SUMMARY  With the successful adoption of link analysis techniques such as PageRank and web spam filtering, current web search engines well support “navigational search”. However, due to the use of a simple conjunctive Boolean filter in addition to the inappropriateness of user queries, such an engine does not necessarily well support “informational search”. Informational search would be better handled by a web search engine using an informational retrieval model combined with enhancement techniques such as query expansion and relevance feedback. Moreover, the realization of such an engine requires a method to process the model efficiently. In this paper we propose a novel extension of an existing top-k query processing technique to improve search efficiency. We add to it the technique utilizing a simple data structure called a “term-document binary matrix,” resulting in more efficient evaluation of top-k queries even when the queries have been expanded. We show on the basis of experimental evaluation using the TREC GOV2 data set and expanded versions of the evaluation queries attached to this data set that the proposed method can speed up evaluation considerably compared with existing techniques especially when the number of query terms gets larger.

key words: web search engine, top-k query processing, early pruning, early termination, term-document binary matrix

1. Introduction

With the successful adoption of link analysis techniques such as PageRank and web spam filtering, current web search engines well support navigational search where a user is looking for a particular web resource that the user has in mind. However, such techniques does not necessarily well support informational search where a user is looking for information about a certain topic that might be on diverse web resources [7], [9].

There are a number of reasons for the relative ineffectiveness. One is that current web search engines adopts simple conjunctive Boolean filters for query processing [9]. From our experience in studying a retrieval effectiveness evaluation of web search engines [10], a web search engine should incorporate an information retrieval (IR) model [15], such as vector space model (e.g., TF-IDF) and probabilistic model (e.g., BM25), to better support informational search.

Another is that user queries are often inappropriate. There are two different cases for this problem. One case is the ambiguity of user queries [7]. User queries are often short and ambiguous such that the queries do not necessarily represent the user’s information need while the volume of web data is huge. Therefore, a simple IR model fails in general. In such a case, the user’s original query needs to be augmented by related terms such as synonyms, hyponyms and hypernyms, by applying query enhancement techniques [7] such as query expansion and relevance feedback. Based on the observation that some queries are hurt by such techniques, Cronen-Townsend et al. [8] improved such techniques with decision mechanisms based on the characteristics of queries. The basic idea is to disable such techniques if the query can be predicted to perform poorly.

The other case is the verbosity of user queries [3], [4]. Long and verbose queries allow users to express their information need using natural language. Such queries account for a small but significant percentage of the queries submitted to web search engines currently. However, a simple IR model can not deal with such a query well due to its complexity. To overcome this problem, Bendersky et al. [3], [4] proposed techniques for transforming such a query into an equivalent one that is more likely to perform well. They use query expansion as the final step of the query transformation.

In all of above cases, combining an IR model with query enhancement techniques is indispensable for better support of informational search on the web. However, the effectiveness issue is beyond the scope of this paper.

Nevertheless, to exploit the various techniques for web search engines, efficiency is another problem. Since expanded queries typically result in 10-30 terms and occasionally in a few hundred terms, evaluation of such long queries may become too slow, as reported by Carpineito et al. [7], and highly optimized techniques are hence needed. Therefore, we hereafter focus on the efficiency issue of evaluating such long queries.

There has been a considerable amount of work on optimization techniques including index compression and caching [21], result caching [14], and top-k query processing [1], [2], [5], [11], [13], [17], [18]. In this paper we fo-

1017

focus on top-k query processing techniques, which find the exact top-k documents without processing the entire posting list for each query term. Here, the exact top-k documents mean k ranked documents that are exactly the same as the k documents obtained from the top of all documents ranked by a given IR model. This approach is especially efficient in the case of large-scale IR systems such as web search engines where k is small and the posting lists can be overwhelmingly long. However, much of previous work on top-k query processing mainly dealt with relatively short queries. Therefore, we have developed a novel extension of an existing top-k query processing technique that enables it to efficiently evaluate long queries using an IR model even when the queries have been expanded [12]. It uses a simple data structure called a “term-document binary matrix,” which indicates which document contains which query term. To the best of our knowledge, there have been no reports on integrating such a data structure into top-k query processing techniques to increase the efficiency. To summarize, our main contributions are as follows.

1. We describe the integration of the term-document binary matrix into the best known top-k query processing technique, the Combined Algorithm (CA), enabling more efficient evaluation of top-k queries even when the queries have been expanded using enhancement techniques such as query expansion and relevance feedback.

2. We describe our experimental evaluation of the extended technique using the TREC GOV2 data set and expanded versions of the evaluation queries attached to this data set and show that it can speed up evaluation considerably compared with CA when the number of query terms gets larger.

This paper is organized as follows. Section 2 discusses related work on top-k query processing. Section 3 presents the model and algorithm that are used. Section 4 describes our approach. Section 5 explains the experimental evaluation done using the TREC GOV2 collection, describes the key results, and presents our conclusions. The key points are summarized and future work is mentioned in Sect. 6.

2. Related Work

There has been a large amount of work on top-k query processing in both IR and database communities. The IR community has a long history of research on efficient evaluation of vector space queries. Earlier work includes [5] and [17]. Although Buckley et al. [5] and Persin et al. [17] dealt with long queries, their techniques are intended for relatively small collections. There has been recent work, e.g., Anh et al. [1], aimed at efficient evaluation for top-k queries for large collections, but the techniques reported mainly focus on relatively short queries consisting of at most ten or so terms. In the database community, seminal work has been done by Fagin [11] who introduced a family of threshold algorithms for efficient evaluation of top-k queries. He introduced the notion of instance optimality and showed that his family of threshold algorithms satisfy this notion. Inspired by his work, many researchers in both the IR and the database communities extended the threshold algorithms. Among them, most relevant to our work are those using upper level information of sorted lists such as intersections of the lists [13], and those using lower level information of sorted lists, such as histograms of value distributions in the lists and/or co-occurrence statistics between the lists [2].

For the former type, Kumar et al. [13] generalized the threshold algorithms to the case in which pre-aggregated intersection lists of different lists are available in addition to the original sorted lists. Although their ideas are relevant to our approach, which uses upper level information in a term-document binary matrix, they deal with conjunctive queries. In contrast, we focus on disjunctive queries, which are essential for an IR model combined with enhancement techniques such as query expansion and relevance feedback, and traditionally studied in the IR community. Also, Pang et al. [16] extended the threshold algorithm by using pre-computed Bloom filters on sorted lists, which allow efficient pruning of candidate documents. Although they deal with disjunctive queries, their algorithm is intended for relatively short queries on relatively small collections.

