0$) whose updates are completed can be extracted. Such accumulators are located and passed to the selection phase as input. Since an accumulator is extracted exactly once, there are always e extraction operations.
- *Selection*: This phase compares each extracted accumulator score with the previously extracted ones and selects the accumulators having the top s scores. This way, the set $\mathcal{S}_{\text{top}}$ of best-matching documents is constructed.
- *Sorting*: The accumulators in $\mathcal{S}_{\text{top}}$ are sorted in decreasing order of their scores, and their document ids are returned to the user in this sorted order.

The asymptotic run-time costs for the creation, update, extraction, selection and sorting phases are represented by Time_{C}, Time_{U}, Time_{E}, Time_{S}, and Time_{R}, respectively. We represent the total run-time cost of an implementation by Time_{T} and the storage cost by S. In all analyses, we strictly have $e \leqslant D$, $e \leqslant u$, $s \leqslant e$, and $u \leqslant QD$. Moreover, we assume $s \ll D$, $Q \ll T$, and $u = O(D)$.

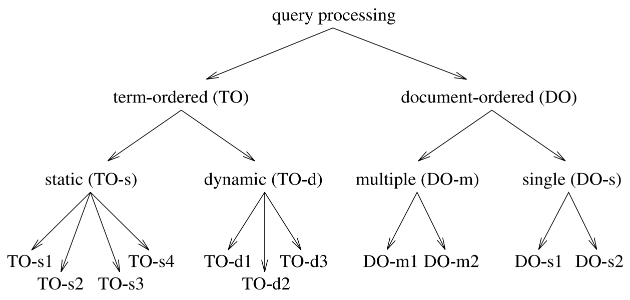

Fig. 1. A classification for query processing implementations.

Depending on the processing order of postings, we make a broad classification of query processing implementations as term-ordered (TO) and document-ordered (DO).We further classify TO processing as static (TO-s) and dynamic (TO-d), according to the strategy used in allocation of accumulators. Similarly, we classify DO processing as multiple (DO-m) and single (DO-s), according to the number of accumulators allocated. For TO-s, TO-d, DO-m, and DO-s approaches, we present 4, 3, 2, and 2 implementations, respectively (Fig. 1). To the best of our knowledge, the implementations TO-s4, TO-d1, TO-d2, TO-d3, DO-m1, and DO-m2 are not discussed in any other publication.

3.1. Implementations for term-ordered (TO) processing

In TO processing, inverted lists are sequentially processed. The postings of a term are completely exhausted before the postings of the next term are processed. Extraction and selection phases are performed in an interleaved manner. In TO-s, D accumulators are allocated statically. In TO-d, at most e accumulators are allocated on demand, thus saving space if D is very high.

3.1.1. Implementations with static accumulator allocation (TO-s)

In TO-s implementations, an array $\mathscr{A}$ of D accumulators is statically allocated. Each array element $a_i = \mathscr{A}[i]$ is used as an accumulator. Before processing a query, accumulator fields are initialized as $a_i.d = i$ and $a_i.s = 0$. Similarity updates for document d_i are performed over $a_i.s$. Creation and update phases are the same for all TO-s implementations. These implementations mainly differ in extraction, selection, and sorting phases. The algorithm for TO-s implementations is given in Fig. 2. In this section, we describe four different TO-s implementations.

3.1.1.1. TO-s1: accumulator array, accumulators with nonzero scores sorted. The most naive implementation is to sort all accumulators in $\mathscr{A}$ in decreasing order of their scores and return the document ids in the first s

```
TO-S($\mathcal{Q}$, $\mathcal{A}$)
    **for** each accumulator $a_i \in \mathcal{A}$ **do**
        *INITIALIZE* $a_i$ as $a_i.d = i$ and $a_i.s = 0$
    **for** each query term $t_{q_j} \in \mathcal{Q}$ **do**
        **for** each posting $p \in \mathcal{I}_{q_j}$ **do**
            *UPDATE* $a_{p.d}.s$ as $a_{p.d}.s + p.w$
    $\mathcal{S}_{\text{top}} = \emptyset$
    *INSERT* the accumulators having the top $s$ scores into $\mathcal{S}_{\text{top}}$
    *SORT* the accumulators in $\mathcal{S}_{\text{top}}$ in decreasing order of their scores
    *RETURN* $\mathcal{S}_{\text{top}}$
```

Fig. 2. The algorithm for TO-s implementations.

accumulators. If $e \ll D$, most accumulators are never updated and their score fields remain zero. In this case, it is better to first pick the nonzero accumulators and then sort those (Witten et al., 1999). Costs for this approach are as follows:

- *Creation*: Array $\mathscr{A}$ of D accumulators is allocated, and its accumulators are initialized. This type of allocation is a one-time $O(D)$-cost operation independent of the number of incoming queries. However, reinitialization of the accumulators between consecutive queries require $O(e)$ operations. Hence, $\text{Time}_{\text{C}} = O(e)$.
- *Update*: Each term q_j is considered in turn, and for each posting $p \in \mathcal{I}_{q_j}$ with $p.d = i$, an update is performed over the corresponding accumulator field $a_i.s$, i.e., $a_i.s = a_i.s + p.w$. This phase involves reading and writing a total of u values between two locations. Hence, $\text{Time}_{\text{U}} = O(u)$.
- *Extraction*: Since it is not known which accumulators have nonzero score fields, the whole $\mathscr{A}$ array must be traversed to locate them. During this traversal, nonzero accumulators are picked and stored at the first e elements of array $\mathscr{A}$. Traversing the whole array and checking the score fields require $O(D)$ comparisons. Hence, $\text{Time}_{\text{E}} = O(D)$.
- *Selection*: This phase involves no work since the top s scores to be selected already reside within the first e array elements. $\text{Time}_{\text{S}} = O(1)$.
- *Sorting*: Sorting the first e array elements in decreasing order of the scores gives a ranking. The document ids in the first s array elements are returned as the set $\mathcal{S}_{\text{top}}$ of best-matching documents. Sorting has a cost of $\text{Time}_{\text{R}} = O(e \lg e)$.

The running time of this implementation is $\text{Time}_{\text{T}} = O(e + u + D + 1 + e \lg e) = O(D + e \lg e)$. The storage overhead is $S = O(D)$.

3.1.1.2. TO-s2: accumulator array, max-priority queue for nonzero accumulators. An improvement over TO-s1 is to use a max-priority queue implemented as a binary heap $\mathscr{H}_{\text{max}}$ to select the top s accumulator scores (Moffat et al., 1994). The max-heap $\mathscr{H}_{\text{max}}$ contains e accumulators, keyed by their scores. This approach avoids the cost of sorting the whole set of nonzero accumulators if $s < e$.

- *Creation*, *Update*: Similar to TO-s1. $\text{Time}_{\text{C}} = O(e)$, $\text{Time}_{\text{U}} = O(u)$. Note that array $\mathscr{A}$ can be used in order to store the accumulators in $\mathscr{H}_{\text{max}}$. Hence, no extra storage is necessary for implementing the max-priority queue.
- *Extraction*: Similar to TO-s1. $\text{Time}_{\text{E}} = O(D)$.
- *Selection*: Extracted accumulators in the first e elements of array $\mathscr{A}$ are treated as elements of heap $\mathscr{H}_{\text{max}}$, using their score fields as the key and document id fields as the data. Since there are e extracted accumulators, the heap can be built with $O(e)$ operations. After building, the root of $\mathscr{H}_{\text{max}}$ keeps the accumulator with the highest score. The top s accumulators are obtained by repeatedly performing s extract-max operation on $\mathscr{H}_{\text{max}}$. $\text{Time}_{\text{S}} = O(e + s \lg e)$.
- *Sorting*: This phase involves no work since accumulators are extracted from $\mathscr{H}_{\text{max}}$ in sorted order during the selection phase. $\text{Time}_{\text{R}} = O(1)$.

$$\text{Time}_{\text{T}} = O(e + u + D + e + s \lg e) + 1 = O(D + s \lg e).\ S = O(D).$$

3.1.1.3. TO-s3: accumulator array, min-priority queue for top s accumulators. A variation over TO-s2 is to employ, instead of a max-priority queue, a min-priority queue implemented as a min-heap $\mathscr{H}_{\text{min}}$ (Witten et al., 1999). At any time, the min-heap $\mathscr{H}_{\text{min}}$ contains at most s accumulators, keyed by their scores.

- *Creation*, *Update*: Similar to TO-s1. $\text{Time}_\text{C} = O(e)$, $\text{Time}_\text{U} = O(u)$.
- *Extraction*: The $\mathscr{A}$ array is traversed, and nonzero accumulators are passed to the selection phase. $\text{Time}_\text{E} = O(D)$.
- *Selection*: As long as the number of accumulators in $\mathscr{H}_\text{min}$ is less than s, extracted accumulators are simply added to $\mathscr{H}_\text{min}$. Once it contains s accumulators, $\mathscr{H}_\text{min}$ is built. After this point, the root of $\mathscr{H}_\text{min}$ keeps a_smin, the accumulator with the minimum score observed so far. The score $a.s$ of each extracted accumulator a is compared with $a_\text{smin}.s$. If the incoming score $a.s$ is less than the current minimum $a_\text{smin}.s$, the accumulator a is simply ignored. Otherwise, accumulator a_smin is removed from $\mathscr{H}_\text{min}$, and the extracted accumulator a is inserted into $\mathscr{H}_\text{min}$. Building the min-heap from the first s extracted accumulators has a cost of $O(s)$. In the worst case, all remaining accumulators must be inserted into $\mathscr{H}_\text{min}$. This has a cost of $O((e-s)\lg s)$. Hence, $\text{Time}_\text{S} = O(s + (e-s)\lg s)$.
- *Sorting*: Accumulators in $\mathscr{H}_\text{min}$ are sorted in decreasing order of scores. $\text{Time}_\text{R} = O(s\lg s)$.

$$\text{Time}_\text{T} = O(e + u + D + (s + (e-s)\lg s) + s\lg s) = O(D + e\lg s).\ S = O(D).$$

3.1.1.4. TO-s4: accumulator array, sth largest score selection. This method relies on the observation that the accumulator with the smallest score to be entered into the set $\mathcal{S}_\text{top}$ of top s accumulators can be located in linear time.

