Efficient Searching on Data Using Forward Search

Supriya Sivapuja¹, Sk. Mohiddin², S Srikanth Babu³, Srikar Babu S.V⁴

¹[M.Tech], PBR Visvodaya Institute of Technology & Science, Kavali, A.P, India
²Assistant Professor, Dept of CSE, PBR Visvodaya Institute of Technology & Science, Kavali, A.P, India
³[M.Tech], PBR Visvodaya Institute of Technology & Science, Kavali, A.P, India
⁴Associate Consultant, OFSS, Bangalore

Abstract—In conventional search systems on data, a user composes a query with different keywords, submits it to the system, and retrieves relevant information. If the user doesn’t know how to issue queries, he tries multiple queries and sees what the result is. Now, a new information retrieving system that searches data as the user types in query keywords. It allows users to discover data as they type, even if there is an error in query keywords. This system has the following features: 1) It extends Autocomplete, which supports multiple keywords in queries in data. 2) It can find high-quality answers that have keywords matching query keywords approximately. 3) Our effective index structures and searching algorithms can achieve a very high interactive speed. 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 to achieves high search efficiency and result quality.

Keywords: search system, Autocomplete, index structures

1. INTRODUCTION

Searching using keywords is a mostly used mechanism for querying data such as XML data. One important advantage of keyword searching is it enables users to search information neither knowing a complex query language such as SQL, nor having prior knowledge about the structure of the underlying data. Query languages such as XPath and XQuery are used to query XML data. These methods are dominant but distant to non-expert users. First, these query languages are hard to understand for non-database users. For example, XQuery is fairly complicated to grasp. Second, these languages require the queries to be posed against the original, difficult, database schemas.

In this search system over XML data using keywords, a user creates a query, submits it to the system, and retrieves relevant information from XML data. This requires the user to have definite information about the structure and content of the basic data repository. If the user has incomplete knowledge about the data, seldom the user feels “left in the dark”, and has to use a try-and-see approach for finding answers. He uses few possible keywords, goes through the returned answers, modifies the keywords, and reissues a new query. He needs to reiterate this step multiple times to find the answers. This system interface is neither well-organized nor user open.

2. LITERATURE SURVEY

Many systems provide solutions to this problem. Frequently used method is Autocomplete, which predicts phrase that the user may type in based on the unfinished string the user has typed. But the problem with Autocomplete is that the system treats a query as a single string if it consist multiple keywords. That means it does not allow keywords to show at different places. One solution to this problem given by Bast and Weber [1], [2] is CompleteSearch in textual documents, which can find related answers by allowing keywords in query, come out at any places in the solution. But it does not support fairly accurate search that is, it cannot allow small errors among query keywords and answers. For allowing users to hunt data as user type, even if there is a slight error of their input keywords type-forward search in textual documents is proposed.

Type-forward search can provide users immediate response as users type in keywords, and it does not require users to type in entire keywords. Type-forward search can help users browse the data, save users typing attempt, and efficiently find the answers. Fig 1 shows the architecture for type-forward search.
We also considered type-forward search in relational databases. Existing methods cannot search XML data in a type-forward search way, and it is not inconsequential to expand existing techniques to support type-forward search in XML data.

This is because XML contains parent-child relationships, and we need to identify appropriate XML subtrees that confine such structural relationships from XML data to answer queries with keywords, as a substitute of single documents.

We suggest TFSX, a type-forward search technique in XML data. TFSX searches the XML data on the fly as user’s type in query keywords, even in the occurrence of small errors of their keywords. TFSX provides a responsive interface for users to discover XML data, and can considerably save users typing attempt.

In this paper, we study challenges that take place physically in this computing model. The foremost challenge is search-effectiveness. Each query with multiple keywords needs to be answered powerfully. For this, the delay between the client browser and server should be as small as possible also within milliseconds.

This short running-time requirement is mainly difficult when the backend repository has a huge amount of data. For this, we suggest effective index structures and algorithms to solve keyword queries in XML data. We observe effective ranking functions and timely termination techniques to gradually discover top-k answers.

2.1 Notations

In general XML document can be organized as a rooted tree with labeled nodes. 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 \( u < v \) ("\( u > v \)", respectively) to denote that node \( u \) is an ancestor (descendant, respectively) of node \( v \).