Among the techniques using lower level information of sorted lists, Bast et al. [2] proposed integrating into the threshold algorithms a novel access scheduling of sorted lists based on statistics for the lists, such as histograms of per-term score distributions in the lists and co-occurrence statistics between the lists. While they did not report the instance optimality of their algorithms, they demonstrated a significant speedup in evaluating relatively short disjunctive queries. Their approach and ours are expected to be enhanced by integration.

Another enhancement technique for information search on the web is to take into account term proximity in an IR model, i.e., the distance between term occurrences in a document. Schenkel et al. [18] developed efficient top-k query processing techniques for a proximity-aware IR model. They focused on a proximity-aware scoring function defined by a linear combination of a standard BM25-based score and a proximity score, and extended an existing top-k query processing technique [20] that was originally intended for a standard IR model such as TF-IDF and BM25. The key idea is to exploit the proximity list that indexes the contribution to the proximity score for each pair of terms for each document, sorted by descending order. They showed that their techniques speeded up evaluation considerably with an improved result quality. However, since the underlying top-k query processing technique intends for relatively short queries as does other existing top-k query processing techniques, those evaluated efficiently by their techniques in [18] are limited to relatively short queries.

A proximity-aware IR model improves the result quality in average. However, in the case where a query is short and ambiguous such that it does not well represent the user information need, the proximity-aware IR model seems to
be of little use, and hence query enhancement techniques such as query expansion and relevance feedback are still required. Therefore, the approaches of Schenkel et al. [18] and us work complementarily. Furthermore, our technique can be easily integrated into their approach and is expected to improve their efficiency.

3. Preliminaries

3.1 Model

We describe the underlying model that is used.

Queries. We assume that a query \( q \) contains \( m \) terms \( t_1, \ldots, t_m \) and that the score of a document \( d \) for \( q \) is of the form \( \text{score}(d) = \text{score}(w(d, t_1), \ldots, w(d, t_m)) \) where \( \text{score} \) is a given scoring function and \( w(d, t_i) \) is the per-term score of \( d \) for a term \( t_i \). The scoring function is said to be monotone, if \( \text{score}(w_1, \ldots, w_m) \leq \text{score}(w'_1, \ldots, w'_m) \) whenever \( w_i \leq w'_i \) for every \( i \).

TF-IDF and BM25 functions are in this class of scoring functions because they are formalized by the simple aggregation function \( \text{score}(w_1, \ldots, w_m) = w_1 + \cdots + w_m \), which is obviously monotone. Hereafter, we assume that a scoring function is of this form.

Indexes. We assume that a posting list \( L_i \) is maintained for each term \( t_i \), that each posting in each \( L_i \) has a unique document ID and a per-term score, i.e., each posting is of the form \( \langle d, w(d, t_i) \rangle \), and that the postings in each \( L_i \) are sorted by the per-term score, \( w(d, t_i) \), of \( d \) for \( t_i \). As in Fagin [11], we consider two modes of access to the posting lists. The first mode is sorted access in which the query processor obtains the per-term score, \( w(d, t_i) \), of \( d \) for \( t_i \) in \( L_i \) by proceeding through \( L_i \) sequentially from the top. The second mode is random access in which the query processor obtains the per-term score, \( w(d, t_i) \), of \( d \) for \( t_i \) in \( L_i \) in one random access. In addition, we assume that we have a list for each \( t_i \) that contains only the IDs of documents containing \( t_i \). The DocID-only lists are used for creating a term-document binary matrix presented in the next section.

3.2 Instance Optimality

We focus on an instance optimal algorithm for efficient top-k document retrieval. Intuitively, instance optimality implies the optimality of an algorithm for every query, not only for the worst case or the average case query. As in Fagin [11], we formalize this notion. Let \( \mathbf{D} \) be the class of databases consisting of posting lists for terms, and let \( \mathbf{A} \) be the class of algorithms that exactly find the top-k documents for every database \( \mathcal{D} \in \mathbf{D} \). We say that an algorithm \( \mathcal{B} \in \mathbf{A} \) is instance optimal over \( \mathbf{A} \) and \( \mathbf{D} \) if, for every \( \mathcal{A} \in \mathbf{A} \) and \( \mathcal{D} \in \mathbf{D} \), we have \( \text{cost}(\mathcal{B}, \mathcal{D}) = O(\text{cost}(\mathcal{A}, \mathcal{D})) \), where \( \text{cost} \) refers to the total amount of resources consumed by the algorithm. If an algorithm \( \mathcal{A} \) does \( s \) sorted accesses with cost \( c_S \) per sorted access on \( \mathcal{D} \), and \( r \) random accesses with cost \( c_R \) per random access on \( \mathcal{D} \), \( \text{cost}(\mathcal{A}, \mathcal{D}) = s \cdot c_S + r \cdot c_R \). In our scenario, \( c_R \gg c_S \).

3.3 Combined Algorithm

We extended CA, the best known top-k query processing technique and one of the threshold algorithms of Fagin [11]. CA combines sorted access with random access and is appropriate when random access is expensive relative to sorted access, i.e., \( c_R \gg c_S \) as in the case of IR systems including web search engines. The outline of CA is following:

1. CA begins by doing sorted access in parallel for each of the sorted lists, and every \( h = \lfloor c_R / c_S \rfloor \) steps (that is, every time the depth of the sorted access increases by \( h \)), random access is done to compute the total score of a document seen so far for which not all per-term scores for the query terms are known.

2. During the execution of the algorithm, the query processor maintains the set of current top-k documents seen so far denoted as \( T_k \), based primarily on the lower bound score defined by Eq. (1) below, and secondarily on the upper bound score defined by Eq. (2) below (If two documents have the same \( \text{score}_{LB}(d) \) value, then ties are broken using the \( \text{score}_{UB}(d) \) values such that the document with the largest \( \text{score}_{UB}(d) \) value wins and arbitrarily among documents that tie for the largest \( \text{score}_{UB}(d) \)).

3. The query processor halts when it observes that the upper bound score of each document left outside \( T_k \) is no longer larger than the lower bound score of the rank-k document in \( T_k \).

The characteristics of CA include early pruning and early termination, which are described next.