- *Creation*, *Update*: Similar to TO-s1. $\text{Time}_\text{C} = O(e)$, $\text{Time}_\text{U} = O(u)$.
- *Extraction*: This phase involves no work. $\text{Time}_\text{E} = O(1)$.
- *Selection*: The accumulator with the sth largest score can be selected in worst-case linear time by the median-of-medians selection algorithm (Cormen, Leiserson, Rivest, & Stein, 2001) over the accumulators in $\mathscr{A}$. Instead of this algorithm, the randomized selection algorithm (Cormen et al., 2001), which has expected linear-time complexity, could be used for run-time efficiency in practice. This algorithm returns $a_{s\text{th}}$, the accumulator having the sth largest score and places the remaining $s-1$ accumulators that should appear in $\mathcal{S}_\text{top}$ in the array elements following $a_{s\text{th}}$. Hence, $\mathcal{S}_\text{top}$ is formed with $O(D)$ operations. $\text{Time}_\text{S} = O(D)$.
- *Sorting*: Accumulators in $\mathcal{S}_\text{top}$ are sorted in decreasing order of scores. $\text{Time}_\text{R} = O(s\lg s)$.

$$\text{Time}_\text{T} = O(e + u + 1 + D + s\lg s) = O(D + s\lg s).\ S = O(D).$$

3.1.2. Implementations with dynamic accumulator allocation (TO-d)

If $e \ll D$, array $\mathscr{A}$ contains too many unused accumulators and hence wastes lots of space. In such a case or the case where array $\mathscr{A}$ is too large to fit into the volatile memory, it may be a good idea to use a dynamic data structure $\mathscr{D}$ and allow on-demand space allocation for accumulators. In this approach, accumulators are stored in nodes of $\mathscr{D}$ and are located using their document ids as keys. In this section, AVL tree (Knuth, 1998), hashing (Horowitz & Sahni, 1978), and skip list (Pugh, 1990) alternatives are investigated for this purpose. In what follows, we discuss these three alternatives, starting with the AVL tree. Our time analyses for the hashing and skip list alternatives are expected-time analyses. The algorithm for TO-d implementations is given in Fig. 3.

3.1.2.1. TO-d1: AVL tree of accumulators, min-priority queue for top s accumulators. In this implementation, an AVL tree $\mathscr{T}$ containing at most e nodes is used to store the accumulators. Each node of $\mathscr{T}$ keeps an accumulator, pointers to its left and right children, and a balance factor. An AVL tree implementation is preferred over a binary search tree implementation since the postings are stored in each inverted list in increasing order of document ids. In the case of a binary search tree implementation, with such a posting storage scheme, new

```
TO-D($\mathcal{Q}$, $\mathcal{D}$)
    for each query term $t_{q_i} \in \mathcal{Q}$ do
        for each posting $p \in \mathcal{I}_{q_i}$ do
            if $\exists$ an accumulator $a \in \mathcal{D}$ with $a.d = p.d$ then
                UPDATE $a.s$ as $a.s + p.w$
            else
                ALLOCATE a new accumulator $a$
                INITIALIZE $a$ as $a.d = p.d$ and $a.s = p.w$
                $\mathcal{D} = \mathcal{D} \cup \{a\}$
    $\mathcal{S}_{\text{top}} = \emptyset$
    for each $a \in \mathcal{D}$ do
        SELECT($\mathcal{S}_{\text{top}}$, $a$)
    SORT the accumulators in $\mathcal{S}_{\text{top}}$ in decreasing order of their scores
    RETURN $\mathcal{S}_{\text{top}}$

SELECT($\mathcal{S}$, $a$)
    if $|\mathcal{S}| < s$ then
        $\mathcal{S} = \mathcal{S} \cup \{a\}$
    else
        LOCATE $a_{\text{smin}}$, the accumulator with the minimum score in $\mathcal{S}$
        if $a.s > a_{\text{smin}}.s$ then
            $\mathcal{S} = (\mathcal{S} - \{a_{\text{smin}}\}) \cup \{a\}$
```

Fig. 3. The algorithm for TO-d implementations.

accumulator insertions may quickly turn the tree into a linked list. Hence, we prefer the AVL tree data structure, which dynamically balances the height of the tree, making accumulator search less costly.

- *Creation*: If an accumulator needs to be updated in $\mathcal{T}$ and it is not already there, a tree node is dynamically allocated to store the accumulator. The cost of node allocation is constant, i.e., $O(1)$. Hence, $\text{Time}_\text{C} = O(e)$.
- *Update*: For each posting p, nodes of $\mathcal{T}$ are searched to locate the accumulator to be updated, where $a.d = p.d$. If the accumulator is found, its score field $a.s$ is updated as $a.s + p.w$. Otherwise, a new node is allocated and inserted into $\mathcal{T}$, initializing the accumulator in the node as $a.d = p.d$ and $a.s = p.w$. The update cost for an accumulator is proportional with the height of the AVL tree. Hence, $\text{Time}_\text{U} = O(u \lg e)$.
- *Extraction*: When all updates are completed, accumulators can be extracted from nodes of $\mathcal{T}$ in any order. Each extracted accumulator is passed to the selection phase. Traversing the AVL tree has a cost of $\text{Time}_\text{E} = O(e)$.
- *Selection*: The min-priority queue mechanism of TO-s3 is used. $\text{Time}_\text{S} = O(s + (e - s)\lg s)$.
- *Sorting*: Similar to TO-s3. $\text{Time}_\text{R} = O(s \lg s)$.

$\text{Time}_\text{T} = O(e + u \lg e + e + (s + (e - s)\lg s) + s \lg s) = O(u \lg e)$. The storage overheads are $O(e)$ for the AVL tree and $O(s)$ for the min-priority queue. $S = O(e)$.

3.1.2.2. TO-d2: hashing of accumulators, min-priority queue for top s accumulators. Another implementation alternative which offers dynamic allocation is hashing. Since e is not known until all postings are completely processed, hashing techniques that require static allocation (such as open addressing) cannot be used. Here, we use hashing with chaining (Horowitz & Sahni, 1978). In this implementation, accumulators are placed into B buckets, where each bucket keeps a linked list of accumulators. The bucket b for an accumulator a is determined by applying a hash function on the document id field (e.g., $b = a.d \bmod B$).

- *Creation*: Selecting the appropriate number B of buckets is the most important step in this implementation. Allocating too many buckets may increase space consumption. On the contrary, if too few buckets are allocated, the number of accumulators per bucket increases. Since accumulators are sequentially searched in each bucket, this increases the query processing time. In this implementation, B pointers are needed to keep the list heads. Each list node stores an accumulator and has a pointer to the next node in the linked list. It is necessary to dynamically allocate a total of e list nodes. Hence, $\text{Time}_\text{C} = O(B + e)$.
- *Update*: For a posting p, the bucket to be searched is determined by hashing $p.d$ to a bucket. The accumulators in a bucket are searched by following the links between list nodes. If an accumulator with $a.d = p.d$ is found, its score is updated. If the end of the list is reached or an accumulator with a greater document id is found, the search ends. In this case, a new node which contains an accumulator is allocated, initialized using p, and then inserted into the list. List nodes are maintained in increasing order of document ids. Each bucket stores e/B list nodes on the average. Hence, these many comparisons are necessary to locate an accumulator. $\text{Time}_\text{U} = O(ue/B)$.
- *Extraction*: Accumulators are extracted from the buckets and passed to the selection phase. Since exactly e nodes must be extracted, $\text{Time}_\text{E} = O(e)$.
- *Selection*, *Sorting*: Similar to TO-s3. $\text{Time}_\text{S} = O(s + (e - s)\lg s)$, $\text{Time}_\text{R} = O(s\lg s)$.

$\text{Time}_\text{T} = O((B + e) + ue/B + e + (s + (e - s)\lg s) + s\lg s) = O(ue/B + e\lg s)$. The storage overheads are $O(B + e)$ for the hash table and $O(s)$ for the min-priority queue. $S = O(B + e)$.

3.1.2.3. TO-d3: skip list of accumulators, min-priority queue for top s accumulators. Yet another alternative is to use a skip list $\mathcal{S}$ to store and search the accumulators. Skip lists balance themselves probabilistically rather than explicitly (e.g., rotations in AVL trees). Although they have bad worst-case time complexities, they have good expected-time complexities for insert and find operations and perform well in practice.

- *Creation*: A list node is dynamically allocated in $\mathcal{S}$ to store an accumulator and a set of forward pointers to the following list nodes. The number of forward pointers in each node is determined randomly, but it is limited from above. Since e list nodes must be allocated, $\text{Time}_\text{C} = O(e)$.
- *Update*: For each posting p, the nodes in $\mathcal{S}$ are searched to locate the accumulator to be updated, where $a.d = p.d$. For this purpose, forward pointers are used and the skip list is traversed in a manner similar to binary search. If the accumulator is located in $\mathcal{S}$, its score field is updated as $a.s = a.s + p.w$. Otherwise, a new node is allocated and inserted into $\mathcal{S}$ after initializing its accumulator as $a.d = p.d$ and $a.s = p.w$. The expected update cost for an accumulator is $O(\lg e)$. Hence, $\text{Time}_\text{U} = O(u\lg e)$.
- *Extraction*: Nodes of $\mathcal{S}$ are visited sequentially, and accumulators are passed to the selection phase. $\text{Time}_\text{E} = O(e)$.
- *Selection*, *Sorting*: Similar to TO-s3. $\text{Time}_\text{S} = O(s + (e - s)\lg s)$, $\text{Time}_\text{R} = O(s\lg s)$.

$\text{Time}_\text{T} = O(e + u\lg e + e + (s + (e - s)\lg s) + s\lg s) = O(u\lg e)$. The storage overheads are $O(e)$ for the skip list and $O(s)$ for the min-priority queue. $S = O(e)$.

