Fuzzy Type-Ahead Search in XML Data

K. L. A. Nivedita, R. Naveen, K. Sravani
M.Tech Student Department of CSE SBIT, Khammam
Assistant Professor Department of CSE SBIT, Khammam
Assistant Professor Department of CSE SBCE, Khammam

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 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 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.*

I. Introduction

Traditional methods use query languages such as XPath and XQuery to query XML data. These methods are powerful but unfriendly to non-expert users. First, these query languages are hard to comprehend for non database 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 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 Auto complete, 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 Auto complete 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 Auto complete 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, because these two keywords appear at different places in the answer. To address this problem, Bast and Weber 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. It allows users to explore data as they type, even in the presence of minor errors of their input keywords.

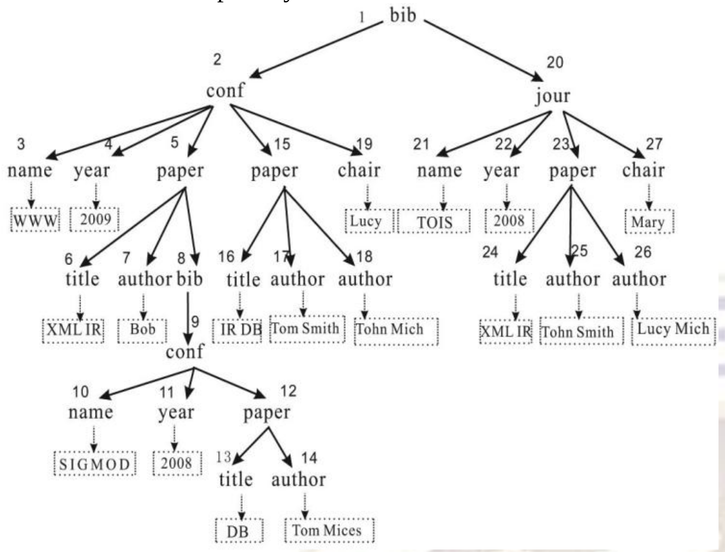

Fig. 1: An XML document.

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. 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 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 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 [2][3].

. 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.

II. Problem Formulation of Fuzzy Type-Ahead Search In XML Data

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 [6].

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 is 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 $ed(s_1,s_2)$, 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.

III .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.

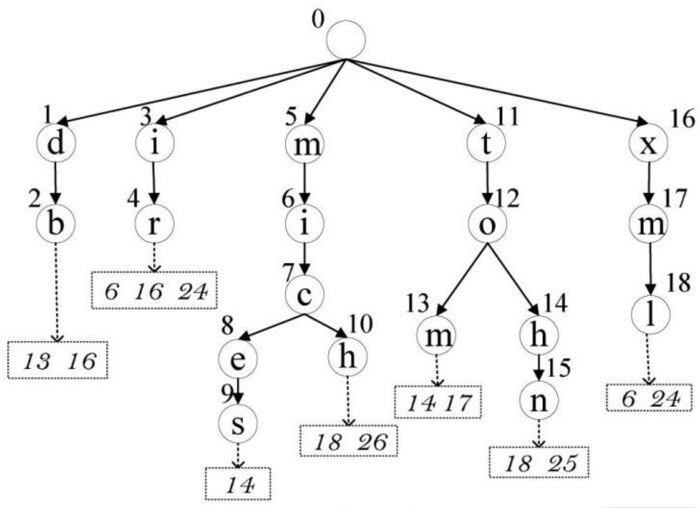

Fig. 2. The trie on top of words in Fig. 1 (a part of words).

3.1 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.

3.2 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.

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 [10].

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.

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.

3.3 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 $k_1,k_2....k_l$. For each keyword k_i ($1 \le i \le l$), 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 U_{ki} for every ki. Finally, we compute the predicted answers on top of lists U_{k1}, U_{k2} . . . U_{kl} .We use the semantics of ELCA to compute the corresponding answers. We use the binary-search-based method to compute ELCAs.

IV .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. 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.

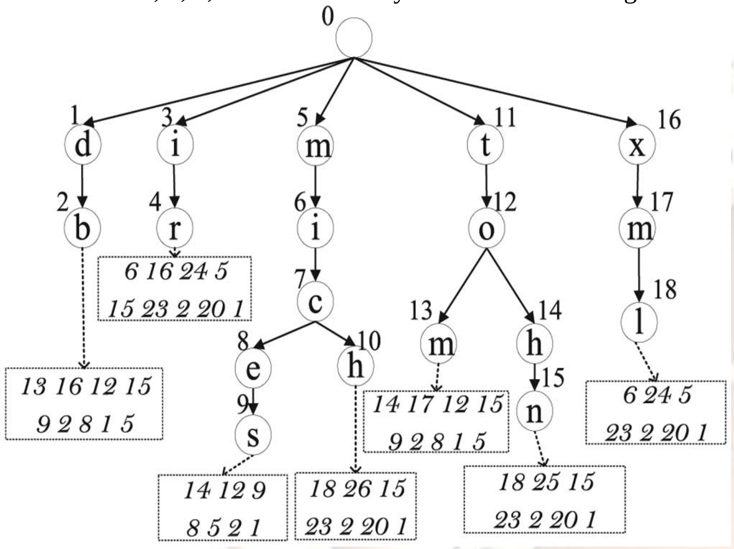

Fig. 3. The extended trie on top of words in Fig. 1 (a part of words).

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. 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. We develop effective ranking techniques to rank XML elements on the inverted lists in the extended trie structure. We can employ threshold-based algorithms to progressively and efficiently identify the top-k relevant answers. Moreover, our approach automatically supports the "OR" semantics.

4.1 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 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 .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.

4.2 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 and then extend the ranking function to support fuzzy search

4.2.1 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.

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 kj. The distance between 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:

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

Where P is the set of pivotal nodes for n and k_j, α is a damping factor between 0 and 1, and Δ(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 trade off, 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.

V. 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.

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, 2010.

[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, 2011.

[4] A. Balmin, V. Hristidis, and Y. Papakonstantinou, “Object rank: Authority-Based Keyword Search in Databases,” Proc. Int’l Conf. Very Large Data Bases (VLDB), pp. 564-575, 2011.

[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, 2010.

[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, 2011.

[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, 2011.

[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, 2011.

[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, 2011.

[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, 2011.

[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-SIGACTSIGAR T Symp. Principles of Database Systems (PODS), 2011.