KEMB: A Keyword-Based XML Message Broker

Guoliang Li, Jianhua Feng, *Member*, *IEEE*, Jianyong Wang, *Senior Member*, *IEEE*, and Lizhu Zhou

Abstract—This paper studies the problem of XML message brokering with user subscribed profiles of keyword queries and presents a KEyword-based XML Message Broker (KEMB) to address this problem. In contrast to traditional-path-expressions-based XML message brokers, KEMB stores a large number of user profiles, in the form of keyword queries, which capture the data requirement of users/applications, as opposed to path expressions, such as XPath/XQuery expressions. KEMB brings new challenges: 1) how to effectively identify relevant answers of keyword queries in XML data streams; and 2) how to efficiently answer large numbers of concurrent keyword queries. We adopt *compact lowest common ancestors* (CLCAs) to effectively identify relevant answers. We devise an automaton-based method to process large numbers of queries and devise an effective optimization strategy to enhance performance and scalability. We have implemented and evaluated KEMB on various data sets. The experimental results show that KEMB achieves high performance and scales very well.

Index Terms—Keyword search, XML data stream, XML message brokers, compact lowest common ancestor (CLCA).

1 Introduction

Extensible Markup Language (XML) is a standard for data exchange on the Internet. Steam-based XML query processing is an emerging query paradigm, where XML data continuously arrives from different sources, and XML queries are evaluated each time when a new document is received. An important feature of stream-based processing is to efficiently process data as it arrives. The XML message broker is an application of stream-based XML query processing where messages need to be filtered and transformed on-the-fly. In contrast to publisher-subscriber systems, XML message brokers select a relevant part of the XML document, instead of the whole matched XML document.

XML message brokers have attracted great interest from academic and industrial communities [2], [4] since they have a lot of applications on the web and have become mediators between the applications and users. An important application is personalized content delivery, where users register with the broker by providing their interests, the applications send messages to the broker, the broker filters these messages based on the user interests, and sends the personalized relevant data to each user.

The existing XML message brokers [2], [4], [6], [9], [13], [16], [24], [26], [38] usually use path expressions, such as XPath/XQuery expressions, to specify user interests. However, users are usually interested in the content of XML streams instead of the structures. Moreover, most of internet users cannot write valid path expressions since 1) XPath/XQuery is very complicated and hard to comprehend, and 2) XPath/XQuery depends on the underlying, sometimes complex, schemas which are usually unknown for internet users. Fortunately, keyword search has been proposed as an alternative means of querying XML documents [15], [28], [41], which is simple and yet familiar to most internet users as it only requires the input of some keywords.

In this paper, we study the problem of keyword-based XML message brokering, where message brokers store a large number of user profiles, in the form of XML keyword queries that capture the data requirement of users/applications. We propose a KEyword-based XML Message Broker (called KEMB) to effectively identify users' interested data, by addressing the recent trends of seamlessly integrating databases and information retrieval. KEMB uses the concept of *compact lowest common ancestors* (CLCAs) to effectively identify relevant answers of keyword queries over XML data streams. KEMB adopts an automaton-based method to facilitate the processing of large numbers of keyword queries and employs an optimization technique by effectively indexing the queries to enhance the performance and scalability. To the best of our knowledge, this is the first attempt to address the problem of keyword-based XML message brokering. To summarize, we make the following contributions:

- We propose KEMB to address the problem of keyword-based XML message brokering with user subscribed profiles of keyword queries. We adopt the concept of CLCAs to effectively identify relevant answers of keyword queries over XML data streams.
- We devise an automaton-based method to process large numbers of keyword queries. We propose maximal coverage rules and coverage graph to accelerate the processing of large numbers of concurrent keyword queries by effectively indexing the keyword queries.
- We have conducted an extensive experimental study and the experimental results show that KEMB achieves high performance and scales very well.

The remainder of this paper is organized as follows: We survey related works in Section 2. We present how to effectively answer keyword search in XML data streams in

• *The authors are with the Tsinghua National Laboratory for Information Science and Technology, Department of Computer Science and Technology, Tsinghua University, Room 10-204, East Main Building, Beijing 100084, China. E-mail: {liguoliang, fengjh, jianyong, dcszlz}@tsinghua.edu.cn.*

Manuscript received 15 Apr. 2008; revised 14 Jan. 2009; accepted 20 Jan. 2010; published online 2 Sept. 2010.
Recommended for acceptance by M. Garofalakis.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-2008-04-0201.
Digital Object Identifier no. 10.1109/TKDE.2010.159.

Section 3 and propose an automation-based method to process large numbers of concurrent keyword queries in Section 4. We conduct extensive experimental studies in Section 5. Finally, Section 6 concludes the paper.

2 Related Work

2.1 XML Message Filtering/Brokering

In this section, we focus our discussion on the work that is most closely related to our approach XML message brokering. The alternative approaches can be broadly classified into two categories: the automaton-based methods and the index-based approaches.

Automaton-based approaches [2], [9], [13], [16] build an automaton based on the XPath/XQuery expressions processed by the system. The transitions in the automaton are triggered by the tags of the XML document being processed. XFilter [2] treats each XPath as a finite state machine (FSM). This approach is not able to fully handle overlap, especially prefix overlap, between expressions. To address this problem, YFilter [9] extends XFilter by building a nondeterministic finite automaton (NFA) for all XPath expressions in the system. Another automaton-based approach, XPush [16], lazily constructs a single deterministic pushdown automaton for XPath expressions.

The index-based algorithms [4], [6], [24], [26], [38] take advantage of precomputed schemes on either XML documents or XPath expressions. XTrie [6] proposes a trie-based index structure, which decomposes the XPath expressions to substrings that only contain parent-child operators to share common substrings among queries. Index-Filter [4] addresses the problem of obtaining all matches for each expression stored in the system. It answers multiple XML path queries by building indexes over the XML document elements to avoid processing large portions of the input document. Filtering by Sequencing Twigs (FiST) [24] is the first-sequence-based XML document filtering system. FiST encodes XML documents and twig patterns into Prüfer sequences and holistically matches twig patterns with coming documents. A branching-sequencing-based XML message broker is proposed [38] to match twig patterns holistically.

In addition, Lakshmanan and Parthasarathy [26] proposed an index structure that manages XPath expressions for solving the filtering problem. Kwon et al. [25] devised a method to take advantage of overlaps in different XPath expressions by using a novel encoding scheme. Fabret et al. [11] provided an implementation that uses a relational database as the matching engine to address the XML message brokering problem. Candan et al. [5] presented AFilter to leverage both prefix and suffix commonalities over filter statements for reducing the overall filtering time. Chan and Ni [7] proposed to optimize the performance of content-based dissemination of XML data by piggybacking useful annotations to the document being forwarded so that a downstream router can leverage the processing done by its upstream router to reduce its own overhead.

However, the path-expressions-based XML message brokers are powerful but unfriendly for internet users. First, users are usually interested in the contents of XML documents instead of the structures. Second, the path expressions are hard to comprehend for nondatabase users. For example, XPath/XQuery expressions are fairly complicated to grasp. Finally, the path expressions require the underlying, sometimes complex, database schemas, which are usually unaware for internet users. Fortunately, keyword search [8], [15], [34] provides an alternative means for querying XML data.

2.2 Keyword Search in XML Data

Keyword search is a proven and popular mechanism for querying in document systems and World Wide Web and has been extensively applied to extract useful and relevant information from the Internet. The database research community has recently recognized the benefits of keyword search and has been introducing keyword search capability into relational databases [1], [3], [10], [14], [18], [20], [33], [36], [37], [39], XML databases [8], [15], [19], [21], [29], [32], [34], [40], [41], [42], [35], graphs [17], [22], [31], and heterogenous data sources [27], [30]. Markowetz et al. [37] process keyword queries on relational streams, and our method is orthogonal to [37]. First, [37] focuses on relational data streams and we emphasize on XML data streams. Second, [37] considers join conditions and generates operate trees to find relevant answers. We use stack-based algorithms to identify compact trees. Third, [37] uses schema information to generate operator trees while we find answers based on XML data without using schema information.

The research most related to our work is the computation of lowest common ancestor (LCAs) to answer keyword queries in XML data, which has been extensively studied in [8]. As an extension of LCA, XRank [15], meaningful LCA (MLCA) [32], smallest LCA (SLCA) [41], grouped distance minimum connecting tree (GDMCT) [19], valuable LCA (VLCA) [29], multiway-SLCA (MSLCA) [40], RACE [28], and XSeek [34] have been proposed to answer keyword queries in XML data.

There are two baseline approaches for determining query results adopted in the existing works. One is to return the subtrees rooted at LCAs (or its variants) [15], [41], named as *subtree return*. The other one is to return the paths in the XML tree from each LCA to its descendants that match an input keyword [19], [21], namely *path return*. XSeek [34] generates return nodes, which can be explicitly inferred from keywords or dynamically constructed according to the entities in the data that are relevant to the search. XSeek is not easy to adapt to XML data streams as it highly depends on the underlying XML schemas, and thus, it is rather hard to infer entities from XML data streams. Liu and Chen [35] proposed a new semantics by considering monotonicity and consistency to reason and identify relevant matches.

Generally, the existing XML keyword search methods need first index XML elements (as inverted lists), and then, answer keyword queries based on the indices. They are hard to adapt to keyword search over XML data streams. Inspired by path-expressions-based XML message brokers and keyword search, we study the problem of keyword-based XML message brokering in this paper.

3 Keyword Search in XML Streams

3.1 Notations

For ease of presentation, we briefly outline the XML data model and introduce some notations in this section. An

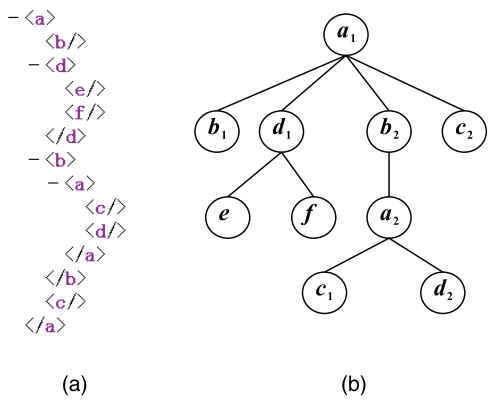

Fig. 1. (a) An XML Document and (b) the tree model.

XML document can be modeled as a rooted, ordered, and labeled tree. Nodes in this rooted tree correspond to elements in the XML document. For any two nodes u and v, $u \prec v$ ($u \succ v$) denotes that node u is an ancestor (descendant) of node v; $u \preceq v$ denotes that $u \prec v$ or $u = v$. We call node u that directly contains input keywords a *content node*. Node u is called to (directly or indirectly) contain keyword k if u directly contains k or u has a descendant v that directly contains k. Fig. 1b gives the tree model of the XML document in Fig. 1a, where b_1 denotes the first element of tag b. To support metadata search, the tag names (e.g., "b" in Fig. 1) are also taken as keywords.

3.2 Query Semantics

We first briefly review the concepts of LCA [8] and SLCA [41] and then introduce our methodology CLCA.

