Efficient Fuzzy Type-Ahead Search in XML Data

Jianhua Feng, *Senior Member*, *IEEE*, and Guoliang Li, *Member*, *IEEE*

Abstract—In a traditional keyword-search system over XML data, a user composes a keyword query, submits it to the system, and retrieves relevant answers. In the case where the user has limited knowledge about the data, often the user feels "left in the dark" when issuing queries, and has to use a try-and-see approach for finding information. In this paper, we study *fuzzy type-ahead search* in XML data, a new information-access paradigm in which the system searches XML data on the fly as the user types in query keywords. It allows users to explore data as they type, even in the presence of minor errors of their keywords. Our proposed method has the following features: 1) Search as you type: It extends Autocomplete by supporting queries with multiple keywords in XML data. 2) Fuzzy: It can find high-quality answers that have keywords matching query keywords approximately. 3) Efficient: Our effective index structures and searching algorithms can achieve a very high interactive speed. We study research challenges in this new search framework. We propose effective index structures and top-k algorithms to achieve a high interactive speed. We examine effective ranking functions and early termination techniques to progressively identify the top-k relevant answers. We have implemented our method on real data sets, and the experimental results show that our method achieves high search efficiency and result quality.

Index Terms—XML, keyword search, type-ahead search, fuzzy search.

1 Introduction

Traditional methods use query languages such as XPath and XQuery to query XML data. These methods are powerful but unfriendly to nonexpert users. First, these query languages are hard to comprehend for nondatabase users. For example, XQuery is fairly complicated to grasp. Second, these languages require the queries to be posed against the underlying, sometimes complex, database schemas. Fortunately, keyword search is proposed as an alternative means for querying XML data, which is simple and yet familiar to most Internet users as it only requires the input of keywords. Keyword search is a widely accepted search paradigm for querying document systems and the World Wide Web. Recently, the database research community has been studying challenges related to keyword search in XML data [19], [12], [37], [54], [49], [32], [40], [55], [35]. One important advantage of keyword search is that it enables users to search information without knowing a complex query language such as XPath or XQuery, or having prior knowledge about the structure of the underlying data.

In a traditional keyword-search system over XML data, a user composes a query, submits it to the system, and retrieves relevant answers from XML data. This information-access paradigm requires the user to have certain knowledge about the structure and content of the underlying data repository. In the case where the user has limited knowledge about the data, often the user feels "left in the dark" when issuing queries, and has to use a try-and-see approach for finding information. He tries a few possible keywords, goes through the returned results, modifies the keywords, and reissues a new query. He needs to repeat this step multiple times to find the information, if lucky enough. This search interface is neither efficient nor user friendly. Many systems are introducing various features to solve this problem. One of the commonly used methods is *Autocomplete,* which predicts a word or phrase that the user may type in based on the partial string the user has typed. More and more websites support this feature. As an example, almost all the major search engines nowadays automatically suggest possible keyword queries as a user types in partial keywords. Both Google Finance (http://finance.google.com/) and Yahoo! Finance (http://finance.yahoo.com/) support searching for stock information interactively as users type in keywords.

One limitation of Autocomplete is that the system treats a query with multiple keywords as a *single string;* thus, it does not allow these keywords to appear at different places. For instance, consider the search box on Apple.com, which allows Autocomplete search on Apple products. Although a keyword query "`iphone`" can find a record "`iphone has some great new features`," a query with keywords "`iphone features`" cannot find this record (as of February 2010), because these two keywords appear at different places in the answer.

To address this problem, Bast and Weber [6], [7] proposed CompleteSearch in textual documents, which can find relevant answers by allowing query keywords appear at any places in the answer. However, CompleteSearch does not support approximate search, that is it cannot allow minor errors between query keywords and answers. Recently, we studied *fuzzy type-ahead search* in textual documents [27]. It allows users to explore data as they type, even in the presence of minor errors of their input keywords. Type-ahead search can provide users instant

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

Manuscript received 20 Feb. 2010; revised 16 July 2010; accepted 25 Sept. 2010; published online 21 Dec. 2010.
Recommended for acceptance by Q. Li.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-2010-02-0103.

1041-4347/12/$31.00 © 2012 IEEE Published by the IEEE Computer Society

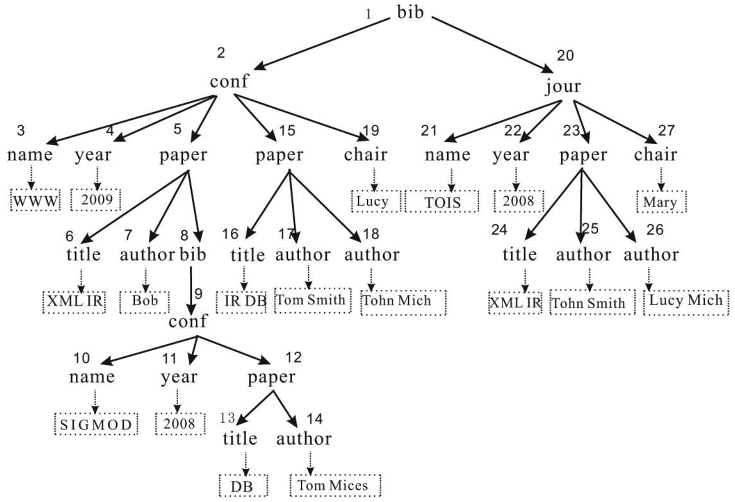


Fig. 1. An XML document.

feedback as users type in keywords, and it does not require users to type in complete keywords. Type-ahead search can help users browse the data, save users typing effort, and efficiently find the information. We also studied type-ahead search in relational databases [34]. However, existing methods cannot search XML data in a type-ahead search manner, and it is not trivial to extend existing techniques to support fuzzy type-ahead search in XML data. This is because XML contains parent-child relationships, and we need to identify relevant XML subtrees that capture such structural relationships from XML data to answer keyword queries, instead of single documents.

In this paper, we propose TASX (pronounced "task"), a *fuzzy type-ahead* search method in XML data. TASX searches the XML data on the fly as users type in query keywords, even in the presence of minor errors of their keywords. TASX provides a friendly interface for users to explore XML data, and can significantly save users typing effort. In this paper, we study research challenges that arise naturally in this computing paradigm. The main challenge is search efficiency. Each query with multiple keywords needs to be answered efficiently. To make search really interactive, for each keystroke on the client browser, from the time the user presses the key to the time the results computed from the server are displayed on the browser, the delay should be as small as possible. An interactive speed requires this delay should be within milliseconds. Notice that this time includes the network transfer delay, execution time on the server, and the time for the browser to execute its JavaScript. This low-running-time requirement is especially challenging when the backend repository has a large amount of data. To achieve our goal, we propose effective index structures and algorithms to answer keyword queries in XML data. We examine effective ranking functions and early termination techniques to progressively identify top-k answers. To the best of our knowledge, this is the first paper to study fuzzy type-ahead search in XML data. To summarize, we make the following contributions:

- We formalize the problem of fuzzy type-ahead search in XML data.
- We propose effective index structures and efficient algorithms to achieve a high interactive speed for fuzzy type-ahead search in XML data.
- We develop ranking functions and early termination techniques to progressively and efficiently identify the top-k relevant answers.
- We have conducted an extensive experimental study. The results show that our method achieves high search efficiency and result quality.

The remainder of this paper is organized as follows: Section 2 gives the preliminaries. We formalize the problem of fuzzy type-ahead search in XML data in Section 3 and propose a lowest common ancestor (LCA)-based method in Section 4. Section 5 introduces a progressive search method. Extensive experimental evaluations are provided in Section 6. We review related work in Section 7 and conclude in Section 8.

2 Preliminaries

2.1 Notations

An XML document can be modeled as a rooted and labeled tree. A node v in the tree corresponds to an element in the XML document and has a label. For two nodes u and v, we use "$u \prec v$" ("$u \succ v$," respectively) to denote that node u is an ancestor (descendant, respectively) of node v. We use "$u \preceq v$" to denote that $u \prec v$ or $u = v$. For example, consider the XML document in Fig. 1, we have paper (node 5) $\preceq$ author (node 7) and paper (node 12) $\succ$ conf (node 2).

A keyword query consists of a set of keywords $\{k_1, k_2, \ldots, k_\ell\}$. For each keyword k_i, we call the nodes in the tree that contain the keyword the *content nodes* for k_i. The ancestor nodes[1] of the content nodes are called the *quasi-content nodes* of the keyword. For example, consider the XML document in Fig. 1, title (node 16) is a content node for keyword "DB," and conf (node 2) is a quasi-content node of keyword "DB."

2.2 Keyword Search in XML Data

In the literature, there are different ways to define the answers to a keyword query on an XML document. A commonly used

[1] A node is not an ancestor nor a descendant of itself.

one is based on the notion of *lowest common ancestor* [20]. Given an XML document D and its XML nodes $v_1, v_2, \ldots, v_m$, we say a node u in the document is the lowest common ancestor of these nodes if $\forall\ 1 \leq i \leq m$, $u \preceq v_i$, and there does not exist another node u' such that $u \prec u'$ and $u' \preceq v_i$.

Intuitively, each LCA of the keyword query is the LCA of a set of content nodes corresponding to all the keywords in the query. Many algorithms for XML keyword search use the notion of LCA or its variants [19], [12], [37], [54], [49], [32], [40], [55]. For a keyword query, the LCA-based algorithm first retrieves *content nodes* in XML data that contain the input keywords using inverted indices. It then identifies the LCAs of the content nodes, and takes the subtrees rooted at the LCAs as the answer to the query. For example, a bibliography XML document is shown in Fig. 1. Suppose a user issues a keyword query "DB Tom." The content nodes of "DB" and "Tom" are {13,16} and {14,17}, respectively. Nodes 2, 12, and 15 are LCAs of the keyword query. Notice that node 2 is the LCA of nodes 13 and 17. Evidently, node 2 is less relevant to the query than nodes 12 and 15, as nodes 13 and 17 correspond to values of different papers.