Early pruning. Let \( \text{score}_{LB}(d) \) be the lower bound score of document \( d \) seen so far:

\[
\text{score}_{LB}(d) = \sum_{t_i \in d} w(d, t_i). 
\]

Let \( \text{score}_{UB}(d) \) be the upper bound score of document \( d \) seen so far:

\[
\text{score}_{UB}(d) = \text{score}_{LB}(d) + \sum_{t_i \in q \setminus d} w_i, 
\]

where \( a_d = \{t_1, \ldots, t_k\} \subseteq q = \{t_1, \ldots, t_m\} \) contains already known query terms in \( d \), with per-term scores \( w(d, t_k), \ldots, w(d, t_k) \), and \( w_i \) is the (smallest) per-term score obtained through the most recent sorted access in sorted list \( L_i \). Let \( \text{mink} \) be the lower bound score of the rank-k document in \( T_k \). The query processor safely prunes \( d \) when \( \text{score}_{UB}(d) \) is no longer larger than \( \text{mink} \). Thus, CA can keep bookkeeping cost small. Early pruning is very important for efficient execution of the algorithm.

Early termination. Let \( S \) be the set of documents seen so far left outside \( T_k \) having \( \text{score}_{UB}(d) \geq \text{mink} \) and \( U \) be the set of documents that have not been seen so far. Let \( \text{score}_{UB}(S) \) be the maximum upper bound score of documents in \( S \):
score_{UB}(S) = \max_{d \in S} score_{UB}(d).  \quad (3)

Let \( score_{UB}(U) \) be the maximum upper bound score of documents in \( U \):

\[
\text{score}_{UB}(U) = \sum_{j=1}^{m} w_j. \quad (4)
\]

The query processor halts, yielding the exact top-k documents, when \( score_{UB}(U) \) are no longer larger than \( mink \), and \( S \) becomes empty. Thus, CA can stop without processing the entire sorted list for each query term. Early termination is especially efficient in the case of large-scale IR systems such as web search engines where \( k \) is small and the posting lists can be overwhelmingly long.

We show the pseudo-code of the CA algorithm.

**Algorithm 1 CA**

01: Set \( T_k = \emptyset \)
02: Set \( S = \emptyset \)
03: Set \( cnt = 0 \)
04: Set \( mink = 0 \)
05: Set \( score_{UB}(U) = \) an large value
06: while
07: \( \text{DoSortedAccess}() \)
08: \( \text{MaintainCAState}() \)
09: if \( cnt = mnh \) then
10: \( \text{DoRandomAccess}() \)
11: \( \text{MaintainCAState}() \)
12: \( cnt = 0 \)
13: end if
14: if \( \text{termination conditions are met} \) then
15: return \( T_k \)
16: end if
17: end while
18: Termination conditions:
19: a) \( T_k \) contains \( k \) distinct documents.
20: b) \( score_{UB}(U) \leq mink \) and \( S = \emptyset \).

\( \text{DoSortedAccess}() \) is defined as follows.

**Procedure** \( \text{DoSortedAccess}() \)

01: for \( t_i \in q \) do
02: \( \text{Let} \ (d, w(d, t_i)) \) be the next posting in \( L_i \)
03: if \( d \notin T_k \cup S \) then
04: if \( score_{UB}(U) > mink \) then
05: Create an element for \( d \) in \( S \)
06: Set \( score(d) = w(d, t_i) \)
07: Set \( ad = \{ t_i \} \)
08: end if
09: else
10: Set \( score(d) += w(d, t_i) \)
11: Set \( ad = ad \cup \{ t_i \} \)
12: end if
13: end for
14: \( cnt += m \)

We call a document \( d \) viable if \( score_{UB}(d) > mink \). \( \text{DoRandomAccess}() \) is defined as follows.

**Procedure** \( \text{DoRandomAccess}() \)

01: Set \( d = \arg \max_{d \in T_k \cup S} \text{score}_{UB}(d) \)
02: \( \text{Do random accesses for} \ d \text{'s missing per-term scores} \) (if \( d \) exists)

\( \text{MaintainCAState}() \) is defined as follows.

**Procedure** \( \text{MaintainCAState}() \)

01: Maintain the last obtained \( w_1, \ldots, w_m \) encountered in the lists
02: Maintain \( T_k \)
03: Update \( mink = \min_{d \in T_k} \text{score}_{UB}(d) \)
04: Maintain \( S \)
05: Update \( score_{UB}(U) \) (using Eq. (4))

This algorithm consists of two phases. The first phase of the algorithm (that is, when \( score_{UB}(U) > mink \), which means that some document in \( U \) might make it into the final top-k documents), is to find all the documents that could qualify for the final top-k documents. The second phase of the algorithm (that is, when \( score_{UB}(U) \leq mink \), which means that no documents in \( U \) could make it into the final top-k documents), is to choose the final top-k documents from the documents seen so far.

4. Our Approach

4.1 The Key Idea

As reported by Bast et al. [2], CA shows poor efficiency when used for retrieving the top-k documents for long queries for a number of reasons. First, the upper bound score, \( score_{UB}(d) \), computed using Eq. (2) becomes looser (larger) when the number of query terms is increased because of the increase in unknown query terms. Looser upper bound scores restrict early pruning, which significantly increases bookkeeping overhead. Secondly, the maximum upper bound scores, \( score_{UB}(S) \) and \( score_{UB}(U) \), also becomes looser (larger) when the number of query terms is increased. This restricts the possibility of early termination. Thus, CA is not effective in this scenario.

To overcome these shortcomings of CA, we propose integrating the simple data structure, \( B_q \), which depends on query \( q \), into CA. The \( B_q(t, d) \) value is 1 if query term \( t \) is contained in document \( d \), and 0 if not. We call this matrix a “term-document binary matrix.” We re-estimate \( score_{UB}(d) \), \( score_{UB}(S) \) and \( score_{UB}(U) \) more tightly by using the term-document binary matrix:

\[
\text{score}_{UB}(d) = \text{score}_{LB}(d) + \sum_{t_i \in q, a_d \in B_q(t, d) = 1} w_i, \quad (5)
\]

\( \text{score}_{UB}(S) \) is re-estimated using Eqs. (3) and (5), and \( \text{score}_{UB}(U) \) is re-estimated using...
\[ \text{score}_{UB}(U) = \sum_{i=1}^{m} w'_i, \]  

where \( w'_i \) is the \( i \)-th largest value of \( w_1, \ldots, w_m \) and \( m' \) is the maximum co-occurrence of query terms in documents in \( U \), i.e., \( m' = \max_{d \in U} \sum_{q \in q} B_q(t, d) \). The \( m' \) can be efficiently computed by calculating the co-occurrence statistics for all documents in a collection based on \( B_q \) at the beginning of the algorithm and then updating the statistics each time an unknown document in \( U \) is seen by doing sorted access during the execution of the algorithm. CA using these tighter upper bounds leads to better early pruning and early termination.