3.2. Implementations for document-ordered (DO) processing

Two important features in the inverted index structure let us devise another query processing strategy. First, the postings of a term are stored in increasing order of document ids. That is, while traversing an inverted list, once a document id is seen in a posting, there cannot be a smaller document id in one of

```
DO-M($\mathcal{Q}$, $\mathcal{M}$)
    $\ell_1 = 0$, $\ell_2 = 1$, $\mathcal{M} = \emptyset$, and $\mathcal{S}_{\text{top}} = \emptyset$
    for each query term $t_{q_i} \in \mathcal{Q}$ do
        $h[i] = 1$, i.e., the current head of $\mathcal{I}_{q_i}$ is its first posting $\mathcal{I}_{q_i}^1$
    while $|\mathcal{L}_\mathcal{Q}| > 0$ do
        while $|\mathcal{L}_\mathcal{Q}| > 0$ and $|\mathcal{M}| < |\mathcal{L}_\mathcal{Q}|$ do
            if $\ell_1 = 0$ then
                $\ell = \ell_2$
            else
                $\ell = \ell_1$
            $p = \mathcal{I}_{q_\ell}^{h[\ell]}$
            if $\exists$ an accumulator $a \in \mathcal{M}$ with $a.d = p.d$ then
                UPDATE $a.s$ as $a.s + p.w$
                $h[\ell] = h[\ell] + 1$
                if $h[\ell] > |\mathcal{I}_{q_\ell}|$ then
                    $\mathcal{L}_\mathcal{Q} = \mathcal{L}_\mathcal{Q} - \{\mathcal{I}_{q_\ell}\}$
                    if $\ell_1 = 0$ then
                        $\ell_2 = \ell_2 + 1$
            else
                ALLOCATE an accumulator $a$
                INITIALIZE $a$ as $a.d = p.d$, $a.s = p.w$, and $a.\ell = \ell$
                $\mathcal{M} = \mathcal{M} \cup \{a\}$
                $h[\ell] = h[\ell] + 1$
                if $\ell_1 = 0$ then
                    $\ell_2 = \ell_2 + 1$
        while $|\mathcal{L}_\mathcal{Q}| > 0$ and $|\mathcal{M}| = |\mathcal{L}_\mathcal{Q}|$ do
            LOCATE $a_{\text{dmin}}$, the accumulator with the minimum document id in $\mathcal{M}$
            $\mathcal{M} = \mathcal{M} - \{a_{\text{dmin}}\}$
            $\ell_1 = a_{\text{dmin}}.\ell$
            SELECT($\mathcal{S}_{\text{top}}$, $a_{\text{dmin}}$)
            if $h[\ell] > |\mathcal{I}_{q_\ell}|$ then
                $\mathcal{L}_\mathcal{Q} = \mathcal{L}_\mathcal{Q} - \{\mathcal{I}_{q_\ell}\}$
    SORT the accumulators in $\mathcal{S}_{\text{top}}$ in decreasing order of their scores
    RETURN $\mathcal{S}_{\text{top}}$
```

Fig. 4. The algorithm for DO-m implementations.

the succeeding postings in that list. Second, the number of query terms is limited. We have Q terms to be processed. These observations allow us to process the inverted lists in parallel instead of processing them consecutively. This way, it is possible to compute a complete score for a document before all postings in the lists are completely processed. In DO processing, update, extraction, and selection phases are performed in an interleaved manner. The implementations differ in their choice for the number of accumulators allocated, the data structures employed to store the accumulators, and the processing order of the list heads.

3.2.1. Implementations with multiple accumulator allocation (DO-m)

Implementations in the DO-m category use a structure $\mathcal{M}$, which contains at most Q accumulators at any time. Also, an array h of Q elements is used to locate the first unprocessed posting in each inverted list, i.e., each element $h[i]$ points at the posting $\mathcal{I}_{q_i}^{h[i]} \in \mathcal{I}_{q_i}$ that will be processed next in list $\mathcal{I}_{q_i}$. Each accumulator $a \in \mathcal{M}$ is associated with a single inverted list. Accumulators contain a list id field, which is initialized as $a.\ell = i$ if accumulator a is associated with inverted list $\mathcal{I}_{q_i}$. Although any posting with a document id of $a.d$ from any inverted list may update the score field $a.s$, only the postings from list $\mathcal{I}_{q_{a.\ell}}$ may initialize $a.d$. The document id $a.d$ of each accumulator a is equal to a document id in one of the postings in $\mathcal{I}_{q_{a.\ell}}$. No two accumulators in $\mathcal{M}$ can have the same document id and list id. The structure $\mathcal{M}$ can be implemented by a sorted array or a dynamic data structure. These alternatives are described below. The algorithm for DO-m implementations is given in Fig. 4.

3.2.1.1. DO-m1: sorted array of accumulators, array of posting pointers, min-priority queue for top s accumulators. In this approach, Q accumulators are kept in an array sorted in decreasing order of document ids.

- *Creation*: An accumulator array $\mathscr{A}$ and an array h for marking current list heads, each of size Q, are allocated. The cost of allocating both arrays is $O(Q)$. After the allocation, each $h[i]$ is initialized to point at the first posting $\mathcal{I}^1_{q_i} \in \mathcal{I}_{q_i}$, i.e., $h[i] = 1$. In processing a query, there are e initializations over the accumulators in $\mathscr{A}$. Hence, $\text{Time}_\text{C} = O(e + Q)$.
- *Update*, *Extraction*: The following procedure is repeated until all postings are processed. If there are less than Q occupied accumulators in $\mathscr{A}$, updates are performed over the accumulators using the postings at the current list heads (pointed by h) which are not currently associated with an accumulator in $\mathscr{A}$. In processing of a posting $p = \mathcal{I}^{h[i]}_{q_i}$, array $\mathscr{A}$ is searched for an accumulator with $a.d = p.d$. If it is found, a is updated using p. Otherwise, a new accumulator is created in $\mathscr{A}$ and is initialized as $a.d = p.d$, $a.s = p.w$, and $a.\ell = i$. If all Q accumulators in $\mathscr{A}$ are occupied, i.e., associated with a list, the accumulator admin with the minimum document id is located, extracted, and passed to the selection phase. Then, $h[a_\text{dmin}.\ell]$ is incremented by 1, and hence it points to the posting $p = \mathcal{I}^{h[a_\text{dmin}.\ell]}_{q_{a_\text{dmin}.\ell}}$ to be processed next. Since the $\mathscr{A}$ array is maintained in decreasing order of document ids, an accumulator can be located in $O(\lg Q)$ time using binary search. Although update of an accumulator is an $O(1)$-time operation once it is located, insertion of a new accumulator after a failed search requires shifting $O(Q)$ accumulators in the array. Considering the fact that there are $u - e$ accumulator updates and e insertions, $\text{Time}_\text{U} = O(u\lg Q + eQ)$. Extraction is simple since the accumulator with the smallest document id is always the last element of the array. $\text{Time}_\text{E} = O(e)$.
- *Selection*, *Sorting*: Similar to TO-s3. $\text{Time}_\text{S} = O(s + (e - s)\lg s)$, $\text{Time}_\text{R} = O(s\lg s)$.

$\text{Time}_\text{T} = O((e + Q) + (u\lg Q + eQ) + e + (s + (e - s)\lg s) + s\lg s) = O(u\lg Q + eQ + e\lg s)$. The storage overheads are $O(Q)$ for the sorted array, $O(Q)$ for the array of posting pointers, and $O(s)$ for the min-priority queue. $S = O(Q + s)$.

3.2.1.2. DO-m2: AVL tree of accumulators, array of posting pointers, min-priority queue for top s accumulators. Instead of a sorted array, an AVL tree $\mathscr{T}$ can be used as a dynamic structure to store the accumulators.

- *Creation*: Array h is allocated and initialized similar to DO-m1. Nodes of AVL tree $\mathscr{T}$ are dynamically allocated. For each accumulator with a distinct document id, a tree node must be allocated although $\mathscr{T}$ contains no more than Q nodes at any time. Hence, $\text{Time}_\text{C} = O(e + Q)$.
- *Update*, *Extraction*: Update and extraction phases are similar to DO-m1. However, in processing a posting, both update of an existing accumulator and insertion of a new one require $O(\lg Q)$ operations in the worst case. Hence, $\text{Time}_\text{U} = O(u\lg Q)$. The accumulator with the smallest document id is contained within the left-most leaf node in $\mathscr{T}$. This leaf node can be reached by following the left links iteratively starting from the root of $\mathscr{T}$ until a node with no children is reached. With this approach, extraction is an $O(\lg Q)$-time operation. However, it is possible to improve this by an implementation trick. If each node keeps a link to its parent node, and the node with the smallest document id in $\mathscr{T}$ is remembered by a pointer, it turns out that extraction is an $O(1)$-time operation. Hence, $\text{Time}_\text{E} = O(e)$.
- *Selection*, *Sorting*: Similar to TO-s3. $\text{Time}_\text{S} = O(s + (e - s)\lg s)$, $\text{Time}_\text{R} = O(s\lg s)$.