To address this limitation of using LCAs as query answers, many methods have been proposed [12], [25], [37], [11], [23], [48], [38] to improve search efficiency and result quality. Guo et al. [19] and Xu and Papakonstantinou [55] proposed *exclusive lowest common ancestor* (ELCA). Given a keyword query $Q = \{k_1, k_2, \ldots, k_\ell\}$ and an XML document D, $u \in D$ is called an ELCA of Q, if and only if there exists nodes $v_1 \in \mathcal{I}_{k_1}, v_2 \in \mathcal{I}_{k_2}, \ldots, v_\ell \in \mathcal{I}_{k_\ell}$ such that u is the LCA of $v_1, v_2, \ldots, v_\ell$, and for every v_i, the descendants of u on the path from u to v_i are not LCAs of Q nor ancestors of any LCA of Q.

An LCA is an ELCA if it is still an LCA after excluding its LCA descendants. For example, the ELCAs to the keyword query "DB Tom" on the data in Fig. 1 are nodes 12 and 15. Node 2 is not an ELCA as it is not an LCA after excluding nodes 12 and 15. Xu and Papakonstantinou [55] proposed a binary-search-based method to efficiently identify ELCAs.

3 Problem Formulation of Fuzzy Type-Ahead Search in XML Data

In this section, we introduce the overview of fuzzy type-ahead search in XML data and formalize the problem.

3.1 Overview

We first introduce how TASX works for queries with multiple keywords in XML data, by allowing minor errors of query keywords and inconsistencies in the data itself. Assume there is an underlying XML document that resides on a server. A user accesses and searches the data through a web browser. Each keystroke that the user types invokes a query, which includes the current string the user has typed in. The browser sends the query to the server, which computes and returns to the user the best answers ranked by their relevancy to the query.

The server first tokenizes the query string into several keywords using delimiters such as the space character. The keywords are assumed as *partial keywords*, as the user may have not finished typing the complete keywords. For the partial keywords, we would like to know the possible words the user intends to type. However, given the limited information, we can only identify a set of complete words in the data set which have *similar prefixes* with the partial keywords. This set of complete words are called the *predicted words*. We use edit distance to quantify the similarity between two words. The edit distance between two words s_1 and s_2, denoted by $\mathrm{ed}(s_1, s_2)$, is the minimum number of edit operations (i.e., insertion, deletion, and substitution) of single characters needed to transform the first one to the second. For example, $\mathrm{ed}(\mathrm{mics}, \mathrm{mices}) = 1$ and $\mathrm{ed}(\mathrm{mics}, \mathrm{mich}) = 1$. For instance, given a partial keyword "mics," its predicted words could be "mices," "mich," "michal," etc.

Then, the server identifies the relevant subtrees in XML data that contain the predicted words for every input keyword. We can use any existing semantics to identify the answer based on the predicted words, such as ELCA [19]. We call these relevant subtrees the *predicted answers* of the query. For example, consider the XML document in Fig. 1. Assume a user types in a keyword query "db mics." The predicted word of "db" is "db." The predicted words of "mics" are "mices" and "mich." The subtree rooted at node 12 is the predicted answer of "db mices." The subtree rooted at node 15 is the predicted answer of "db mich." Thus, TASX can save users time and efforts, since they can find the answers even if they have not finished typing all the complete keywords or typing keywords with minor errors.

3.2 Problem Formulation

We formalize the problem of fuzzy type-ahead search in XML data as follows:

Definition 1 (Fuzzy Type-Ahead Search in XML Data). *Given an XML document D, a keyword query $Q = \{k_1, k_2, \ldots, k_\ell\}$, and an edit-distance threshold τ. Let the predicted-word set be $W_{k_i} = \{w | w$ is a tokenized word in D and there exists a prefix of w, k_i', $\mathrm{ed}(k_i, k_i') \leq \tau\}$. Let the predicted-answer set be $R_Q = \{r | r$ is a keyword-search result of query $\{w_1 \in W_{k_1}, w_2 \in W_{k_2}, \ldots, w_\ell \in W_{k_\ell}\}\}$. For the keystroke that invokes Q, we return the top-k answers in R_Q for a given value k, ranked by their relevancy to Q.*

We treat the data and query string as lowercase strings. We will focus on how to efficiently find the predicted answers, among which we can find the best top-k relevant answers using a ranking function.

There are two challenges to support fuzzy type-ahead search in XML data. The first one is how to interactively and efficiently identify the *predicted words* that have prefixes *similar* to the input partial keyword after each keystroke from the user. The second one is how to progressively and effectively compute the top-k *predicted answers* of a query with multiple keywords, especially when there are many predicted words. We introduce effective index structures and incremental computing algorithms to address the first challenge (Section 4). We devise effective ranking functions, early termination techniques, efficient algorithms, and forward-index structures to address the second challenge (Section 5).

4 LCA-Based Fuzzy Type-Ahead Search

This section proposes an LCA-based fuzzy type-ahead search method. We use the semantics of ELCA [55] to identify relevant answers on top of predicted words.

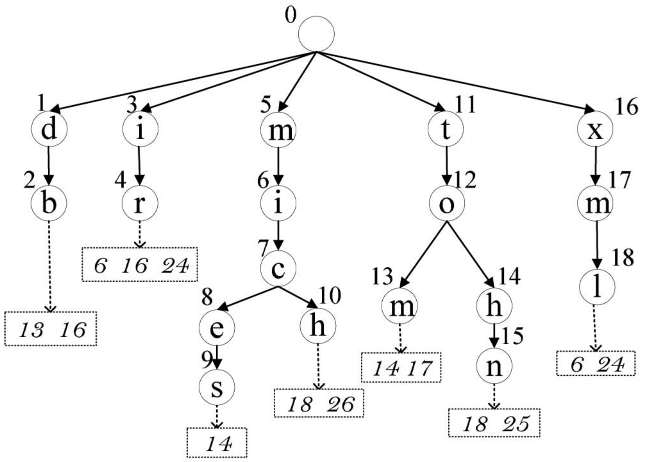


Fig. 2. The trie on top of words in Fig. 1 (a part of words).

4.1 Index Structures

We use a trie structure to index the words in the underlying XML data. Each word w corresponds to a unique path from the root of the trie to a leaf node. Each node on the path has a label of a character in w. For each leaf node, we store an inverted list of IDs of XML elements that contain the word of the leaf node. For instance, consider the XML document in Fig. 1. The trie structure for the tokenized words is shown in Fig. 2. The word "mich" has a node ID of 10. Its inverted list includes XML elements 18 and 26.

4.2 Answering Queries with a Single Keyword

We first study how to answer a query with a single keyword using the trie structure. Each keystroke that a user types invokes a query of the current string, and the client browser sends the query string to the server.

4.2.1 Exact Search

We first consider the case of exact search. One naive way to process such a query on the server is to answer the query from scratch as follows: we first find the trie node corresponding to this keyword by traversing the trie from the root. Then, we locate the leaf descendants of this node, and retrieve the corresponding predicted words and the predicted XML elements on the inverted lists.

For example, suppose a user types in query string "mich" letter by letter. When the user types in the character "m," the client sends the query "m" to the server. The server finds the trie node corresponding to this keyword (node 5). Then, it locates the leaf descendants of node 5 (nodes 9 and 10), and retrieves the corresponding predicted words ("mices" and "mich") and the predicted XML elements (elements 14, 18, and 26). When the user types in the character "i," the client sends a query string "mi" to the server. The server answers the query from scratch as follows: it first finds node 6 for this string, then locates the leaf descendants of node 6 (nodes 9 and 10). It retrieves the corresponding predicted words ("mices" and "mich"). Other queries invoked by keystrokes are processed in a similar way. One limitation of this method is that it involves a lot of recomputation without using the results of earlier queries.

We can use a caching-based method to incrementally find the trie node for the input keyword. We maintain a session for each user. Each session keeps the keywords that the user has typed in the past and the corresponding trie node. We use a hashtable to maintain such information. When a session times out, the kept information will be deleted. The goal of keeping the information is to use it answer subsequent queries incrementally as follows: assume a user has typed in a query string $c_1c_2 \dots c_x$ letter by letter. Let $p_i = c_1c_2 \dots c_i$ be a prefix query $(1 \leq i \leq x)$. Suppose n_i is the trie node corresponding to p_i. After the user types in a prefix query p_i, we store node n_i for p_i. For each keystroke the user types, for simplicity, we first assume that the user types in a new character c_{x+1} at the end of the previous query string and submit a new query $p_{x+1} = c_1c_2 \dots c_xc_{x+1}$. To incrementally answer the new query, we first check whether node n_x that has been kept for p_x has a child with a label of c_{x+1}. If so, we locate the leaf descendants of node n_{x+1}, and retrieve the corresponding predicted words. Otherwise, there is no word that has a prefix of p_{x+1}, and we can just return an empty answer.

For example, suppose a user has typed in "mic." After this query is submitted and processed, the server has stored node 5 for the prefix query "m," node 6 for the prefix query "mi," and node 7 for "mic." If the user types in "h" after "mic," we check whether node 7 kept for "mic" has a child with label "h." Here, we find node 10, and retrieve the corresponding predicted word "mich."

In general, the user may modify the previous query string arbitrarily, or copy and paste a completely different string. In this case, for the new query string, among all the keywords typed by the user, we identify the cached keyword that has the *longest* prefix with the new query. Then, we use this prefix to incrementally answer the new query, by inserting the characters after the longest prefix of the new query one by one.

4.2.2 Fuzzy Search

Obviously, for exact search, given a partial keyword, there exists at most one trie node for the keyword. We retrieve the leaf descendants of this trie node as the predicted words. However, for fuzzy search, there could be multiple trie nodes that are *similar* to the partial keyword within a given edit-distance threshold, called *active nodes*. For example, both nodes "mices" and "mich" on the trie in Fig. 2 are active nodes for "mics."

We can incrementally compute active nodes as follows: given a partial query $p_x = c_1c_2 \dots c_x$, suppose we have computed the active-node sets of p_x, $\mathcal{A}_{p_x}$. Then, for a new query $p_{x+1} = c_1c_2 \dots c_xc_{x+1}$, we use $\mathcal{A}_{p_x}$ to compute $\mathcal{A}_{p_{x+1}}$ as follows: for any active node $n \in \mathcal{A}_{p_x}$, its descendants could be similar to p_{x+1}, we consider three edit operations of insertion, deletion, and substitution to compute active nodes under node n and put them into $\mathcal{A}_{p_{x+1}}$ [27]. Given a null string ϕ, we initialize $\mathcal{A}_\phi = \{n|$ the level of node n is no larger than τ (the level of the root is 0)$\}$. Accordingly, we can incrementally compute $\mathcal{A}_{p_{x+1}}$ based on $\mathcal{A}_{p_x}$ [27]. To facilitate incremental computation, for each user, we use a session to maintain active nodes of each keyword using a hashtable.

