Efficient Fuzzy Type-Ahead Search in XML Data

#V.REDYA JADAV¹, Associate Professor, Computer Science and Engineering Department
#G.JYOSTHNA² M.Tech, Computer Science and Engineering Department
# Bomma Institute of Technology and Science, Khammam, A.P State, INDIA

Abstract— Keywords are suitable for query XML streams without schema information. In current forms of keywords search on XML streams and rank functions do not always represent users’ intentions. This paper addresses this problem in another aspect. In this paper, the skyline Top-$K$ keyword queries, a novel kind of keyword queries on XML streams, are presented. For such queries, skyline is used to choose results on XML streams without considering the complicated factors influencing the relevance to queries. With skyline query processing techniques, two techniques, are presented to process skyline Top-$K$ keyword single queries and multi-queries on XML streams efficiently. Extensive experiments are performed to verify the effectiveness and efficiency of these techniques presented in this paper. According to the experimental results, the algorithms are not sensitive to the parameters such as the number of keywords, the number of results, the number of queries, and the runtime is approximately linear to the size of document.

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.

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 Auto complete 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 searching 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.

1. Introduction:

We propose XIR, a novel method for processing partial match queries on heterogeneous XML documents using information retrieval (IR) techniques. A partial match query is defined as the one having the descendent-or-self axis “//” in its path expression. In its general form, a partial match query has branch predicates forming branching paths. The objective of XIR is to efficiently support this type of queries for large-scale documents of heterogeneous schemas. XIR has its basis on the conventional schema-level methods using relational tables and significantly improves their efficiency using two techniques: an inverted index technique and a novel prefix match join. The former indexes the labels in label paths as keywords in texts, and allows for finding the label paths matching the queries more efficiently than string match used in the conventional methods. The latter supports branching path expressions, and allows for finding the result nodes more efficiently than containment joins used in the conventional methods. We compare the efficiency of XIR with those of XRel and XParent using XML documents crawled from the Internet. The results show that XIR is more efficient than both XRel and XParent by several orders of magnitude for linear path expressions, and by several factors for branching path expressions.

Current data sharing and integration among various organizations require a central and trusted authority to first collect data from all data sources and then integrate the collected data. This process tends to complicate the update of data and to compromise data sources’ privacy. In this paper, a repository for integrating data from various data sharing services without central authorities is presented. The major differences between our repository and existing central authorities are: 1) Our repository collects data from data sharing services based on users' integration requirements rather than all the data from the data sharing services as existing central authorities. 2) While existing central authorities have full control of the collected data, the capability of our repository is restricted to computing the integration results required by users and cannot get other information about the data or use it for other purposes. 3) The data collected by our repository cannot be used to generate other results except that of the specified data integration request, and hence the compromise of our repository can only reveal the results of the specified data integration request, while the compromise of central authorities will reveal all 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 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 user’s 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 Complete Search in textual documents, which can find relevant answers by allowing query keywords appear at any places in the answer. However, Complete Search 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 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 sub trees 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 user’s 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 Java Script. 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.

![Fig. 1. An XML document](image1)

- In an XML tree, every two nodes are connected through their LCA.
- Not all connected trees are relevant, even if the size is small.
- The focus is defining query results to prune irrelevant sub tree.

![Fig 2. Result Definition on XML & Trees](image2)

**2. 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. Schemafree 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 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 sub trees to answer keyword queries. It first identifies the minimum connected tree, which is a sub tree 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.

This three-hour tutorial categorizes existing work and covers the following topics.

### 2.1 Generating Search Results

Unlike traditional database applications where query results are fully specified by structured queries, the result task in keyword search is to define query results which automatically gather relevant information that is generally fragmented and scattered across multiple places (e.g., records in different relations in RDBMSs, different databases in Web/distributed databases, and elements/nodes in XML or graph-structured data).