4.2 The Algorithm

Now we present our version of the CA algorithm called “BMCA,” short for “term-document-binary-matrix-based Combined Algorithm.”

**Algorithm 2 BMCA**

1: Set \( T_k = \emptyset \)
2: Set \( S = \emptyset \)
3: Set \( \text{cnt} = 0 \)
4: Set \( \text{mink} = 0 \)
5: Set \( \text{score}_{UB}(U) = \) an large value
6: CreateBinaryMatrix()
7: for \( n = 1 \) to \( m' \) do
8: \( \text{Set } c_n = |\{d \mid \sum_{q \in q} B_q(t, d) = n\}| \)
9: end for
10: Set \( m' = \max\{n \mid c_n > 0\} \)
11: while
12: \( \text{DoOptimizedSortedAccess()} \)
13: \( \text{MaintainBMCAState()} \)
14: if \( \text{cnt} = \text{mink} \) then
15: \( \text{DoOptimizedRandomAccess()} \)
16: \( \text{MaintainBMCAState()} \)
17: \( \text{cnt} = 0 \)
18: end if
19: if termination conditions are met then
20: return \( T_k \)
21: end if
22: end while

23: Termination conditions:
24: a) \( T_k \) contains \( k \) distinct documents.
25: b) \( \text{score}_{UB}(U) \leq \text{mink} \) and \( S = \emptyset \).

CreateBinaryMatrix() of BMCA is defined as follows.

**Procedure CreateBinaryMatrix()**

1: Create a zero-initialized m-bit array as \( B_q \)
2: for all \( t \in q \) do
3: \( \text{Load the document-ID only list of } t \)
4: for all \( d \) in the list do
5: \( \text{Set } B_q(t, d) = 1 \)
6: end for
7: end for

DoOptimizedSortedAccess() is defined as follows.

**Procedure DoOptimizedSortedAccess()**

1: \( R = \text{ChooseLists()} \)
2: for all \( L_i \in R \) do
3: \( \text{Let } (d, w(d, t_i)) \text{ be the next posting in } L_i \)
4: \( \text{if } d \notin T_k \cup S \text{ then} \)
5: \( \text{if } \text{score}_{UB}(U) > \text{mink} \text{ then} \)
6: \( \text{Create an element for } d \text{ in } S \)
7: \( \text{Set } \text{score}(d) = w(d, t_i) \)
8: \( \text{Set } a_d = \{t_i\} \)
9: \( \text{Set } n = \sum_{q \in q} B_q(t, d) \)
10: \( \text{cnt} -= 1 \)
11: end if
12: else
13: \( \text{Set } \text{score}(d) += w(d, t_i) \)
14: \( \text{Set } a_d = a_d \cup \{t_i\} \)
15: end if
16: \( \text{cnt} += 1 \)
17: if \( \text{cnt} = \text{mink} \) then
18: break
19: end if
20: end for

ChooseLists() in the above pseudo-code is defined as follows.

**Procedure ChooseLists()**

1: if \( \text{score}_{UB}(U) > \text{mink} \text{ then} \)
2: \( \text{Set } R = \{L_i \mid m' \text{ lists that have the largest } m'w'_i \text{ values} \} \)
3: else
4: \( \text{Set } R = \{L_i \mid \exists d \in T_k \cup S \text{ such that } t_i \notin a_d \text{ and } B_q(t_i, d) = 1\} \)
5: end if
6: return \( R \)

DoOptimizedRandomAccess() is defined as follows.

**Procedure DoOptimizedRandomAccess()**

1: \( \text{Set } d = \text{arg max}_{d \in T_k \cup S} |\{q \mid \sum_{q \in q} B_q(t, d) \text{ score}_{UB}(d)\} - |a_d|\text{ missing per-term scores (if } d \text{ exists} \}) \)

MaintainBMCAState() is defined as follows.

**Procedure MaintainBMCAState()**

1: \( \text{Maintain the last obtained } w_1, \ldots, w_m \text{ encountered in the lists} \)
2: \( \text{Maintain } T_k \)
3: \( \text{Update } \text{mink} = \min_{d \in T_k} \text{score}_{UB}(d) \)
4: \( \text{Maintain } S \)
5: \( \text{Update } m' = \max\{n \mid \text{cnt} > 0\} \)
6: \( \text{Update } \text{score}_{UB}(U) \text{ (using Eq. (6))} \)
This algorithm optimizes the execution of the sorted and random access, which is described next.

Optimization of Sorted Access. In the first phase (i.e., $score_{UB}(U) > mink$), the algorithm employs a strategy to efficiently reduce $score_{UB}(U)$ to early finish the phase. To do so, using Eq. (6), the algorithm does sorted access enough to decrease $score_{UB}(U)$. In the second phase, the algorithm employs a strategy to avoid useless sorted access. To do so, using $B_q$, the algorithm does sorted access only to the lists where at least one per-term score of some document in $T_k \cup S$ is expected to be obtained. These are illustrated in Procedure ChooseLists().

Optimization of Random Access. By $score_{UB}(d)$ computed using Eq. (5), which is tighter than that computed using Eq. (2) of CA, the algorithm chooses a more viable document for random access than does CA. This leads to an increase in $mink$ and thereby earlier pruning and termination.

We illustrate the execution (especially the sorted access strategy) of BMCA along with that of CA. Let us consider query $q = \{t_1, t_2, t_3\}$, and $k = 2$, i.e., we are to retrieve the two documents with the highest scores. Figure 1 shows the first three iterations of CA and BMCA for $L_1$, $L_2$ and $L_3$.

First, let us consider the case of CA. The algorithm does sorted access in parallel to each of the lists at each iteration. When we finished the first three iterations, we have $score_{UB}(U) = 0.30 + 0.35 + 0.40 = 1.05$, $T_2 = \{d_1, d_2\}$ and $mink = score_{UB}(d_2) = 0.95$. Therefore, $score_{UB}(U) > mink$. This means that the algorithm must continue the first phase (phase I).