For example, consider the XML document in Fig. 2, we have \( \text{paper} \) (node 15) \( \leq \) \( \text{author} \) (node 17) and \( \text{conf} \) (node 9) \( > \) \( \text{paper} \) (node 5). A keyword query consists of a set of keywords \( \{k_1, k_2, \ldots, k_j\} \). 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 of the content nodes are called the quasi-content nodes of the keyword. For example, consider the XML document in Fig. 2, \( \text{title} \) (node 24) is a content node for keyword “XML,” and \( \text{jour} \) (node 20) is a quasi-content node of keyword “XML.”

2.2 Keyword Search in XML Data

There are many ways to identify the answers to a query on an XML document. A frequently used one is based on the idea of lowest common ancestor [3]. 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 for all, \( I \leq i \leq m, u \leq v_i \), and there does not exist another node \( u’ \) such that \( u < u’ \) and \( u’ \leq v_i \).

For a keyword query, the LCA-based algorithm primarily 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 \( \{13, 16\}, \{14, 17\} \) 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 resolve the problem of use of LCAs as query solutions, various methods have been projected to progress search effectiveness and result quality. One of the solutions is exclusive lowest common ancestor (ELCA) planned by Guo et al. [4] and Xu and Papakonstantinou [5]. An LCA is an ELCA if it is still an LCA after excluding its LCA descendants.

For example, the ELCA 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 proposed a binary-search-based method to efficiently identify ELCA.

3. TYPE-FORWARD SEARCH IN XML DATA
3.1 Overview
We initially begin how TAFX works for multiple keyword queries in XML data, by allowing small errors of query keywords and inconsistencies in the data itself. Suppose there is an original 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 consists existing string. The browser sends the query to the server, which computes and profits to the user the best solutions ranked by their relevancy to the keyword query.

The server primarily tokenizes the query into a number of keywords using delimiters such as the space character. The keywords are taken as partial keywords, as the user may have not completed typing the entire keywords. For the partial keywords, we would like to know the feasible words the user intends to type. However, given the partial 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. Then we use edit distance to quantify the resemblance between two words. The edit distance between two words s1 and s2, denoted by ed(s1, s2), is the lowest 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(cs; ces) = 1 and ed(cs; ch) = 1. For instance, given a partial keyword “cs,” its predicted words could be “ces,” “ch,” “chal,” etc.

Next, server identifies the related subtrees in XML data for every input keyword that contain the predicted words. We can use any presented semantics to recognize the answer based on the predicted words, such as ELCA. We call these related subtrees as the predicted answers of the query.

Therefore, TAFX can save users time and efforts, because they can find the answers even if they have not completed typing all the entire keywords or typing keywords with small errors.

3.2 Description (TYPE-FORWARD SEARCH IN XML DATA):
Given an XML document D, a keyword query Q= { k1, k2, . . . , ki}, and an edit-distance threshold T. Let the predicted-word set be \( W_k = \{ w | w \) is a tokenized word in \( D \) and there exists a prefix of \( w \), \( k_i \), \( ed(k_i, k_i) \leq T \} \). Let the predicted-answer set be \( R_k = \{ r | r \) is a keyword-search result of query \( \{ w_1 \in W_{k_1}, w_2 \in W_{k_2}, \ldots, w_l \in W_{k_l} \} \). For the keystroke that invokes \( Q \), we return the top-k answers in \( R_k \) for a given value \( k \), ranked by their relevancy to \( Q \).

We take the data and query as lowercase strings. There are two challenges to maintain type-forward search in XML data. One is how to interactively and powerfully identify the predicted words that have prefixes similar to the input partial keyword after each keystroke from the user. Second one is how to gradually and efficiently figure the top-k predicted answers of a query with multiple keywords, particularly when there are many predicted words.

We use 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.

4. LCA-BASED TYPE-FORWARD SEARCH
Here we use the concept of ELCA to see related answers on top of predicted words.

4.1 Index Structures
We use a tree arrangement to index the words in the original XML data. Every word \( w \) corresponds to a single path from the root of the tree 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 have the word of the leaf node.

For instance, consider the XML document in Fig. 2. The tree structure for the tokenized words is shown in Fig. 3. The word “tom” has a node ID of 13. Its inverted list includes XML elements 14 and 17.

4.2 Answering Queries with a Single Keyword
Here we answer queries based on the keywords present in that query. We perform two types of searches to answer queries with a single keyword using the tree structure as follows.