#### 2.1.1 Query Result Definition

When the data is modeled as a tree, lowest common ancestor (LCA) is a fundamental form to define the query results. That is, a result is a sub tree rooted at the LCA of a set of nodes that collectively match query keywords [3, 5, 11, 25, 33, 37, 47, 57, 58]. A query result on a graph data model is commonly defined as a sub tree of the data graph where no node or edge can be removed without losing connectivity or keyword matches. Since finding the smallest result, which is the group Steiner tree, is NP-hard, variations and relaxation of the definition have been proposed in order to attain reasonable efficiency [4, 10, 13, 22, 24, 29]. Furthermore, besides the data that match query keywords, studies have been performed on identifying data that do not match keywords, but are implicitly relevant [16, 26, 35, 38, and 50].

#### 2.1.2 Ranking Functions

Keyword searches are inherently ambiguous, and not all query results are equally relevant to a user. Various ranking schemes have been proposed to order the query results into a sorted list so that users can focus on the top ones, which are hopefully the most relevant ones. Various ranking schemes are used in existing work, which consider both the properties of data nodes (e.g., TF*IDF, node weight, and page-rank style ranking) and the properties of the whole query result (e.g., number of edges, weights on edges, size normalization, redundancy penalty) [3, 5, 6, 8, 10, 11, 13, 16, 24, 29, 28, 34, 39, 43, 44, 51, 53, 54, 56, 59].

#### 2.1.3 Result Generation and Top-k Query Processing

We will introduce representative algorithms for query result generation and efficient top-k query processing. For keyword search on
XML data, encoding and indexing schemes [5, 11, 33, 47, and 57] as well as materialized views [36] have been exploited. For keyword search on relational databases, existing approaches are mainly based on candidate network (CN) generation, and differ on processing and optimization techniques to execute the CNs. We will distinguish the algorithms for monotonic ranking functions [17, 34] and non monotonic ranking functions [39]. For keyword search on graph-structured data, there are exhaustive search based on dynamic programming [8], efficient generation of top-k answers [10], heuristics-based approaches [4, 22, 30, 31, 52], and approaches leveraging pre computing or indexing [6, 9, 13, 29, 41]. We will also compare and discuss the challenges of keyword search processing techniques when the data schema is available [1, 8, 15, 39, 46, 54, 56] versus when it is absent [8, 10, 13, 24].

2.2 Improving Search Quality

To improve search quality and users' search experience, various techniques have been proposed, such as result snippets, result clustering, query cleaning, etc, which have been successfully used in text search. However, they pose new challenges in the context of searching structured data.

2.2.1 Result Snippets.

To compensate the inaccuracy of ranking functions, result snippets should be generated [18, 19]. The principle of result snippets is orthogonal to that of ranking functions: let users quickly judge the relevance of query results by providing a brief quotable passage of each query result, so that users can choose and explore relevant ones among many results.

2.2.2 Result Clustering.

In face of query ambiguity, instead of displaying a mixture of query results of different semantics, it is more desirable to cluster query results based on their similarity, so that the user can quickly browse all possible interpretations of query semantics and choose the sets of results that are relevant [14, 27, 54].

2.2.3 Query Cleaning.

Query cleaning involves semantic linkage and spelling corrections of database-relevant query key words, followed by segmentation of nearby query keywords so that each segment corresponds to a high quality data term. Compared to query cleaning on textual documents, query cleaning for structured data brings great potentials with new challenges [42].

2.2.4 Evaluation

We will discuss evaluation framework for keyword search engines. One is based on empirical evaluation using benchmark data, such as INEX (Initiative for the Evaluation of XML Retrieval) [20], a benchmark for XML keyword searches. The other is formal evaluation, which evaluates an approach based on a set of axioms that capture broad intuitions [37].

2.3 Applications of Keyword Search in Information Integration and Analysis

Supporting keyword search is not only helpful for users to access a single database, but also benefits information integration. As the number of potentially-related data sources continues to grow rapidly, the existing approach of using predefined forms and associated query templates can not adequately support diverse data sources and
meet diverse user needs. Keyword search provides a light-weight mechanism to access multiple data sources without labor-intensive information integration upfront.