Thus, given a partial keyword p_x, we first compute its active-node set $\mathcal{A}_{p_x}$. Then, for each active node $n \in \mathcal{A}_{p_x}$, we retrieve inverted lists of n's leaf descendants and compute the union of such inverted lists, denoted as U_n. Finally, we compute the union of U_n for $n \in A_{p_x}$, denoted as $\overline{U}_{p_x}$, i.e., $\overline{U}_{p_x} = \bigcup_{n \in A_{p_x}} U_n$. We call $\overline{U}_{p_x}$ the *union list* of p_x. Obviously, $\overline{U}_{k_i}$ is exactly the set of XML elements that have prefixes similar to k_i.

Example 1. Consider the trie structure in Fig. 2 and suppose the given edit-distance threshold $\tau = 1$. Assume a user types in a partial keyword "mics" letter by letter. We compute active-node sets as follows: first, we initialize $A_\phi = \{\phi(0), \mathrm{d}(1), \mathrm{i}(1), \mathrm{m}(1), \mathrm{t}(1), \mathrm{x}(1)\}$, where the number in the parenthesis denotes the edit distance between the active node and the partial keyword. Then, we incrementally compute $A_m = \{\phi(1), \mathrm{d}(1), \mathrm{i}(1), \mathrm{m}(0), \mathrm{mi}(1), \mathrm{t}(1), \mathrm{x}(1), \mathrm{xm}(1)\}$, $A_{\mathrm{mi}} = \{\mathrm{m}(1), \mathrm{mi}(0), \mathrm{mic}(1)\}$, $A_{\mathrm{mic}} = \{\mathrm{mi}(1), \mathrm{mic}(0), \mathrm{mice}(1), \mathrm{mich}(1)\}$, and $A_{\mathrm{mics}} = \{\mathrm{mic}(1), \mathrm{mices}(1), \mathrm{mich}(1)\}$. We traverse the subtrees of active nodes 7, 9, and 10, get predicted words "mices" and "mich," compute $U_{\mathrm{mices}} = \{14\}$, $U_{\mathrm{mich}} = \{18, 26\}$, and get the union list $\overline{U}_{\mathrm{mics}} = \{14, 18, 26\}$.

4.3 Answering Queries with Multiple Keywords

Now, we consider how to do fuzzy type-ahead search in the case of a query with multiple keywords. For a keystroke that invokes a query, we first tokenize the query string into keywords, $k_1, k_2, \ldots, k_\ell$. For each keyword k_i ($1 \leq i \leq \ell$), we compute its corresponding active nodes, and for each such active node, we retrieve its leaf descendants and corresponding inverted lists. Then, we compute union list $\overline{U}_{k_i}$ for every k_i as discussed in Section 4.2.[2] Finally, we compute the predicted answers on top of lists $\overline{U}_{k_1}, \overline{U}_{k_2}, \ldots, \overline{U}_{k_\ell}$.

We use the semantics of ELCA [55] to compute the corresponding answers. We use the binary-search-based method to compute ELCAs [55]. We will introduce an effective ranking function in considering fuzzy search in Section 5.2.

Example 2. Consider the trie structure in Fig. 2 and suppose the given edit-distance threshold $\tau = 1$. Assume a user types in a query "db mics" letter by letter. As the user types in the keyword "db," for each keystroke, we incrementally answer the query as discussed before. We identify predicted word "db" and compute the union list $\overline{U}_{db} = \{13, 16\}$. When the user types in "db mics," we find the active nodes $\{\mathrm{mic}(7), \mathrm{mices}(9), \mathrm{mich}(10)\}$, identify the predicted words of the active nodes, "mices" and "mich," and compute the union list $\overline{U}_{\mathrm{mics}} = \{14, 18, 26\}$. Then, we compute the ELCAs on top of the two union lists $\overline{U}_{\mathrm{db}}$ and $\overline{U}_{\mathrm{mics}}$, get the ELCAs (XML elements 12 and 15), and return the subtrees rooted at the two ELCAs. Accordingly, we can incrementally answer the keyword query "db mics."

5 Progressive and Effective Top-k Fuzzy Type-Ahead Search

The LCA-based fuzzy type-ahead search algorithm in XML data has two main limitations. First, they use the "AND" semantics between input keywords of a query, and ignore the answers that contain some of the query keywords (but not all the keywords). For example, suppose a user types in a keyword query "DB IR Tom" on the XML document in Fig. 1. The ELCAs to the query are nodes 15 and 5. Although node 12 does not have leaf nodes corresponding to all the three keywords, it might still be more relevant than node 5 that contains many irrelevant papers. Second, in order to compute the best results to a query, existing methods need find candidates first before ranking them, and this approach is not efficient for computing the best answers. A more efficient algorithm might be able to find the best answers without generating all candidates.

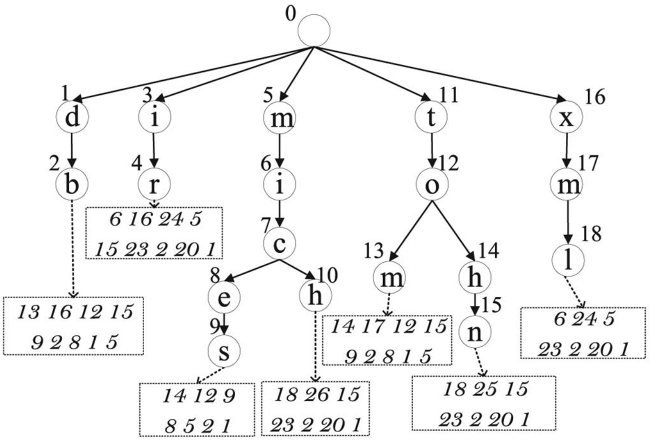


Fig. 3. The extended trie on top of words in Fig. 1 (a part of words).

To address these limitations, we develop novel ranking techniques and efficient search algorithms. In our approach, each node on the XML tree could be potentially relevant to a keyword query, and we use a ranking function to decide the best answers to the query. For each leaf node in the trie, we index not only the content nodes for the keyword of the leaf node, but also those quasi-content nodes whose descendants contain the keyword. For instance, consider the XML document in Fig. 1. For the keyword "DB," we index nodes 13, 16, 12, 15, 9, 2, 8, 1, and 5 for this keyword as shown in Fig. 3. For the keyword "IR," we index nodes 6, 16, 24, 5, 15, 23, 2, 20, and 1. For the keyword "Tom," we index nodes 14, 17, 12, 15, 9, 2, 8, 1, and 5. The nodes are sorted by their relevance to the keyword (we will discuss how to evaluate relevance of nodes to a keyword in Section 5.2.1). Fig. 3 gives the extended trie structure.

For instance, assume a user types in a keyword query "DB IR Tom." We use the extended trie structure to find nodes 15 and 12 as the top-2 relevant nodes. We propose *minimal-cost trees* (MCTs) to construct the answers rooted at nodes 15 and 12 (Section 5.1). We develop effective ranking techniques to rank XML elements on the inverted lists in the extended trie structure (Section 5.2). We can employ threshold-based algorithms [15] to progressively and efficiently identify the top-k relevant answers (Section 5.3). Moreover, our approach automatically supports the "OR" semantics.

5.1 Minimal-Cost Tree

In this section, we introduce a new framework to find relevant answers to a keyword query over an XML

[2] Note that if the user only modifies k_i, we only need to compute $\overline{U}_{k_i}$, as $\forall j \neq i$, $\overline{U}_{k_j}$ has been cached.

document. In the framework, each node on the XML tree is potentially relevant to the query with different scores. For each node, we define its corresponding answer to the query as its subtree with paths to nodes that include the query keywords. This subtree is called the "minimal-cost tree" for this node. Different nodes correspond to different answers to the query, and we will study how to quantify the relevance of each answer to the query for ranking (Section 5.2).

Consider an XML document D, a keyword k_i, and a content node or quasi-content node for k_i, n. Let P denote a subset of n's descendant nodes which are content nodes of k_i. $p \in P$ is a *pivotal node* for k_i and n, if node p has the minimal distance to node n among all nodes in P. The path from node n to a pivotal node is called the *pivotal path* of this pivotal node.[3] For example, consider the XML document in Fig. 1. Given a keyword "DB," node 9 is a quasi-content node for "DB." Node 13 is a pivotal node for node 9 and keyword "DB," and the path $n_9 \rightarrow n_{12} \rightarrow n_{13}$ is the corresponding pivotal path, where n_9, n_{12}, n_{13} denote nodes 9, 12, and 13, respectively. Intuitively, a pivotal node for node n and k_i is much more relevant to node n for k_i than other content nods. Thus, given a node n and a keyword query Q, we combine all pivotal paths as an answer of query Q. Now, we give a formal definition.

Given a keyword query, each node n in the XML document is potentially relevant to the query. We introduce the notion of *minimal-cost tree* rooted at node n to define the answer to the query.

Definition 2 (MINIMAL-COST TREES). *Given an XML document D, a node n in D, and a keyword query $Q = \{k_1, k_2, \ldots, k_\ell\}$, a minimal-cost tree of query Q and node n is the subtree rooted at n, and for each keyword $k_i \in Q$, if node n is a quasi-content node of k_i, the subtree includes the pivotal paths for k_i and node n.*

To answer a keyword query, we first identify the predicted words for each input keyword. Then, we construct the minimal-cost tree for every node in the XML tree based on the predicted words, and return the best ones with the highest scores. Later, we will discuss how to rank a minimal-cost tree in Section 5.2, and give progressive and efficient search algorithms in Section 5.3. Here to better understand our method, we give an example to show how to construct a minimal-cost tree as below.