4.2.1 Accurate Search
One immature way to process such a query on the server is to answer the query as follows: we initially find the tree node equivalent to this keyword by traversing the tree from the root. Then, we trace the leaf descendants of this node, and get back the corresponding predicted words and the predicted XML elements on the inverted lists. The drawback of this method is that it involves a lot of recomputation without using the outcome of previous queries.
To solve this problem we use a caching-based method to incrementally get the tree node for the entered keyword. We preserve a session for all users. Every session keeps the keywords that the user has typed in the earlier period and the related tree node. Now a hash-table is used to maintain such information. If session times away, then the reserved data will be deleted. The objective of maintenance the information is to use it answer later queries incrementally.

In most cases, the user may adjust the earlier query string randomly, or copy and paste an entirely different string. Then we discover the cached keyword that has the best prefix with the new query. To incrementally answer the new query we use this prefix, by inserting the characters after the best prefix of the new query one at a time.

4.2.2 Forward Search

Clearly, in accurate search, given a partial keyword, there exists at most one tree node for the keyword. We recover the leaf descendants of this tree node as the predicted words. For forward search, there could be various tree nodes that are related to the partial keyword within a specified edit-distance threshold, called active nodes.

To assist incremental computation, we use a session for each user to keep active nodes of each keyword by means of a hashtable. Thus, given a partial keyword \( p_n \), we first calculate its active-node set \( A_{p_n} \). Then, for each active node \( n \in A_{p_n} \), we get back inverted lists of \( n \)'s leaf descendants and calculate the union of such inverted lists, denoted as \( U_n \). Finally, we calculate the union of \( U_n \) for \( n \in A_{p_n} \), denoted as \( U_{p_n} \), i.e., \( U_{p_n} = U_n \bigcup A_{p_k} U_n \). We call \( U_{p_n} \) the union list of \( p_n \). Clearly, \( U_{p_k} \) is accurately the set of XML elements that have prefixes alike to \( k \).

4.3 Answering Queries with Multiple Keywords

Here, we consider type-forward search to query with multiple keywords. We first tokenize the query into keywords, for each keystroke that invokes a query \( k_1, k_2, \ldots, k_l \). For each one keyword \( k_i \) \((1 \leq i \leq l)\), we calculate its equivalent active nodes, and for each such active node, we get back its leaf descendants and equivalent inverted lists. Next, we calculate union list \( U_{k_i} \) for each \( k_i \). Lastly, we calculate the predicted answers on top of lists \( U_{k_1}, U_{k_2}, \ldots, U_{k_l} \). We use the semantics of ELCA [5] to calculate the equivalent answers. We use the binary-search-based method to calculate ELCAks [5].

5. TOP-K TYPE-FORWARD SEARCH

The type-forward search algorithm in XML data that based on LCA has two major restrictions. Primarily, it uses the “AND” semantics among enter keywords of a query, and disregard the answers that include only few keywords of query. Next, to compute best results, in existing techniques need to find candidates before ranking. But this one is inefficient. So there is a need of effective algorithms to find best answers without generating all candidates.

To deal with these restrictions, we build up novel ranking techniques and efficient search algorithms. In our technique, every node on the XML tree could be potentially related to a query keyword, and to choose the best answers to the query we use a ranking function. For every leaf node in the tree, we index content nodes for the keyword of the leaf node, and also quasi-content nodes whose descendants have the keyword.

For example, consider the XML document in Fig. 1. For the keyword “XML,” we index nodes 6, 24, 5, 23, 2, 20 and 1 for this keyword as shown in Fig. 4. For the keyword “MICES,” we index nodes 14, 9, 8, 5, 2 and 1.
$C \ Q$, if node $n$ is a quasi-content node of $k$, the subtree includes the pivotal paths for $k$ and node $n$.

Identifying the predicted words for each input keyword is the first step to answer a keyword query. Next based on the predicted words, minimal-cost tree can be constructed for each node in the XML tree and return a node, which have maximum score as best node.

The major benefit of this description is that, even if a node does not have descendant nodes that contain all the keywords in the query, this node could still be measured as a probable answer. That means, this description is relax the statement in presented semantics that all the query keywords need to show in the descendants of a reply node.

5.2 MCT- Ranking

In this part, we talk about a minimal-cost tree ranking. We initially establish a ranking function for accurate search and then extend the ranking function to support forward search.

5.2.1 Accurate Search – Ranking

For minimal-cost tree ranking, we primary estimate the relevance among the root node and every input keyword, and then join these relevance scores for all input keyword as the total score of the minimal-cost tree.