Next, let us consider the case of BMCA. In Fig. 1 we show the term-document binary matrix only for top-ranked documents in $L_1$, $L_2$ and $L_3$, for which we assume that we have $c_3 = 2$, $c_2 = 20$ and $c_1 = 50$ at the beginning of the algorithm, i.e., $d_1$ and $d_3$ contain all the query terms while the other documents contain at most two query terms. The term-document binary matrix at each step of the algorithm represents that which query terms of which documents have been seen so far from $L_1$, $L_2$ and $L_3$.

We show that unlike CA, BMCA can break phase I by the first three iterations. The algorithm does sorted access in parallel to each of the lists until the second iteration because $score_{UB}(U) > mink$ and because $m_1 = 3$. After the second iteration, however, we have $m_1 = 2$ because $c_1 = 0$ due to that both $d_1$ and $d_3$ have appeared so far in some lists. Hence, as shown in Fig. 1, $score_{UB}(U)$ is updated using the two largest $w_i$ values: $score_{UB}(U) = 0.70 + 0.45$ (The $m'_1$ postings that have the $m'_1$ largest $w_i$ values after each iteration is underlined in Fig. 1). However, since we have $T_2 = \{d_1, d_2\}$ and $mink = score_{UB}(d_2) = 0.95$, $score_{UB}(U) > mink$, which means that the algorithm must continue phase I. Unlike the previous iterations, the algorithm does sorted access only to $L_2$ and $L_3$ at the next iteration, which have the two largest $w_i$ values used to compute $score_{UB}(U)$ as above. After the third iteration, as shown in Fig. 1, $score_{UB}(U)$ is updated using the two largest $w_i$ values decreased than before: $score_{UB}(U) = 0.40 + 0.40$, which leads that $score_{UB}(U) < mink$. Consequently, the algorithm breaks phase I after the third iteration and start the second phase (phase II). Note that the algorithm can break phase I by doing 8 sorted accesses while CA can break phase I by doing at least 12 sorted accesses. As shown in this example the algorithm finish phase I much earlier by doing optimized sorted access.

At this point we have $T_2 = \{d_1, d_2\}$ and $S = \{d_3, d_4, d_6\}$. Since we know from the term-document binary matrix that $d_4$ and $d_6$ in $S$ have the final scores 0.45 and 0.40 respectively and that the final scores are no longer larger than $mink = 0.95$, $d_4$ and $d_6$ can be pruned off safely. In addition, since the term-document binary matrix indicates that the only missing per-term scores are the ones of $d_2$ and $d_3$ for $t_3$, the algorithm has only to do sorted access to $L_3$ in phase II (that is, the algorithm avoids useless sorted access to $L_1$ and $L_2$). As shown in this example, BMCA are able to optimize sorted access not only during phase I but also during phase II.

4.3 Exactness of BMCA

We show in Appendix A that BMCA is exact as well as CA is.

Theorem 1: Let $score$ be a simple aggregation function. Then BMCA exactly finds the top-k documents.

4.4 Instance Optimality of BMCA

When an adaptive sorted access scheduling method other than round robin style scheduling is introduced into CA, it is not necessarily guaranteed that the resulting algorithm is still instance optimal. However, we show in Appendix B that BMCA is still instance optimal under a certain condition. We say that a database $\mathcal{D}$ satisfies the distinct property if for each $i$, no two documents in $\mathcal{D}$ have the same score in the $i$-th list, that is, if for each $i$, the per-term scores in the $i$-th list are distinct.

Theorem 2: Let $score$ be a simple aggregation function. Let $\mathcal{D}$ be the class of databases consisting of posting lists for terms that satisfy the distinct property, and $\mathcal{A}$ be the class of algorithms that exactly find the top-k documents for $score$ for every database $\mathcal{D} \in \mathcal{D}$. Then BMCA is instance optimal over $\mathcal{A}$ and $\mathcal{D}$.

5. Evaluation

5.1 Setup

We used the TREC GOV2 data set. The corpus contains about 25 million web documents crawled from the gov domain during early 2004. The uncompressed size of the corpus is 426 GB. The data set was used for the TREC Terabyte Track[6], which consisted of three experimental tasks: an adhoc retrieval task, a nemed page finding task, and an ef-
ficiency task. Adhoc retrieval is an informational search task while named page finding is a navigational search task. The efficiency task extends both the adhoc retrieval task and the named page finding task, and investigates efficiency and scalability issues in IR systems.

The objective of this research is to realize an optimization technique that can evaluate long queries for informational search efficiently. To examine the retrieval efficiency of the proposed method for such long queries, we created evaluation queries in the following way:

1. We focused on the evaluation queries attached to the
1023

FUJITA and OYAMA: EFFICIENT TOP-K DOCUMENT RETRIEVAL FOR LONG QUERIES USING TERM-DOCUMENT BINARY MATRIX

Table 1 Index size in GB for TREC GOV2 data set.

<table>
<thead>
<tr>
<th>Disk size (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorted lists</td>
</tr>
<tr>
<td>26.2</td>
</tr>
<tr>
<td>DocID-only lists</td>
</tr>
<tr>
<td>6.0</td>
</tr>
</tbody>
</table>

adhoc retrieval task. The queries consist of 150 topics in total (topics 701-750 for TREC 2004, topics 751-800 for TREC 2005 and topics 801-850 for TREC 2006), and each topic has the title field, which essentially contains a keyword query, similar to a query that might be entered into a web search engine.

2. According to our experiment on the retrieval effectiveness of query expansion done using the Indri search engine as shown in Appendix C, we used 8-, 16-, 32- and 64-term expanded queries as evaluation queries for our experiments. In the experiments described here, no stopwords were removed.

We implemented CA and BMCA on the Zettair search engine, an open source fast text search engine developed by the RMIT Search Engine Group.† Because the Zettair search engine provides a standard technique for processing the entire posting list for each query term for the pivoted cosine model [19], we implemented this model in CA and BMCA. For handling sorted access, both CA and BMCA were implemented such that they obtained a partition of 64 KB in each list per sorted access. This partition contained about 13,000 postings. For handling random access, we integrated the additional data structure, which assigned each document to a document vector containing all terms in that document with nonzero per-term scores, along with their scores. Moreover, both CA and BMCA were implemented such that they obtained a document vector in one random access. In addition, while we described both algorithms in Sects. 3 and 4 such that they did one random access every time they did \( mh \) sorted accesses, they were extended such that they did \( \nu \) random accesses every time they did \( \mu \) sorted accesses.‡ We created a term-document binary matrix for a 8-, 16-, 32- or 64-term expanded query as an array of 8-, 16-, 32- or 64-bit integers. Table 1 shows the amount of disk space required to store DocID-only lists for all terms compared with that required to store sorted lists for all term. Although BMCA needs DocID-only lists as additional data, we see from the table that DocID-only lists are compact: The DocID-only lists for all terms are about 23% of the size of the sorted lists for all terms.