Definition 1 (LCA). *Given m nodes, $n_1, n_2, \ldots, n_m$, `ca` is a common ancestor (CA) of these m nodes, if `ca` is an ancestor of each n_i for $1 \leq i \leq m$. `lca` is the LCA of these m nodes, denoted as* $\texttt{lca} = \mathrm{LCA}(n_1, n_2, \ldots, n_m)$*, if `lca` is a CA of these m nodes and $\nexists u, u \succ$ `lca`, which is also a CA of these m nodes.*

Existing methods usually compute the LCAs of content nodes to answer keyword queries in XML documents. However, it is inefficient to compute all the LCAs as there may be large numbers of content nodes. To address this problem, SLCA [41] is proposed to improve search efficiency. The basic idea behind SLCA is that, if node v contains all input keywords, its ancestors will be less meaningful than v. Hence, SLCA introduces the concept of *smallest tree*, which is a tree that contains all keywords but contains no subtrees that also contain all keywords. However, SLCA wrongly prunes some LCAs, which have descendants that are also LCAs, and causes a serious problem—negative, which is not an *adhoc* problem but ubiquitous in the XML documents with nested structures. For example, consider query $\{a, c, d\}$ and the XML document in Fig. 1, the LCAs are a_1 and a_2, but a_1 is a false negative for SLCA as a_1 has a LCA descendant a_2. However, a_1 should contribute to an answer as $\{a_1, d_1, c_2\}$ contains the three input keywords. To address this problem and effectively answer queries, we adopt the concept of CLCA [12], [28].

Definition 2 (CLCA). *Given a query $\mathcal{K} = \{k_1, k_2, \ldots, k_m\}$, and suppose v_i is a content node w.r.t. k_i for $1 \leq i \leq m$. $w = \mathrm{LCA}(v_1, v_2, \ldots, v_m)$ is said to dominate v_i w.r.t. $\mathcal{K}$, if, $\forall v'_j$, a content node w.r.t. $k_j (j \neq i)$, $w \succeq \mathrm{LCA}(v'_1, \ldots, v'_{i-1}, v_i, v'_{i+1}, \ldots, v'_m)$. w is a CLCA w.r.t. $\mathcal{K}$, if w dominates each v_i for $1 \leq i \leq m$.*

A CLCA is the LCA of some relevant nodes, and the irrelevant nodes cannot share a CLCA. For instance, recall query $\{a, c, d\}$ although a_1 is the LCA of a_1, c_1, and d_1, it is not their CLCA as c_1 is dominated by a_2 while d_1 is dominated by a_1. It is easy to figure out that a_1 is the CLCA of a_1, d_1, and c_2; and a_2 is the CLCA of a_2, c_1, and d_2. Note that SLCA wrongly prunes some LCAs which have LCA descendants and CLCA can avoid such false negatives introduced by SLCA. Moreover, the false negative problem is not an *adhoc* problem but ubiquitous over the XML documents with nested structures. For example, recall query $\{a, c, d\}$, the CLCAs are a_1 and a_2, but a_1 is not a valid SLCA as it has a LCA descendant a_2.

In terms of "AND" semantics, CLCA is the same as exclusive LCA (ELCA) [42] and XRank [15]. In terms of "OR" semantics, they are different and CLCA captures more compact structures. For example, in Fig. 1, suppose node e has two children b_3 and c_3 with keywords b, c, respectively. Consider "OR" semantics, node e is not an ELCA for query $\{a, b, c, d\}$, but it is a CLCA. We have compared different semantics [12]. Interested readers are referred to [12] for more details. In this paper, we use CLCA semantics and identify *path returns* (the subtrees rooted at CLCAs and containing the paths from each CLCA to its descendants that are dominated by the CLCA) as answers. For example, the *path returns* of query $\{a, c, d\}$ are $<a_1><d_1/><c_2/></a_1>$ and $<a_2><c_1/><d_2/></a_2>$. We first consider "AND" semantics, and then, extend to support "OR" semantics.

3.3 Keyword Search in XML Streams

In this section, we propose KALE to effectively process keyword search over XML streams. Different from KEMB, KALE answers one keyword query at a time, as opposed to simultaneously answering large numbers of concurrent keyword queries.

We adopt a stack-based method for event-driven processing. Arriving XML documents are parsed with an event-based SAXParser. The events raised during parsing are used to drive the execution; in particular, 1) "start-of-document" events drive KALE to initialize the stack, 2) "start-of-element" events cause KALE to push elements, and 3) "end-of-element" events drive KALE to pop elements. During elements processing, if the element in the stack contains all input keywords, it is a CLCA and we identify the answer. If the element contains a part of input keywords, KALE takes the subtree rooted at it as a subtree of its parent, i.e., the element directly below it in the stack; otherwise, we discard such element. We present Lemma 1 to identify CLCAs and construct the answers rooted at CLCAs.

Lemma 1. *Given a keyword query in KALE, an element e in the stack is a CLCA, if during execution, e contains all input*

Algorithm 1: KALE Algorithm

```
Input: K={k1,k2,...km} and an XML stream D
Output: {R_K|R_K is a set of path returns rooted at CLCAs}
1  begin
2    SD=SAXParser(D);
3    while e∈SD (in document order) do
4      switch event type of e do
5        case Start-of-Document
6          S = new Stack(); /*Initialize Stack S*/
7        case Start-of-Element
8          S.Push(e);
9        case End-of-Element
10         EoE();
11 end
```

Procedure EoE Procedure

```
1  begin
2    e=S.Pop();
3    if e contains all the keywords in K then
4      R_e = the subtree associated with e;
5      OutputResult(R_e);
6    else if e contains a part of keywords in K then
7      r=S.top().subtree;
8      r=r→e; /* take the subtree rooted at e as a subtree of r */
9      S.top().subtree = r;
10 end
```

Fig. 2. KALE algorithm.

keywords. The subtree associated with e must be a path return which matches the keyword query.

Proof. Given any element in the runtime stack, e, and for any children of e, d, if d contains all the keywords, the descendants of d must have been popped out from the runtime stack. As e contains all keywords, and any children of e cannot dominate the keywords denoted by e, e must be a CLCA. Hence, the subtree associated with e must be a *path return*. □

Based on above observations, we devise a stack-based algorithm in an event-driven fashion (Fig. 2). KALE first parses the XML documents using SAXParser (line 2), and then, for each incoming element e, if e is a type of "start-of-document," KALE initializes the stack (line 6); if e is a type of "start-of-element," KALE pushes e into the stack (line 8); if e is a type of "end-of-element," KALE calls its subroutine EoE to check whether this element is a CLCA (line 10). EoE pops the topmost element e from the stack (line 2). If e contains all keywords, e must be a CLCA and the subtree associated with it must be a *path return* (Lemma 1). EoE outputs the result (lines 3-5). If e contains some input keywords, the ancestors of e may constitute an answer with subsequent elements. EoE takes the subtree associated with e as a subtree of its parent (lines 6-9). It is easy to figure out that the time complexity of the algorithm is $O(D * N)$, where D is the depth of the XML document and N is the number of elements in the XML document. To further go into our algorithm, we walk through our algorithm with a running example.

Example 1. Consider the XML document in Fig. 1 and a keyword query $\mathcal{K} = \{a, c, d\}$. When elements $<a_1>$ and $<b_1>$ arrive, KALE pushes them into the stack. Upon the coming of $</b_1>$, as b contains no keyword, KALE pops it from the stack. Upon the coming of $</d_1>$, KALE pops d from the stack. As d is an input keyword, KALE takes the subtree rooted d as a subtree of its parent a_1. When $</a_2>$ arrives, KALE pops a_2 from the stack. As the subtree associated with a_2 contains all keywords, a_2 must be a CLCA and the subtree associated with a_2 (as circled by the rectangle) must be a *path return* (Lemma 1). We can proceed to walk through our algorithm and get two results (Fig. 3).

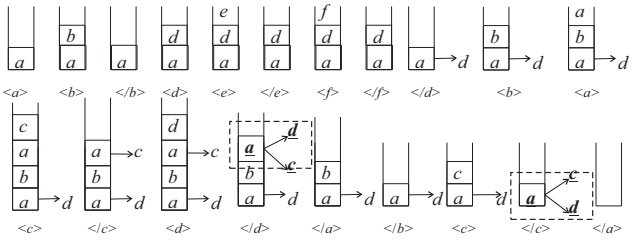

Fig. 3. A running example on query $\{a, c, d\}$.

4 Keyword-Based XML Message Broker

4.1 Problem Statement

XML message brokers take user profiles of keyword queries and XML data streams as input, filter XML subtrees that contain input keywords and satisfy search semantics (such as CLCA), and finally, deliver the relevant answers to corresponding users. As XML message brokers should provide fast, on-the-fly matching of XML subtrees to large numbers of user profiles, we propose a Keyword-based XML Message Broker (KEMB) to address this issue. A formal description of keyword-based XML message brokers is defined as follows:

Definition 3 (Keyword-based XML Message Brokers). *Given 1) a set $\mathcal{Q} = \{\mathcal{Q}_1, \mathcal{Q}_2, \ldots, \mathcal{Q}_n\}$ of keyword queries and 2) a stream of XML documents, compute, for each document $\mathcal{D}$, the complete set of* path returns *corresponding to the subtrees of $\mathcal{D}$, which match $\mathcal{Q}_i$ for $1 \leq i \leq n$.*

Different from XML document filtering, KEMB should identify all *path returns* as query results, which satisfy the CLCA semantics, as opposed to the whole XML document that matches user profiles. In contrast to KALE, KEMB should provide the capability of effectively processing large numbers of concurrent keyword queries.

Note that, during the KEMB execution to answer multiple keyword queries, although an element e in the runtime stack is a CLCA for a query, KEMB cannot discard the elements in the subtree rooted at e as they may answer other queries with subsequent elements; similarly, even if e contains all input keywords for query $\mathcal{Q}_i$, e may not be a CLCA of $\mathcal{Q}_i$, as one of e's children may be the CLCA if the child contains all input keywords of $\mathcal{Q}_i$.

For example, consider the four queries in Table 1 and the XML document in Fig. 1. When $</a_2>$ arrives although a_2 contains all the input keywords of $\mathcal{Q}_3$ and is a CLCA, its descendant d_2 cannot be discarded as it may answer $\mathcal{Q}_2$ with subsequent elements b_2 and a_2. Upon the coming of

TABLE 1
User Subscribed Profiles, $\mathcal{Q}$

Query ID	Keyword Queries
Q_1	$a\ b\ c$
Q_2	$a\ b\ d$
Q_3	$a\ d$
Q_4	$e\ f$

$</b_2>$ although b_2 contains "a, d," b_2 is not a CLCA of $\mathcal{Q}_3$ as a_2 is the CLCA.

Accordingly, it is not straightforward to process large numbers of concurrent keyword queries and KEMB raises new challenges: 1) how to efficiently identify CLCAs and *path returns* for large numbers of keyword queries; 2) how to take full advantage of the overlaps of large numbers of concurrent keyword queries; and 3) how to discard the *outdated elements in an eager manner*. That is, once finding some *outdated elements*, which will not answer any query with subsequent elements, throwing them away from the runtime stack immediately. We propose Lemma 2 to check whether an element is a CLCA to address the first challenge. We will discuss how to address the other two challenges in Sections 4.2 and 4.3, respectively.

Lemma 2. *Given an element in an XML document $\mathcal{D}$, e, and any keyword query $\mathcal{Q}_i$ in $\mathcal{Q}$. e is a CLCA w.r.t. $\mathcal{Q}_i$, if e contains all the keywords in $\mathcal{Q}_i$ after removing e's children, which also contain all the keywords in $\mathcal{Q}_i$.*

Proof. Given an element in the runtime stack, e, if e contains all keywords in $\mathcal{Q}_i$ after removing e's children that also contain all such keywords, any children of e cannot dominate those keywords denoted by e. Thus, e must be a CLCA. □

Example 2. Consider the four queries in Table 1 and the XML document in Fig. 1. For query $\mathcal{Q}_2$, as a_1 contains all input keywords of $\mathcal{Q}_2$ after removing its child b_2, which also contains all the keywords, a_1 is a CLCA as formalized in Lemma 2.