$\text{Time}_\text{T} = O((e + Q) + u\lg Q + e + (s + (e - s)\lg s) + s\lg s) = O(u\lg Q + e\lg s)$. The storage overheads are $O(Q)$ for the AVL tree, $O(Q)$ for the array of posting pointers, and $O(s)$ for the min-priority queue. $S = O(Q + s)$.

```
DO-s($\mathcal{Q}$, $a_{\mathrm{dmin}}$)
    $\mathcal{S}_{\mathrm{top}} = \emptyset$
    for each query term $t_{q_i} \in \mathcal{Q}$ do
        $h[i] = 1$, i.e., the current head of $\mathcal{I}_{q_i}$ is its first posting $\mathcal{I}_{q_i}^1$
    while $|\mathcal{L}_{\mathcal{Q}}| > 0$ do
        LOCATE $p_{\mathrm{dmin}}$, the posting with the minimum document id
                among all $\mathcal{I}_{q_i}^{h[i]} \in \mathcal{I}_{q_i}$, where $\mathcal{I}_{q_i} \in \mathcal{L}_{\mathcal{Q}}$
        INITIALIZE $a_{\mathrm{dmin}}$ as $a_{\mathrm{dmin}}.d = p_{\mathrm{dmin}}.d$ and $a_{\mathrm{dmin}}.s = 0$
        for each posting $p = \mathcal{I}_{q_i}^{h[i]}$ where $\mathcal{I}_{q_i} \in \mathcal{L}_{\mathcal{Q}}$ do
            if $p.d = p_{\mathrm{dmin}}.d$ then
                UPDATE $a_{\mathrm{dmin}}.s$ as $a_{\mathrm{dmin}}.s + p.w$
                $h[i] = h[i] + 1$
                if $h[i] > |\mathcal{I}_{q_i}|$ then
                    $\mathcal{L}_{\mathcal{Q}} = \mathcal{L}_{\mathcal{Q}} - \{\mathcal{I}_{q_i}\}$
        SELECT($\mathcal{S}_{\mathrm{top}}$, $a_{\mathrm{dmin}}$)
    SORT the accumulators in $\mathcal{S}_{\mathrm{top}}$ in decreasing order of their scores
    RETURN $\mathcal{S}_{\mathrm{top}}$
```

Fig. 5. The algorithm for DO-s implementations.

3.2.2. Implementations with single accumulator allocation (DO-s)

Implementations in the DO-s category require the use of only a single accumulator a_{dmin} at any time. All updates are performed on this single accumulator. Here, we describe two different implementations that belong to this category. The algorithm for DO-s implementations is given in Fig. 5.

3.2.2.1. DO-s1: single accumulator, array of posting pointers, min-priority queue for top s accumulators. In this very simple approach, two passes are made over the list heads. In the first pass, the smallest document id among the currently unprocessed postings is determined. In the second pass, the postings with this smallest document id are picked and used to update a_{dmin}.

- *Creation*: The single accumulator a_{dmin}, which stores the information about the currently minimum document id, is allocated. The h array is allocated and initialized as in DO-m1. The cost of reinitializing a_{dmin} is $O(e)$. Hence, $\mathrm{Time_C} = O(e + Q)$.
- *Update*, *Extraction*: A pass is made over the postings pointed by the h array. Within these postings, a posting p_{dmin} with the minimum document id $p_{\mathrm{dmin}}.d$ is found. Accumulator a_{dmin} is initialized as $a_{\mathrm{dmin}}.d = p_{\mathrm{dmin}}.d$ and $a_{\mathrm{dmin}}.s = 0$. With a second pass over these postings, the postings that have this minimum document id are found. The score field $a_{\mathrm{dmin}}.s$ of accumulator a_{dmin} is updated using the weights in each such posting. $h[i]$ for each inverted list $\mathcal{I}_{q_i}$ that contains such a posting is incremented to point at the next posting in the list. Once all updates over a_{dmin} is completed, a_{dmin} is passed to the selection phase. This procedure is repeated until all postings are consumed. Since two passes are made over h for each distinct document id, $\mathrm{Time_U} = O(eQ)$. Extracting a_{dmin} is an $O(1)$-time operation. Hence, $\mathrm{Time_E} = O(e)$.
- *Selection*, *Sorting*: Similar to TO-s3. $\mathrm{Time_S} = O(s + (e - s)\lg s)$, $\mathrm{Time_R} = O(s \lg s)$.

$\mathrm{Time_T} = O((e + Q) + eQ + e + (s + (e - s)\lg s) + s\lg s) = O(eQ + e\lg s)$. The storage costs are $O(1)$ for the accumulator, $O(Q)$ for the array of posting pointers, and $O(s)$ for the min-priority queue. $S = O(Q + s)$.

3.2.2.2. DO-s2: single accumulator, min-priority queue for posting pointers, min-priority queue for top s accumulators. In this implementation, instead of the h array in the DO-s1 implementation, a min-priority queue is used so that there is no need for the first pass, which searches for the minimum document id. Here, we describe an improved version of the implementation described by Kaszkiel, Zobel, and Sacks-Davis (1999).

- *Creation*: Similar to DO-s1. However, h is a min-priority queue implemented as a min-heap of postings pointers, keyed by the document ids in the postings they point at. $\text{Time}_\text{C} = O(e + Q)$.
- *Update, Extraction*: The min-priority queue h is built using the postings at the list heads. The following procedure is repeated until all postings are processed. The root of h stores posting p_dmin, i.e., the posting with the minimum document id among the current list heads. a_dmin is initialized as $a_\text{dmin}.d = p_\text{dmin}.d$ and $a_\text{dmin}.s = 0$. h is traversed in reverse order (starting from the Qth element down to the first element), and the postings with $p.d = p_\text{dmin}.d$ are located. Each such posting p is used to update a_dmin as $a_\text{dmin}.d = p.d$ and $a_\text{dmin}.s = a_\text{dmin}.s + p.s$. Then, posting p is replaced by the next posting in the inverted list that p belongs to, and h is heapified at the node containing p. This approach avoids building the heap (Kaszkiel et al., 1999) at each pass. After the posting p_dmin at the root performs its update, a_dmin is extracted and passed to the selection phase. In this approach, the heap is heapified exactly once for each posting, and hence $\text{Time}_\text{U} = O(u \lg Q)$. Extraction has a cost of $\text{Time}_\text{E} = O(e)$.
- *Selection*, *Sorting*: Similar to TO-s3. $\text{Time}_\text{S} = O(s + (e - s)\lg s)$, $\text{Time}_\text{R} = O(s \lg s)$.

$\text{Time}_\text{T} = O((e + Q) + u \lg Q + e + (s + (e - s)\lg s) + s \lg s) = O(u \lg Q + e \lg s)$. The storage overheads are $O(1)$ for the accumulator, $O(Q)$ for the min-priority queue of posting pointers, and $O(s)$ for the min-priority queue of top s accumulators. $S = O(Q + s)$.

4. Experimental results

4.1. Experimental platform

In the experiments, a Pentium IV 2.54 GHz PC, which has 2 GB of main memory, 512 KB of L2 cache, and 8 KB of L1 cache, is used. As the operating system, Mandrake Linux, version 13 is installed. All algorithms are implemented in C and are compiled in gcc with O2 optimization option. Due to the randomized nature of some of the implementations, experiments are repeated 10 times, and the average values are reported. All experiments are conducted after booting the system into the single user mode.

As the document collection, results of a large crawl performed over the '.edu' domain, i.e., the educational US Web sites, is used. The entire collection is around 30 GB and contains 1,883,037 Web pages (documents). After cleansing and stop-word elimination, there remains 3,325,075 distinct index terms. The size of the inverted index constructed using this collection is around 2.7 GB.

In query processing, four different query sets ($\mathcal{Q}_\text{short}$, $\mathcal{Q}_\text{medium}$, $\mathcal{Q}_\text{long}$, and $\mathcal{Q}_\text{huge}$) are tried. Each query set contains 100 queries, expect for $\mathcal{Q}_\text{huge}$, which contains a single query. The query terms are selected from the sentences within the documents of the collection. Queries in $\mathcal{Q}_\text{short}$, which simulate Web queries, are made up of between 1 and 5 query terms. Queries in $\mathcal{Q}_\text{medium}$ contain between 6 and 25 query terms. This type of queries is observed in relevance feedback. Queries in $\mathcal{Q}_\text{large}$ contain between 26 and 250 query terms and simulate queries observed in text classification. $\mathcal{Q}_\text{huge}$ is included for experimental purposes and the results, although mentioned in the text, are partially reported. Properties of the query sets are given in Table 2. This table also presents the minimum, maximum, and average e and u values observed during the experiments.

For each query set, three answer sets ($\mathcal{S}_\text{small}$, $\mathcal{S}_\text{large}$, and $\mathcal{S}_\text{full}$), each with a different top document count s, are tried. $\mathcal{S}_\text{small}$ and $\mathcal{S}_\text{large}$ expect the query processing system to return the first 10 and 1000 best-matching documents, respectively. $\mathcal{S}_\text{full}$ expects all documents with a nonzero score to be returned to the user. Prop-

Table 2
The minimum, maximum, and average values of the number of query terms (Q), number of extracted accumulators (e), and number of updated accumulators (u) for different query sets

	$\mathcal{Q}_{\text{short}}$	$\mathcal{Q}_{\text{medium}}$	$\mathcal{Q}_{\text{long}}$	$\mathcal{Q}_{\text{huge}}$
$\vert\mathcal{Q}\vert$	100	100	100	1
Q_{min}	1	6	26	2500
Q_{max}	5	25	250	2500
Q_{avr}	3.0	14.6	142.1	2500
e_{min}	4	331,524	1,218,640	1,866,703
e_{max}	1,363,584	1,637,894	1,839,661	1,866,703
e_{avr}	375,166	1,109,691	1,723,229	1,866,703
u_{min}	4	367,068	2,625,452	111,028,126
u_{max}	1,964,216	6,861,180	38,760,201	111,028,126
u_{avr}	451,931	2,310,010	16,468,300	111,028,126

Table 3
The minimum, maximum, and average values of the number of top documents (s) for answer sets produced after processing query set $\mathcal{Q}_{\text{short}}$

	$\mathcal{S}_{\text{small}}$	$\mathcal{S}_{\text{large}}$	$\mathcal{S}_{\text{full}}$
$\vert\mathcal{S}\vert$	10	1000	e
s_{min}	4	4	4
s_{max}	10	1000	1,363,584
s_{avr}	9.94	994	375,166

erties of these answer sets, and the minimum, maximum, and average number of top documents actually returned as answer to queries in $\mathcal{Q}_{\text{short}}$ are displayed in Table 3.

