Improving the Time and Space Performance for Processing Keyword Queries on XML Databases

YAHUI CHANG AND WEI-HSI LEE
Department of Computer Science and Engineering
National Taiwan Ocean University
Keelung, 202 Taiwan
E-mail: yahui@ntou.edu.tw

For an XML database, it is important to automatically reason meaningful answers for a given keyword query. Particularly, the MaxMatch algorithm is proposed to output contributors, which represent more (or equal number of) keywords than those contained by their sibling nodes. Such concept can distinguish the importance of sibling nodes and has attracted a lot of attention. In this paper, we intend to improve the time and space performance of the original approach. We first propose the LevelPrune algorithm, which only needs to construct the smallest necessary intermediate tree and can also reduce the times of processing nodes. We then extend the idea of constructing trees level by level to optimize physical representations, and design the corresponding algorithm LevelPrune+. According to the experimental results, our proposed algorithms outperform other state-of-the-art approaches by an order of magnitude in some cases. Moreover, the index size of LevelPrune+ is significantly reduced compared with that of the original approach.

Keywords: XML database, query processing, keyword search, time performance, space performance

1. INTRODUCTION

As the XML (eXtensible Markup Language) technology emerged as the de facto standard for information sharing and data exchange on the Web, XML data management and query processing have attracted a lot of attention from the academic and business communities. Since the syntaxes of the standard XML queries, e.g., XQuery, are very complex for ordinary users, the style of keyword queries, which has been long adopted to search documents, is therefore getting attention. For example, consider the tree structure depicted in Fig. 1. It corresponds to an XML document representing the players in 1998 Major League1, where elements and attributes are denoted as nodes, the nesting relationships are depicted by edges, and each node is associated with a unique Dewey number [17]. Suppose a user wishes to know which team a player named “Jim” belongs to and his position. He only needs to issue a simple keyword query such as $Q_1 = (Jim, POSITION, TEAM_NAME)$.

The advantage of keyword queries is that it provides a convenient interface for users to obtain desired information from XML documents, but irrelevant data may be returned due to lacking exact query semantics. Therefore, many researchers investigate how to automatically infer reasonable answers for users based on the tree structure of XML data, and advocate the LCA-based technique, where an LCA (Lowest Common

Received August 15, 2013; revised November 11, 2013; accepted January 5, 2014.
Communicated by Wang-Chien Lee.
* This work was partially supported by the National Science Council under No. NSC 102-2221-E-019-063-.
1 http://www.cafeconleche.org/books/biblegold/examples/.

433
Ancestor) contains every keyword under its subtree at least once [7, 11, 14, 20, 23]. Particularly, the concept of \textit{SLCA} (Smallest Lowest Common Ancestor) [20], which excludes those LCAs which have descendant nodes that also contain all query keywords, is very welcome. Based on such concept, the researchers in [14] further proposed to return only \textit{contributors}, which represent more (or equal number of) keywords than those contained by their sibling nodes. Consider $Q_1$ and the XML tree in Fig. 1, where two SLCA nodes, \textit{i.e.}, TEAM (1.2) and TEAM (1.3), are identified. Since the subtree rooted at node 1.2.2 contains keywords \textit{Jim} and \textit{POSITION}, but the subtree rooted at node 1.2.3 only contains the keyword \textit{POSITION}, the latter subtree is not a contributor and is pruned by the former subtree. The answers are the gray areas depicted in Fig. 1.