Our system was a Redhat Linux server equipped with a 2.50-GHz Intel(R) Xeon quad-core processor, 8 GB memory and 8 SATA-II disk drives configured as RAID5 disk array. All runs were done on a single core.

5.2 Results

We compared the retrieval efficiency of our algorithm, BMCA, with that of the original algorithm, CA, and that of a standard technique, in which the entire posting list for each query term is processed. We measured the average execution time per query.

Figure 2 shows the average execution times for 8-, 16-, 32- and 64-term queries for various values of \( k \) for the TREC GOV2 data set. BMCA significantly outperformed the standard technique for every query size and every \( k \). Moreover, while it remained comparable to CA for small \( k \) for every query size, it considerably outperformed CA for large \( k \) especially for large query sizes. In particular, while the retrieval efficiency of both BMCA and CA deteriorated as \( k \) increased for every query size, CA degradation was much greater, and its retrieval efficiency actually became worse than that of the standard technique for query size \( = 64 \) and \( k = 100 \). BMCA showed the increased retrieval efficiency, yielding a speedup of up to a factor of about 2 over CA. This is because our term-document-binary-matrix-based algorithm leads to better early pruning and termination even for large \( k \) for long queries.

Table 2 shows the average time required to create a term-document binary matrix for a expanded query and the memory used to store the term-document binary matrix. These times account for about 35% to 43% of the average execution times when \( k = 10 \), and that cancels the merit of BMCA when \( k \) is small. We think that the memory usage is not so large even for a 64-terms query, considering the amount of available RAM per processor core on a currently available multi-core system. Therefore, we expect that the proposed method allows to run concurrent queries up to at least the number of processor cores on such a system.

Figure 3 illustrates the effects of the early pruning and early termination property of BMCA. It shows the number of documents retained at each iteration \( |T_k \cup S| \) during the execution of BMCA for, as an example, a 32-term expanded query from the title field of TREC TOPIC 741 and \( k = 100 \). BMCA pruned more documents than CA by an order of magnitude and terminated after about three fourths of the number of iterations done by CA. That is, BMCA did about three fourths of sorted and random accesses done by CA. We statistically analyzed this property of BMCA. Figure 4 shows the average total number of sorted accesses per query done by CA and BMCA for 8-, 16-, 32- and 64-term queries for various values of \( k \), and Fig. 5 shows the average maximum number of documents per query retained by both algorithms for 8-, 16-, 32- and 64-term queries for various values of \( k \). The results shown in these figures indicate that, for long queries, CA incurs substantial I/O overhead due to more sorted and random access along with considerable CPU overhead due to the explosive increase of retained documents, resulting in efficiency degradation for large \( k \) for long queries. In contrast, BMCA incurs much less I/O overhead due to efficient early pruning and early termination based on the term-document binary matrices along with

†http://www.seg.rmit.edu.au/zettair/

‡\( \mu \) and \( \nu \) were set as follows: \( \mu = 8 \), 16, 32 and 64 for 8-, 16-, 32- and 64-term queries for the first iteration, respectively, and \( \mu = 400 \) for every query size after the first iteration. \( \nu = 200 \) for every query size for every iteration.
much less CPU overhead due to a reduced number of retained documents, even for large k for long queries. As a result, BMCA speeds up evaluation considerably compared with CA when the number of query terms gets larger.

Table 2  Average creation times and memory usage of a term-document binary matrix for 8-, 16-, 32- and 64-term queries for TREC GOV2 data set.

<table>
<thead>
<tr>
<th>Number of Documents Retained at Each Iteration During Execution</th>
<th>(a) 8-term queries</th>
<th>(b) 16-term queries</th>
<th>(c) 32-term queries</th>
<th>(d) 64-term queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (seconds)</td>
<td>Memory (MB)</td>
<td>Time (seconds)</td>
<td>Memory (MB)</td>
<td>Time (seconds)</td>
</tr>
<tr>
<td>8-term queries</td>
<td>0.72</td>
<td>24.0</td>
<td>16-term queries</td>
<td>1.19</td>
</tr>
</tbody>
</table>

6. Conclusion

Our integration of a simple data structure, the “term-document binary matrix,” into the CA algorithm resulted in efficient evaluation of top-k (disjunctive) queries, even ones expanded using enhancement techniques such as query expansion and relevance feedback. Experimental evaluation using the TREC GOV2 data set and expanded versions of the evaluation queries attached to this data set showed that the proposed algorithm, which has instance optimality, speeded up evaluation considerably compared with CA when the number of query terms gets larger.

The creation of the term-document binary matrices from the DocID-only lists imposes some overhead due to our simple implementation of the functionality. We plan to re-implement the functionality using state-of-the-art techniques, e.g., Ref. [21], which should greatly reduce the overhead. We also plan to investigate the combination of our approach with that of Bast et al. [2] to achieve a more sophisticated random access scheduling.

References

Fig. 4 Average total number of sorted accesses done by CA and BMCA for 8-, 16-, 32- and 64-term queries for various values of k for TREC GOV2 data set.

Fig. 5 Average maximum number of documents retained by CA and BMCA for 8-, 16-, 32- and 64-term queries for various values of k for TREC GOV2 data set.


Appendix A: Proof of Theorem 1

Assume that BMCA returns $T_k = \{d_1, d_2, \ldots, d_k\}$. Let $d'$ be a document in $U$. From the termination conditions, we need only show that $score(d_i) \geq score(d')$ for each $i$. By the monotonicity of $score(d)$ and the definition of $mink$, $score(d_i) \geq score_{UB}(d_i) \geq mink$.

By the termination conditions, $mink \geq score_{UB}(U) \geq score_{UB}(d')$, where the right inequality follows by the property of $score_{UB}(U)$. Also by the monotonicity of $score(d)$, $score_{UB}(d') \geq score(d')$.

Combining the inequalities, we have $score(d_i) \geq score(d')$, for each $i$, as desired.