Example 3. Consider the XML document in Fig. 1 and given a keyword query $Q = \{\texttt{DB}, \texttt{Tom}, \texttt{WWW}\}$. Nodes 3, 13, 14, 16, and 17 are content nodes of the three keywords; nodes 1, 2, 5, 8, 9, 12, and 15 are their quasi-content nodes. Node 3 is the pivotal node for node 2 and "WWW." Node 16 is the pivotal node for node 2 and "DB." Node 17 is the pivotal node for node 2 and "Tom." The MCT of node 2 is the subtree rooted at node 2, which contains the paths: $n_2 \rightarrow n_3$, $n_2 \rightarrow n_{15} \rightarrow n_{16}$, and $n_2 \rightarrow n_{15} \rightarrow n_{17}$.

The main advantage of this definition is that, even if a node does not have descendant nodes that include all the keywords in the query, this node could still be considered as a potential answer. In other words, this definition is relaxing the assumption in existing semantics that all the query keywords need to appear in the descendants of an answer node. As we will see in the next section, this definition still allows us to do effective indexing to answer queries efficiently.

5.2 Ranking Minimal-Cost Trees

In this section, we discuss how to rank a minimal-cost tree. We first introduce a ranking function for exact search in Section 5.2.1 and then extend the ranking function to support fuzzy search in Section 5.2.2.

5.2.1 Ranking for Exact Search

To rank a minimal-cost tree, we first evaluate the relevance between the root node and each input keyword, and then combine these relevance scores for every input keyword as the overall score of the minimal-cost tree. We propose two ranking functions to compute the relevance score between the root note n to an input keyword k_i. The first one considers the case that n contains k_i. The second one considers the case that n does not contain k_i but has a descendant containing k_i.

Our first ranking method models each node n as a document that includes the terms contained in the tag name or text values (#PCDATA) of n. We can then use the idea of TF/IDF in IR literature to score the relevance of node n to a keyword. Given an XML document D, a node $n \in D$, and a keyword k_i contained in n, we denote $tf(k_i, n)$ as the number of occurrences of k_i in the subtree rooted at n, $idf(k_i)$ as the inverse document frequency of k_i (i.e., the ratio of the number of nodes in the XML document to the number of nodes that contain k_i), and $ntl(n)$ as the normalized term length of n, i.e., $ntl(n) = \frac{|n|}{|n_{max}|}$, where $|n|$ denotes the number of terms contained in n and n_{max} denotes the node with the maximal number of terms.

If n contains k_i, we use existing ranking methods [40] to compute the relevance of node n to keyword k_i:

$$\mathrm{SCORE}_1(n, k_i) = \frac{ln\big(1 + tf(k_i, n)\big) * ln\big(idf(k_i)\big)}{(1 - s) + s * ntl(n)}. \qquad (1)$$

In the formula, s is a constant, which is widely studied in the information-retrieval community and usually set to 0.2 [39].

Example 4. Consider the XML document in Fig. 1 and a query $Q = \{\texttt{XML}, \texttt{IR}, \texttt{Tohn}\}$. For node 24, we have

$$\mathrm{SCORE}_1(n_{24}, \texttt{XML}) = \frac{ln(1+1) * ln(28/3)}{0.2 + 0.8} = 1.55$$

and $\mathrm{SCORE}_1(n_{24}, \texttt{IR}) = \frac{ln(1+1)*ln(28/4)}{0.2+0.8} = 1.35$. For node 25, we have

$$\mathrm{SCORE}_1(n_{25}, \texttt{Tohn}) = \frac{ln(1+1) * ln(28/3)}{0.2 + 0.8} = 1.55.$$

However, if n does not contain k_i, the first ranking function cannot quantify the relevancy between node n and keyword k_i. To address this issue, we extend the first ranking function and propose the second ranking function. Given a keyword k_j, a quasi-content node n for k_j, suppose p is the pivotal node for n and k_j. The distance between n

[3] In general, there can be more than one pivotal nodes for k_i and n.

and p can indicate how relevant the node n is to keyword k_j. The smaller the distance between n and p, the larger relevancy score between n and k_j should be. Based on this observation, we proposed the second ranking function to compute the relevance between n and k_j as follows:

$$\mathrm{SCORE}_2(n, k_j) = \sum_{p \in P} \alpha^{\delta(n,p)} * \mathrm{SCORE}_1(p, k_j), \qquad (2)$$

where P is the set of pivotal nodes for n and k_j, α is a damping factor between 0 and 1, and $\delta(n, p)$ denotes the distance between node n and node p. As the distance between n and p increases, n becomes less relevant to k_j. As a tradeoff, our experiments suggested that a good value for α is 0.8, and our method achieves the best performance at this point. This is because it will degrade the importance of ancestor nodes for a smaller α and thus may miss meaningful and relevant results; on the contrary, it will involve some duplicates and less important results for a larger α.

Based on the two ranking functions, given a query $Q = \{k_1, k_2, \ldots, k_\ell\}$ and a node n, we take the sum of the scores of node n to every k_i as the overall score of node n to Q:

$$\mathrm{SCORE}(n, Q) = \sum_{i=1}^{\ell} \mathrm{SCORE}(n, k_i), \qquad (3)$$

where $\mathrm{SCORE}(n, k_i)$ denotes the score of node n to keyword k_i. $\mathrm{SCORE}(n, k_i) = \mathrm{SCORE}_1(n, k_i)$ if n contains k_i; otherwise, $\mathrm{SCORE}(n, k_i) = \mathrm{SCORE}_2(n, k_i)$ if n has a descendant that contains keyword k_i. We give a running example to show how to compute the score of a minimal-cost tree.

Example 5. Recall EXAMPLE 4, given a keyword query $Q = \{\texttt{XML}, \texttt{IR}, \texttt{Tohn}\}$, for node 23, we have

$$\begin{aligned} \mathrm{SCORE}(n_{23}, \texttt{XML}) &= \alpha^1 * \mathrm{SCORE}(n_{24}, \texttt{XML}) = 0.8 * 1.55 \\ &= 1.24; \mathrm{SCORE}(n_{23}, \texttt{IR}) \\ &= \alpha^1 * \mathrm{SCORE}(n_{24}, \texttt{IR}) = 0.8 * 1.35 \\ &= 1.08; \mathrm{SCORE}(n_{23}, \texttt{Tohn}) \\ &= \alpha^1 * \mathrm{SCORE}(n_{25}, \texttt{Tohn}) = 0.8 * 1.55 \\ &= 1.24; \mathrm{SCORE}(n_{23}, Q) = 1.08 + 1.24 \\ &\quad + 1.24 = 3.56. \end{aligned}$$

5.2.2 Ranking for Fuzzy Search

Given a keyword query $Q = \{k_1, k_2, \ldots, k_\ell\}$, in terms of fuzzy search, a minimal-cost tree may not contain the exact input keywords, but contain predicted words for each keyword. Consider a minimal-cost tree rooted at n, suppose the predicted words for every input keyword in the subtree are $\{w_1, w_2, \ldots, w_\ell\}$. We propose how to quantify the similarity between k_i and w_i as follows: as k_i may be a partial keyword and users may type in more letters and complete the keyword, k_i could be similar to prefixes of w_i. The prefix of w_i which has the minimal edit distance to k_i among all the prefixes is called the *best similar prefix* for k_i and w_i, denoted as a_i. Intuitively, the best similar prefix of w_i could be considered to be most similar to k_i. For example, suppose k_i = "mics" and w_i = "miceslucy". The best similar prefix is a_i = "mices". We use the best similar prefix to quantify the similarity between k_i and w_i, and incorporate the similarity into ranking functions in order to support fuzzy search. Moreover, we will highlight the best similar prefixes in the answer such as "**mices**" for keyword "**mics**" and predicted word "**miceslucy**."

Intuitively, the smaller edit distance between k_i and a_i, w_i is more relevant to k_i. In addition, as a_i is a prefix of w_i, we use $\frac{|a_i|}{|w_i|}$ to quantify their similarity. Thus, we propose a new function to quantify similarity between k_i and w_i:

$$sim(k_i, w_i) = \gamma * \frac{1}{1 + \mathrm{ed}(k_i, a_i)^2} + (1 - \gamma) * \frac{|a_i|}{|w_i|}, \qquad (4)$$

where γ is a tuning parameter between 0 and 1. As the former is more important, γ is close to 1. Our experiments suggested that a good value for γ is 0.95, and our method achieves the best performance at this point. We extend the ranking function in (3) by incorporating this similarity function to support fuzzy search as below

$$\mathrm{SCORE}(n, Q) = \sum_{i=1}^{\ell} sim(k_i, w_i) * \mathrm{SCORE}(n, w_i). \qquad (5)$$

5.3 Progressively Finding Top-k Minimal-Cost Trees

In this section, we propose how to progressively find the top-k relevant minimal-cost trees. In the trie index, for each leaf trie node, we keep the content nodes and quasi-content nodes in the XML document as shown in Fig. 3, and corresponding scores and pivotal paths for the keyword of the leaf node, sorted by the relevancy to the keyword.[4]

Given a keyword query $Q = \{k_1, k_2, \ldots, k_\ell\}$, for each partial keyword k_i, we first compute its predicted words as discussed in Section 4. Then, we compute the union of inverted lists of k_i's predicted words, $\overline{U}_{k_i}$, sorted by corresponding scores (i.e., $\mathrm{SCORE}(n, w_i)$). Then, we can use existing threshold-based NRA algorithm [16] to progressively and efficiently compute the top-k answers of on top of every $\overline{U}_{k_i}$. To better understand our method, we give a running example to describe how to compute the top-k answers.

Example 6. Consider the XML document in Fig. 1 and suppose the edit-distance threshold $\tau = 1$. Assume a user types in a keyword query "db mics." We first identify active node 2 for "db" and active nodes 7, 8, and 10 for "mics" as discussed in Section 4. Then, we compute the union list of "db," $\overline{U}_{\mathrm{db}} = \{13, 16, 12, 15, 9, 2, 8, 1, 5\}$, and the union list of "mics," $\overline{U}_{\mathrm{mics}} = \{14, 18, 12, 26, 9, 15, 8, 23, 5, 2, 20, 1\}$. Next, we use the threshold-based NRA algorithm on top of the two sorted union lists to compute the top-k answers. Assume we want to identify top-2 answers. We get nodes 12 and 15 by accessing some elements on the two union lists and construct minimal-cost subtrees rooted at them to answer the query.