4.2. Experiments on execution time

Fig. 6 presents the running times of implementations for different types of query and answer sets. Among the static-accumulator implementations in the TO-s category, for $\mathcal{S}_{\text{small}}$ and $\mathcal{S}_{\text{large}}$, the min-priority queue implementation TO-s3 performs the best if queries contain a few terms, i.e., when $\mathcal{Q}_{\text{short}}$ is used. For the same answer sets, the linear-time selection scheme TO-s4 performs slightly better than TO-s3 if $\mathcal{Q}_{\text{medium}}$ or $\mathcal{Q}_{\text{long}}$ is used. For the answer set $\mathcal{S}_{\text{full}}$, the best results are achieved by the max-priority queue implementation TO-s2. The TO-s1 implementation, which requires sorting the nonzero accumulators, is outperformed in all experiments, but the gap between TO-s1 and the others closes as the queries get longer. For $\mathcal{Q}_{\text{huge}}$ and $\mathcal{S}_{\text{full}}$ combination, TO-s1 is almost as good as TO-s2 and TO-s3.

Among the dynamic-accumulator implementations in the TO-d category, for $\mathcal{Q}_{\text{short}}$ and $\mathcal{Q}_{\text{medium}}$, the hashing implementation TO-d2 performs the best. For this implementation, we used an adaptive bucket size $B = u/Q$ due to the time-space trade-off mentioned in Section 3.1.2.2. For query sets $\mathcal{Q}_{\text{long}}$, the best results are achieved by TO-d2 and the AVL tree implementation TO-d1, which perform almost equally well. Increasing the number of terms in queries seems to favor TO-d1, which is the fastest implementation for $\mathcal{Q}_{\text{huge}}$.

In the DO-m category, although the run-time complexity for the AVL tree implementation DO-m2 is better than that of the sorted array implementation DO-m1, in practice, DO-m1 is faster than DO-m2 for $\mathcal{Q}_{\text{short}}$ and $\mathcal{Q}_{\text{medium}}$. This shows that the cost of rotations in the AVL tree implementation is higher than

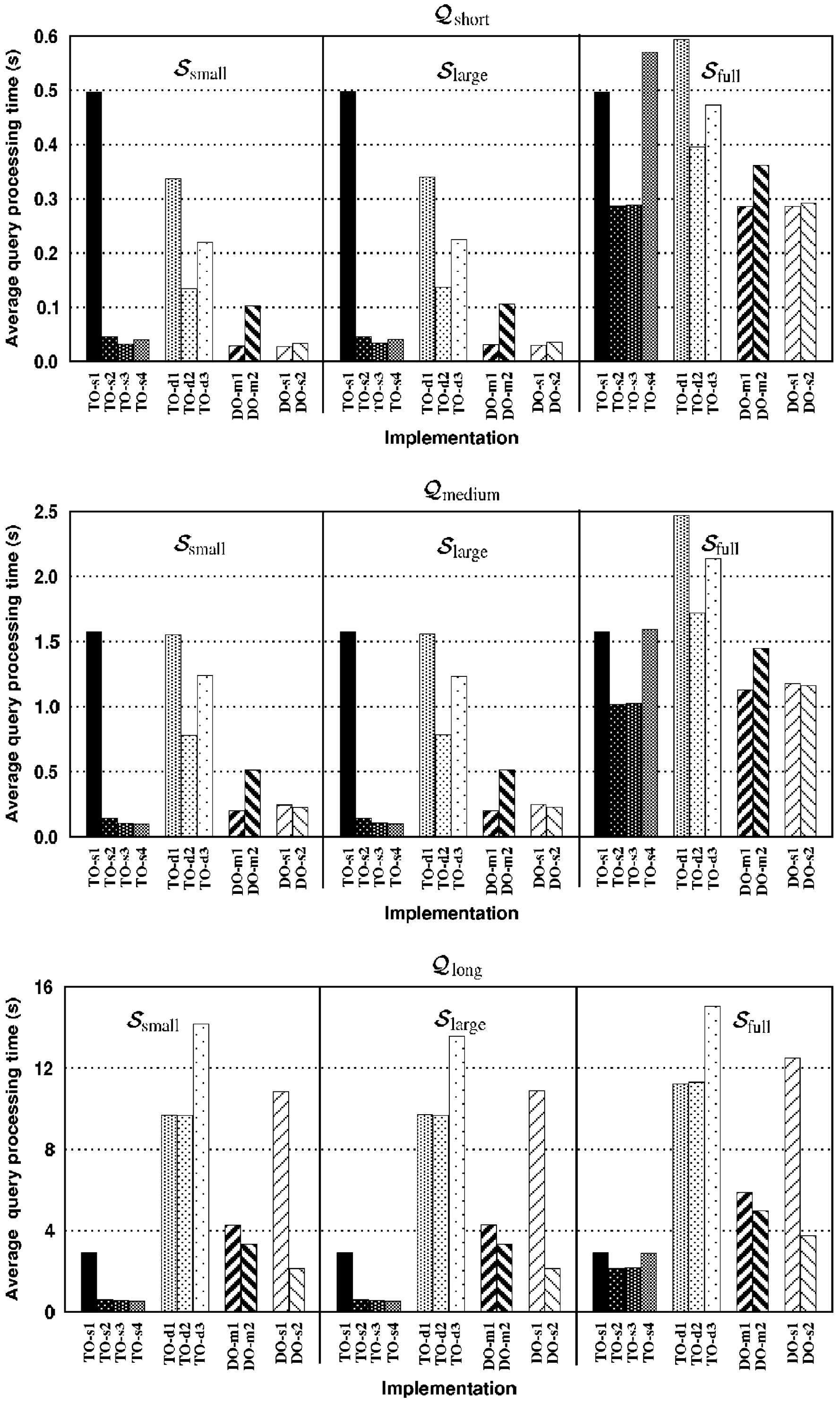

Fig. 6. Query processing times of the implementations for different query and answer set sizes.

the cost of accumulator shifts in the sorted array implementation. However, if queries get longer, DO-m2 starts to perform better than DO-m1. Interestingly, for $\mathcal{Q}_{\text{huge}}$, DO-m2 runs 11 times faster than DO-m1 on the average.

In the DO-s category, for short queries, the two-pass DO-s1 implementation is faster than the one-pass DO-s2 implementation. As the number of query terms increase, DO-s2 starts to perform better. This can be explained by the fact that visiting the list heads in the first pass of DO-s1 brings an additional overhead, which dominates when queries are long. It is observed that, for $\mathcal{Q}_{\text{huge}}$, DO-s2 runs 35 times faster than DO-s1.

Among all implementations, if all documents with a nonzero score are returned, TO-s2 performs the best with TO-s3 displaying close performance. Otherwise, if answers are partially returned, performance depends on the number of query terms. For example, if queries are short DO-s1 is the best choice, whereas TO-s4 is the fastest implementation for medium and long query sizes.

It should also be noted that, for aggregate querying scenarios, the winners may change. For example, in the case the user is interested in the top 10 documents and 40% or more of the queries come from $\mathcal{Q}_{\text{short}}$ while the remaining 60% or less are of type $\mathcal{Q}_{\text{medium}}$ requiring all top documents, then TO-s3 is preferable

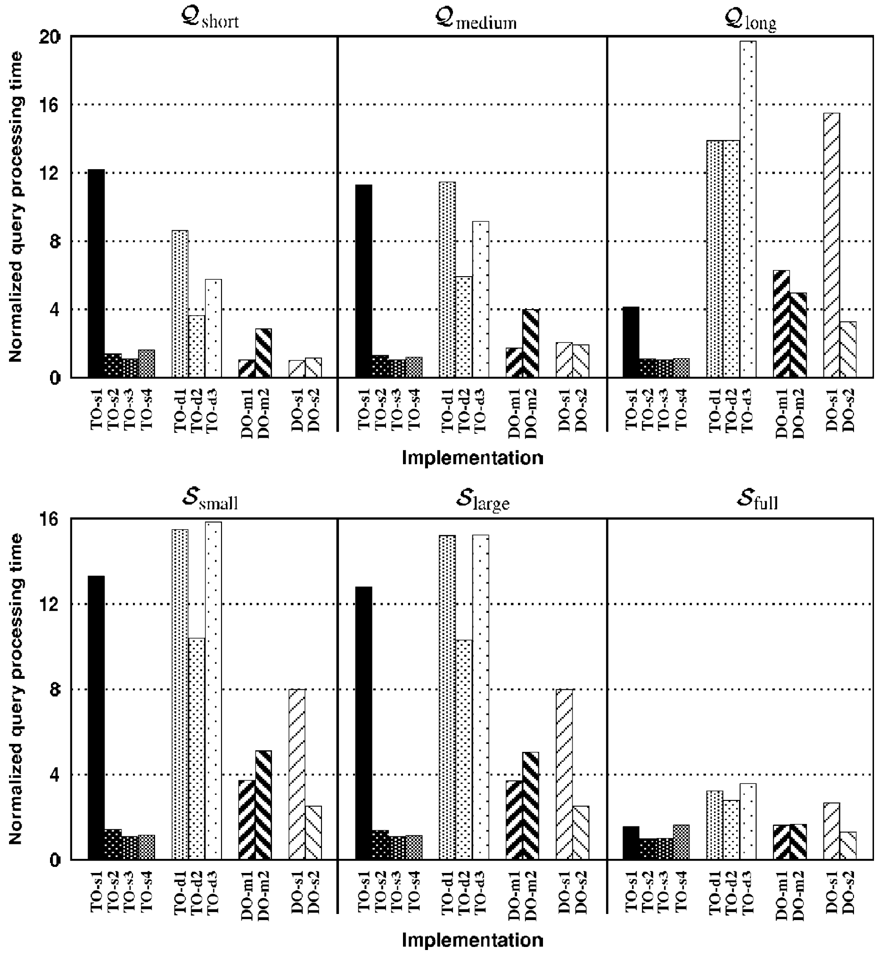

Fig. 7. Normalized query processing times of the implementations for different query and answer set sizes.