4.2 Query Indexing

To effectively index keyword queries by sharing their overlaps, in this section, we present an effective mechanism to process large numbers of keyword queries.

Automaton-based approaches [2], [9], [13], [16] have been proposed to utilize the overlaps of path queries. We also adopt nondeterministic finite automaton to process multiple keyword queries. However, KEMB is different from the existing path-expressions-based methods as 1) KEMB should effectively and efficiently identify *path returns* rooted at CLCAs as the answer; and 2) KEMB is more difficult than path-expressions-based XML message brokers as there is no constraint between input keywords in KEMB while XPath expressions are labeled and ordered, and must follow XPath-constraints.

KEMB uses an NFA-based approach to identify commonalities among keyword queries and share the processing among them. In our approach, rather than representing each path query as a FSM individually, KEMB combines all queries into a single FSM in the form of a NFA. The NFA has two key features: (1) there is one accepting state for each keyword query; and (2) the common subqueries of keyword queries are represented only once.

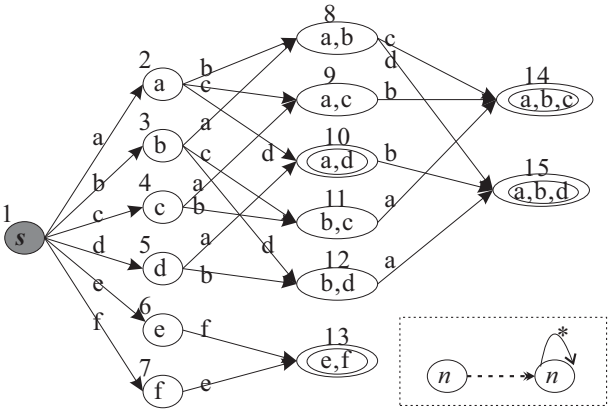

Fig. 4. An NFA on top of queries in Table 1.

Fig. 4 shows an example of such an NFA representing the four queries in Table 1. The NFA represents multiple keyword queries. Each query in the NFA has only a single accepting state and the NFA contains multiple accepting states. Note that the common subqueries are shared. A circle denotes a state and each state is assigned with a unique number (SID), the shaded circle represents the initial state, and two concentric circles denote an accepting state and such states are also marked with the queries they represent [9]. A directed edge represents a transition, and the symbol on an edge represents the input that triggers the transition [9]. The output function of the NFA is a mapping from the set of accepting states to a partitioning of the queries in the system, where each partition contains the queries that share the accepting state [9]. Once arriving at an accepting state, we identify *path returns* and deliver them to corresponding users as stated in Lemma 2.

Having presented the basic NFA model used by KEMB, we describe an incremental process for NFA construction and maintenance. Given a keyword query $\mathcal{Q}_i$, we construct an FSM as follows: Each of its subsets represents a state. The empty set represents the initial state and $\mathcal{Q}_i$ itself denotes an accepting state. Given two subsets of $\mathcal{Q}_i$, $\mathcal{S}_1$ and $\mathcal{S}_2$, if $\mathcal{S}_1 \subset \mathcal{S}_2$ and $|\mathcal{S}_1| = |\mathcal{S}_2| - 1$, there is a transition from $\mathcal{S}_1$ to $\mathcal{S}_2$ with the trigger of keyword k, where $k \notin \mathcal{S}_1$ and $k \in \mathcal{S}_2$, i.e., $\{k\} = \mathcal{S}_2 - \mathcal{S}_1$. Given multiple keyword queries, we first create their corresponding FSMs individually, and then, combine them by sharing the common states to construct the NFA. Note that each state in the combined NFA has a self-loop (which denotes that it can transit to itself) with a trigger of "$*$" as shown in Fig. 4, which will be explained in detail later. It is easy to figure out that NFA supports update, insertion, and deletion of keyword queries well. For deletion of a query, we first locate the accepting state for the query. Then, we remove the previous states of the accepting state iteratively. For each of the previous states, if it is not an accepting state and has only one transition, we remove it from the NFA and process its previous states similarly; otherwise, we keep it and terminate.

Having described the logical construction of the NFA, we present how to implement the NFA. For efficient execution, the NFA is implemented using a hash table-based approach, which has been shown to have low time

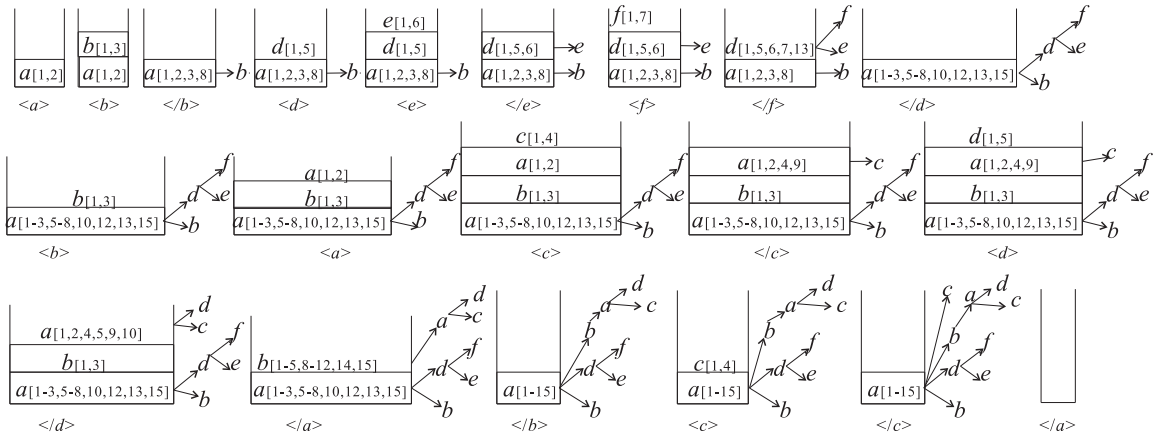

Fig. 5. A running example of KEMB on queries in Table 1.

complexity for inserting/deleting states, inserting/deleting transitions, and performing transitions [9]. In our approach, a data structure is created for each state, including: 1) the *SID* of the state; and 2) a small hash table that contains all legal transitions from the state. The transition hash table for each state contains [*keyword*, *SID*] pairs where *keyword* (the key) indicates the trigger of the outgoing transition and *SID* identifies the child state that the transition can lead to.

KEMB executes the NFA in an event-driven fashion; as an arriving document is parsed, the events raised by the parser callback the handlers and drive the transitions in the NFA. KEMB employs a stack-based mechanism to execute the event-based NFA. In contrast to KALE, KEMB maintains "active states" associated with an element in the runtime stack, which may contribute to answers with subsequent elements. Different from YFilter, KEMB avoids backtracking to process branches and predicates. KEMB employs the following handlers:

1. Start Document Handler: When an XML document arrives to be parsed, the NFA execution begins at the initial state.
2. Start Element Handler: When a new element is read from the stream, the NFA execution transits the initial state to "active states" with the trigger of keywords directly contained in the element. KEMB pushes the element and "active states" into the runtime stack. Note that the initial state is also added to the set of "active states" as there is a self-loop with trigger of "$*$." This is because some subsequent elements may trigger the initial state to other states and thus may generate potential query results. For example, in Fig. 5, when $<b_1>$ arrives, KEMB pushes b_1 associated with two active states 1 and 3 into the stack. As some subsequent elements, such as e and f, may transit the initial state to states 6, 7, 13 and get an answer of query $\mathcal{Q}_4$.
3. End Element Handler: When an "end-of-element" is encountered, KEMB checks whether there are some accepting states associated with the topmost element e in the stack. For each accepting state (if any), KEMB checks whether e is a CLCA based on Lemma 2. If it is, KEMB constructs *path return* and delivers the result to corresponding users. It is important to note that, unlike a traditional NFA, whose goal is to find one accepting state for an input, the NFA execution must find all matching queries. Thus, even after an accepting state has been reached for a document, the execution must continue until the document has been completely processed. Then, KEMB pops the topmost element e from the stack. If the subtree, denoted as $\mathcal{T}$, associated with e contains input keywords, KEMB takes $\mathcal{T}$ as a subtree of e's parent, p, i.e, the element directly below it in the stack. Subsequently, KEMB updates "active states" associated with p as follows: For each keyword in $\mathcal{T}$, k, and each active state associated with p, α, KEMB checks: 1) k is looked up in α's hash table. If it is present, the corresponding SID is added to a set of "target states," which will be the "active states" for the next element. 2) As there is a self-loop for symbol "$*$," α is also added to the set of "target states," as some subsequent elements may trigger the state to other active states. 3) After all current active states have been checked in this manner, the set of "target states" is pushed onto the top of the runtime stack. They then become "active states" for the next event.

Example 3. Consider the XML document in Fig. 1 and user profiles in Table 1. When $<b_1>$ arrives, KEMB pushes b_1 associated with two active states 1 and 3 into the stack. When $</b_1>$ arrives, KEMB pops b_1 from the stack and takes it as a subtree of its parent a_1. Then, KEMB updates the active states w.r.t. a_1, i.e., {1,2}, by triggering the two states with keyword b. We get two target states 3 and 8 by, respectively, triggering states 1 and 2 with b. We note that each state has a self-loop marked with "$*$" as subsequent elements may trigger the state to some other active states. Thus, we should keep the original states 1 and 2 active. Because other elements, such as d, may transit states 1 and 2 to active states 5 and 10, respectively. Accordingly, we get the target states, $\{1, 2, 3, 8\}$, i.e., "active states" for the next element a. Similarly, we proceed to walk through the running example as illustrated in Fig. 5.

The order of keywords to trigger the active states is nondetermined, that is, the elements with different order

will not result in different "target states." Lemma 3 guarantees the correctness. In this paper, we select keywords in document order to transit "active states."

Lemma 3. *Consider a set of active states $\mathcal{A}$ (including the initial state), $\mathcal{T}_1$ and $\mathcal{T}_2$ are target sets by triggering $\mathcal{A}$ with ordered keywords $\mathcal{L}_1 = ``k_1, \ldots, k_i, \ldots, k_j, \ldots, k_n''$ and $\mathcal{L}_2 = ``k_1, \ldots, k_j, \ldots, k_i, \ldots, k_n,''$ respectively, we have $\mathcal{T}_1 = \mathcal{T}_2$.*

Proof. Suppose the keyword set associated with $\mathcal{A}$ is $\mathcal{K}$, and the keyword sets w.r.t. $\mathcal{L}_1$ and $\mathcal{L}_2$ are $\mathcal{K}_1$ and $\mathcal{K}_2$, respectively. It is easy to figure out $\mathcal{T}_1$ and $\mathcal{T}_2$ are, respectively, the sets of the states for all the subsets of $\mathcal{K} \cup \mathcal{K}_1$ and $\mathcal{K} \cup \mathcal{K}_2$. As $\mathcal{K}_1 = \mathcal{K}_2$, $\mathcal{K} \cup \mathcal{K}_1 = \mathcal{K} \cup \mathcal{K}_2$. Hence, we have $\mathcal{T}_1 = \mathcal{T}_2$. □

For each element, when pushing it into the stack, we need generate its active states; when popping it from the stack, we need compute its parent's active states. Thus, the time complexity of KEMB is $O(D * N * AS)$, where AS is the average number of active states for each element, D is the depth of the XML document, and N is the number of elements in the XML document.