Notice that it is very expensive to construct the union lists of every input keyword as there may be multiple predicted words and many inverted lists. Instead, we can

[4] We omit the scores and pivotal paths in the figure.

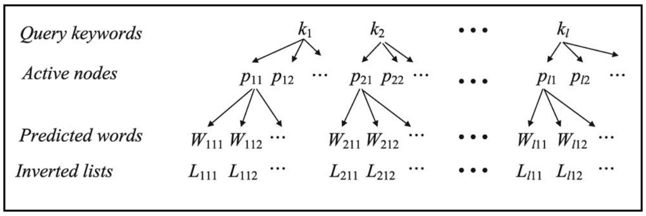


Fig. 4. Active nodes, predicted words, and corresponding inverted lists for query $Q = \{k_1, k_2, \ldots, k_\ell\}$.

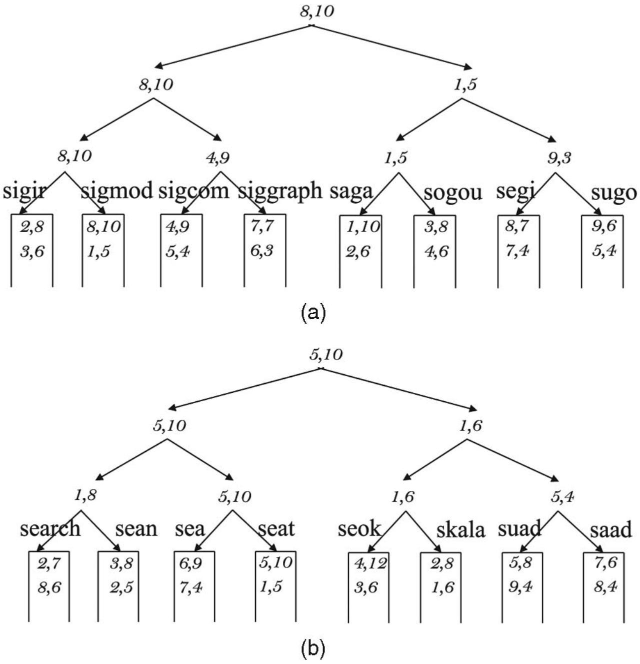
generate a *partial virtual list* on the fly. We only use the elements in the partial virtual list to compute the top-k answers. The partial virtual list can avoid accessing all the elements of inverted lists of predicted words. It only needs to access those with higher scores, and if we have computed the top-k answers using the partial accessed elements, we can do an early termination and do not need to visit other elements on the inverted lists.

For ease of presentation, we first introduce some notations. Given a keyword query $Q = \{k_1, k_2, \ldots, k_\ell\}$, for each partial keyword k_i, let $A_{p_{k_i}} = \{p_{i1}, p_{i2}, \ldots\}$ denote its active-node set. For each active node p_{ij}, let $W_{p_{ij}} = \{w_{ij1}, w_{ij2}, \ldots\}$ denote its leaf-descendant set and $L_{p_{ij}} = \{L_{ij1}, L_{ij2}, \ldots\}$ denote the corresponding inverted-list set as illustrated in Fig. 4. Let $U_{p_{ij}} = \cup_{L' \in L_{p_{ij}}} L'$.

For each keyword k_i, consider one of k_i's predicted words w_i and the corresponding inverted list L_i. For each XML element n on list L_i, we compute its score to k_i: $\mathrm{SCORE}(n, k_i) = sim(k_i, w_i) * \mathrm{SCORE}(n, w_i)$. We can construct a max heap for the elements on every inverted list of k_i's predicted words where scores are computed using this function. The top element on each max heap has the maximal relevancy score to k_i. We use the threshold-based NRA algorithm [15] to identify the top-k answers on top of the top elements of every max heap. When deleting the top element, we adjust the max heap to generate the next element with the largest score to k_i. The time complexity of constructing the max heap for k_i is $O(\varpi_{k_i})$ [3], where ϖ_{k_i} denotes the number of inverted lists of k_i's predicted words. The time complexity of deleting the top element and adjusting the max heap is $O\big(log(\varpi_{k_i})\big)$. When accessing elements in the inverted lists, we do an early termination once we get top-k answers. Note that we only need to construct a virtual list on the fly.

Example 7. Assume a user types in a query "sig sea." Suppose the predicted words and inverted lists for the two keywords are illustrated in Fig. 5. We need not construct the union lists for the two keywords on the fly. Instead, we build two max heaps as illustrated in Fig. 5. In the similarity function (4), we set $\gamma = 1$ for simplicity in the running example. Consider the two predicted words "saga" and "sogou," the best similar prefixes for keyword "sig" are "sag" and "sog," respectively. For element 1 on the inverted list of "saga," we have $\mathrm{SCORE}(1, \mathrm{sig}) = \mathrm{sim}(\mathrm{sig}, \mathrm{sag}) * \mathrm{SCORE}(1, \mathrm{saga}) = 5.$ Thus, the maximal score of the first elements of "saga" and "sogou" is 5. Similarly, we can construct the two max heaps for the two keywords.

Then, suppose we want to compute the top-3 answers. We first pop the top elements of the two max heaps $\langle 8, 10\rangle$ and $\langle 5, 10\rangle$, where 8 is an element and 10 is the corresponding score, and get a threshold 20. Then, we delete the two top elements, adjust the max heaps, and get the next top elements $\langle 4, 9\rangle$ and $\langle 6, 9\rangle$. When we have visited elements $\{\langle 8, 10\rangle; \langle 4, 9\rangle; \langle 2, 8\rangle; \langle 7, 7\rangle; \langle 3, 6\rangle\}$ and $\{\langle 5, 10\rangle; \langle 6, 9\rangle; \langle 3, 8\rangle; \langle 2, 7\rangle; \langle 8, 6\rangle\}$, we can get the top-3 answers $\langle 8, 16\rangle$, $\langle 2, 15\rangle$, and $\langle 3, 14\rangle$.



(a)

(b)

Fig. 5. An example for answering query "sig sea" using the max heap. (a) The max heap for "sig." (b) The max heap for "sea."

5.4 Improvement Using Forward Index

In this section, we propose the forward index to improve search performance. We can utilize "random access" based on the forward index to do an early termination in the algorithms. That is, given an XML element and an input keyword, we can get the corresponding score of the keyword and the element using the forward index, without accessing inverted lists. Fagin et al. have proved that the threshold-based algorithm using random access is optimal over all algorithms that correctly find the top k answers [15]. Thus, in this section, we propose a forward index to implement random access.

Given an XML element e, we construct a trie structure to maintain the keywords contained in the element as discussed in Section 4.1. Each leaf node in the forward index keeps the score of element e to the corresponding word of the leaf node. Thus, given any partial keyword, we can efficiently check whether e contains a word having prefixes similar to the keyword using the forward index as discussed in Section 5.3.[5] In this way, we can use the forward index for random access [15] and employ the TA algorithm to progressively identify the top-k answers, instead of using the NRA algorithm [15].

Given an XML element, if there is a large number of keywords under the element, the forward index of this element will be large, and it is expensive to maintain the forward index and find similar words from the forward

[5] We can also get the corresponding score.

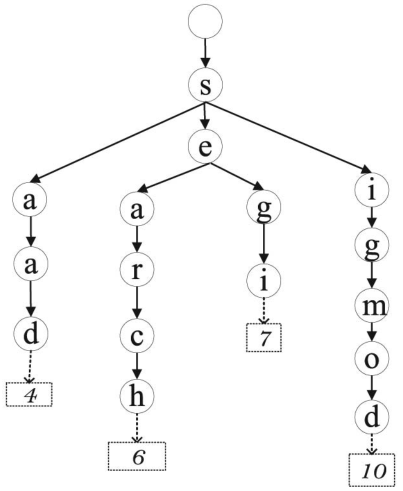


Fig. 6. The forward index for element 8 in Fig. 5. Element 8 contains keywords "`saad`," "`search`," "`segi`," and "`sigmod`."

index. We employ a cost-based method to select forward index for materialization. The time complexity of sorted access is $O(1)$ and that of random access is $O(\tau * AN)$, where τ is the edit-distance threshold and AN is the number of active nodes [27]. Suppose the average number of active nodes is A and the average inverted-list length is I. If $\tau * A > I$, we will not maintain the forward index, since we can only use sorted access to scan the inverted lists.

Example 8. Recall EXAMPLE 7, suppose we have built the forward index for element 8 (in Fig. 5) as shown in Fig. 6, where element 8 contains keywords "`saad`," "`search`," "`segi`," and "`sigmod`." We show how to identify the top-3 answers for query "`sig sea`." When we get the element 8 from the max heap of "`sig`," we can use its forward index to check whether it contains keywords that have prefixes similar to "`sea`." We find "`search`" and get its corresponding score 6. Thus, the score of element 8 is $10 + 6 = 16$. Similarly, we can get the scores of other elements. We only need access elements $\{\langle 8, 10\rangle;$ $\langle 4, 9\rangle$; $\langle 2, 8\rangle$; $\langle 7, 7\rangle\}$ for "`sig`," and $\{\langle 5, 10\rangle$; $\langle 6, 9\rangle$; $\langle 3, 8\rangle$; $\langle 2, 7\rangle\}$ for "`sea`." We need not access $\langle 3, 6\rangle$ for "`sig`" and $\langle 8, 6\rangle$ for "`sea`." Thus, the forward index can avoid accessing unnecessary elements. Similarly, we can use forward index to improve performance.

6 Experimental Study

We have implemented our method on real applications using our proposed techniques. We employed the data sets DBLP[6] and XMark.[7] The sizes of DBLP and XMark were 510 and 113 MB, respectively. We randomly selected 100 queries for each data set and Table 1 gives some sample queries.