![Fig. 1. A sample XML tree.](image)

The concept of contributors can output meaningful answers most of the time, so it has got a lot of attention and it is important to identify the corresponding answers efficiently. The \textit{MaxMatch} algorithm proposed in the original paper [14] consists of two parts: (1) identifying SLCA, and (2) outputting contributors. Specifically, for the first part, the MaxMatch algorithm will retrieve all matches, where a match is a node whose tag name or content contains a given query keyword, and then apply the procedures given in [20] to find SLCA. As to the second part, MaxMatch first groups matches based on SLCA. For each group, it then constructs an intermediate tree in postorder that consists of all nodes along the path from each match up to the SLCA, which is called a \textit{match tree}. It then traverses each match tree in preorder and compares the match information between siblings to identify contributors. Many research efforts have been proposed to improve the performance of MaxMatch, where most of them focus on the first part [6, 22]. However, actually the second part requires a lot more time than the first part does. As will be shown later in Table 4 of Section 7, the second part usually constitutes 85.67%-98.6% of the total computation time. Therefore, we will focus on improving the performance of the second part, \textit{i.e.}, outputting contributors, in this paper.

Based on the above brief description about MaxMatch, observe that the nodes in the match tree are first created in postorder for propagating the match information of descendants to ancestors, and are then examined in preorder for identifying contributors. In this paper, we improve this procedure by utilizing the characteristics of Dewey encodings, to support the top-down manner of constructing intermediate trees and identifying contributors right away. Therefore, our approach visits nodes in the intermediate trees only once instead of twice as in MaxMatch, and achieves better efficiencies. Besides, we formalize the concept of the \textit{smallest necessary intermediate tree} by identifying a class
of nodes in the match tree which cannot be part of answers, and avoid creating those nodes at all. This makes the intermediate trees created by our approach usually consist of less nodes than those constructed by MaxMatch. We have designed the LevelPrune algorithm to construct the intermediate tree level by level, and this leveling idea is further used to design a special way of data layouts and encoding methods to improve the space utilization as well. Specifically, the contributions of this paper are summarized as follows:

- We propose the LevelPrune algorithm, which is based on the characteristics of Dewey encodings and applies the simple array structure to efficiently construct the intermediate tree level by level. We show that the intermediate tree it creates is the smallest necessary one for identifying contributors, and explain how it can only process nodes in the tree once instead of twice as in MaxMatch.
- We further discuss how to achieve better space utilization by a special encoding scheme and index compression mechanism based on the leveling idea, and propose the corresponding LevelPrune+ algorithm.
- We have conducted an extensive performance study from all aspects. In terms of space performance, experimental results show that the sizes of the intermediate tree and the index created by our approach are a lot smaller than those of MaxMatch. As to the time performance, experimental results demonstrate that LevelPrune and LevelPrune+ usually outperform other state-of-the-art algorithms by an order of magnitude.

The rest of this paper is organized as follows. We first discuss related work in Section 2 and briefly introduce the MaxMatch algorithm in Section 3. The basic idea and the core data structures of our approach are discussed in Section 4. We then describe the LevelPrune algorithm in Section 5. The physical representation designed to improve the space utilization is further given in Section 6. Finally, we perform the experimental studies in Section 7 and conclude this paper in Section 8.

2. RELATED WORK

XML keyword search has been an important research issue. The earliest attempts apply the techniques seen in the information retrieval (IR) field. There have been many relevant research results and we only discuss some representative ones here. For example, researchers study how to efficiently compute the scores of XML elements [1], or assign appropriate scoring functions to reflect the characteristics of both contents and structures for XML data [2, 7]. In addition, the authors in [9] identify the specific goals that a semantically meaningful result snippet should meet, and construct the system eXtract to generate effective snippets for XML search results. The authors in [3] discuss the guidelines for a search engine and propose a novel XML TF*IDF ranking strategy to rank the individual matches of all possible search intentions. The authors in [6, 19] design novel algorithms to return Top-K answers. The works in [4, 5] study how to efficiently process XML queries with both structural and full-text constraints.

Another track of research directions, which our paper belongs to, mainly explores the tree structure represented by the XML data to automatically infer all meaningful
results for users. In addition to the SLCA approach [20] and the MaxMatch system [14] discussed in Section 1, there are the concept of Minimum Connecting Trees [8], MLCA [11], entities and attributes [15], ELCA [23], and SFQI [18]. Besides, the authors in [16] identify the node that is nearest to a given node among all the nodes associated with the input keyword. The authors in [10] determine the top-k SLCA nodes on probabilistic XML data.