Appendix B: Proof of Theorem 2

Assume that algorithm $A \in A$ is run over database $D \in D$. Assume that $A$ halts at depth $d^A$ (that is, if $d^A_i$ is the number of documents seen under sorted access to list $L_i$, for $1 \leq i \leq m$, then $d^A = \max_{i=1,\ldots,m}(d^A_i)$). Assume that $A$ sees a distinct documents. In particular, $d^A \leq a$.

Let $x^A_i$ be the per-term score of the last document seen under sorted access to $L_i$, for $1 \leq i \leq m$. Define the threshold value $\tau_A$ to be $score(x^A_1, \ldots, x^A_m)$. For the final top-k documents $d$ returned by $A$,

$score(d) \geq \tau_A$.

Later, we shall show that BMCA halts on $D \in D$ in at most $b = (a + 1)m(1 + m^2h)$ iterations of DoOptimizedSortedAccess() combined with DoOptimizedRandomAccess(). Hence, the cost of BMCA is

$cost(BMCA) \leq bmcS + \frac{bm}{h}(m - 1)c_R$,

$= \frac{bmcS}{h}(m - 1)c_R\left(c_R/c_S\right)$,

$= accS + ccS$,

where

$c = m(1 + m^2h) + (m - 1)/h(c_R/c_S))$.

On the other hand, since $A$ sees $a$ distinct documents, $cost(A) \geq acS$.

Therefore,

$cost(BMCA) \leq c \cdot cost(A) + ccS$.

This shows that BMCA is instance optimal, as desired.

Lemma 1: If BMCA iterates DoOptimizedSortedAccess() $(a + 1)m + 1$ times, then, at the $n$-th iteration $(1 \leq n \leq (a + 1)m + 1)$, all the selected lists to do sorted access satisfy the following condition.

$d^h_i \geq a + 1 \forall$ selected $L_i$,

where $d^h_i$ is the number of documents seen under sorted access to $L_i$ so far.

Proof. To show this we will use proof by contradiction. Assume that, for each iteration until the $((a + 1)m + 1)$-th iteration, at least one list selected to do sorted access satisfied $d^h_i \leq a$. This means that the number of postings that reside in lists by depth $a + 1$ is $(a + 1)m + 1$. However, the number of such postings is $(a + 1)m$. This implies a contradiction. Therefore, at least one iteration (the $n$-th iteration), all the selected lists to do sorted access satisfy $d^h_i \geq a + 1$, as desired.

Lemma 2: If BMCA iterates DoOptimizedSortedAccess() $n - 1$ times, then, for any document $d$ not seen by BMCA,

$score_{UB}(d) < \tau_A$.

Proof. Let $C$ be a set of $m'$ selected lists to do sorted access at the $n$-th iteration that are corresponding to largest $m'$ $w^*_d$. Let $d$ be a document not seen by BMCA by the $(n - 1)$-th iteration. Let $C_d$ be a set of lists containing nonzero per-term scores of $d_i$ that is $C_d = \{L_i | B_d(t_i, d) = 1\}$. In particular, $|C_d| \leq |C| = m'$. Then, by the definition of $C$,
Moreover, since that, from Lemma 1, \( d'^n_i \geq a + 1 \), for every \( L_i \in C \) and that \( d'^n_i \leq a, d'^n_i < d^n_i \). Hence, due to the distinct property of \( D \),

\[
\sum_{L_i \in C} w_i \leq \sum_{L_i \in C} w' \leq \sum_{L_i \in C} w_j.
\]

On the other hand, since that, from Lemma 1, \( d'^n_i \geq a + 1 \), for every \( L_i \in C \) and that \( d'^n_i \leq a, d'^n_i < d^n_i \). Hence, due to the distinct property of \( D \),

\[
\sum_{L_i \in C} w_i < \sum_{L_i \in C} w' \leq \sum_{i=1}^{m} x'^n = \tau_{\mathcal{A}}.
\]

Therefore,

\[
\text{score}_{UB}(d) < \tau_{\mathcal{A}}.
\]

Lemma 3: If BMCA iterates DoOptimizedSortedAccess() \( n-1 \) times, then the number of documents seen by BMCA that satisfy \( \text{score}_{UB}(d) \geq \tau_{\mathcal{A}} \) is at most \((a + 1)m^2\).

Proof. Since DoOptimizedSortedAccess() does sorted access to at most \( m \) lists in one iteration, the number of documents seen by BMCA after \( n-1 \) iterations of DoOptimizedSortedAccess() is at most \( m(n-1) \leq (a + 1)m^2 \).

Lemma 4: If BMCA iterates DoOptimizedSortedAccess() \( n - 1 + (a + 1)m^3h \) times, then, for each document \( d \) seen by BMCA that satisfies \( \text{score}_{UB}(d) \geq \tau_{\mathcal{A}} \), the final score \( \text{score}(d) \) is determined.

Proof. If BMCA iterates DoOptimizedSortedAccess() \( n - 1 + (a + 1)m^3h \) times, then, DoOptimizedRandomAccess() is done at least \((a + 1)m^2\) times after \( n-1 \) iterations of DoOptimizedSortedAccess(). We assume that the \( n_i \)-th iteration of DoOptimizedSortedAccess() is followed by DoOptimizedRandomAccess(), for \( 1 \leq l \leq (a + 1)m^2 \). Let \( d_l \) be the document for that DoOptimizedRandomAccess() does random access, for \( 1 \leq l \leq (a + 1)m^2 \). Here, we use the notation \( \text{score}_{UB}^{n_i}(d_l) \) as \( \text{score}_{UB}(d_l) \) estimated after the \( n_i \)-th iteration of DoOptimizedSortedAccess(). So, by the definition of choosing a document to do random access,

\[
\text{score}_{UB}^{n_i}(d_l) \geq \text{score}_{UB}^{n_i}(d_{l+1}).
\]

Moreover, since that \( \text{score}_{UB}(d) \) decreases in monotone,

\[
\text{score}_{UB}^{n_i}(d_{l+1}) \geq \text{score}_{UB}^{n_i+1}(d_{l+1}).
\]

Hence,

\[
\text{score}_{UB}^{n_i+1}(d_{l+1}) \geq \ldots \geq \text{score}_{UB}^{n_i+1+m^2}(d_{l+1}^m) + \text{score}_{UB}^{n_i+1+m^2}(d_{l+1}^m) + \ldots + \text{score}_{UB}^{n_i+1+m^2}(d_{l+1}^m).
\]

We will show that, by doing DoOptimizedRandomAccess() to \( d_{l+1}^m \), the final score \( \text{score}(d) \) is determined for every document \( d \) seen by BMCA that satisfies \( \text{score}_{UB}(d) \geq \tau_{\mathcal{A}} \).