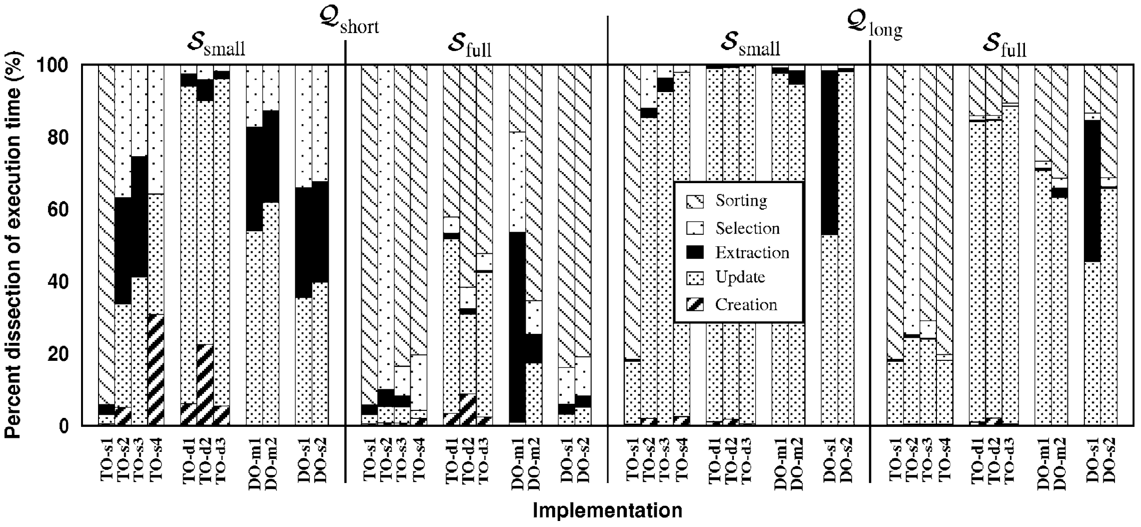

Fig. 8. Percent dissection of execution times of query processing implementations according to the five different phases.

to both DO-s1 and TO-s2 in that it provides the best average query processing time. Taking this fact into consideration, we also present normalized running times in Fig. 7. In order to generate this figure, the execution times are first normalized with the smallest execution time. Then, the normalized time values are averaged and displayed across each query and answer set category.

According to Fig. 7, DO-s1 and DO-m1 perform better than the rest for query set $\mathcal{Q}_{\text{short}}$. For $\mathcal{Q}_{\text{medium}}$ and $\mathcal{Q}_{\text{long}}$, TO-s3 is better than the others. For $\mathcal{S}_{\text{small}}$ and $\mathcal{S}_{\text{large}}$, TO-s3 is again the best. For $\mathcal{S}_{\text{full}}$, TO-s2 very slightly outperforms TO-s3. On the overall, the local winners of the four categories are TO-s3, TO-d2, DO-m1, and DO-s2, where TO-s3 is also the global winner.

Fig. 8 displays the percent dissection of execution times for different query processing phases, i.e., creation, update, extraction, selection, and sorting. According to this figure, for TO-s1, the bottleneck is at the sorting phase. However, for most implementations, the sorting overhead is relatively less important, except for the case of short queries with all results retrieved. Overhead of the selection phase is more apparent for short queries. Especially, in the small answer set case, a considerable percentage of execution times for TO-s2, TO-s3, TO-s4, DO-s1, and DO-s2 implementations is occupied by the overhead of this phase. The extraction phase seems to be relatively important for DO-m1 and DO-s1 implementations. The respective reasons of this high overhead for DO-m1 and DO-s1 are the high amount of accumulator shift operations and inverted list head traversals. In general, except for the case of short queries with all answers returned, the update phase incurs the highest overhead. This overhead is especially high for TO-d implementations. The creation overhead is usually negligible.

4.3. Experiments on scalability

In this section, we provide some experimental results that evaluate scalability of the implementations with increasing number of query terms, increasing number of extracted postings, increasing answer set sizes, and increasing number of documents. In the plots, instead of displaying the actual data curves which contain many data points, we give curves fitted by regression and limit the number of data points to 11 in order to simplify drawings and ease understanding. For the same purpose, we provide a single representative curve in cases where more than one curves have a very similar behavior and hence overlap.

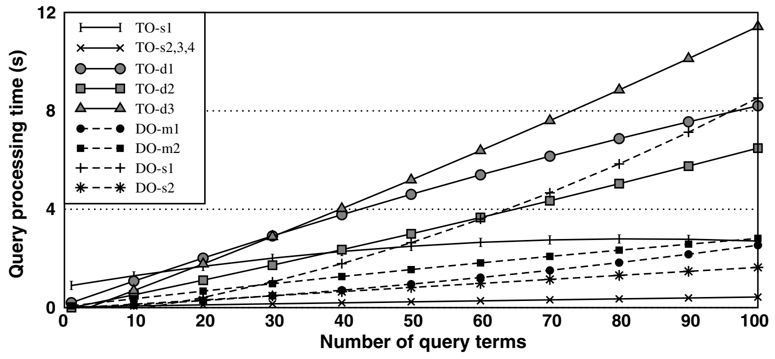

Fig. 9. Query processing times for varying number of query terms (Q).

4.3.1. *Effect of number of query terms (Q)*

Fig. 9 shows the query processing performance for varying number of query terms. This plot is obtained by submitting 100 queries, where ith query contains i query terms, and retrieving highly ranked 10 documents at each query. As expected, DO-s1 is the implementation most affected from increasing query sizes. Other DO implementations as well as TO-d implementations are also affected since increasing number of query terms results in more posting updates, i.e., increases the overhead of the update phase. The impact on TO-s implementations is relatively limited since update operations are not costly and extraction and selection operations have a considerable overhead for this type of implementations.

4.3.2. *Effect of number of extracted accumulators (e)*

In order to investigate the effect of the number of extracted postings on the query processing performance, we used a query set consisting of 100 queries, where each query has a single term. The queries are such that the ith query incurs $1000 \times i$ extraction operations. As a result, the top 10 documents are retrieved. Fig. 10 shows the performance variation for increasing number of extracted accumulators. Except for TO-s1, the TO-s implementations are not affected much by the increasing number of extractions since they anyway traverse the whole accumulator array and check every score field. The different behavior of TO-s1 is basically due to the overhead of sorting. Among the TO-d implementations, TO-d2 seems to scale best with increasing e. DO implementations perform quite well since there is only a single term in the queries.

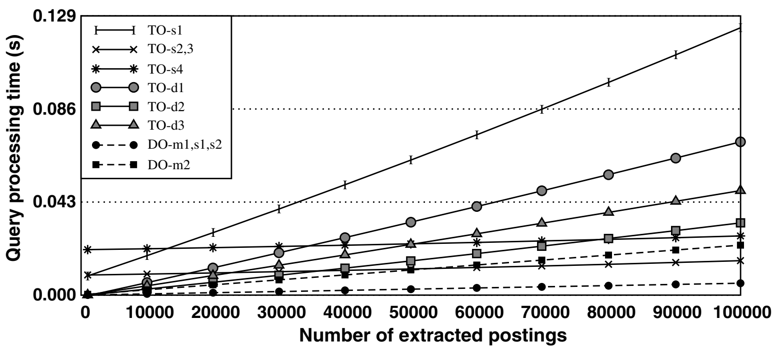

Fig. 10. Query processing times for varying number of extracted accumulators (e).

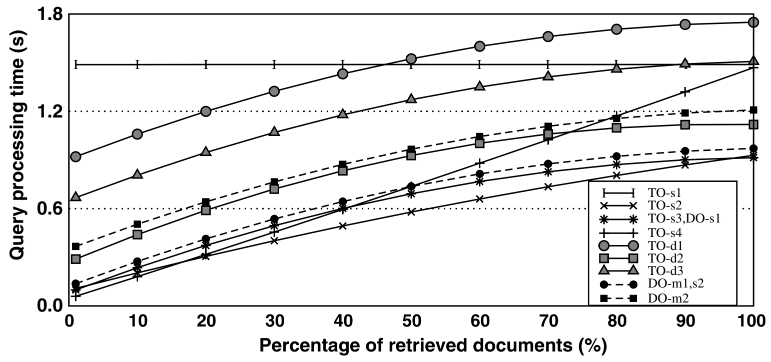

Fig. 11. Query processing times for varying number of retrieved documents (s).

4.3.3. Effect of number of retrieved documents (s)

Fig. 11 shows how the performance is affected by increasing size of answer sets. To obtain this plot, we used a single query containing a very frequent term ('university') so that the number of documents returned is high in case all documents with a nonzero score are requested. We had 100 experiments, where, for the ith experiment, the size of the answer set equals i% of the documents with a nonzero score, i.e., $s_i = i \times e/100$. According to Fig. 11, as expected, the number of returned documents has no effect on TO-s1 since all nonzero documents are anyway sorted. For TO-s2, the curve is almost linear since the complexity of the selection phase is $s \lg e$ and e is fixed. The linear behavior of TO-s4 is also due to the linear-time selection heuristic employed. All other implementations have a similar behavior which complies with their $O(e \lg s)$ complexity. The performance gap between the curves is due to the overheads of other phases. An interesting observation obtained from Fig. 11 is that a trade-off can be made between TO-s2, TO-s3, and TO-s4 implementations depending on the percentage of retrieved documents.

4.3.4. Effect of dataset size (D)

In this section, we investigate the scalability of the implementations with respect to the document collection size. In the experiments, we use document collections of three different sizes ($\mathcal{D}_{\text{small}}$, $\mathcal{D}_{\text{medium}}$, and $\mathcal{D}_{\text{large}}$). $\mathcal{D}_{\text{small}}$ and $\mathcal{D}_{\text{medium}}$ are subsets of the original collection $\mathcal{D}_{\text{large}}$, which was used in the rest of the experiments. Table 4 gives the number of documents and number of distinct terms in these collections. In all experiments, we use the medium-length query set $\mathcal{Q}_{\text{medium}}$ with $\mathcal{S}_{\text{small}}$ and $\mathcal{S}_{\text{full}}$ as the answer sets.