The field of research most related to our work improves the efficiency of identifying meaningful answers. For example, the researchers in [6] propose a new labeling scheme to facilitate the computation of SLCAs, and the work in [22] further improves the efficiency of identifying both SLCA and ELCA based on the IDList index. In this research, we apply the original approach in [20] to compute the SLCA due to two reasons. First, our work focuses on the part of identifying contributors, not SLCAs. Second, the overall performance of our approach can be further improved by applying the above research results. As to efficiently producing the output, the SingleProbe algorithm [12] constructs a single match tree based on all matches, and identifies SLCA nodes while traversing the match tree. The advantage of such approach is that it only processes matches once, but the single match tree may be larger than those created by MaxMatch, and takes more time to process. We will perform detailed experimental studies with this algorithm later in Section 7. In contrast, the work in [13] avoids unnecessary index accesses when outputting the tag of XML data, which can be incorporated to our framework. The concept of the tightest matched subtree proposed in [21] intends to avoid producing redundant nodes, and will produce different outputs from ours. Therefore, we will not compare the performance with these two works.

3. PRELIMINARIES

In this section, we formally define relevant notions and explain the MaxMatch algorithm. The sample XML tree given in Fig. 1 will be used in the running examples throughout this paper.

3.1 Definitions

Some of the following definitions are adopted from [14], but are slightly modified from their original representations to be easily used in our framework. We also apply $Q_1$, i.e., $(Jim, POSITION, TEAM\_NAME)$, as an example to explain these definitions.

**Definition 1:** A node is a *match* if its tag name or content contains a given query keyword.

**Definition 2:** Given a query consisting of $w$ keywords: $k_1, ..., k_w$, let the node $t$ be an LCA of a set of matches $S$ corresponding to those keywords. The *match tree* rooted at $t$, denoted as $mTrees(t)$, consists of the nodes along the path from each match in $S$ up to $t$.

**Definition 3:** The *descendant matches* of a node $n$, denoted as $dMatch(n)$ or $n.dMatch$, is a set of query keywords, each of which has at least one match in the subtree rooted at $n$. $dMatch(n)$ could be seen as a bit array of size $w$ or a decimal value for simplicity, where $w$ is the number of keywords, and each keyword corresponds to its own bit.
Definition 4: A node $n$ is a contributor if (i) $n$ is a descendant of a given SLCA or $n$ itself is one of the SLCA's, and (ii) $n$ does not have a sibling $v$ such that $dMatch(v) \supset dMatch(n)$.

Definition 5: A match node $n$ is considered relevant if (i) $n$ has an ancestor-or-self $t$ such that $t$ is one of the SLCA's, and (ii) each node on the path from $n$ to $t$ is a contributor.

Definition 6: Given an XML tree and a set of query keywords, a query result is defined as all of the relevant matches (including value children, if any) contained in the subtree rooted at an SLCA. The paths from the SLCA to each relevant match will be also output, and each of such trees is called an answer tree.

Example 1: Observe that node 1.2.2.1.1 is a match for the keyword Jim, nodes 1.2.2 and 1.2.3.2 are matches for the keyword POSITION, and node 1.2.1 is a match for the keyword TEAM_NAME, so $mTree(1.2)$ is the subtree depicted in Fig. 2. Also note that $dMatch(1.2.2) = \{Jim, POSITION\}$ and $dMatch(1.2.3) = \{Position\}$, so node 1.2.3 is not a contributor by Definition 4. This in turn makes node 1.2.3.2 not a relevant match. Therefore, the answer tree based on node 1.2 will be the match tree in Fig. 2 excluding the subtree rooted at node 1.2.3.