We recommend two ranking functions to calculate the relevance score involving the root node $n$ to an input keyword $k$. The first case is if $n$ contains $k$. The second case is if $n$ does not contain $k$ but has a descendant containing $k$.

Our first ranking method models every 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 $k$ contained in $n$, we denote $tf(k,n)$ as the number of occurrences of $k$ in the subtree rooted at $n$, $idf(k)$ as the inverse document frequency of $k$, (i.e., the ratio of the number of nodes in the XML document to the number of nodes that contain $k$), and $ntl(n)$ as the normalized term length of $n$, i.e., $ntl(n) = \frac{|n|}{ntl_{max}}$, where $|n|$ denotes the number of terms contained in $n$ and $ntl_{max}$ denotes the node with the maximal number of terms.

If $n$ contains $k$, we use presented ranking methods [7] to calculate the relevance of node $n$ to keyword $k$,

$$\text{SCORE}_1(n,k) = \frac{\sum_{i=1}^{n} \text{TF}(k,n) \cdot \text{IDF}(k)}{1 + \sum_{i=1}^{n} \text{TF}(k,n) \cdot \text{IDF}(k)}$$

In the above method, $s$ is a constant, which is widely considered in the information-retrieval community and usually set to 0.2 [8].

On the other hand, if $n$ does not have $k$, the earliest ranking function cannot quantify the relevancy between node $n$ and keyword $k$. To solve this problem, we extend the first ranking function and propose the second ranking function. Given a keyword $k$, a quasi-content node $n$ for $k$, suppose $p$ is the pivotal node for $n$ and $k$. The distance between $n$ and $p$ can specify how relevant the node $n$ is to keyword $k$. The minor the distance between $n$ and $p$, the bigger relevancy score between $n$ and $k$ should be.

$$\text{SCORE}_2(n,k) = \sum_{p \in \text{path}} \alpha * \text{SCORE}_1(P, k)$$

where $P$ is the set of pivotal nodes for $n$ and $k$, $\alpha$ is a damping factor between 0 and 1, and $\text{path}(n,p)$ denotes the distance between node $n$ and node $p$. As the distance between $n$ and $p$ increases, $n$ becomes less relevant to $k$.

But our experiments recommended that a good value for $\alpha$ is 0.8, it achieves the best performance at this point. This is because it will corrupt the importance of ancestor nodes for a smaller $\alpha$ and thus may ignore important and related outcome, on the opposing, it will occupy some duplicates and fewer significant results for a larger $\alpha$.

Based on the two ranking functions, given a query $Q = \{k_1, k_2, \ldots, k_l\}$ 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 $Q$,

$$\text{SCORE}(n,Q) = \sum_{k_i \in Q} \text{SCORE}(n,k_i)$$

where $\text{SCORE}(n,k)$ denotes the score of node $n$ to keyword $k$. $\text{SCORE}(n,k) = \text{SCORE}_2(n,k)$ if $n$ contains $k$; if not, $\text{SCORE}(n,k) = \text{SCORE}_2(n,k)$ if $n$ has a descendant that contains keyword $k$.

5.2.2 Forward Search - Ranking

In the forward search a minimal-cost tree may not hold the correct input keywords, but have predicted words for each keyword in a specified query. Think a minimal-cost tree rooted at $n$, assume the predicted words for each input keyword in the subtree are $\{w_1, w_2, \ldots, w_l\}$. We suggest 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 further letters and complete the keyword, $k_i$ could be related to prefixes of $w_i$. The prefix of $w_i$ which has the minimum edit distance to $k_i$ among all the prefixes is called the top related prefix for $k_i$. $w_i$ is more related to $k_i$. Naturally, the top related prefix of $w_i$ could be measured to be most related to $k_i$. Unthinkingly, the minor edit distance between $k_i$ and $w_i$ is more related to $k_i$. Besides, as $w_i$ is a prefix of $w_i$, we use $\text{sim}(k_i, w_i)$ to quantify their similarity. Therefore, we suggest a new function to measure resemblance between $k_i$ and $w_i$:

$$\text{sim}(k_i, w_i) = \gamma \ast \frac{4}{1 + \sum_{j=1}^{l} \text{edit}(k_i, w_i)} + (1 - \gamma) \ast \frac{|k_i|}{|w_i|}$$

where $\gamma$ is a tuning parameter between 0 and 1. The previous is more vital, $\gamma$ is close to 1. Our experiments recommended that a good value for $\gamma$ is 0.95, and our scheme achieves the best performance. We expand the
ranking function in (3) by incorporating this similarity function to support forward search.