We implemented the hybrid algorithm of XRANK [19] for the LCA-based method. We used the Dewey inverted list and hash index. We implemented XRANK's ranking functions. We used the cache for incremental computation. We set up a server using Apache[8] and FastCgi.[9] The server was running a program implemented in C++ and compiled with the GNU C++ compiler. We used Ajax and JavaScript to allow the client browser to interact with the server and display the results. We conducted the evaluation on a PC running a Ubuntu operating system with an Intel(R) Xeon(R) CPU X5450@3.00 GHz CPU and 4 GB RAM.

TABLE 1
The Selected Queries on DBLP Data Set

IDs	Queries	Typed Queries
Q_1	xml keyword search	xml keyw sear
Q_2	faloutsos similarity icde	*falot* simil icd
Q_3	nick koudas approximate	nic *koua* appr
Q_4	approximate string divesh	appro str *diva*
Q_5	keyword search sigmod	keyw sear sigm
Q_6	interactive vldb 2006	intera vld 20
Q_7	keyword search papakonstantinou	keyw sear papa
Q_8	schema xquery jagadish	sche xque jag
Q_9	xrank search shanmugasundaram	xran *shen*
Q_{10}	xsearch vldb cohen	xse vld coh

Table 2 shows the data set sizes, trie-index sizes, forward-index sizes, and index-construction time. As XMark contains many more distinct keywords than DBLP, the index size on XMark is larger than that on DBLP. We implemented cache-based algorithms. We used the cold cache. For exact search, we cached the corresponding trie node for each keyword; for fuzzy search, we cached similar trie nodes of each keyword. The number of similar trie nodes of a keyword is usually small; thus, the cache size is not large. In our experiments, for each user, the cache size is about 0.5 to 2 KB.

6.1 Result Quality

This section evaluates result quality of the LCA-based method and MCT-based method. We evaluate query results by human judgement. As XMark data set captures more complicated structures than DBLP data set, we used XMark data set and generated 100 keyword queries. Answer relevance of the selected queries was judged from discussions of researchers in our database group. As users are usually interested in the top-k answers, we employed the top-k precision, i.e., the ratio of the number of answers deemed to be relevant in the first k results to k, to compare the LCA-based method and the MCT-based method. Table 3 shows the average top-k precision of the selected 100 queries. We see that our MCT-based search method achieves much higher result quality than the LCA-based method. This is attributed to our effective ranking functions that rank both content nodes and quasi-content nodes and incorporate structural information into our ranking functions.

TABLE 2
Data Sets and Index Costs

Dataset	DBLP	XMark
Dataset sizes	510 MB	113 MB
Trie-index sizes	36 MB	56 MB
Inverted-list sizes	52 MB	88 MB
Forward-index sizes	66 MB	95 MB
Index-construction time	54 secs	75 secs
Index sizes of XRANK	217 MB	307 MB

[6] http://dblp.uni-trier.de/xml/.
[7] http://monetdb.cwi.nl/xml/.
[8] http://www.apache.org/.
[9] http://www.fastcgi.com/.

TABLE 3
Top-k Precision by Human Judgement

Precision (%)	Top-1	Top-10	Top-50	Top-100
LCA	52	70	62	64
MCT	87	91	88	92

6.2 Server Running Time

This section evaluates the server running time. We first compared the LCA-based method and the MCT-based progressive search method in Section 6.2.1. Then, we evaluated the effectiveness of using max heap and forward index in Section 6.2.2.

6.2.1 LCA-Based Method versus MCT-Based Method

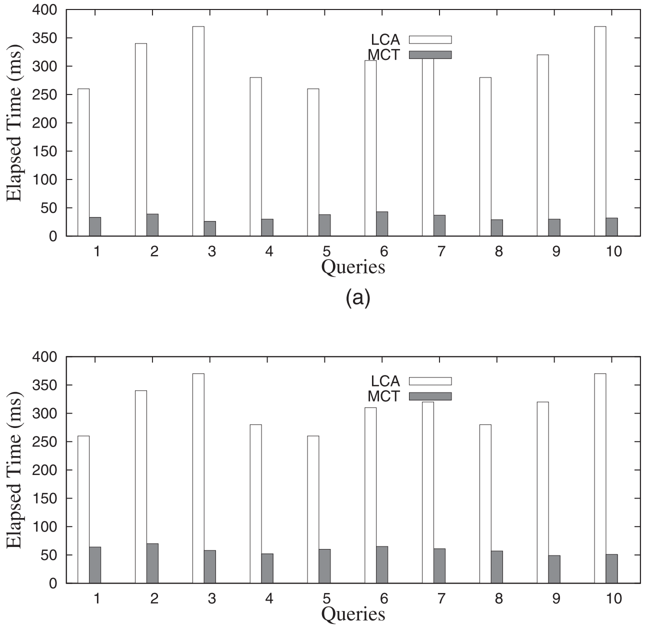
This section compares search efficiency of the LCA-based method and the MCT-based progressive search method. We used the 10 queries on DBLP data set as shown in Table 1. For each query, we measured the running time on the server to find the top-100 answers, which included the time to find predicted words of each keyword and the time to find the relevant subtrees of queries. We used Q_{10} as an example, and the results are shown Table 4. Fig. 7 gives the total server time for different queries.

We observe that the MCT-based search method is better than the LCA-based method and achieves much higher search performance in terms of both exact search and fuzzy search. This is attributed to our effective index structures and threshold-based computing algorithms.

6.2.2 Using Max Heap and Forward Index

This section evaluates the effectiveness of our proposed techniques using max heap and the forward index. We selected 1,000 queries for each data set and made some minor errors for some keywords in the queries. Fig. 8 illustrates the experimental results.

We observe that the heap-based method outperforms the MCT-based method. We need not construct the union lists of every input keyword on the fly, which could be very expensive for large numbers of predicted words. In addition, the forward index can improve search performance, because it can avoid accessing many unnecessary elements on inverted lists. Notice that the search time on XMark data set is a bit larger than that on DBLP data set. This is because the structure of XMark data set is more complicated than that of DBLP data set.



(a)

(b)

Fig. 7. Search time (LCA versus MCT). (a) Exact search. (b) Fuzzy search (edit-distance threshold $\tau = 2$).

6.3 Round-Trip Time

Since different locations can have different network delays to a server, we want to know whether our techniques can support an interactive speed for users from different locations. For this purpose, we asked several colleagues from different countries to access our DBLP prototype server at Tsinghua University, China, and issue the queries shown in Table 1. The places included China, US, and Australia. For each location, we measured the round-trip time from the time the user typed in a letter to the time the results are displayed on the browser. This time includes the network delay, query-execution time on the server, and JavaScript time on the client browser. We employed the heap-based method and used the forward index for progressive search. The experimental results are shown in Fig. 9.

The results show that the server running time is always less than 1/3 of the total round-trip time. The JavaScript program on the browser took about 40 to 60 ms. The more data returned to the user, the more time the JavaScript program needs to display the data. The relative low speed was due to the fact that the JavaScript program needs to be

TABLE 4
Running Time: Finding Predicted Words and Predicted Answers

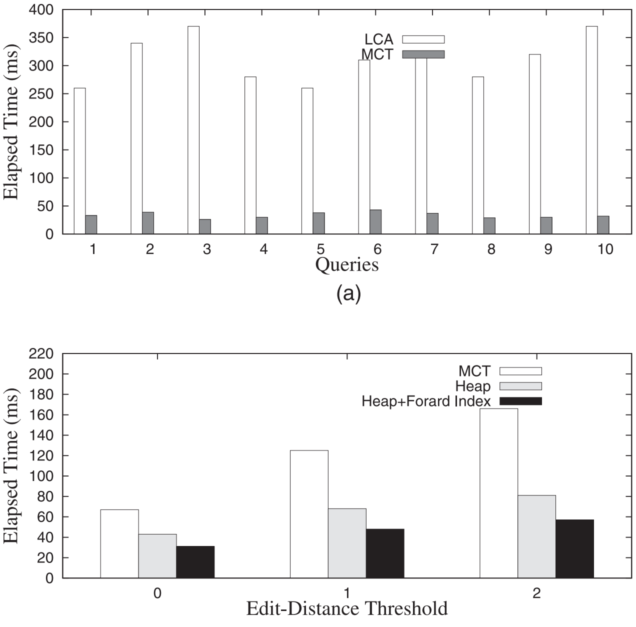
Time (ms) / Query	xs	xse	xse vl	xse vld	xse vld co	xse vld coh
Find predicted words	12	7	10	5	17	4
Find top-100 predicted answers (LCA)	90	70	320	301	290	268
Find top-100 predicted answers (MCT)	4	3	45	40	48	43

(a)

Time (ms) / Query	xs	xse	xse vl	xse vld	xse vld co	xse vld coh
Find predicted words	18	13	16	9	25	8
Find top-100 predicted answers (LCA)	134	102	432	410	370	341
Find top-100 predicted answers (MCT)	8	7	58	49	71	63

(b)

(a) Exact search. (b) Fuzzy search (edit-distance threshold $\tau = 2$).



(a)

(b)

Fig. 8. Search time (using Max heap and Forward Index). (a) DBLP data set. (b) XMark data set.

interpreted by the browser. The network delay depends on the location of the user. For example, for the user from China, the network delay was about 40 ms, which is about 1/3 of the total round-trip time. For all the users from different locations, the total round-trip time for a query was always below 350 ms, and all of them experienced an indeed interactive interface. For large-scale systems processing queries from different countries, we can solve the possible network-delay problem by using distributed data centers.

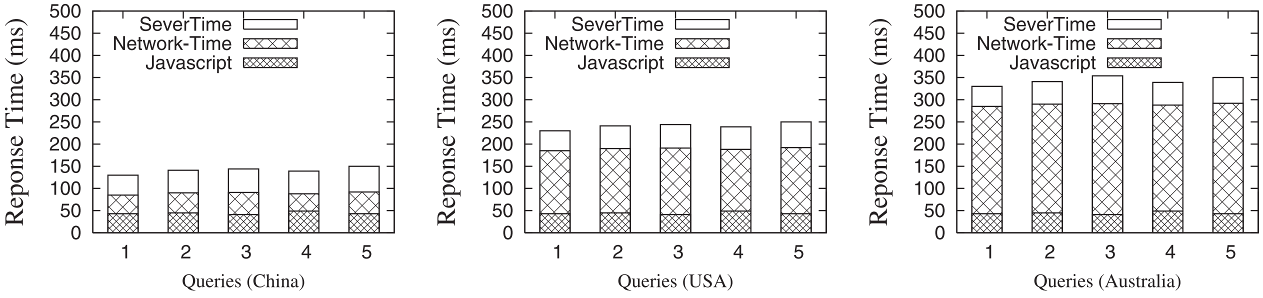
6.4 Scalability

This section evaluates the scalability of our algorithms. As an example, we used the DBLP data set. We varied the number of publications in the data set from 100K, 200K to one million. Fig. 10 shows the elapsed time of building the index structure, the sizes of indexes, and the average search time for 100 queries.