The following definition, which summarizes the $dMatch$ values of all its child nodes, is additionally proposed by MaxMatch to assist in identifying contributors:

Definition 7: The $dMatchSet$ value of a node $n$, denoted as $dMatchSet(n)$ or $n\_dMatch\_Set$, is a bit array of size $2^w$ to record the $dMatch$ values of $n$’s children. All the bits are initialized as 0 (standing false) in the beginning. The $j$th bit (from left to right), $j \leq 2^w - 1$, is set to 1 (standing true) if it has at least one child $n_c$ such that $dMatch(n_c) = j$.

In Fig. 2, we denote the $dMatchSet$ bit arrays (noted as $dms$) along with the $dMatch$ bit arrays (noted as $dm$) for each node. Here the keywords (from left to right) correspond to the first (right-most), the second, and the third bits of $dMatch$. For example, $dMatch(1.2) = 111_{16}$ because of the matches 1.2.1, 1.2.2.1.1, and 1.2.3.2. Also, $dMatchSet(1.2) = 00111000_{16}$ since the $dMatch$ values of its child nodes 1.2.3, 1.2.2, 1.2.1 are 2, 3, 4, respectively. The usage of these structures will be explained further in the next subsection.


3.2 MaxMatch

The MaxMatch algorithm is briefly described on the left of Fig. 3. As shown in L1-L4, it retrieves matches corresponding to each query keyword, computes the SLCAs by the algorithm given in [20], merges all the matches into a single sorted node list, and groups them according to their SLCA ancestors. Then, for each SLCA group, as shown in L5-L7, it invokes procedure ConstructTree to construct the match tree in postorder and compute the correct value of dMatch and dMatchSet for each node. It then invokes procedure PreorderTrav to traverse each match tree in preorder and output the answer tree.

The functionality of the statements in L1-L4 is pretty clear. For the following, assume that procedure GroupMatches outputs a set of groups. Each group (i, M) consists of an SLCA node t and the sorted matches M under the subtree rooted at t. Procedure ConstructTree will build match trees in the manner of postorder tree traversal. Each tree is composed of the nodes on the paths from every match m up to the SLCA node t. For each node n from m (exclusively) up to t, the jth bit of n.dMatch is set to 1 if n matches the jth keyword. Besides, n.dMatch is further updated according to the bitwise OR operator with np.dMatch, and np.dMatchSet[n.dMatch] is set to true, where np and np are the parent and the child of n, respectively.

After constructing the match tree, procedure PreorderTrav performs a preorder tree traversal to identify contributors by function IsContributor (the right of Fig. 3), which takes a node n as the input and determines whether it is a contributor or not. In the for loop, np.dMatchSet[i] = true indicates that n has a sibling whose dMatch value is j. If bitwise AND(i, j) = i, i will be a proper subset of j, and the corresponding node is not a contributor. Consider query Q1, where the values of dMatch(1.2.2) and dMatch(1.2.3) are 011 and 010, respectively. Since dMatch(1.2.3) < dMatch(1.2.2) and AND(dMatch(1.2.3), dMatch(1.2.2)) = dMatch(1.2.3), node 1.2.3 is eventually pruned by node 1.2.2.

3.3 Time and Space Complexity of MaxMatch

Let |M| denote the total frequencies of query keywords, w be the number of query keywords, d be the maximum depth of the XML tree, and |D| be the size of the XML tree. The total time complexity of MaxMatch is \(O(\min\{|D|, \sum|M|})\). Interested readers please see the original paper [14] for the detailed analysis.
As to the space complexity, the algorithm first retrieves all matches and represent them in a node list, which requires $O(d|M|)$ space, since the size of a node depends on the longest Dewey encoding. After identifying all SLCAs, it constructs all corresponding matches trees, with the size in $O(n\min(|D|, d|M|))$. Each node in the match tree is associated with a $w$-sized $dmatch$ array and a $2^w$-sized $dmatchset$ array, which dominate others, so the total space complexity is $O(n\min(|D|, d|M|) \cdot 2^w)$.