Note that if the number of keywords in a query is too large, for example, larger than 10, KEMB is inefficient and we will not add it into the NFA. Instead, we use KALE to answer such queries, which need not construct an NFA. Thus, for the queries with small lengths, we combine them into an NFA and use KEMB to answer the queries. For the queries with large lengths, we use KALE.

The big difference between KALE and KEMB is that 1) KEMB should maintain "active states" during the NFA execution; and 2) when an "end-of-element," e, arrives, KEMB checks whether there are some accepting states associated with e. If so, KEMB identifies CLCAs, constructs *path returns*, and delivers the results to corresponding users. However, even if e is a CLCA, KEMB cannot discard its descendants as they may answer other queries with subsequent elements. Recall Example 3, consider that $</a_2>$ arrives although a_2 is a CLCA w.r.t. $\mathcal{Q}_3$, we cannot discard d_2 as it will answer $\mathcal{Q}_2$ with subsequent elements a_2 and b_2. In addition, consider that $</b_2>$ arrives although b_2 contains a and d, it is not a CLCA of $\mathcal{Q}_2$ as its child a_2 is the CLCA.

KEMB is suboptimal as during NFA execution it cannot discard *outdated elements*, which will not answer any query with subsequent elements, in an *eager manner*. That is, once finding outdated elements, KEMB does not need to maintain useless elements and should throw them away from the stack. For example, recall Example 3, when $</d_1>$ arrives, as d_1 is a CLCA w.r.t. $\mathcal{Q}_4$, we discard e and f as they will not answer any query with subsequent elements. Hence, how to discard outdated elements in an eager manner is a challenge. To address this issue, we introduce an effective strategy in Section 4.3.

4.3 Maximal Coverage Rule

In this section, we devise a novel strategy to identify and discard outdated elements and corresponding states in an eager manner. We note that this strategy can facilitate the processing of large numbers of keyword queries, as KEMB need not maintain the outdated elements and transit the useless states in the runtime stack. We will experimentally prove that this strategy can improve search performance in Section 5.

For ease of presentation, we begin by introducing some notations. Given a set $\mathcal{Q} = \{\mathcal{Q}_1, \mathcal{Q}_2, \ldots, \mathcal{Q}_n\}$ of keyword queries, an XML data stream $\mathcal{D}$ and an element in $\mathcal{D}$, e, let $\Sigma = \cup_{i=1}^{n} \mathcal{Q}_i$ denote the input keyword set and $\mathcal{CK}(e)$ denote the set of keywords which are contained in the subtree rooted at e. $\forall k \in \Sigma$, $\mathcal{S}_k = \{\mathcal{Q}_i | k \in \mathcal{Q}_i\}$, which is the set of keyword queries that contain input keyword k. $\mathcal{C}_k = \cup_{\mathcal{Q}_i \in \mathcal{S}_k} \mathcal{Q}_i$, which denotes the set of input keywords that are contained in a same keyword query with input keyword k. $\mathcal{C}_k$ is called a coverage for k. $\mathcal{K}_k = \{l | l \in \Sigma \text{ and } \mathcal{C}_l = \mathcal{C}_k\}$, which is a set of keywords with the same coverage. For example, considering the four keyword queries in Table 1, we have $\Sigma = \{a, b, c, d, e, f\}$, $\mathcal{S}_d = \{\mathcal{Q}_2, \mathcal{Q}_3\}$, $\mathcal{C}_d = \{a, b, d\}$, $\mathcal{C}_e = \{e, f\}$, $\mathcal{C}_f = \{e, f\}$, and $\mathcal{K}_k = \{e, f\}$. Consider the XML document in Fig. 1, $\mathcal{CK}(d_1) = \{d, e, f\}$. $\mathcal{CK}(a_2) = \{a, c, d\}$.

Based on above notations, we are ready to introduce the concept of *maximal coverage rule* (MCR).

Definition 4 (Maximal Coverage Rule). *Given a set $\mathcal{Q} = \{\mathcal{Q}_1, \mathcal{Q}_2, \ldots, \mathcal{Q}_n\}$ of keyword queries. $\forall k \in \Sigma$, $\mathcal{C}_k \rightarrow k$ is called a coverage rule. $\mathcal{C}_k \Rightarrow \mathcal{K}_k$ is called a maximal coverage rule.*

Lemma 4. *Given two keywords k_i and k_j, if $\mathcal{C}_{k_i} = \mathcal{C}_{k_j}$, we have, 1) $\mathcal{K}_{k_i} = \mathcal{K}_{k_j}$ and 2) $\mathcal{C}_{k_i} \Rightarrow \mathcal{K}_{k_i} \equiv \mathcal{C}_{k_j} \Rightarrow \mathcal{K}_{k_j}$.*

Proof. We first prove 1), and then, 2) is obvious.

We begin by proving that $\forall k \in \mathcal{K}_{k_i}$, $k \in \mathcal{K}_{k_j}$ as follows: If $k \in \mathcal{K}_{k_i}$, we have $\mathcal{C}_k = \mathcal{C}_{k_i}$ based on Definition 4. Thus, $\mathcal{C}_k = \mathcal{C}_{k_j} = \mathcal{C}_{k_i}$. Hence, $k \in \mathcal{K}_{k_j}$ based on Definition 4. Similarly, we can prove that $\forall k \in \mathcal{K}_{k_j}$, $k \in \mathcal{K}_{k_i}$. Therefore, $\mathcal{K}_{k_i} = \mathcal{K}_{k_j}$. □

Note that if $\mathcal{C}_{k_i} = \mathcal{C}_{k_j}$, then $\mathcal{K}_{k_i} = \mathcal{K}_{k_j}$ (Lemma 4). Thus, $\mathcal{C}_{k_i} \Rightarrow \mathcal{K}_{k_i}$ is equivalent to $\mathcal{C}_{k_j} \Rightarrow \mathcal{K}_{k_j}$. We denote $\mathcal{C}_k \Rightarrow \mathcal{K}_k$ as $\mathcal{C} \Rightarrow \mathcal{K}$ if there is no ambiguous in the context.

The novel idea behind MCR is that, given an input keyword k, only the input keywords in $\mathcal{C}_k$ are highly relevant to k. Similarly, given a coverage $\mathcal{C}_k$, only the keywords in $\mathcal{K}_k$ are highly relevant to $\mathcal{C}_k$. Given an MCR $\mathcal{C} \Rightarrow \mathcal{K}$, if element e in the runtime stack contains all keywords in $\mathcal{C}$ ($\mathcal{CK}(e) \supseteq \mathcal{C}$), we can discard e descendant d if $\mathcal{CK}(d) \subseteq \mathcal{K}$, as d will not contribute to any answer with subsequent elements. This is because $\mathcal{CK}(e)$ contains all related keywords of $\mathcal{CK}(d)$ and the answers for d must be in the subtree rooted at e. On the contrary, we cannot discard them as they may contribute to other potential answers. Therefore, we can use such features to identify the outdated elements. For example, recall Example 3, consider that $</d_1>$ arrives, as $\mathcal{CK}(d_1) = \{d, e, f\}$, $\mathcal{CK}(e_1) = \{e\}$, $\mathcal{CK}(f_1) = \{f\}$ and there is an MCR $\{e, f\} \Rightarrow \{e, f\}$, e_1 and f_1 can be discarded as they will not answer any keyword query.

Based on above observations, we propose Lemma 5 to discard the outdated elements in an eager manner. If some elements do not satisfy the conditions of Lemma 5, we must keep them in the runtime stack as they may contribute to other answers.

Lemma 5. *Given an element of an XML data stream, e, KEMB can directly discard e's descendant d in an eager manner, if $\exists\, \mathcal{C} \Rightarrow \mathcal{K}$, 1) $\mathcal{CK}(e) \supseteq \mathcal{C}$ and 2) $\mathcal{CK}(d) \subseteq \mathcal{K}$.*

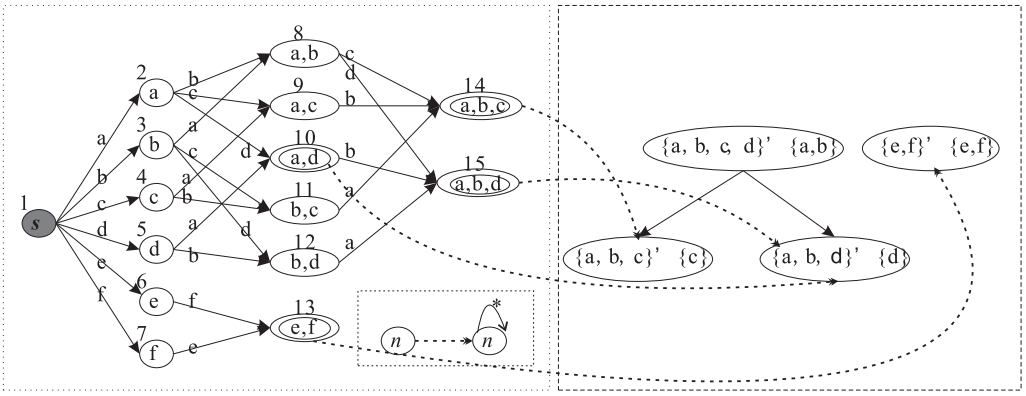

Fig. 6. (a) An NFA and (b) the coverage graph.

Proof. Suppose $k \in \mathcal{K}$, we have $\mathcal{C} = \mathcal{C}_k$. As $\mathcal{S}_k = \{\mathcal{Q}_i | k \in \mathcal{Q}_i\}$ and $\mathcal{C}_k = \cup_{\mathcal{Q}_i \in \mathcal{S}_k} \mathcal{Q}_i$, all the keyword queries which contain k must be in $\mathcal{S}_k$ and $\mathcal{C}_k$ contains all the corresponding keywords. As $\mathcal{CK}(e) \supseteq \mathcal{C}$ and $\mathcal{CK}(d) \subseteq \mathcal{K}$, d cannot answer any keyword query with subsequent elements, as the answers associated with d have been already identified. Therefore, we can discard d safely. □

To effectively generate MCRs, we first construct the coverage rules for every $k \in \Sigma$, and then, merge the coverage rules with the same coverage to generate the MCRs. For example, recall the four queries in Table 1. Consider keyword a, we have $\mathcal{C}_a = \mathcal{Q}_1 \cup \mathcal{Q}_2 \cup \mathcal{Q}_3 = \{a, b, c, d\}$ and $\{a, b, c, d\} \rightarrow a$. Similarly, $\{a, b, c, d\} \rightarrow b$, $\{a, b, c\} \rightarrow c$, $\{a, b, d\} \rightarrow d$, $\{e, f\} \rightarrow e$, and $\{e, f\} \rightarrow f$. We merge the coverage rules with the same coverage to generate MCRs, $\{a, b, c, d\} \Rightarrow \{a, b\}$, $\{a, b, c\} \Rightarrow \{c\}$, $\{a, b, d\} \Rightarrow \{d\}$, and $\{e, f\} \Rightarrow \{e, f\}$. It is easy to figure out that the number of MCRs is no larger than $|\Sigma|$.

We propose an effective algorithm MCRGen to generate MCRs (Fig. 7). MCRGen first initializes a set (denoted as $MCRSet$) of MCRs as ϕ (line 2). Then, for each input keyword, MCRGen constructs its coverage $\mathcal{C}$ (line 5-line 7), if there is a same coverage in $MCRSet$, we update the coverage rule by inserting the keyword into the coverage rule (line 9); otherwise, MCRGen inserts it into $MCRSet$ (line 11). It is easy to figure out that the complexity of the algorithm is $O(|\cup Q_i|^2)$.