We observe that our method scales very well with the increase of the data set. In particular, the size of the trie is sublinear with the number of records. With the increase of the data sizes, the average search time also increased sublinearly. This is because of two main reasons. First, the time of finding the predicted words depends on the number of nodes on the trie, which increases sublinearly as the data size increases. Second, our method to incrementally compute the predicted words and progressively identify the predicted answers can save a lot of computation.

7 Related Work

Keyword search in XML data has attracted great attention recently. Xu and Papakonstantinou [54] proposed smallest lowest common ancestor (SLCA) to improve search efficiency. Sun et al. [49] studied multiway SLCA-based keyword search to enhance search performance. Schema-free XQuery [37] employed the idea of meaningful LCA, and proposed a stack-based sort-merge algorithm by considering XML structures and incorporating a new function mlcas into XQuery. XSEarch [12] focuses on the semantics and the ranking of the results, and extends keyword search. It employs the semantics of meaningful relation between XML nodes to answer keyword queries, and two nodes are meaningfully related if they are in a

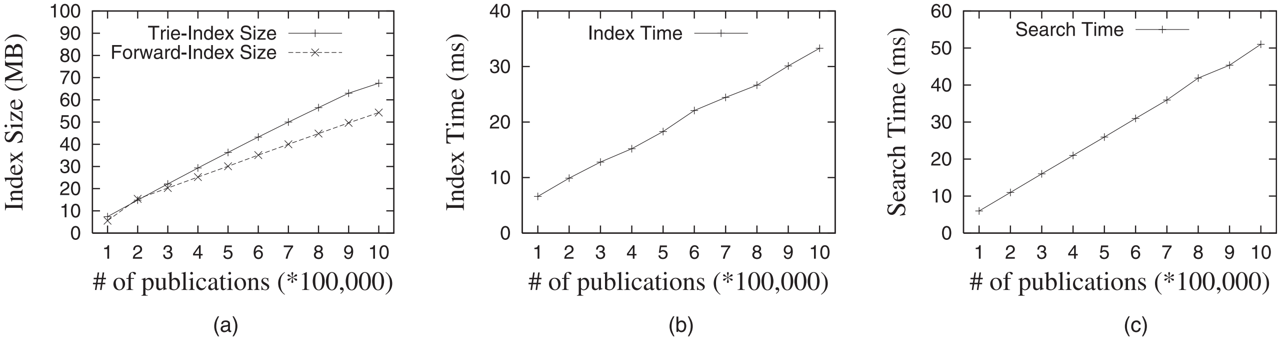


(a)

(b)

(c)

Fig. 9. Round-trip time for different locations.



(a)

(b)

(c)

Fig. 10. Scalability on DBLP data set by varying numbers of selected publications, 100K, 200K, . . . , 1M. (a) Index size. (b) Index time. (c) Search time.

same set, which can be given by administrators or users. Li et al. [32] proposed valuable LCA (VLCA) to improve the meaningfulness and completeness of answers and devised a new efficient algorithm to identify the answers based on a stack-based algorithm. XKeyword [25] is proposed to offer keyword proximity search over XML documents, which models XML documents as graphs by considering IDREFs between XML elements. Hristidis et al. [23] proposed grouped distance minimum connecting tree (GDMCT) to answer keyword queries, which groups the relevant subtrees to answer keyword queries. It first identifies the minimum connected tree, which is a subtree with minimum number of edges, and then groups such trees to answer keyword queries. Shao et al. [48] studied the problem of keyword search on XML views. XSeek [40] studied how to infer the most relevant return nodes without elicitation of user preferences. Liu and Chen [41] proposed to reason and identify the most relevant answers. Huang et al. [26] discussed how to generate snippets of XML keyword queries. Bao et al. [5] proposed to address the ambiguous problem of XML keyword search through studying *search for* and *search via* nodes. Different from [35], we extended it to support fuzzy type-ahead search in XML data.

In addition, the database research community has recently studied the problem of keyword search in relational databases [1], [24], [8], [22], [4], [39], [42], [43], graph databases [28], [21], [14], and heterogenous data sources [36]. DISCOVER-I [24], DISCOVER-II [22], BANKS-I [8], BANKS-II [28], and DBXplorer [1] are recent systems to answer keyword queries in relational databases. DISCOVER and DBXplorer return the trees of tuples connected by primary-foreign-key relationships that contain all query keywords. DISCOVER-II extended DISCOVER to support keyword proximity search in terms of disjunctive (OR) semantics, different from DISCOVER which only considers the conjunctive (AND) semantics. BANKS proposed to use Steiner trees to answer keyword queries. It first modeled relational data as a graph where nodes are tuples and edges are foreign keys, and then found Steiner trees in the graph as answers using an approximation to the Steiner tree problem, which is proven to be an NP-hard problem. BANKS-II improved BANKS-I by using bidirectional expansion on graphs to find answers. He et al. [21] proposed a partition-based method to efficiently find Steiner trees using the BLINKS index. Ding et al. [14] proposed to use dynamic programming for identifying Steiner trees. Dalvi et al. [13] studied disk-based algorithms for keyword search on large graphs, using a new concept of "supernode graph."

More recently, Kimelfeld and Sagiv [29] discussed keyword proximity search in relational databases from theory viewpoint. They showed that the answer of keyword proximity search can be enumerated in ranked order with polynomial delay under data complexity. Golenberg et al. [17] presented an incremental algorithm for enumerating subtrees in an approximate order which runs with polynomial delay and can find all top-k answers. Markowetz et al. [44] studied the problem of keyword search on relational data streams. They proposed several optimization techniques using mesh to answer keyword queries over streams. Guo et al. [18] studied the problem of data topology search on biological databases. Sayyadian et al. [47] incorporated schema mapping into keyword search and proposed a new method to answer keyword search across heterogenous databases. Liu et al. [39] incorporated IR ranking techniques to rank answers on relational data. They employed the techniques of phrase-based and concept-based models to improve result quality. Luo et al. [42] proposed a new ranking method that adapts state-of-the-art IR ranking functions and principles into ranking tree-structured results composed of joined database tuples. They incorporated the idea of skyline to rank answers. Balmin et al. proposed Object-Rank [4] to improve results quality by extending hub-and-authority [30] ranking-based method. This method is effective in ranking objects, pages, and entities, but it may cannot effectively rank tree-structured results (e.g., Steiner trees), since it does not consider structure compactness of an answer in its ranking function. Richardson and Domingos [46] proposed to combine page content and link structure to answer queries.

Tao and Yu [50] proposed to find co-occurring terms of query keywords in addition to the answers, in order to provide users relevant information to refine the answers. Koutrika et al. [31] proposed data clouds over structured data to summarize the results of keyword searches over structured data and use them to guide users to refine searches. Zhang et al. [57] and Felipe et al. [16] studied keyword search on spatial databases by combining inverted lists and R-tree indexes. Tran et al. [51] studied top-k keyword search on RDF data using summarized RDF graph. Qin et al. [45] studied three different semantics of m-keyword queries, namely, connect-tree semantics, distinct core semantics, and distinct root semantics, to answer keyword queries in relation databases. The search efficiency is achieved by new tuple reduction approaches that prune unnecessary tuples in relations effectively followed by processing the final results over the reduced relations. Chu et al. [10] proposed to combine forms and keyword search, and studied effective summary techniques to design forms. Yu et al. [56] and Vu et al. [52] studied keyword search over multiple databases in P2P environment. They emphasized on how to select relevant database sources in P2P environments. Chen et al. [9] gave an excellent tutorial of keyword search in XML data and relational databases. The recent integration of DB and IR was reported in [2], [7], [53].

Type-ahead search is a new topic to query relational databases. Li et al. [34] studied type-ahead search in relational databases, which allows searching on the underlying relational databases on the fly as users type in query keywords. Ji et al. [27] studied fuzzy type-ahead search on a set of tuples/documents, which can on the fly find relevant answers by allowing minor errors between input keywords and the underlying data. A straightforward method for type-ahead search in XML data is to first find all predicted words, and then use existing search semantics, e.g., LCA and ELCA, to compute relevant answers based on the predicted words. However, this method is very time consuming for finding top-k answers. To address this problem, we propose to progressively find the most relevant answers. For exact search, we propose to incrementally compute predicted words. For fuzzy search, we use existing techniques [27] to compute predicted words of query keywords. We extend the ranking functions in [34] to support fuzzy search, and propose new index structures and efficient algorithms to progressively find the most relevant answers.

This paper extended the poster paper [33] by adding efficient algorithms and ranking techniques to support fuzzy search.

8 Conclusion

In this paper, we studied the problem of fuzzy type-ahead search in XML data. We proposed effective index structures, efficient algorithms, and novel optimization techniques to progressively and efficiently identify the top-k answers. We examined the LCA-based method to interactively identify the predicted answers. We have developed a minimal-cost-tree-based search method to efficiently and progressively identify the most relevant answers. We proposed a heap-based method to avoid constructing union lists on the fly. We devised a forward-index structure to further improve search performance. We have implemented our method, and the experimental results show that our method achieves high search efficiency and result quality.

Acknowledgments

This work was partly supported by the National Natural Science Foundation of China under Grant No. 61003004 and No. 60873065, the National Grand Fundamental Research 973 Program of China under Grant No. 2011CB302206, National S&T Major Project of China under Grant No. 2011ZX01042-001-002, a project of Tsinghua University under Grant No. 20111081073, and the "NExT Research Center" funded by MDA, Singapore, under the Grant No. WBS:R-252-300-001-490.