4. BASIC IDEAS AND DATA STRUCTURES

In this section, we will explain the main idea of our approach and the underlying data structures. The query $Q_2 = (MLB, James, POSITION)$ will be used as the running example. This query only corresponds to one SLCA node, which is the root node 1.

4.1 Basic Ideas

The main purpose of this paper is to reduce the number of nodes which need to be processed for determining contributors, and only construct the smallest necessary intermediate tree, which is defined and justified as follows:

**Definition 8**: Given an XML tree $T$, an input query $Q$, and one of the corresponding SLCA nodes $t$, let the nodes in $mTree(t)$ be denoted as $V$. Classify all nodes in $V$ into three disjoint types: (1) $v_c$: node itself is a contributor; (2) $v_{pc}$: node itself is not a contributor, but its parent node is a contributor; (3) $v_{pnc}$: both the node and its parent node are not contributors. Define the smallest necessary intermediate tree $S$ as a subtree of $mTree(t)$ formed by all $v_c$ and $v_{pc}$ nodes.

**Example 2**: Consider Query $Q_2$. Its corresponding classification of nodes is shown in Fig. 4, where nodes $v_c$, $v_{pc}$ and $v_{pnc}$ are represented by solid circles, hollow circles, and crosses, respectively. For example, node 1.2 is a $v_c$ and node 1.3 is a $v_{pc}$. This also makes node 1.3.2 a $v_{pnc}$.

**Theorem 1**: Given an $mTree(t)$, where each node only represents $dMatchSet$ and $dMatch$ values without extra information, the nodes in $S$ will be the minimum set of nodes in $mTree(t)$ which needs to be examined to identify the answer tree.
Proof: It is trivial that \( S \) must include all contributors, i.e., \( v_c \) nodes. Besides, since a contributor is determined by comparing its \( dMatch \) values with those of its sibling nodes, which are either \( v_c \) or \( v_{pc} \), all the \( v_{pc} \) nodes also need to be examined.

Recall that MaxMatch first constructs a match tree in postorder to determine the values of \( dMatch \) and \( dMatchSet \), and then examines nodes in \( S \) to identify the answer tree in preorder. It is obvious that those \( v_{pc} \) nodes are constructed in the first step but are not used in the second step. Our intention is therefore to directly construct the tree \( S \) and determine which nodes to output at the same time. In the following subsection, we will discuss how the proposed data structures are designed to help achieve this task.

4.2 The \( kwMatch \) and \( dMatch \)-range Arrays

The challenge of our approach mainly lies in how to construct the tree \( S \) from top to bottom and also being able to determine the \( dMatch \) and \( dMatchSet \) values of a node. Specifically, the following two things have to be done: (1) inferring the child nodes from a given node, and (2) computing the \( dMatch \) value when constructing a node. Note that the nodes in \( S \) are actually part of the match tree, so we can utilize the information stored in \( kwMatch \). Recall that for a keyword \( w \), \( kwMatch[w] \) represents all elements (in Dewey encoding) which correspond to the keyword \( w \) in a node list, as shown in Fig. 3 and explained in Section 2.2. Since the Dewey encoding represents all ancestors of a particular node, we can also infer the descendant information from a Dewey encoding in the reverse direction. For example, suppose the \( kwMatch \) corresponding to the keyword \( POSITION \) consists of the match with the encoding 1.2.2.2. From the first and second components of this encoding, we can infer that node 1 has a child node 1.2, and the \( dMatch \) sets of both nodes contain the keyword \( POSITION \).

Obviously, an important underlying task is to retrieve the component values of Dewey encodings level by level efficiently. Therefore, instead of using node lists as in MaxMatch, we redesign the structure and let each \( kwMatch_i \) be an \( r_i \times c_i \) two dimensional integer array, where \( r_i \) is the number of corresponding matches, and \( c_i \) is the largest number of components for all Dewey encodings of those matches\(^2\). Therefore, for the \( m \)th match corresponding to the \( i \)th keyword, its \( n \)th component value of the Dewey encoding will be represented in \( kwMatch_i[m,n] \). By adopting such array structure, we can efficiently produce the required nodes in a top-down manner.