4.4 Coverage Graph

During the NFA execution, if finding an accepting state, we can utilize Lemma 5 to discard outdated elements. However, given an element e in the runtime stack, it is still a challenge to efficiently identify the *relevant* MCRs, the coverages of which are contained by $\mathcal{CK}(e)\big(\mathcal{C} \subseteq \mathcal{CK}(e)\big)$. To address this issue, we propose to construct a *coverage graph*.

For ease of presentation, we introduce some notations. $\mathcal{C} \Rightarrow \mathcal{K}$ is called to cover $\mathcal{Q}_i \in \mathcal{Q}$ if $\mathcal{Q}_i \subseteq \mathcal{C}$; $\mathcal{C} \Rightarrow \mathcal{K}$ is called to directly cover $\mathcal{Q}_i$, if $\mathcal{Q}_i \subseteq \mathcal{C}$ and $\nexists \mathcal{C}' \Rightarrow \mathcal{K}'$, $\mathcal{Q}_i \subseteq \mathcal{C}'$ and $\mathcal{C}' \subset \mathcal{C}$. $\mathcal{C} \Rightarrow \mathcal{K}$ is called to be covered by a set of keywords S, if $\mathcal{C} \subseteq S$. Given two MCRs, $\mathcal{R}^u = \mathcal{C}^u \Rightarrow \mathcal{K}^u$ and $\mathcal{R}^v = \mathcal{C}^v \Rightarrow \mathcal{K}^v$, $\mathcal{R}^u$ is called to contain $\mathcal{R}^v$ if $\mathcal{C}^v \subset \mathcal{C}^u$. $\mathcal{R}^u$ is called to directly contain $\mathcal{R}^v$ if $\mathcal{C}^v \subset \mathcal{C}^u$ and $\nexists \mathcal{R}^w = \mathcal{C}^w \Rightarrow \mathcal{K}^w$, $\mathcal{C}^v \subset \mathcal{C}^w$ and $\mathcal{C}^w \subset \mathcal{C}^u$. Based on these notations, we introduce the concept of *coverage graph*.

Definition 5 (Coverage Graph). *The coverage graph is a directed acyclic graph, where vertexes are MCRs. Given two vertexes u and v, there is an arc from u to v, iff, the MCR w.r.t. u, denoted as $\mathcal{R}^u$, directly contains $\mathcal{R}^v$. It is obvious that there is a path from u to v, denoted as $u \rightsquigarrow v$, iff, $\mathcal{R}^u$ contains $\mathcal{R}^v$.*

To efficiently construct the coverage graph, we first sort MCRs according to their sizes in descending order, where the size of an MCR $\mathcal{C} \Rightarrow \mathcal{K}$ refers to the number of keywords in $\mathcal{C}$. We then always select MCR_{max} with maximal size and insert it into the coverage graph as follows: We add a new vertex ω which maintains MCR_{max} into the coverage graph, and identify the vertexes, the MCRs of which directly contain MCR_{max}. For each such vertex v, we add an arc from v to ω. Fig. 6*b* gives the coverage graph on top of queries in Table 1.

Note that we discard outdated elements only if there are some accepting states for an element in the runtime stack. Thus, in the NFA, each accepting state preserves some pointers to the vertexes in the coverage graph, the MCRs of which directly cover the keyword query for the accepting state (the dotted lines in Fig. 6). Such vertexes are called *source vertexes* for the accepting state. Given a source vertex s, we classify MCRs in the coverage graph into three categories:

Algorithm 2: MCRGen Algorithm

```
   Input: Q={Q1,Q2,...,Qn}
   Output: MCRSet={C⇒K}
1  begin
2      MCRSet = φ;
3      for k ∈ Σ=∪_{i=1}^{n} Qi do
4          C = φ;
5          for Qi ∈ Q do
6              if k ∈ Qi then
7                  C ∪=Qi;
8          if ∃C⇒K ∈ MCRSet then
9              MCRSet.update(C ⇒ K, C ⇒ K ∪ {k});
10         else
11             MCRSet ← {C ⇒ {k}};
12 end
```

Fig. 7. MCRGen algorithm.

Algorithm 3: FindRMCR Algorithm

Input: An accepting state α, the NFA and the coverage graph, and an element e.
Output: $RMCRSet$=$\{\mathcal{R}^v|\mathscr{C}^v \subseteq \mathcal{CK}(e)$ and $\mathcal{R}^v$=$\mathscr{C}^v \Rightarrow \mathscr{K}^v$ denotes the MCR w.r.t. the vertex $v.\}$

```
1  begin
2      RMCRSet = φ;
3      S = FindSourceVertexes(α);
4      for s ∈ S do
5          if C^s ⊆ CK(e) then
6              RMCRSet∪={R^d | s⇝d};
7              RMCRSet∪=FindAncestors(s,e);
8          else
9              RMCRSet∪=FindDescendants(s,e);
10 end
```

Procedure FindAncestors Procedure

```
1  begin
2      A={a|there is an arc from a to s};
3      for a ∈ A do
4          if C^a ⊆ CK(e) then
5              RMCRSet∪={R^a};
6              RMCRSet∪=FindAncestors(a,e);
7  end
```

Procedure FindDescendants Procedure

```
1  begin
2      D={d|there is an arc from s to d};
3      for d ∈ D do
4          if C^d ⊆ CK(e) then
5              RMCRSet∪={R^d' | d⇝d'};
6          else
7              RMCRSet∪=FindDescendants(d,e);
8  end
```

Fig. 8. FindRMCR algorithm.

1. $\mathcal{O} = \{\mathcal{R}^d | s \rightsquigarrow d\}$. $\mathcal{R}^d \in \mathcal{O}$ may be a relevant MCR. Moreover, there is a salient feature that, if $\mathcal{R}^d$ is a relevant MCR, $\mathcal{R}^r(\forall r, d \rightsquigarrow r)$ must be a relevant MCR.
2. $\mathcal{I} = \{\mathcal{R}^a | a \rightsquigarrow s\}$. $\mathcal{R}^a \in \mathcal{I}$ may be a relevant MCR. There is a key feature that, if $\mathcal{R}^a$ is not a relevant MCR, $\mathcal{R}^v(\forall v, v \rightsquigarrow a)$ cannot be a relevant MCR for s.
3. $\mathcal{N} = \{\mathcal{R}^u | u \not\rightsquigarrow s \text{ and } s \not\rightsquigarrow u\}$. $\mathcal{R}^u \in \mathcal{N}$ cannot be a relevant MCR for s.

Based on these categories, we only need to check whether the MCRs of vertexes in $\{u | u \rightsquigarrow s$ or $s \rightsquigarrow u\}$ are relevant MCRs. Moreover, we can skip many irrelevant MCRs based on 1 and 2 so as to efficiently identify relevant MCRs. We devise an effective algorithm FindRMCR to identify relevant MCRs according to the three categories as shown in Fig. 8.

Given an accepting state α and an element in the runtime stack e, FindRMCR first locates to the source vertexes according to the NFA and the coverage graph based on the links as illustrated in Fig. 8 (line 3). Then, for each source vertex s, if $\mathcal{C}^s \subseteq \mathcal{CK}(e)$ ($\mathcal{C}^s \Rightarrow \mathcal{K}^s$ denotes the MCR for s), the MCRs of such vertexes in $\{\mathcal{R}^d | s \rightsquigarrow d\}$ must be relevant MCRs (line 6), and FindRMCR identifies other relevant MCRs, the corresponding vertexes of which have paths to e, calling its subroutine FindAncestors (line 7); otherwise, identifies other MCRs by calling subroutine FindDescendants (line 9).

FindAncestors identifies relevant MCRs among the vertexes which have paths to e iteratively. It first finds the vertexes (line 2), which have arcs to e, and then, for each such vertex a, if $\mathcal{C}^a \subseteq \mathcal{CK}(e)$, FindAncestors adds $\mathcal{R}^a$ into the result set (line 5) and identifies other relevant MCRs by calling itself (line 6); otherwise, there are no relevant MCRs. Similar to FindAncestors, FindDescendants also identifies relevant MCRs iteratively. It first finds the vertexes (e has arcs to such vertexes in line 2), and then, for each such vertex d, if $\mathcal{C}^d \subseteq \mathcal{CK}(e)$, FindAncestors adds the relevant MCRs in $\{\mathcal{R}^{d'} | d \rightsquigarrow d'\}$ into the result set (line 5); otherwise, identifies other relevant MCRs by calling itself (line 7). The complexity of FindAncestors is $O(DCG)$, where DCG is the depth of the coverage graph that is no larger than the number of queries. The complexity of FindAncestors is $O(CG)$, where CG is the number of nodes in the coverage graph. The complexity of FindRMCR is $O(AV * (CG + DCG))$, where AV is the average number of vertexes of a given state.

4.4.1 Update of Coverage Graph

When a query is added, for each query keyword, we first locate the maximal coverage rule for the keyword. If the coverage does not contain the query keyword, we update the coverage by adding the query keyword; otherwise, we need not do any update. When a query is removed, for each query keyword, we first locate the maximal coverage rule for the keyword. If the occurrence number of the keyword in the coverage is larger than one (the occurrence number is the number of queries than contain the keyword, which can be kept in the coverage graph), we decrease the occurrence number by one; otherwise, we update the coverage by removing the query keyword. If a maximal coverage rule is updated, we only need update its associated edges on the coverage graph.

4.5 KEMB Algorithm

We present how to incorporate MCRs and the coverage graph into KEMB so as to efficiently discard outdated elements in an eager manner. In contrast to KALE algorithm, during the NFA execution, when an "end-of-element" e arrives, KEMB checks whether there are accepting states for e. If so, for each accepting state α, KEMB checks whether e is a CLCA for α, constructs *path returns*, and delivers the result to corresponding users. Then, KEMB locates to the source vertexes and identifies the relevant MCRs based on the coverage graph. If e and each relevant MCR satisfy Lemma 5, KEMB discards outdated elements.

We devise an effective algorithm KEMB to answer large numbers of concurrent keyword queries (Fig. 10). KEMB first constructs the NFA, generates MCRs, and constructs the coverage graph in line 2 (i.e., the step of **Query Indexing** in KEMB, which can be performed offline). Then, KEMB parses the XML document (line 3). For each incoming event, if it is "start-of-document," KEMB initializes the stack and begins at the initial state (line 7; if it is "start-of-element," pushes e and "active states" by triggering the initial state with keywords directly contained in e (line 9); if it is "end-of-element," calls its subroutine EoE$^+$ to identify results and discards outdated elements (line 11). EoE$^+$ first pops the topmost element e (line 2). For each accepting state of e,

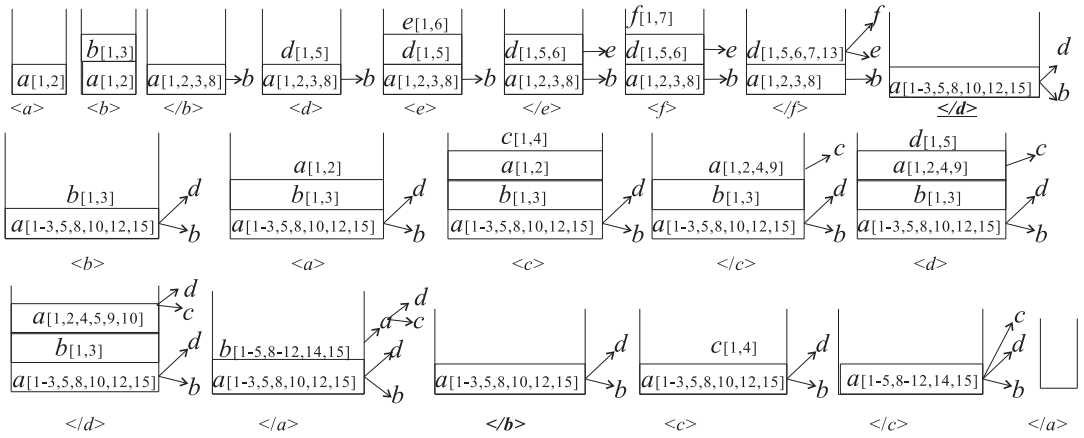

Fig. 9. A running example of KEMB with discarding outdated elements on queries in Table 1.

denoted by α, KEMB constructs the *path returns* rooted at e and delivers the results to corresponding users (line 4), identifies the relevant MCRs (line 5), and discards outdated elements as stated in Lemma 5 (line 6). Then, EoE$^+$ takes $\mathcal{T}_e$ (the subtree rooted at e by eliminating the outdated elements) as a subtree of the top element (line 7) and triggers each active state of the topmost element with keywords contained in $\mathcal{K}$ to generate "active states" for the next element, where $\mathcal{K}$ is the keyword set contained in $\mathcal{T}_e$ (lines 9-11).