Fig. 12 shows the average query processing times for collections of different sizes. To better illustrate the scalability of the implementations with increasing dataset size, we also provide Table 5. This table provides the speedups, which is calculated as $\text{QPT}(\mathcal{D})/\text{QPT}(\mathcal{D}')$, where QPT is the average query processing time, for two document collections $\mathcal{D}$ and $\mathcal{D}'$ such that $|\mathcal{D}| > |\mathcal{D}'|$. According to Table 5, for $\mathcal{Q}_{\text{medium}}$ and $\mathcal{S}_{\text{small}}$ combination, there is almost no scalability problem for most of the implementations as we increase the size of the document collection from small to medium, i.e., the query processing times double as the collection size doubles. However, scalability begins to become an issue when we further increase the size of the

Table 4
The number of documents (D) and distinct terms (T) in collections of varying size

	$\mathcal{D}_{\text{small}}$	$\mathcal{D}_{\text{medium}}$	$\mathcal{D}_{\text{large}}$
D	472,533	943,672	1,883,037
T	1,467,932	2,201,992	3,325,075

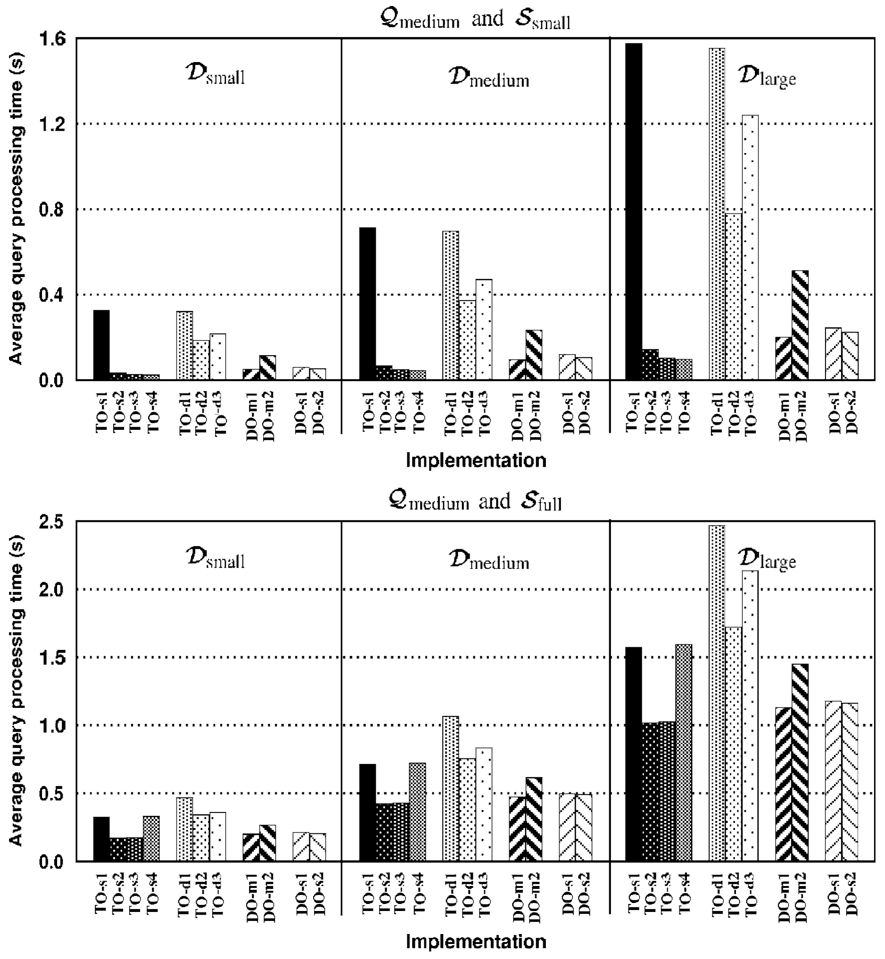

Fig. 12. Average query processing times for collections with varying number of documents (D).

Table 5
Scalability of implementations with different collection sizes

Imp.	$\mathcal{Q}_{\text{medium}}$ and $\mathcal{S}_{\text{small}}$		$\mathcal{Q}_{\text{medium}}$ and $\mathcal{S}_{\text{full}}$	
	$\frac{\text{QPT}(\mathcal{D}_{\text{medium}})}{\text{QPT}(\mathcal{D}_{\text{small}})}$	$\frac{\text{QPT}(\mathcal{D}_{\text{large}})}{\text{QPT}(\mathcal{D}_{\text{medium}})}$	$\frac{\text{QPT}(\mathcal{D}_{\text{medium}})}{\text{QPT}(\mathcal{D}_{\text{small}})}$	$\frac{\text{QPT}(\mathcal{D}_{\text{large}})}{\text{QPT}(\mathcal{D}_{\text{medium}})}$
TO-s1	2.2	2.2	2.2	2.2
TO-s2	2.0	2.1	2.5	2.4
TO-s3	2.0	2.1	2.5	2.4
TO-s4	2.0	2.1	2.2	2.2
TO-d1	2.2	2.2	2.3	2.3
TO-d2	2.0	2.1	2.2	2.3
TO-d3	2.2	2.6	2.3	2.6
DO-m1	2.0	2.1	2.4	2.4
DO-m2	2.0	2.2	2.3	2.4
DO-s1	2.0	2.0	2.3	2.4
DO-s2	2.0	2.1	2.4	2.4

document collection. The best scalability is observed for DO-s1, whereas the least scalable implementation is TO-d3. In general, the implementations are less scalable in case all answers are returned. This is basically due to the increasing overhead of the sorting phase, which does not scale well.

4.4. Experiments on space consumption

Fig. 13 displays the peak space consumption of each implementation. This value is equal to the maximum amount of space allocation for inverted lists, accumulators, and some auxiliary data structures, observed at any time while running the query processor for a query and answer set pair. It excludes the space for the general data structures which are utilized for each query. In all implementations, a data structure is immediately de-allocated at the moment it is no longer needed.

In TO implementations, the peak space consumption is reached when space for accumulators plus an inverted list is allocated. In TO-s implementations, the peak consumption is reached when the space for

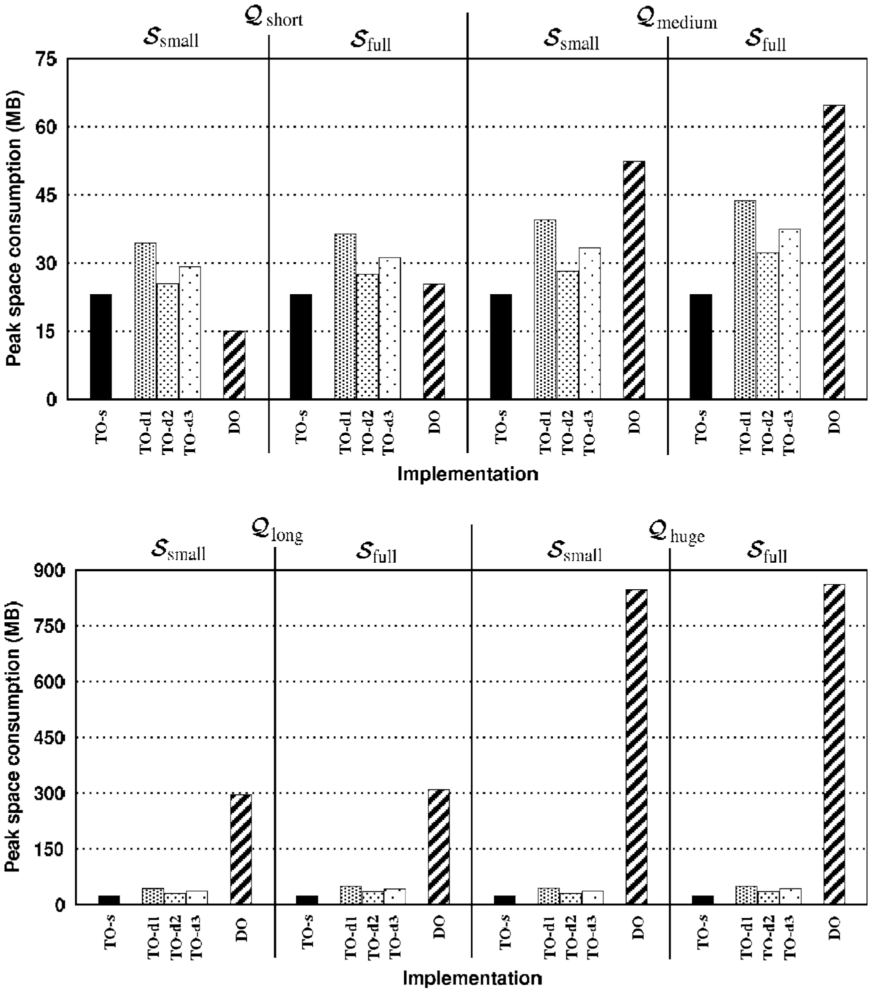

Fig. 13. Peak space consumption (in MB) observed for different implementations.

the inverted list with the highest number of postings is allocated. In DO implementations, it is reached when the space for all inverted lists is allocated and the number of accumulators is at the maximum.

According to Fig. 13, for short queries, DO implementations are the most space-efficient. However, there is a rapid increase in the space needs of this type of implementations as the queries get longer. This is basically because the storage amount of postings dominates that of accumulators since more inverted lists must be in the memory at the same time. For $\mathcal{Q}_{\text{medium}}$, $\mathcal{Q}_{\text{long}}$, and $\mathcal{Q}_{\text{huge}}$, TO-s implementations require the least amount of space. Among TO-d implementations, TO-d2 is the most space-efficient implementation.

5. Concluding discussion

Time complexities for different phases of the algorithms are summarized in Table 6. According to this table, in general, TO-s implementations differ in their selection phase whereas the update phase is discriminating for TO-d and DO implementations. Table 7 gives the total time and space complexities. The provided space complexities in Table 6 do not encapsulate the space cost of inverted lists, which is $O(e)$ for the TO implementations and $O(u)$ for the DO implementations.