![Fig. 5. Sample KwMatch arrays.](image)

**Example 3:** The three \( kwMatch \) structures of \( Q_2 \) over the sample XML tree are shown

---

\(^2\) Note that \( c_i \) is bounded by the depth of the XML tree.
in Fig. 5. Observe \( \text{kwMatch}_2 \). From the first component value of each Dewey encoding, we can infer that node 1 contains the query keyword POSITION. From all the second component values, we can further infer that its child nodes 1.2, 1.3, 1.4 also contain this query keyword, but not the child node 1.1.

To easily compute the \( dMatch \) value of a node, we additionally associate each constructed node a data structure called \( \text{dmatch-range} \), or \( \text{DR} \) in short, to indicate where its descendants are represented in each \( \text{kwMatch} \) array. Suppose the input query consists of \( w \) keywords. Each \( \text{dmatch-range} \) will be an integer array with dimension \( 2w \). Consider node \( n \) and the \( i \)th keyword. \( n.\text{DR}[2i] \) and \( n.\text{DR}[2i+1] \) will represent the first position and the last position of \( n \)’s descendant nodes in \( \text{kwMatch}_i \), respectively. For example, the \( \text{DR} \) array of node 1 is equal to \([0, 0, 0, 0, 0, 3]\), with respect to the three \( \text{kwMatch} \) structures in Fig. 5. It covers all indexes of the three \( \text{kwMatch} \) structures since all nodes are descendants of node 1.

We now explain how to use the \( \text{dmatch-range} \) structure of a node to infer its \( dMatch \) value. Consider node \( n \). Initially, we assign all component values of the associated \( \text{dmatch-range} \) array as “-1”. For the \( i \)th keyword, we then access \( \text{kwMatch}_i \) to determine the values of \( n.\text{DR}[2i] \) and \( n.\text{DR}[2i+1] \). If \( n.\text{DR}[2i] \) is not negative after the process, node \( n \) must have at least one descendant node which matches the \( i \)th keyword. We can therefore use this information to obtain the full \( dMatch \) array of node \( n \). Note that the \( \text{dmatch-range} \) array can be efficiently constructed by linearly scanning the \( \text{kwMatch} \) structures, since the encodings of nodes are represented in an increasing order in \( \text{kwMatch} \), and it is pretty simple to determine the ancestor/descendant relationship based on Dewey encodings. For example, consider node 1.2 and the 2nd keyword “POSITION”. Since \( \text{kwMatch}_2[0] = 1.2.2.2 \) and 1.2 is the prefix of 1.2.2.2, node 1.2.2.2 represents a descendant of node 1.2. Similarly, \( \text{kwMatch}_2[1] \), i.e., 1.2.3.2, is also a descendant node, but \( \text{kwMatch}_2[2] \), i.e., 1.3.2.2, is not. Therefore, we can determine that the values of \( \text{DR}[4] \) and \( \text{DR}[5] \) for node 1.2 are 0 and 1, respectively.

Another advantage of the \( \text{dmatch-range} \) array is that it can be also used to infer the corresponding SLCA group for an SLCA node. As an example, recall that the \( \text{dmatch-range} \) structure for the SLCA node 1 is \([0, 0, 0, 0, 0, 3]\), so the corresponding SLCA group consists of all six matches represented in the three \( \text{kwMatch} \) structures. By using this structure, we do not need to merge all matches into a single node list for performing grouping, as procedure MergeMatches in MaxMatch (L3 of Fig. 3) does.