In the algorithm, we need visit each XML element and generate active states. Then, for each accepting state, we need discard outdated elements, thus the complexity is $O\big(D * N * AS + Ans * AV * (CG + DCG)\big)$, where D is the depth of the input XML document, N is the number of XML elements, AS is the average number of active states for each element, Ans is the number of answers, AV is the average number of source vertexes for each accepting state, CG is the number of nodes in the coverage graph, and DCG is the depth of the coverage graph. Having walked through the logical construction and physical implementation of KEMB, we give a running example.

Example 4. Recall Example 3, when $</d_1>$ arrives, as state 13 is an accepting state, KEMB identifies MCR $\{e, f\} \Rightarrow \{e, f\}$ in the coverage graph and discards elements e, f. When $</b_2>$ arrives, as states 10,14,15 are accepting states, KEMB identifies MCRs $\{a, b, c\} \Rightarrow \{c\}$, $\{a, b, c, d\} \Rightarrow \{a, b\}$, $\{a, b, d\} \Rightarrow \{d\}$ and discards a, b, c, d. We walk through the example as shown in Fig. 9. We see that our strategy discards many outdated elements by comparing Fig. 9 with Fig. 5. We will experimentally show the benefits of our methods in Section 5.

Algorithm 4: KEMB Algorithm

```
Input: Q={Q1,Q2,...Qm} and an XML stream D
Output: PR={PR_i | PR_i is a path return w.r.t. Q_i}
1  begin
2      Construct_NFA&Generate_MCRs&Construct_CG(Q);
3      SD=SAXParser(D);
4      while e∈SD (in document order) do
5          switch event type of e do
6              case Start-of-Document
7                  Initialize Stack S & Begin at Initial State;
8              case Start-of-Element
9                  S.Push(e,Ω); /*Ω is the active state set for e*/
10             case End-of-Element
11                 EoE+();
12 end
```

Procedure EoE$^+$ Procedure

```
1  begin
2      e=S.Pop();
3      for α ∈ A(A is the accepting state set w.r.t. e) do
4          OutputResult(e,α);
5          PR←FindRMCR(e,α);
6      Discard(T_e);   /*T_e is the subtree rooted at e*/
7      S.top.subtree=S.top.subtree←T_e;/*take T_e as a subtree*/
8      for k ∈ K(K is the keyword set w.r.t. T_e) do
9          for α∈Ω(Ω is the active state set w.r.t. S.top()) do
10             Ψ← the active state set by triggering α with k;
11             S.top.ass∪=Ψ; /*update the active states (ass)*/
12 end
```

Fig. 10. KEMB algorithm.

4.6 Supporting OR Semantics Efficiently

In this section, we discuss how to extend our method to support the OR semantics efficiently. In the runtime stack, the above-discussed method can generate all CLCAs in terms of the AND semantics. To support the OR semantics, when finding a CLCA in the runtime stack satisfying the AND semantics, we check each of its children c as follows: If we remove all of c's descendants, the CLCA still contains the same keywords, then c may be a CLCA in terms of the OR semantics [12], and we still keep the subtree rooted at c in the stack. We will compute CLCAs in terms of the OR semantics under the subtree rooted at c later. After generating all CLCAs satisfying the AND semantics, the runtime stack still contains some keyword information, and we identify other CLCAs in terms of the OR semantics as

TABLE 2
Characteristics of Three DTDs

Datasets	# of elements	# of attributes	depth
NITF	123	510	12
DBLP	36	14	6
TreeBank	102	0	36

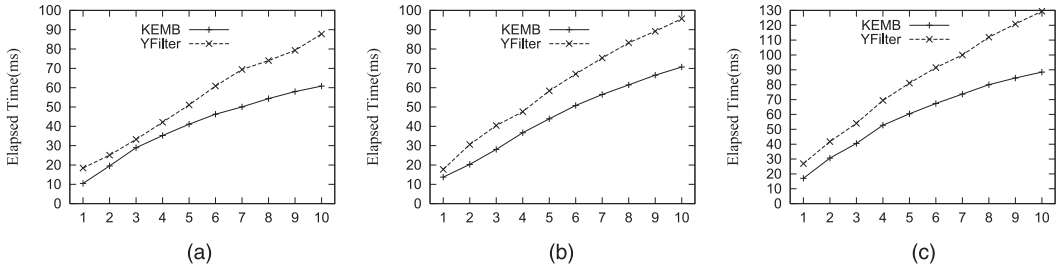

Fig. 11. Elapsed time of indexing queries. (a) # of queries (*10,000) - NITF. (b) # of queries (*10,000) - DBLP. (c) # of queries (*10,000) - TreeBank.

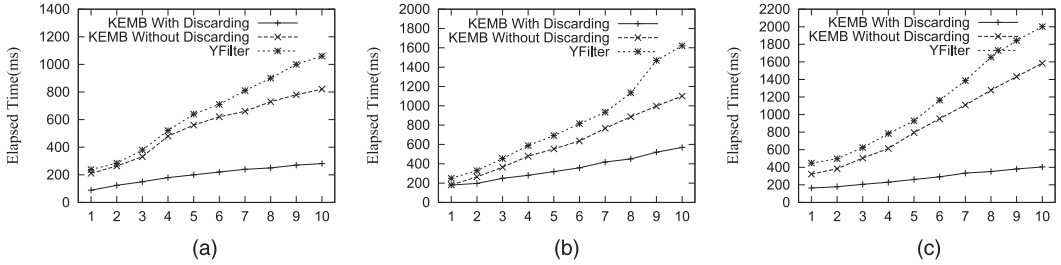

Fig. 12. Elapsed time of the NFA execution. (a) # of queries (*10,000) - NITF. (b) # of queries (*10,000) - DBLP. (c) # of queries (*10,000) - TreeBank

follows: For each element in the stack from the bottom to the top, e, if we remove its children which contain the same keywords as e, e still contains the same keywords, then e must be a CLCA in terms of the OR semantics. We generate the path tree rooted at e including e's descendants which contain smaller number of distinct keywords than e [12], and take it as an answer. Similarly, we can generate all CLCAs in terms of the OR semantics.

5 Experiment Study

In this section, we examine the performance of KEMB. We employed the news industry text format (NITF) DTD[1] as a data set. We used IBM's XML Generator[2] to generate 100 XML documents based on the NITF DTD. The sizes of all the generated XML documents are in the range of 100-2,000 KB. We also used the real data set TreeBank[3] and DBLP[4] in our experiments. For DBLP and TreeBank data sets, we extracted data from the XML document to form 100 small documents. The sizes of the extracted documents are about 1-10 MB.

We used the selected DTDs, NITF, DBLP, and TreeBank (extracted from the data set) to generate the workloads for our experiments. We used the tools provided in YFilter [9] to generate XPath queries. Some characteristics of these DTDs are shown in Table 2. Note that all of the DTDs allow an infinite level of nesting due to loops involving multiple elements.

Given a DTD, the tools used to run the experiments include a DTD parser, a query generator, an XML generator, and an event-based XML parser supporting the SAX interface. The DTD parser which was developed using a WUTKA DTD parser[5] outputs parent-child relationships between elements, and statistical information for each element including the probability of an attribute occurring in an element (randomly chosen between 0 and 1) and the maximum number of values an element or an attribute can take (randomly chosen between 1 and 20). The output of the DTD parser is used by the query generator and the document generator. YFilter provides a query generator that creates a set of XPath queries based on the workload parameters and we used the default parameters in our experiments. For each data set, we generated 10 sets of XPath queries by varying query numbers from 10,000 to 100,000. Then, we generated the keyword queries by taking terms of each XPath query as keywords of the transformed keyword query.

To offer insight into the comparison between XPath-based methods and keyword-based methods, we compared KEMB with YFilter [9] to show the benefits of our method. We evaluated YFilter on the XPath queries and KEMB on the corresponding keyword queries.

All the algorithms were implemented in Java. The codes of YFilter were provided by YFilter group.[6] We ran all the experiments on an Intel(R) Core(TM) 2.0 GHz CPU machine with 2 GB memory running windows XP.

5.1 Indexing Keyword Queries

This section evaluates the performance of indexing large numbers of keyword queries. We show the elapsed time of indexing keyword queries (including constructing the NFA and generating MCRs and the coverage graph) in Fig. 11.

We observe that the elapsed time of KEMB scales well with the number of keyword queries. KEMB also outperforms YFilter. Because YFilter has to deal with the branch nodes, predicates, wildcard, "/" and "//" in XPath queries while KEMB only needs to index simple input keywords. Moreover, KEMB can share many overlaps among keyword queries, and thus, achieves much better performance. For example, in Fig. 11, the elapsed time of indexing keyword queries on NITF data set varies a little with the increase of

1. http://www.nitf.org/.
2. http://www.alphaworks.ibm.com/tech/xmlgenerator.
3. http://www.cs.washington.edu/research/xmldata sets/.
4. http://dblp.uni-trier.de/xml.
5. http://www.wutka.com/dtdparser.html.
6. http://yfilter.cs.umass.edu/.

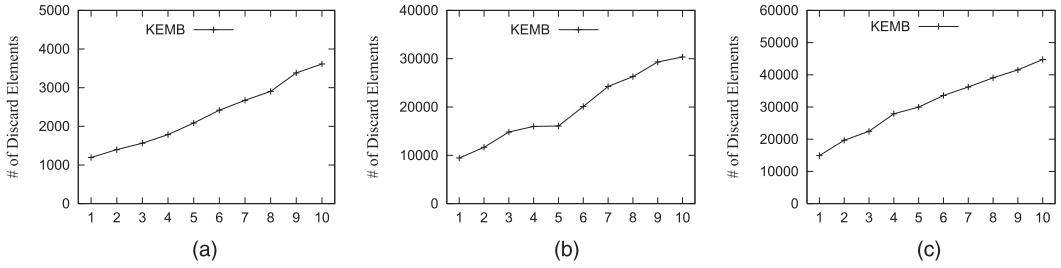

Fig. 13. # of discarded elements. (a) # of queries (*10,000) - NITF. (b) # of queries (*10,000) - DBLP. (c) # of queries (*10,000) - TreeBank.