2.3.1 Database Selection.

We will discuss techniques that summarize underlying databases by a keyword relationship graph, and select the most relevant data sources with respect to a user keyword search based on derived summaries [29, 43, 53,59].

2.3.2 Query Generation.

We will discuss techniques that allow a casual user to author new query templates and Web forms by posing keyword searches. The keyword searches are matched against source relations and their attributes to create multiple ranked queries linking the keyword matches. The set of queries is attached to a Web query form, which can be reused by anyone with related information needs [49].

2.3.3 Analytical Processing.

Online Analytical Processing (OLAP) tools provide elaborate query languages that allow users to group and aggregate data in various ways, and to explore interesting trends and patterns in the data. However, the complexity of issuing such analytic queries is overwhelming. It is highly desirable, yet very challenging, to combine intuitive keyword-based search with the power of OLAP, to allow users to easily analyze complex data [51, 56, 61].

4 PROBLEM FORMULATION OF FUZZY TYPE-AHEAD SEARCH IN XML DATA

We introduce the overview of fuzzy type ahead search in XML data and formalize the problem.

4.1 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 invoke 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. These sets 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 s1 and s2, denoted by ed(s1, s2), 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, ed(mics, mices)= 1 and ed(mics, michÞ)= 1. For instance, given a partial keyword “mics,” its predicted words could be “mices,” “mich,” “michal,” etc. Then, the server identifies the relevant sub trees 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 sub trees 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.

4.2 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 = \{k1, K2,…… k'l\} and an edit-distance threshold T. Let the predicted-word set be Wki = \{w/w is a tokenized word in D and there exists a prefix of w, k’i, ed(ki, k’i)<=T\}. Let the predicted-answer set be RQ = \{r/r is a keyword-search result of query \{w1€ 2 Wk1,w2 € Wk2 , . . . ,w'l€ Wkl\}\}. For the keystroke that invokes Q, we return the top-k answers in RQ 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. We devise effective ranking functions, early termination techniques, efficient algorithms, and forward-index structures to address the second challenge.

5 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.

5.1 Index Structures

We use a tries structure to index the words in the underlying XML data. Each word w corresponds to a unique path from the root of the tries 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 tries 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.
5.2 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.

5.2.1 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 hash table 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 \( c_1c_2 \ldots c_x \) letter by letter. Let \( p_i = c_1c_2 \ldots c_i \) be a prefix query \( (1 \leq i \leq x) \). Suppose is the trie node corresponding to \( p_i \). After the user types in a prefix query \( p_i \), we store node 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 \( =c_1c_2 \ldots c_i c_{x+1} \). To incrementally answer the new query, we first check whether node \( p_i \) that has been kept for \( p_i \) has a child with a label of \( c_{x+1} \). If so, we locate the leaf descendants of node \( p_i \), 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.

### 5.2.2 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 \( p_1 p_2 \ldots p_x \), suppose we have computed the active-node sets \( A_{p_1} \ldots A_{p_x} \). Then, for a new query \( p_1 p_2 \ldots p_x \), we use \( A_{p_1} \) to compute \( A_{p_2} \) as follows: for any active node \( n \in A_{p_1} \), its descendants could be similar to \( pxp_1 \), we consider three edit operations of insertion, deletion, and substitution to compute active nodes under node \( n \) and put them into \( A_{p_2} \). [27]. Given an null string \( \emptyset \), we initialize \( A_{\emptyset} = \{ n | \text{the level of node } n \text{ is no larger than } T \} \). Accordingly, we can incrementally compute \( A_{p_1} \) based on [27].

To facilitate incremental computation, for each user, we use a session to maintain active nodes of each keyword using a hash table. Thus, given a partial keyword \( p_x \), we first compute its active-node set \( A_{p_x} \). Then, for each active node \( n \in A_{p_x} \), we retrieve inverted lists of \( n \)'s leaf descendants and compute the union of such inverted lists.