\[ \text{SCORE}(n,Q) = \sum_{l \in Q} \text{SCORE}(n,w_l) \]  

(5)

5.3 Finding of Top-k MCT gradually

In this part, we propose how to gradually find the top-k related minimal-cost trees. For each leaf tree node in the tree index, we maintain the content nodes and quasi-content nodes in the XML document as shown in Fig. 4, and corresponding scores and pivotal paths for the keyword of the leaf node, sorted by the relevancy to the keyword.

Specified a keyword query \( Q = \{k_1, k_2, \ldots, k_l\} \) for each partial keyword \( k_i \), we initially calculate its predicted words. Next, we calculate the union of inverted lists of \( k_i \)’s predicted words, \( U_{n_i} \), sorted by corresponding scores (i.e., \( \text{SCORE}(n,w_l) \)). After that, we can use presented NRA algorithm [16] based on threshold to gradually and powerfully calculate the top-k answers of on top of every \( U_{n_i} \).

Constructing the union lists of every input keyword is very expensive if many predicted words and many inverted lists present. As an alternative, we can create a partial virtual list on the fly. We only use partial virtual list elements to calculate the top-k answers. The partial virtual list can access those with higher scores not all. Then only we can do an early termination and do not need to visit other elements on the inverted lists.

Figure 5 Active nodes, predicted words, and corresponding inverted lists for query \( Q = \{k_1, k_2, \ldots, k_l\} \)

5.4 Forward Index to improve search performance

Here, we recommend the forward index to get better search performance. To do an early termination we can use “random access” based on the forward index in the algorithms.

To be precise, specified an XML element and an input keyword, we can get the related 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 finest one over all algorithms that appropriately get the top k answers [6].

We suggest a forward index to execute random access. Given an XML element \( e \), we create a tree structure to preserve the keywords contained in the element. Each leaf node in the forward index keeps the score of element \( e \) to the corresponding word of the leaf node. Therefore, given any partial keyword, we can powerfully check whether \( e \) contains a word having prefixes related to the keyword using the forward index.

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 index if there is a large number of keyword under the element for a given XML element.

We utilize a cost-based method to select forward index for materialization. The time complexity of sorted access is \( O(1) \) and random access is \( O(T \ast AN) \), where \( T \) is the edit-distance threshold and \( AN \) is the number of active nodes [9]. Suppose the average number of active nodes is \( A \) and the average inverted-list length is \( I \). If \( T \ast A > I \), we will not continue the forward index, since we can only use sorted access to scan the inverted lists.

6. TENTATIVE STUDY

We have implemented our method on real applications using our proposed techniques. We employed the data sets DBLP and MEDLINE. We randomly selected 100 queries for each data set and Table 1 gives some sample queries. We implemented the hybrid algorithm of XRANK [4] 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 Apache8 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

<table>
<thead>
<tr>
<th>IDS</th>
<th>Queries</th>
<th>Typed Queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>xml keyword search</td>
<td>xml keyw search</td>
</tr>
<tr>
<td>Q2</td>
<td>falsoob similarity alike</td>
<td>false siml kd</td>
</tr>
<tr>
<td>Q3</td>
<td>nick koudas approximate</td>
<td>nick four appr</td>
</tr>
<tr>
<td>Q4</td>
<td>approximate string divrad</td>
<td>appp str acru</td>
</tr>
<tr>
<td>Q5</td>
<td>keyword search sigmod</td>
<td>keyw sear sigm</td>
</tr>
<tr>
<td>Q6</td>
<td>interactive vldb 2006</td>
<td>intera vld 2f</td>
</tr>
<tr>
<td>Q7</td>
<td>keyword search popakonstantinou</td>
<td>keyw sear papa</td>
</tr>
<tr>
<td>Q8</td>
<td>schema xquery jagadish</td>
<td>sche xque jag</td>
</tr>
<tr>
<td>Q9</td>
<td>xrank search shanthusasundaram</td>
<td>xran sloz</td>
</tr>
<tr>
<td>Q10</td>
<td>search vldb cohen</td>
<td>xse vld cohen</td>
</tr>
</tbody>
</table>

Table 2 shows the data set sizes, tree-index sizes, forward-index sizes, and index-construction time. As MEDLINE contains many more distinct keywords than DBLP, the index size on MEDLINE is larger than that on DBLP. We implemented cache-based algorithms.