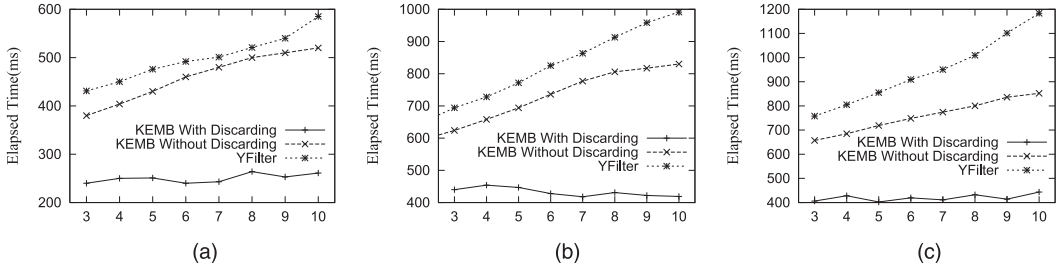

Fig. 14. Elapsed time by varying # of keywords (# of $\mathrm{Queries} = 20{,}000$). (a) # of keywords - NITF. (b) # of keywords - DBLP. (c) # of keywords - TreeBank.

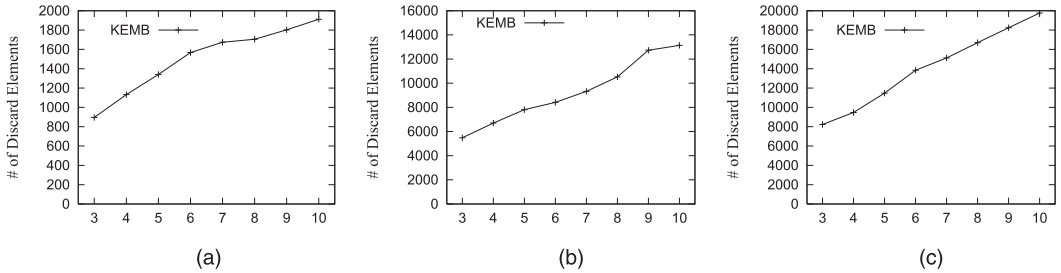

Fig. 15. # of discarded elements by varying # of keywords (# of $\mathrm{Queries} = 20{,}000$). (a) # of keywords - NITF. (b) # of keywords - DBLP. (c) # of keywords - TreeBank.

the number of keyword queries; while YFilter performs substantially worse than KEMB as it needs to index the XPath queries and cannot share the overlaps of keywords.

5.2 Efficiency of KEMB

This section evaluates the filter efficiency of KEMB on various data sets. We computed the elapsed time of KEMB with/without discarding elements, and compared with YFilter. The experimental results obtained are shown in Fig. 12. Moreover, we give the number of discarded elements in Fig. 13.

We see that KEMB scales well with the number of keyword queries. We note that KEMB with discarding elements based on MCRs and the coverage graph indeed improves the performance. Because KEMB need not maintain the outdated elements and transit the useless states. For example, on TreeBank data set, KEMB with discarding elements only costs 400 ms to answer 100,000 keyword queries and discards about 45,000 outdated elements.[7] While YFilter costs nearly 2,000 ms. KEMB achieves much higher performance than YFilter, as it is rather expensive to deal with complicated XPath queries, which involve branch nodes, predicates, wildcard "$*$," double slash "//," and slash "/."

To offer insight into the performance, we evaluated our algorithms by varying the number of keywords. We grouped the generated keyword queries according to the number of keywords in each query. There are eight groups with the numbers of keywords varying from 3 to 10. Each group has 20,000 queries. Fig. 14 gives the elapsed time of processing such queries and Fig. 15 illustrates the number of discarded elements.

We observe that with the increase of the numbers of input keywords, KEMB can discard more outdates elements, and thus, achieves much higher efficiency than YFilter. This is because the keyword queries with more keywords will result in more relevant elements, and thus, more outdated elements will be involved. Thus, KEMB varies slightly with the increase of the numbers of input keywords, while the performance of YFilter drops down sharply. For example, on DBLP data set, KEMB only costs 420 ms when the number of keywords is 10, while YFilter

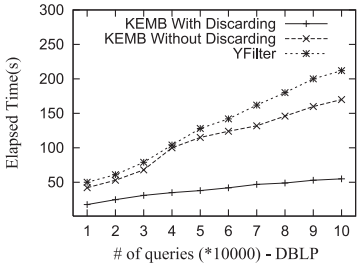

Fig. 16. Elapsed time of the NFA execution (DBLP).

7. The numbers of elements in the selected documents on NITF, DBLP, and TreeBank are, respectively, 19340, 151003, and 170164.

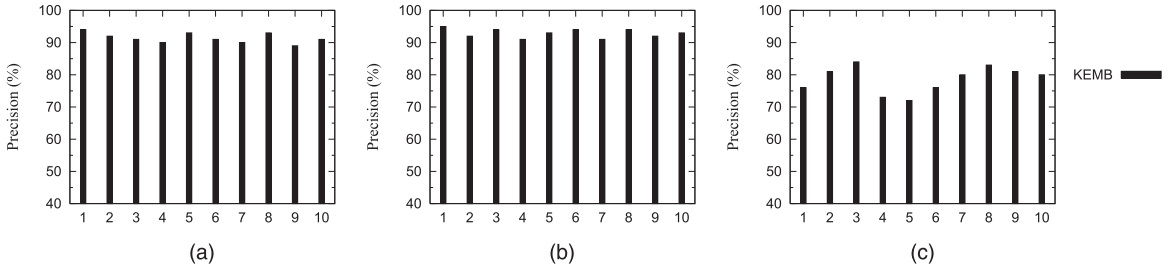

Fig. 17. Precision on different numbers of queries. (a) # of queries (*10,000) - NITF. (b) # of queries (*10,000) - DBLP. (c) # of queries (*10,000) - TreeBank.

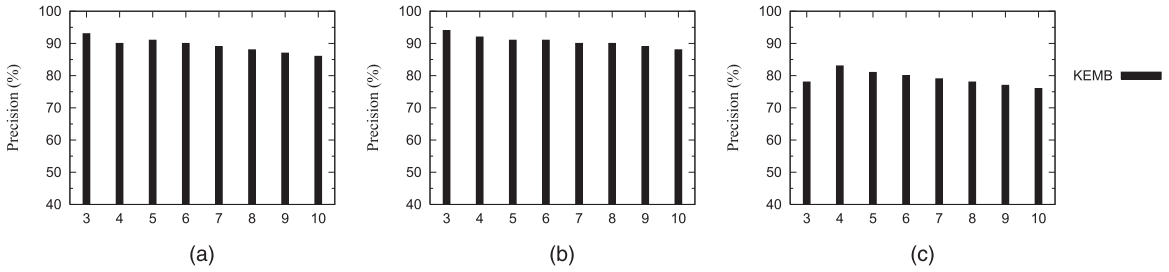

Fig. 18. Precision on different numbers of keywords (# of $\mathrm{Queries} = 20{,}000$). (a) # of keywords - NITF. (b) # of keywords - DBLP. (c) # of keywords - TreeBank.

costs nearly 1,000 ms. Moreover, KEMB can discard 13,000 outdated elements.

We also used the DBLP data set (470 MB) as a single, really huge document to test our algorithm. Fig. 16 gives the experimental results. We can see that our method can efficiently identify relevant answers on a large document.

5.3 Effectiveness of KEMB

This section evaluates the filter effectiveness of KEMB on various data sets. We took the results of XPath queries as the accurate answers to evaluate the effectiveness. We used the well-known metrics precision and recall to evaluate our algorithms. Fig. 17 illustrates the precision of KEMB by varying different numbers of queries and Fig. 18 shows the precision with different numbers of keywords. We observe that KEMB achieves high precision on NITF and DBLP data sets and gets good quality on TreeBank data set. Although KEMB has few false positives since keyword queries are not as powerful as XPath queries, keyword search is user friendly.

In addition, we compared our method with SLCA [41]. We evaluate the answer by human judgement. A group of users (20 people from different research areas) participated in the user study. An answer is deemed to be relevant if the majority of the users think it is relevant. To assess recall, the users browsed the entire XML document to verify whether relevant results were missed by the search. As the XML documents used in this experiment were small (about 1 MB), the users can compute the real recall through browsing the data. Table 3 gives the average precision and recall for all queries. We note that our method outperforms SLCA as our method can avoid false negatives introduced by SLCA, especially on TreeBank data set with highly nested structures.

5.4 Usability Study

To evaluate the usability of different algorithms, we used the INEX 2005 data set.[8] The INEX corpus is composed of the full texts, marked up in XML, consisting of 16,819 articles of the IEEE Computer Society's publications from 12 magazines and six transactions, covering the period of 1995-2004, and totaling 735 megabytes in size. The collection has a suitably complex XML structure (192 different models in DTD) and contains scientific articles of varying length. On average an article contains 1,532 XML nodes, where the average depth of a node is 6.9.

The INEX 2005 collection consists of 47 content and structure (CAS) topics and 40 Content-Only + Structure (CO+S) topics. Here, we focus on the CO+S topics. For each CO+S topic, we generated a keyword query by selecting some keywords from the title. For example, we generate the keyword query of topic 231 as "markov chains graph related algorithms."

INEX 2005 proposes a new set of measures, the eXtended Cumulated Gain (XCG) measures. The XCG measures are a family of evaluation measures that are an extension of the cumulated gain (CG)-based metrics and which aim to consider the dependency of XML elements (e.g., overlap and near misses) within the evaluation [23]. We used the normalized extended cumulated gain (nxCG) to evaluate an answer. We compared with the best results at INEX 2005 and gave the virtual rank of our method at INEX 2005. Table 4 illustrates the results. We see that our method achieves high performance at the Focused task. This is because the Focused retrieval requires the results with the right granularity, and without overlap, and our method generates overlap-free subtrees rooted CLCAs as answers. In addition, CLCAs emphasize on the structural information to answer keyword

TABLE 3
Precision/Recall on Various Data Sets

Datasets	NITF		DBLP		TreeBank	
Approaches	CLCA	SLCA	CLCA	SLCA	CLCA	SLCA
Precision (%)	90.32	84.2	92.54	86.3	86.42	75.1
Recall (%)	100	100	100	100	100	80.2

8. http://inex.is.informatik.uni-duisburg.de/2005/.

TABLE 4
Evaluation Results on INEX 2005 CO+S Topics: (a) Thorough Retrieval Strategies and (b) Focused Retrieval Strategies

Approaches	nxCG (strict)			nxCG(Generalized)		
	10	25	50	10	25	50
Best results at INEX'05	0.1004	0.1189	0.1931	0.3037	0.2771	0.2546
KEMB	0.0615	0.0743	0.0831	0.2215	0.2587	0.2931
Rank of KEMB at INEX'05	8	14	26	19	7	1

(a)

Approaches	nxCG (strict)			nxCG(Generalized)		
	10	25	50	10	25	50
Best results at INEX'05	0.1401	0.543	0.1902	0.2688	0.2325	0.2190
KEMB	0.0615	0.0753	0.0812	0.2303	0.2542	0.2886
Rank of KEMB at INEX'05	14	15	20	8	1	1

(b)

queries and the Focused retrieval strategy benefits more from the structural hints.

6 Conclusions

In this paper, we have studied the problem of keyword-based XML message brokering with user subscribed profiles of keyword queries. We have presented an algorithm KALE to efficiently and effectively answer keyword queries in XML data streams. We have devised an automaton-based algorithm KEMB to answer large numbers of concurrent keyword queries. We have demonstrated techniques to effectively index large numbers of queries by sharing the overlaps for improving the performance and scalability. We have demonstrated an effective mechanism to discard outdated elements in an eager manner. We have conducted an extensive experimental study, and the results show that KEMB achieves high performance and scales very well.