### 5. LEVELPRUNE

Our proposed LevelPrune algorithm is listed in Fig. 6. As discussed in Section 4, the matches are represented in a two-dimensional \( \text{kwMatch} \) arrays (L1), and the \( \text{dmatch-range} \) arrays are used to infer each SLCA group (L3) after getting all the SLCAs (L2). In this section, we continue to discuss how the core \( \text{PruneMatch} \) procedure (L5) can utilize these data structures to avoid processing the intermediate tree twice. We will first discuss the supporting function \( \text{FindNextChild} \), which is invoked to construct the nodes in intermediate trees, and then present the \( \text{PruneMatch} \) procedure.

---

Note that the keyword sequence and the array index both start from 0.
Algorithm LevelPrune
Input: keyword
Output: a set of answer trees
1: kwMatch ← FindMatch(keyword)
2: SLCA ← FindSLCA(kwMatch)
3: group ← GroupMatches(kwMatch, SLCA)
4: for each group[i] corresponding to the SLCA t do
5: PruneMatch(t)

Fig. 6. Algorithm LevelPrune.

5.1 Function FindNextChild

Recall that a contributor is determined based on comparing the dMatch values among all sibling nodes. For a parent node n, Function FindNextChild (Fig. 7) is designed to create the child node nc along with its dMatch value, where nc has the smallest Dewey encoding among all unproduced child nodes. Therefore, by repeatedly invoking Function FindNextChild, n’s child nodes will be produced in preorder sequence.

The major part of the algorithm, i.e., the for loop in L2-L4, determines the value of the variable least_no, which represents the smallest number among the last component values of all remaining child encodings. Note that if the encoding of the parent node n consists of l components, all child nodes of n will have l + 1 components (L1). Also, since n.DR[2i] points to the first position of n’s descendant nodes in kwMatch, and the kwMatch array presents encodings in an increasing order, we can infer that n.DR[2i] will represent the smallest encoding among all its descendant nodes in kwMatch. Therefore, as the same idea of the Merge-Sort algorithm, to find the child with the smallest encoding, we only need to compare all the encodings pointed by n.DR[2i], 0 ≤ i < w. For the remaining part of this algorithm, the codes in L5 determine that node n has no descendant nodes in kwMatch to examine and stop this algorithm. Otherwise, we will first determine the encoding of nc (L6), and then update the dmatch-range arrays for the parent node n and the identified child node nc in the following for loop.

Function FindNextChild
Input: node n
Output: node nc with its dmatch_range
1: least_no ← 0; n.level ← n.level + 1
2: for each query keyword i do
3: if n.DR[2i] ≠ −1 and (least_no = 0 or least_no > kwMatch[n.DR[2i], n.level]) then
4: least_no ← kwMatch[n.DR[2i], n.level]
5: if least_no = 0 then return Null
6: nc.encoding = append(n.encoding, least_no)
7: for each query keyword i do
8: if n.DR[2i] = −1 or least_no ≠ kwMatch[n.DR[2i], n.level] then continue
9: update n.DR[2i], n.DR[2i + 1], nc.DR, and nc.DR[2i + 1];
10: return nc

Fig. 7. Function FindNextChild.

Example 4: Consider query Q2, whose corresponding kwMatch structures are shown in Fig. 5 and 1.DR = [0, 0, 0, 0, 0, 3]. From the values of DR[0], DR[2], and DR[4], we
identify that the smallest encoding among the child nodes of node 1 is 1.1, so we output node 1.1 along with its dmatch-range and update the dmatch-range of node 1, as shown in Fig. 8. In that figure, we also use the dotted arrow to show the range in kwMatch covered by the dmatch-range structure. We can clearly see that node 1.1 has descendant nodes only in kwMatch₁, while node 1 still has unprocessed descendant nodes in kwMatch₂ and kwMatch₃. After three more iterations, we produce all child nodes of node 1 along with their dmatch-range arrays, as shown in Fig. 9. Observe that now the dmatch-range array of node 1 has all the component values as −1.