It should be noted that different variants, which perform well under certain circumstances, can be created by slight modifications over the algorithms presented in this work. For example, TO-s4 can be modified so that in the extraction phase nonzero accumulators are placed in the first e elements, and the median-of-medians selection algorithm can be run only on these accumulators. In our experiments on this variant (although not reported here), we observed that this implementation is the fastest in processing short queries.

Similarly, DO-s2 can be modified using a pruning strategy such that only the postings having the minimum document id and their left and right children in the heap are checked. This approach performs well on long queries, but the bookkeeping overhead dominates at short queries. Similar optimizations are possible for space consumption. For example, TO-s2 and TO-s3 can be modified such that the accumulator array keeps only the scores. This decreases the space consumption to half of its original as long as $s \leqslant D/2$. Although our results indicate that TO-d implementations perform poorly, for querying scenarios where D and Q are high but e is low, implementations in TO-d category can be both time- and space-efficient.

Table 6
The run-time analyses of different phases in each implementation technique

Impl.	Time_C	Time_U	Time_E	Time_S	Time_R
TO-s1	$O(e)$	$O(u)$	$O(D)$	$O(1)$	$O(e\lg e)$
TO-s2	$O(e)$	$O(u)$	$O(D)$	$O(e+s\lg e)$	$O(1)$
TO-s3	$O(e)$	$O(u)$	$O(D)$	$O(s+(e-s)\lg s)$	$O(s\lg s)$
TO-s4	$O(e)$	$O(u)$	$O(1)$	$O(D)$	$O(s\lg s)$
TO-d1	$O(e)$	$O(u\lg e)$	$O(e)$	$O(s+(e-s)\lg s)$	$O(s\lg s)$
TO-d2	$O(B+e)$	$O(ue/B)$[a]	$O(e)$	$O(s+(e-s)\lg s)$	$O(s\lg s)$
TO-d3	$O(e)$	$O(u\lg e)$[a]	$O(e)$	$O(s+(e-s)\lg s)$	$O(s\lg s)$
DO-m1	$O(e+Q)$	$O(u\lg Q+eQ)$	$O(e)$	$O(s+(e-s)\lg s)$	$O(s\lg s)$
DO-m2	$O(e+Q)$	$O(u\lg Q)$	$O(e)$	$O(s+(e-s)\lg s)$	$O(s\lg s)$
DO-s1	$O(e+Q)$	$O(eQ)$	$O(e)$	$O(s+(e-s)\lg s)$	$O(s\lg s)$
DO-s2	$O(e+Q)$	$O(u\lg Q)$	$O(e)$	$O(s+(e-s)\lg s)$	$O(s\lg s)$

[a] Expected time complexities are given.

Table 7
The total time and space complexities for different implementations

Impl.	Time	Space
TO-s1	$O(D + e\lg e)$	$O(D)$
TO-s2	$O(D + s\lg e)$	$O(D)$
TO-s3	$O(D + e\lg s)$	$O(D)$
TO-s4	$O(D + s\lg s)$	$O(D)$
TO-d1	$O(u\lg e)$	$O(e)$
TO-d2	$O(ue/B + e\lg s)$[a]	$O(B + e)$
TO-d3	$O(u\lg e)$[a]	$O(e)$
DO-m1	$O(u\lg Q + eQ + e\lg s)$	$O(Q + s)$
DO-m2	$O(u\lg Q + e\lg s)$	$O(Q + s)$
DO-s1	$O(eQ + e\lg s)$	$O(Q + s)$
DO-s2	$O(u\lg Q + e\lg s)$	$O(Q + s)$

[a] Expected time complexities are given.

To summarize, the results show that there is no single, superior implementation. Depending on the properties of the computing system, document collection, user queries, and answer sets, each implementation has its own advantages. Currently, we are working on a hybrid system which will, depending on the parameters, intelligently select and execute the most appropriate implementation taking both time and space efficiency into consideration. Clearly, for a better analysis, the experiments need to be repeated on a larger document collection where D and T are much higher. For this purpose, we have started a large crawl of the Web and plan to repeat the experiments on this larger collection.

References

Baeza-Yates, R., & Ribeiro-Neto, B. (1999). *Modern information retrieval*. New York: Addison-Wesley.

Bell, T. C., Moffat, A., Nevill-Manning, C. G., Witten, I. H., & Zobel, J. (1993). Data compression in full-text retrieval systems. *Journal of the American Society for Information Science, 44*(9), 508–531.

Bohannon, P., McIlroy, P., & Rastogi, R. (2001). Main-memory index structures with fixed-size partial keys. *ACM SIGMOD Record, 30*(2), 163–174.

Buckley, C., & Lewit, A. (1985). Optimizations of inverted vector searches. In *Proceedings of the 8th international ACM SIGIR conference on research and development in information retrieval* (pp. 97–110). Montreal, Canada.

Can, F., Altingovde, I. S., & Demir, E. (2004). Efficiency and effectiveness of query processing in cluster-based retrieval. *Information Systems, 29*(8), 697–717.

Clarke, C. L. A., Cormack, G. V., & Tudhope, E. A. (2000). Relevance ranking for one to three term queries. *Information Processing and Management, 36*(2), 291–311.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2001). *Introduction to algorithms* (2nd ed.). Cambridge, MA: MIT Press.

Croft, W. B., & Savino, P. (1988). Implementing ranking strategies using text signatures. *ACM Transactions on Office Information Systems, 6*(1), 42–62.

Elmasri, R., & Navathe, S. (2003). *Fundamentals of database systems* (4th ed.). Reading, MA: Addison-Wesley.

Frakes, W. B., & Baeza-Yates, R. (1992). *Information retrieval: Data structures and algorithms*. Englewood Cliffs, NJ: Prentice Hall.

Goldman, R., Shivakumar, N., Venkatasubramanian, S., & Garcia-Molina, H. (1998). Proximity search in databases. In *Proceedings of the 24th international conference on very large data bases* (pp. 26–37). New York, USA.

Harman, D. W. (1986). An experimental study of factors important in document ranking. In *Proceedings of the 9th international ACM SIGIR conference on research and development in information retrieval* (pp. 186–193). Pisa, Italy.

Harman, D., & Candela, G. (1990). Retrieving records from a gigabyte of text on a multicomputer using statistical ranking. *Journal of the American Society for Information Science, 41*(8), 581–589.

Harper, D. J. (1980). Relevance feedback in document retrieval systems: An evaluation of probabilistic strategies. *Ph.D. Thesis*. The University of Cambridge.

Horowitz, E., & Sahni, S. (1978). *Fundamentals of computer algorithms*. Potomac, MD: Computer Science Press.

Hristidis, V., Gravano, L., & Papakonstantinou, Y. (2003). Efficient IR-style keyword search over relational databases. In *Proceedings of the 29th international conference on very large data bases* (pp. 850–861). Berlin, Germany.

Ilyas, F., Aref, G., & Elmagarmid, K. (2004). Supporting top-k join queries in relational databases. *The VLDB Journal—The International Journal on Very Large Data Bases, 13*(3), 207–221.

Kaszkiel, M., Zobel, J., & Sacks-Davis, R. (1999). Efficient passage ranking for document databases. *ACM Transactions on Information Systems, 17*(4), 406–439.

Knuth, D. (1998) (2nd ed.). *The art of computer programming: Sorting and searching* (Vol. 3). Reading, MA: Addison-Wesley.

Lee, D. L., Chuang, H., & Seamons, K. (1997). Document ranking and the vector-space model. *IEEE Software, 14*(2), 67–75.

Lehman, T. J., & Carey, M. J. (1986). A study of index structures for main memory database management systems. In *Proceedings of the 12th international conference on very large data bases* (pp. 294–303). Kyoto, Japan.

Long, X., & Suel, T. (2003). Optimized query execution in large search engines. In *Proceedings of the 29th international conference on very large databases*. Berlin, Germany.

Lucarella, D. (1988). A document retrieval system based upon nearest neighbor searching. *Journal of Information Science, 14*(1), 25–33.

Moffat, A., Zobel, J., & Sacks-Davis, R. (1994). Memory efficient ranking. *Information Processing and Management, 30*(6), 733–744.

Persin, M. (1994). Document filtering for fast ranking. In *Proceedings of the 17th international ACM SIGIR conference on research and development in information retrieval* (pp. 339–348). Dublin, Ireland.

Pugh, W. (1990). Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM, 33*(6), 668–676.

Salton, G., & McGill, M. J. (1983). *Introduction to modern information retrieval*. New York: McGraw-Hill.

Smeaton, A. F., & van Rijsbergen, C. J. (1981). The nearest neighbor problem in information retrieval: an algorithm using upperbounds. In *Proceedings of the 4th international ACM SIGIR conference on research and development in information retrieval* (pp. 83–87). Oakland, California.

Tomasic, A., Garcia-Molina, H., & Shoens, K. (1994). Incremental updates of inverted lists for text document retrieval. In *Proceedings of the 1994 ACM SIGMOD international conference on management of data* (pp. 289–300). Minneapolis, Minnesota.

Turtle, H., & Flood, J. (1995). Query evaluation: strategies and optimizations. *Information Processing and Management, 31*(6), 831–850.

Wilkinson, R., Zobel, J., & Sacks-Davis, R. (1995). Similarity measures for short queries. In *Fourth text retrieval conference (TREC-4)* (pp. 277–285). Gaithersburg, Maryland.

Witten, I. H., Moffat, A., & Bell, T. C. (1999). *Managing gigabytes: Compressing and indexing documents and images* (2nd ed.). San Francisco, CA: Morgan Kaufmann.

Wong, W. Y. P., & Lee, D. K. (1993). Implementations of partial document ranking using inverted files. *Information Processing and Management, 29*(5), 647–669.

Zobel, J., & Moffat, A. (1995). Adding compression to a full-text retrieval system. *Software Practice and Experience, 25*(8), 891–903.

Zobel, J., Moffat, A., & Sacks-Davis, R. (1992). An efficient indexing technique for full-text database systems. In *Proceedings of the 18th international conference on very large databases* (pp. 352–362). Vancouver, Canada.