Acknowledgments

This work is partly supported by the National Natural Science Foundation of China under Grant No. 61003004, the National Grand Fundamental Research 973 Program of China under Grant No. 2011CB302206, and National S & T Major Project of China under Grant No. 2011ZX01042-001-002.

References

[1] S. Agrawal, S. Chaudhuri, and G. Das, "Dbxplorer: A System for Keyword-Based Search over Relational Databases," *Proc. Int'l Conf. Data Eng. (ICDE),* pp. 5-16, 2002.

[2] M. Altinel and M.J. Franklin, "Efficient Filtering of XML Documents for Selective Dissemination of Information," *Proc. Int'l Conf. Very Large Data Bases (VLDB),* pp. 53-64, 2000.

[3] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan, "Keyword Searching and Browsing in Databases Using Banks," *Proc. Int'l Conf. Data Eng. (ICDE),* pp. 431-440, 2002.

[4] N. Bruno, L. Gravano, N. Koudas, and D. Srivastava, "Navigation- vs. Index-Based XML Multi-Query Processing," *Proc. Int'l Conf. Data Eng. (ICDE),* pp. 139-150, 2003.

[5] K.S. Candan et al, "Afilter: Adaptable XML Filtering with Prefix-Caching and Suffix-Clustering," *Proc. Int'l Conf. Very Large Data Bases (VLDB),* pp. 559-570, 2006.

[6] C.Y. Chan, P. Felber, M.N. Garofalakis, and R. Rastogi, "Efficient Filtering of XML Documents with XPath Expressions," *Proc. Int'l Conf. Data Eng. (ICDE),* pp. 235-244, 2002.

[7] C.-Y. Chan and Y. Ni, "Efficient XML Data Dissemination with Piggybacking," *Proc. ACM SIGMOD,* 2007.

[8] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv, "XSearch: A Semantic Search Engine for XML," *Proc. Int'l Conf. Very Large Data Bases (VLDB),* 2003.

[9] Y. Diao, P.M. Fischer, M.J. Franklin, and R. To, "YFilter: Efficient and Scalable Filtering of XML Documents," *Proc. Int'l Conf. Data Eng. (ICDE),* 2002.

[10] B. Ding, J.X. Yu, S. Wang, L. Qin, X. Zhang, and X. Lin, "Finding Top-k Min-Cost Connected Trees in Databases," *Proc. Int'l Conf. Data Eng. (ICDE),* 2007.

[11] F. Fabret, H.-A. Jacobsen, F. Llirbat, J. Pereira, K.A. Ross, and D. Shasha, "Implementing a Scalable XML Publish/Subscribe System Using a Relational Database System," *Proc. ACM SIGMOD,* pp. 479-490, 2004.

[12] J. Feng, G. Li, J. Wang, and L. Zhou, "Finding and Ranking Compact Connected Trees for Effective Keyword Proximity Search in XML Documents," *Proc. Information System,* http://dx.doi.org/10.1016/j.is.2009.05.004, 2009.

[13] T.J. Green, G. Miklau, M. Onizuka, and D. Suciu, "Processing XML Streams with Deterministic Automata," *Proc. Int'l Conf. Database Theory (ICDT),* pp. 173-189, 2003.

[14] L. Guo, J. Shanmugasundaram, and G. Yona, "Topology Search over Biological Databases," *Proc. Int'l Conf. Data Eng. (ICDE),* 2007.

[15] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram, "XRank: Ranked Keyword Search over XML Documents," *Proc. ACM SIGMOD,* pp. 16-27, 2003.

[16] A.K. Gupta and D. Suciu, "Stream Processing of XPath Queries with Predicates," *Proc. ACM SIGMOD,* pp. 419-430, 2003.

[17] H. He, H. Wang, J. Yang, and P. Yu, "Blinks : Ranked Keyword Searches on Graphs," *Proc. ACM SIGMOD,* 2007.

[18] V. Hristidis, L. Gravano, and Y. Papakonstantinou, "Efficient IR-Style Keyword Search over Relational Databases," *Proc. Int'l Conf. Very Large Data Bases (VLDB),* pp. 850-861, 2003.

[19] V. Hristidis, N. Koudas, Y. Papakonstantinou, and D. Srivastava, "Keyword Proximity Search in XML Trees," *IEEE Trans. Knowledge and Data Eng.,* vol. 18, no. 4, pp. 525-539, Apr. 2006.

[20] V. Hristidis and Y. Papakonstantinou, "Discover: Keyword Search in Relational Databases," *Proc. Int'l Conf. Very Large Data Bases (VLDB),* pp. 670-681, 2002.

[21] V. Hristidis, Y. Papakonstantinou, and A. Balmin, "Keyword Proximity Search on XML Graphs," *Proc. Int'l Conf. Data Eng. (ICDE),* pp. 367-378, 2003.

[22] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar, "Bidirectional Expansion for Keyword Search on Graph Databases," *Proc. Int'l Conf. Very Large Data Bases (VLDB),* pp. 505-516, 2005.

[23] G. Kazai and M. Lalmas, "INEX 2005 Evaluation Measures," *Proc. INitiative for the Evaluation of XML Retrieval (INEX),* http://www.dcs.gla.ac.uk/mounia/Papers/inex-2005-metrics.pdf., 2005.

[24] J. Kwon, P. Rao, B. Moon, and S. Lee, "FiST: Scalable XML Document Filtering by Sequencing Twig Patterns," *Proc. Int'l Conf. Very Large Data Bases (VLDB),* pp. 217-228, 2005.

[25] J. Kwon, P. Rao, B. Moon, and S. Lee, "Predicate-Based Filtering of XPath Expressions," *Proc. Int'l Conf. Data Eng. (ICDE),* 2006.

[26] L.V.S. Lakshmanan and S. Parthasarathy, "On Efficient Matching of Streaming XML Documents and Queries," *Proc. Int'l Conf. Extending Database Technology (EDBT),* pp. 142-160, 2002.

[27] G. Li, J. Feng, J. Wang, X. Song, and L. Zhou, "Sailer: An Effective Search Engine for Unified Retrieval of Heterogeneous XML and Web Documents," *Proc. Int'l Conf. World Wide Web (WWW),* pp. 1061-1062, 2008.

[28] G. Li, J. Feng, J. Wang, B. Yu, and Y. He, "Race: Finding and Ranking Compact Connected Trees for Keyword Proximity Search over XML Documents," *Proc. Int'l Conf. World Wide Web (WWW),* pp. 1045-1046, 2008.

[29] G. Li, J. Feng, J. Wang, and L. Zhou, "Effective Keyword Search for Valuable LCAs over XML Documents," *Proc. Conf. Information and Knowledge Management (CIKM),* pp. 31-40, 2007.

[30] G. Li, B.C. Ooi, J. Feng, J. Wang, and L. Zhou, "Ease: An Effective 3-In-1 Keyword Search Method for Unstructured, Semi-Structured and Structured Data," *Proc. ACM SIGMOD,* pp. 903-914, 2008.

[31] G. Li, X. Zhou, J. Feng, and J. Wang, "Progressive Top-k Keyword Search in Relational Databases," *Proc. Int'l Conf. Data Eng. (ICDE),* 2009.
[32] Y. Li, C. Yu, and H.V. Jagadish, "Schema-Free XQuery," *Proc. Int'l Conf. Very Large Data Bases (VLDB),* pp. 72-84, 2004.
[33] F. Liu, C. Yu, W. Meng, and A. Chowdhury, "Effective Keyword Search in Relational Databases," *Proc. ACM SIGMOD,* pp. 563-574, 2006.
[34] Z. Liu and Y. Chen, "Identifying Meaningful Return Information for XML Keyword Search," *Proc. ACM SIGMOD,* 2007.
[35] Z. Liu and Y. Chen, "Reasoning and Identifying Relevant Matches for XML Keyword Search" *Proc. Very Large Data Bases Endowment,* vol. 1, no. 1, pp. 921-932, 2008.
[36] Y. Luo, X. Lin, W. Wang, and X. Zhou, "Spark: Top-k Keyword Query in Relational Databases," *Proc. ACM SIGMOD,* 2007.
[37] A. Markowetz, Y. Yang, and D. Papadias, "Keyword Search on Relational Data Streams," *Proc. ACM SIGMOD,* 2007.
[38] A. Raj and P. Kumar, "Branch Sequencing Based XML Message Broker Architecture," *Proc. Int'l Conf. Data Eng. (ICDE),* 2007.
[39] M. Sayyadian, H. Le Khac, A. Doan, and L. Gravano, "Efficient Keyword Search across Heterogeneous Relational Databases," *Proc. Int'l Conf. Data Eng. (ICDE),* 2007.
[40] C. Sun, C.Y. Chan, and A.K. Goenka, "Multiway SLCA-Based Keyword Search in XML Data," *Proc. Int'l Conf. World Wide Web (WWW),* pp. 1043-1052, 2007.
[41] Y. Xu and Y. Papakonstantinou, "Efficient Keyword Search for Smallest LCAs in XML Databases," *Proc. ACM SIGMOD,* pp. 527-538, 2005.
[42] Y. Xu and Y. Papakonstantinou, "Efficient LCA Based Keyword Search in XML Data," *Proc. Int'l Conf. Extending Database Technology (EDBT),* pp. 535-546, 2008.

Guoliang Li received the PhD degree in computer science from the Tsinghua University, Beijing, China, in 2009. Since then, he has worked as an assistant professor in the Department of Computer Science and Technology, Tsinghua University, Beijing, China. His research interests mainly include integrating databases and information retrieval, data cleaning, and data integration.

Jianhua Feng received the BS, MS and PhD degrees in computer science and technology from the Tsinghua University. He is currently working as a professor in the Department of Computer Science and Technology in Tsinghua University. His main research interests include native XML database, data mining, and keyword search over structure and semistructure data. He has published papers in the international top conferences and top journals, such as ACM SIGMOD, ACM SIGKDD, VLDB, IEEE ICDE, WWW, ACM CIKM, ICDM, SDM, IEEE TKDE, Data Mining and Knowledge Discovery, Information Systems, and so on. He is a member of the IEEE and a senior member of the China Computer Federation (CCF).

Jianyong Wang received the PhD degree in computer science from the Institute of Computing Technology, Chinese Academy of Sciences, in 1999. He is currently an associate professor in the Department of Computer Science and Technology, Tsinghua University, Beijing, China. He was ever an assistant professor at the Peking University and visited Simon Fraser University, University of Illinois at Urbana-Champaign, and University of Minnesota at Twin Cities before joining Tsinghua University in Dec. 2004. His research interests mainly include data mining and knowledge discovery, and web information management. He has coauthored more than 40 research papers in some leading international conferences, such as ACM SIGKDD, ACM SIGMOD, VLDB, IEEE ICDE, SIAM SDM, IEEE ICDM, EDBT, IEEE IPDPS, and ACM CIKM, and some top international journals, such as ACM TODS, DMKD, and IEEE TKDE. According to Google Scholar, all his coauthored papers have attracted more than 2,200 citations. He is a recipient of the 2009 HP Labs Innovation Research Award, the 2009 Okawa Foundation Research Grant (Japan), WWW '08 best posters award, and the Year 2007 Program for New Century Excellent Talents in University, State Education Ministry of China. He is a senior member of the IEEE and a member of the ACM SIGKDD.

Lizhu Zhou received the master of science degree from the University of Toronto, in 1983. Currently, he is a full professor at the Tsinghua University, China. His major research interests include database systems, web data processing, and digital resource management. He is a member of the ACM.