### 6 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 ELCA 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.

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
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. 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. Moreover, our approach automatically supports the “OR” semantics.

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

6.1 Minimal-Cost Tree

In this section, we introduce a new framework to find relevant answers to a keyword query over an XML 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 sub tree with paths to nodes that include the query Key words. This sub tree 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.

Consider an XML document D, a keyword ki, and a content node or quasi-content node for ki, n. Let P denote a subset of n’s descendant nodes which are content nodes of ki. P€ P is a pivotal node for ki 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 n9 -> n12 -> n13 is the corresponding pivotal path, where n9, n12, n13 denote nodes 9, 12, and 13, respectively. Intuitively, a pivotal node for node n and ki is much more relevant to node n for ki than other content nodes. 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.

7. MIN-MAX HEAPS

Given a set S of values, a min-max heap on S is a binary tree T with the following properties:
1) T has the heap-shape
2) T is min-max ordered: values stored at nodes on even (odd) levels are smaller (greater) than or equal to the values stored at
their descendants (if any) where the root is at level zero. Thus, the smallest value of $S$ is stored at the root of $T$, whereas the largest value is stored at one of the root’s children; an example of a min-max heap is shown in Figure 4.

A min-max heap on $n$ elements can be stored in an array $A[1 \ldots n]$. The $i$th location in the array will correspond to a node located on level $L(\log_i)n$ in the heap. A max-min heap is defined analogously; in such a heap, the maximum value is stored at the root, and the smallest value is stored at one of the root’s children. It is interesting to observe that the Hasse diagram for a min-max heap (i.e., the diagram representing the order relationships implicit within the structure) is rather complex in contrast with the one for a traditional heap (in this case, the Hasse diagram is the heap itself); Figure 5 shows the Hasse diagram for the example of Figure 4.

Algorithms processing min-max heaps are very similar to those corresponding to conventional heaps. Creating a min-max heap is accomplished by an adaption of Floyd’s [4] linear-time heap construction algorithm. Floyd’s algorithm builds a heap in a bottom-up fashion. When the algorithm examines the subtree rooted at $A[i]$ then both subtrees of $A[i]$ are max-ordered, whereas the subtree itself may not necessarily be max-ordered. The Trickle Down step of his algorithm exchanges the value at $A[i]$ with the maximum of its children. This step is then applied recursively to this maximum child to maintain the max-ordering. In min-max heaps, the required ordering must be established between an element, its children, and its grandchildren. The procedure must differentiate between min- and max-levels. The resulting description of this procedure follows:

```plaintext
procedure TrickleDownMax(i)
   -- i is the position in the array
   if i is on a min level then
      TrickleDownMin(i)
   else
      TrickleDownMax(i)
   endif

procedure TrickleDownMin(i)
   if A[i] has children then
      m := index of smallest of the
         children and grandchildren
         (if any) of A[i]
      if A[m] is a grandchild of A[i] then
         if A[m] < A[i] then
            swap A[i] and A[m]
         endif
         if A[m] > A[parent(m)] then
            swap A[m] and A[parent(m)]
         endif
         TrickleDownMin(m)
      endif
   else if A[m] is a child of A[i] then
      if A[m] < A[i] then
         swap A[i] and A[m]
      endif
   endif
```