To show this we will use proof by contradiction. Assume that, after doing DoOptimizedRandomAccess() to \( d_{l+1}^m \), there was another document \( d' \) that satisfies \( \text{score}_{UB}(d') \geq \tau_{\mathcal{A}} \) and for that the final score \( \text{score}(d') \) is not determined.

This means that there are more than \((a + 1)m^2\) documents that satisfy \( \text{score}_{UB}(d) \geq \tau_{\mathcal{A}} \) after \( n-1 \) iterations of DoOptimizedSortedAccess(), which contradicts Lemma 3. Therefore, after doing DoOptimizedRandomAccess() to \( d_{l+1}^m \), the final score \( \text{score}(d) \) is determined for every document \( d \) seen by BMCA that satisfies \( \text{score}_{UB}(d) \geq \tau_{\mathcal{A}} \), as desired.

Proposition 1: BMCA halts on \( D \in \mathcal{D} \) in at most \((a + 1)m(1 + m^3h)\) iterations of DoOptimizedSortedAccess() combined with DoOptimizedRandomAccess().

Proof. We will show that, if BMCA iterates DoOptimizedSortedAccess() \( n - 1 + (a + 1)m^3h \) times, then the termination conditions are satisfied. Turning our attention to the final top-k documents \( d \) returned by \( \mathcal{A} \),

\[
\text{score}_{UB}(d) \geq \text{score}(d) \geq \tau_{\mathcal{A}}.
\]

So, from Lemma 2, such documents have been seen by BMCA after \( n-1 \) iterations of DoOptimizedSortedAccess(), which means that there are at least \( k \) documents in \( T_k \) after \( n-1 + (a + 1)m^3h \) iterations of DoOptimizedSortedAccess(). On the other hand, from Lemma 4, the final score \( \text{score}(d) \) is determined for each document \( d \) seen by BMCA that satisfies \( \text{score}_{UB}(d) \geq \tau_{\mathcal{A}} \) after \( n-1 + (a + 1)m^3h \) iterations of DoOptimizedSortedAccess(), which means that documents outside \( T_k \) are no longer viable. Therefore, the termination conditions are satisfied, as desired.

Appendix C: Retrieval Effectiveness of Query Expansion

We show in Table A-1 the experimental evaluation on the retrieval effectiveness of query expansion done using the Indri search engine.
### Table A-1 Retrieval effectiveness evaluation for title and expanded queries on TREC GOV2 data set.

<table>
<thead>
<tr>
<th>TREC 2004</th>
<th>map</th>
<th>bpref</th>
<th>P5</th>
<th>P10</th>
</tr>
</thead>
<tbody>
<tr>
<td>title query</td>
<td>0.28</td>
<td>0.35</td>
<td>0.52</td>
<td>0.51</td>
</tr>
<tr>
<td>expanded query (8-term)</td>
<td>0.28</td>
<td>0.35</td>
<td>0.53</td>
<td>0.50</td>
</tr>
<tr>
<td>expanded query (16-term)</td>
<td>0.28</td>
<td>0.35</td>
<td>0.52</td>
<td>0.49</td>
</tr>
<tr>
<td>expanded query (32-term)</td>
<td>0.29</td>
<td>0.36</td>
<td>0.49</td>
<td>0.51</td>
</tr>
<tr>
<td>expanded query (64-term)</td>
<td>0.29</td>
<td>0.37</td>
<td>0.51</td>
<td>0.51</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TREC 2005</th>
<th>map</th>
<th>bpref</th>
<th>P5</th>
<th>P10</th>
</tr>
</thead>
<tbody>
<tr>
<td>title query</td>
<td>0.34</td>
<td>0.39</td>
<td>0.60</td>
<td>0.59</td>
</tr>
<tr>
<td>expanded query (8-term)</td>
<td>0.35</td>
<td>0.40</td>
<td>0.63</td>
<td>0.62</td>
</tr>
<tr>
<td>expanded query (16-term)</td>
<td>0.36</td>
<td>0.41</td>
<td>0.67</td>
<td>0.63</td>
</tr>
<tr>
<td>expanded query (32-term)</td>
<td>0.37</td>
<td>0.42</td>
<td>0.68</td>
<td>0.63</td>
</tr>
<tr>
<td>expanded query (64-term)</td>
<td>0.38</td>
<td>0.42</td>
<td>0.66</td>
<td>0.63</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TREC 2006</th>
<th>map</th>
<th>bpref</th>
<th>P5</th>
<th>P10</th>
</tr>
</thead>
<tbody>
<tr>
<td>title query</td>
<td>0.31</td>
<td>0.37</td>
<td>0.56</td>
<td>0.51</td>
</tr>
<tr>
<td>expanded query (8-term)</td>
<td>0.29</td>
<td>0.36</td>
<td>0.49</td>
<td>0.49</td>
</tr>
<tr>
<td>expanded query (16-term)</td>
<td>0.30</td>
<td>0.37</td>
<td>0.48</td>
<td>0.47</td>
</tr>
<tr>
<td>expanded query (32-term)</td>
<td>0.32</td>
<td>0.38</td>
<td>0.50</td>
<td>0.50</td>
</tr>
<tr>
<td>expanded query (64-term)</td>
<td>0.32</td>
<td>0.38</td>
<td>0.52</td>
<td>0.51</td>
</tr>
</tbody>
</table>

---

**Etsuro Fujita** received the B.S. from Waseda University in 1994 and the M.S. from Kyoto University in 1996. He is an engineer of NTT Open Source Software Center and currently a PhD student in the School of Multidisciplinary Sciences, the Graduate University for Advanced Studies (SOKENDAI), supervised by Prof. Keizo Oyama. His research interests are database systems, informatation integration, and information retrieval. He is a member of IPSJ.

**Keizo Oyama** received the B.E., M.E. and Dr.Eng. from the University of Tokyo in 1980, 1982 and 1985, respectively. He is a professor of National Institute of Informatics (NII), Research Organization of Information and Systems, and the School of Multidisciplinary Sciences, the Graduate University for Advanced Studies (SOKENDAI). His research interests are web information access, information retrieval systems, and full-text search technologies. He is a member of IPSJ, JSIMS and DBSJ.