Volume 2, Issue 2 March – April 2013
The quality of the result can be assessed by LCA-based method and MCT-based methods. We see that our MCT-based search method achieves much higher result quality than the LCA-based method by human judgment. 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.

The places included Canada, US, and UK. For each location, we measured the time of search 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 evaluated the efficiency of computing the prefixes on the tree that are similar to a query keyword. We implemented three methods to compute similar prefixes. (1) Incremental: We computed the active nodes of a query using the cached active nodes of previous prefix queries, using the incremental algorithm. This algorithm is applicable when the user types a query letter by letter. (2) Non-Incremental: We computed active nodes from scratch. This case happens when a user copies and pastes a long query, and none of the active nodes of any prefix queries has been computed. It also corresponds to the traditional search case, where a user submits a query and clicks the “Search” button. (3) Gram-Based: We built gram inverted lists on all prefixes with at least three letters using the method described in [10]. The index structure can be used to compute similar prefixes for keywords with at least four letters. Table 3 shows the performance results of these three methods.

### Table 3: Memory size for computing prefixes similar to a keyword

<table>
<thead>
<tr>
<th>Method</th>
<th>Memory for storing indexes</th>
<th>Memory for computing similar words</th>
<th>Total memory usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incremental</td>
<td>36</td>
<td>4</td>
<td>40</td>
</tr>
<tr>
<td>Non-Incremental</td>
<td>36</td>
<td>0.2</td>
<td>36.2</td>
</tr>
<tr>
<td>Gram-Based</td>
<td>187</td>
<td>0</td>
<td>192</td>
</tr>
<tr>
<td>Incremental</td>
<td>185</td>
<td>12</td>
<td>177</td>
</tr>
<tr>
<td>Non-Incremental</td>
<td>185</td>
<td>0.5</td>
<td>185.5</td>
</tr>
<tr>
<td>Gram-Based</td>
<td>713</td>
<td>21</td>
<td>733</td>
</tr>
</tbody>
</table>

### 7. CONCLUSION

In this paper, we considered type-forward search on data like XML. We proposed an efficient incremental algorithm to respond single-keyword queries that are treated as prefix conditions. We also considered different algorithms for computing the answers to a query with multiple keywords. We developed well-organized algorithms for incrementally computing answers to queries by using cached results of prior queries in order to get a high interactive speed on huge data sets. We examined the LCA-based method to interactively discover the predicted answers, and developed a minimal-cost-tree-based search method to capably and step by step recognize the nearly all relevant answers. We devised a forward-index structure to further improve search performance. There are numerous troubles for type-forward search. Firstly, is about how to support ranking queries capably. Another one is how to deal with huge amounts of data when the index structures cannot fit into the memory.

### REFERENCES


AUTHORS

1Supriya sivapuja has received her MCA degree at Annamacharya Institute of Technology & Sciences, Rajampet affiliated to JNTUA in 2009 and pursuing M.Tech in Computer Science and Engineering at PBR Visvodaya Institute of Technology & Science (2011-2013), Kavali affiliated to JNTUA. She worked as Assistant Professor in Audisankara College of Engineering and Technology, Gudur, Andhra Pradesh, India and had 2 years of experience in teaching.

2Sk. Mohiddin has received his B.Tech in Information Technology and M.Tech degree in Computer science from JNTU, Hyderabad in 2006 and JNTU, Anantapur in 2010 respectively. He is dedicated teaching field from the last 7 years. He has guided 6 P.G and 20 U.G students. His research areas included Data Mining. At present he is working as Assistant Professor in PBR Visvodaya Institute of Technology & Science, Kavali, Andhra Pradesh, India.

3S. Srikanth Babu has received his MCA degree at A.V.S College of Engineering &Technology, Venkatachelam affiliated to JNTUA in 2010 and pursuing M.Tech in Computer Science and Engineering at PBR Visvodaya Institute of Technology & Science (2011-2013), Kavali affiliated to JNTUA. He worked as Assistant Professor and had 1 year of teaching experience.

4Srikar Babu S.V. has received his B.Tech degree in Electronics and Communication Engineering at Priyadarshini College of engineering & Technology, Kanaparthipuduru affiliated to JNTUA in 2012. Currently, he is working as Associate Consultant in OFSS, Bangalore.