The procedure Trickle Down Max is the same except that the relational operators are reversed. The operations DeleteMin and DeleteMax are analogous to deletion in conventional heaps. Specifically, the required element is extracted and the vacant position is filled with the last element of the heap. The min max ordering is maintained after applying the Trickle Down procedure.
An element is inserted by placing it into the first available leaf position and then reestablishing the ordering on the path from this element to the root. An efficient algorithm to insert an element can be designed by examining the Hasse diagram (recall Figure 5). The leaf-positions of the heap correspond to the nodes lying on the middle row in the Hasse diagram. To reestablish the min-max ordering, the new element is placed into the next available leaf position, and must then move up the diagram toward the top, or down toward the bottom, to ensure that all paths running from top to bottom remain sorted. Thus the algorithm must first determine whether the new element should proceed further down the Hasse diagram (i.e., up the heap on max-levels) or up the Hasse diagram (i.e., up the heap on successive min-levels). Once this has been determined, only grandparents along the path to the root of the heap need be examined—either those lying on min-levels or those lying on max-levels. The algorithms are as follows:

```plaintext
procedure BubbleUp(i)
    -- i is the position in the array
    if i is on a min-level then
        if i has a parent cand A[i] > A[parent(i)] then
            swap A[i] and A[parent(i)]
            BubbleUpMax(parent(i))
        else
            BubbleUpMin(i)
        endif
    else
        if i has a parent cand A[i] < A[parent(i)] then
            swap A[i] and A[parent(i)]
            BubbleUpMin(parent(i))
        else
            BubbleUpMax(i)
        endif
    endif

procedure BubbleUpMin(i)
    if A[i] has grandparent then
        if A[i] < A[grandparent(i)] then
            swap A[i] and A[grandparent(i)]
            BubbleUpMin(grandparent(i))
        endif
    endif
```

The cand (conditional and) operator in the above code evaluates its second operand only when the first operand is true.
BubbleUpMax is the same as BubbleUpMin except that the relational operators are reversed. From the similarity with the traditional heap algorithms, it is evident that the min-max heap algorithms will exhibit the same order of complexity (in terms of comparisons and data movements). The only difference rests with the actual constants: for construction and deletion the constant is slightly higher, and for insertion the constant is slightly lower. The value of the constant for each operation is summarized in Table I: the reader is referred to [2] for a detailed derivation of these values. All logarithms are base 2.

<table>
<thead>
<tr>
<th>Operation</th>
<th>min-heap</th>
<th>min-max heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create</td>
<td>$2n$</td>
<td>$7n/3$</td>
</tr>
<tr>
<td>Insert</td>
<td>$\log(n+1)$</td>
<td>$0.5 \log(n+1)$</td>
</tr>
<tr>
<td>DeleteMin</td>
<td>$2 \log(n)$</td>
<td>$2.5 \log(n)$</td>
</tr>
<tr>
<td>DeleteMax</td>
<td>$0.5 \frac{n}{2} + \log(n)$</td>
<td>$2.5 \log(n)$</td>
</tr>
</tbody>
</table>

Slight improvements in the constants can be obtained by employing a technique similar to the one used by Gonnet and Munro [S] for traditional heaps. The resulting new values are shown in Table II; again, details of the derivation can be found in [2]. In Table II the function $g(x)$ is defined as follows:

$$g(x) = 0 \text{ for } x \leq 1 \text{ and } g(n) = g(\lceil \log(n) \rceil) + 1.$$  

8. CONCLUSIONS

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.

The min-max heap structure is based on the idea of alternating the relations “greater than or equal to all descendants” and “smaller than or equal to all descendants” between consecutive tree levels; the order relation implied is herein referred to as min max ordering and can be applied to a number of structures implementing priority queues, such as P-trees, leftist-trees.

9. REFERENCES


V.Redya received the B.E, degrees in computer science and engineering from Osmania University, Hyderabad, AP, and India. He is received M.Tech in Computer Science and Engineering at Jawaharlal Nehru Technological University, Hyderabad, AP
and India. He is pursuing Ph.D program in computer science and Engineering.

G.Jyosthna received the B.Tech, degrees in computer science and engineering from Jawaharlal Nehru Technological University, Hyderabad, AP, and India. She is Pursuing M.Tech in Computer Science and Engineering at BOMMA INSTITUTE OF TECHNOLOGY and SCIENCE from Jawaharlal Nehru Technological University, Hyderabad, AP and India.

** Journal Impact factor 2.6 **