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] S. Amer-Yahia, D. Hiemstra, T. Roelleke, D. Srivastava, and G. Weikum, "Db&ir Integration: Report on the Dagstuhl Seminar 'Ranked Xml Querying'," *SIGMOD Record,* vol. 37, no. 3, pp. 46-49, 2008.

[3] M.D. Atkinson, J.-R. Sack, N. Santoro, and T. Strothotte, "Min-max Heaps and Generalized Priority Queues," *Comm. ACM,* vol. 29, no. 10, pp. 996-1000, 1986.

[4] A. Balmin, V. Hristidis, and Y. Papakonstantinou, "Objectrank: Authority-Based Keyword Search in Databases," *Proc. Int'l Conf. Very Large Data Bases (VLDB),* pp. 564-575, 2004.

[5] Z. Bao, T.W. Ling, B. Chen, and J. Lu, "Effective XML Keyword Search with Relevance Oriented Ranking," *Proc. Int'l Conf. Data Eng. (ICDE),* 2009.

[6] H. Bast and I. Weber, "Type Less, Find More: Fast Autocompletion Search with a Succinct Index," *Proc. Ann. Int'l ACM SIGIR Conf. Research and Development in Information Retrieval (SIGIR),* pp. 364-371, 2006.

[7] H. Bast and I. Weber, "The Completesearch Engine: Interactive, Efficient, and towards Ir&db Integration," *Proc. Biennial Conf. Innovative Data Systems Research (CIDR),* pp. 88-95, 2007.

[8] 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.

[9] Y. Chen, W. Wang, Z. Liu, and X. Lin, "Keyword Search on Structured and Semi-Structured Data," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 1005-1010, 2009.

[10] E. Chu, A. Baid, X. Chai, A. Doan, and J.F. Naughton, "Combining Keyword Search and Forms for Ad Hoc Querying of Databases," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 349-360, 2009.

[11] S. Cohen, Y. Kanza, B. Kimelfeld, and Y. Sagiv, "Interconnection Semantics for Keyword Search in Xml," *Proc. Int'l Conf. Information and Knowledge Management (CIKM),* pp. 389-396, 2005.

[12] 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),* pp. 45-56, 2003.

[13] B.B. Dalvi, M. Kshirsagar, and S. Sudarshan, "Keyword Search on External Memory Data Graphs," *Proc. Int'l Conf. Very Large Data Bases (VLDB),* pp. 1189-1204, 2008.

[14] 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),* pp. 836-845, 2007.

[15] R. Fagin, A. Lotem, and M. Naor, "Optimal Aggregation Algorithms for Middleware," *Proc. ACM SIGMOD-SIGACT-SIGART Symp. Principles of Database Systems (PODS),* 2001.

[16] I.D. Felipe, V. Hristidis, and N. Rishe, "Keyword Search on Spatial Databases," *Proc. Int'l Conf. Data Eng. (ICDE),* pp. 656-665, 2008.

[17] K. Golenberg, B. Kimelfeld, and Y. Sagiv, "Keyword Proximity Search in Complex Data Graphs," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 927-940, 2008.

[18] L. Guo, J. Shanmugasundaram, and G. Yona, "Topology Search over Biological Databases," *Proc. Int'l Conf. Data Eng. (ICDE),* pp. 556-565, 2007.

[19] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram, "Xrank: Ranked Keyword Search over Xml Documents," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 16-27, 2003.

[20] D. Harel and R.E. Tarjan, "Fast Algorithms for Finding Nearest Common Ancestors," *SIAM J. Computing,* vol. 13, no. 2, pp. 338-355, 1984.

[21] H. He, H. Wang, J. Yang, and P.S. Yu, "Blinks: Ranked Keyword Searches on Graphs," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 305-316, 2007.

[22] 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.

[23] 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.

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

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

[26] Y. Huang, Z. Liu, and Y. Chen, "Query Biased Snippet Generation in Xml Search," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 315-326, 2008.

[27] S. Ji, G. Li, C. Li, and J. Feng, "Efficient Interactive Fuzzy Keyword Search," *Proc. Int'l Conf. World Wide Web (WWW),* pp. 371-380, 2009.

[28] 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.

[29] B. Kimelfeld and Y. Sagiv, "Finding and Approximating Top-k Answers in Keyword Proximity Search," *Proc. ACM SIGMOD-SIGACT-SIGART Symp. Principles of Database Systems (PODS),* pp. 173-182, 2006.

[30] J.M. Kleinberg, "Authoritative Sources in a Hyperlinked Environment," *J. ACM,* vol. 46, no. 5, pp. 604-632, 1999.

[31] G. Koutrika, Z.M. Zadeh, and H. Garcia-Molina, "Data Clouds: Summarizing Keyword Search Results over Structured Data," *Proc. Int'l Conf. Extending Database Technology: Advances in Database Technology (EDBT),* pp. 391-402, 2009.

[32] 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.

[33] G. Li, J. Feng, and L. Zhou, "Interactive Search in Xml Data," *Proc. Int'l Conf. World Wide Web (WWW),* pp. 1063-1064, 2009.

[34] G. Li, S. Ji, C. Li, and J. Feng, "Efficient Type-Ahead Search on Relational Data: A Tastier Approach," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 695-706, 2009.

[35] G. Li, C. Li, J. Feng, and L. Zhou, "Sail: Structure-Aware Indexing for Effective and Progressive Top-k Keyword Search over XML Documents," *Information Sciences,* vol. 179, no. 21, pp. 3745-3762, 2009.

[36] 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 Int'l Conf. Management of Data,* pp. 903-914, 2008.

[37] Y. Li, C. Yu, and H.V. Jagadish, "Schema-Free Xquery," *Proc. Int'l Conf. Very Large Data Bases (VLDB),* pp. 72-83, 2004.

[38] Y. Li, C. Yu, and H.V. Jagadish, "Enabling Schema-Free Xquery with Meaningful Query Focus," *VLDB J.,* vol. 17, no. 3, pp. 355-377, 2008.

[39] F. Liu, C.T. Yu, W. Meng, and A. Chowdhury, "Effective Keyword Search in Relational Databases," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 563-574, 2006.

[40] Z. Liu and Y. Chen, "Identifying Meaningful Return Information for Xml Keyword Search," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 329-340, 2007.

[41] Z. Liu and Y. Chen, "Reasoning and Identifying Relevant Matches for Xml Keyword Search," *Proc. VLDB Endowment,* vol. 1, no. 1, pp. 921-932, 2008.

[42] Y. Luo, X. Lin, W. Wang, and X. Zhou, "Spark: Top-k Keyword Query in Relational Databases," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 115-126, 2007.

[43] Y. Luo, W. Wang, and X. Lin, "Spark: A Keyword Search Engine on Relational Databases," *Proc. Int'l Conf. Data Eng. (ICDE),* pp. 1552-1555, 2008.

[44] A. Markowetz, Y. Yang, and D. Papadias, "Keyword Search on Relational Data Streams," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 605-616, 2007.

[45] L. Qin, J.X. Yu, and L. Chang, "Keyword Search in Databases: The Power of Rdbms," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 681-694, 2009.

[46] M. Richardson and P. Domingos, "The Intelligent Surfer: Probabilistic Combination of Link and Content Information in Pagerank," *Proc. Neural Information Processing Systems (NIPS),* pp. 1441-1448, 2001.

[47] M. Sayyadian, H. LeKhac, A. Doan, and L. Gravano, "Efficient Keyword Search across Heterogeneous Relational Databases," *Proc. Int'l Conf. Data Eng. (ICDE),* pp. 346-355, 2007.

[48] F. Shao, L. Guo, C. Botev, A. Bhaskar, M.M.M. Chettiar, F.Y. 0002, and J. Shanmugasundaram, "Efficient Keyword Search over Virtual XML Views," *Proc. Int'l Conf. Very Large Data Bases (VLDB),* pp. 1057-1068, 2007.

[49] 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.

[50] Y. Tao and J.X. Yu, "Finding Frequent Co-Occurring Terms in Relational Keyword Search," *Proc. Int'l Conf. Extending Database Technology: Advances in Database Technology (EDBT),* pp. 839-850, 2009.

[51] T. Tran, H. Wang, S. Rudolph, and P. Cimiano, "Top-k Exploration of Query Candidates for Efficient Keyword Search on Graph-Shaped (RDF) Data," *Proc. Int'l Conf. Data Eng. (ICDE),* pp. 405-416, 2009.

[52] Q.H. Vu, B.C. Ooi, D. Papadias, and A.K.H. Tung, "A Graph Method for Keyword-Based Selection of the Top-k Databases," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 915-926, 2008.

[53] G. Weikum, "Db&ir: Both Sides Now," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 25-30, 2007.

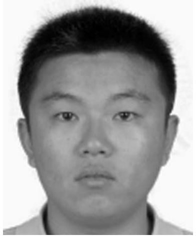
[54] Y. Xu and Y. Papakonstantinou, "Efficient Keyword Search for Smallest Lcas in XML Databases," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 537-538, 2005.

[55] Y. Xu and Y. Papakonstantinou, "Efficient LCA Based Keyword Search in XML Data," *Proc. Int'l Conf. Extending Database Technology: Advances in Database Technology (EDBT),* pp. 535-546, 2008.

[56] B. Yu, G. Li, K.R. Sollins, and A.K.H. Tung, "Effective Keyword-Based Selection of Relational Databases," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 139-150, 2007.

[57] D. Zhang, Y.M. Chee, A. Mondal, A.K.H. Tung, and M. Kitsuregawa, "Keyword Search in Spatial Databases: Towards Searching by Document," *Proc. Int'l Conf. Data Eng. (ICDE),* pp. 688-699, 2009.

Jianhua Feng received the BS, MS, and PhD degrees in computer science and technology from Tsinghua University. He is currently working as a professor in the Department of Computer Science and Technology at 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 top international conferences and journals, such as ACM SIGMOD, ACM SIGKDD, VLDB, IEEE ICDE, WWW, ACM CIKM, ICDM, SDM, *VLDB Journal*, *IEEE Transactions on Knowledge and Data Engineering*, *Data Mining and Knowledge Discovery*, *Information Systems*, and so on. He is a senior member of the IEEE and a senior member of the China Computer Federation (CCF).



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