5.2 Procedure PruneMatch

Before presenting the PruneMatch procedure, we first introduce the dMatchList structure. It is a temporary structure constructed for a parent node n, and consists of an array of node lists to represent all n’ produced child nodes based on the state of their dMatch arrays. Specifically, suppose there are w query keywords. The dimension of the dMatchList array is 2^w, corresponding to all possible combinations of query keywords. Each dMatchList component represents a linked list of child nodes of n which have the same dMatch value. We also utilized one more structure, called node_seq. It is an inte-
ger array representing which dMatchList component a node is appended into. Particularly, if \( \text{node}_\text{seq}[i] = j \), it means that the \( i \)th node is represented in \( \text{dMatchList}[j] \). This structure is used to assist us in outputting nodes in preorder sequence, as will be shown later.

**Example 5:** The dMatchList for \( Q_2 \) over the sample XML tree (in Fig. 1) is shown in Fig. 10, where the corresponding dMatch set of each dMatchList component is shown on the left. The \( \text{node}_\text{seq} \) structure is shown on the bottom.

Procedure PruneMatch is listed in Fig. 11, which considers two levels, \( \text{curNodeList} \) and \( \text{nextNodeList} \), at a time. The variable \( \text{nc}_\text{cnt} \) is used to represent the sequence number of each produced child node, and the final value represents how many child nodes the node \( n \) has. Besides, \( \text{dMatch} \) represents a binary number, and the function \( \text{num} \) converting a binary number to a decimal number. This procedure first appends the input node \( n \) into \( \text{curNodeList} \), initializes the variable \( \text{nc}_\text{cnt} \), and examines each node in \( \text{curNodeList} \) in sequence (L3-L18). For each such node, we output it and initialize the dMatchList structure. The for loop in L5-L8 then produces a child node \( n_c \) of \( n \), obtains its dMatch value, and puts \( n_c \) in the correct component of dMatchList. The following actions are based on how many child nodes are produced. If there is only one, which is definitely a contributor, we append it into nextNodeList for further processing (L10-L11). Otherwise, we need to determine if a child node is a contributor. The processing sequence of nodes is based on node_seq (L13), and the for loop in L15-L16 performs the same test as in Function IsContributor of MaxMatch (see Fig. 3). If a child node is identified as a contributor, we append it into nextNodeList for further processing (L17-L18).

<table>
<thead>
<tr>
<th>Procedure PruneMatch</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> node ( n )</td>
</tr>
<tr>
<td><strong>Output:</strong> the part of the answer tree rooted at ( n )</td>
</tr>
<tr>
<td>1: append ( n ) into ( \text{curNodeList} ); ( \text{nc}_\text{cnt} ) $\leftarrow$ 0</td>
</tr>
<tr>
<td>2: while ( \text{curNodeList} ) not empty do</td>
</tr>
<tr>
<td>3: for each node ( n ) in ( \text{curNodeList} ) do</td>
</tr>
<tr>
<td>4: output ( n ); initialize dMatchList</td>
</tr>
<tr>
<td>5: for each ( n_c ) produced by ( \text{FindNextChild}(n) ) do</td>
</tr>
<tr>
<td>6: for each ( i ) in query keyword do</td>
</tr>
<tr>
<td>7: ( \text{if} \ n.c.DR[2i] \neq -1 \text{ then set the } i \text{th bit of dMatch to 1} )</td>
</tr>
<tr>
<td>8: append ( n_c ) into ( \text{dMatchList}[\text{num}(n_c.dMatch)]; \text{node}<em>\text{seq}[\text{nc}</em>\text{cnt}] \leftarrow \text{num}(n_c.dMatch); \text{nc}_\text{cnt}++ )</td>
</tr>
<tr>
<td>9: ( \text{if} \ \text{nc}_\text{cnt} = 0 \text{ then do nothing} )</td>
</tr>
<tr>
<td>10: ( \text{else if} \ \text{nc}_\text{cnt} = 1 \text{ then} )</td>
</tr>
<tr>
<td>11: ( v \leftarrow \text{get the only node from} \ \text{dMatchList}[n_c.dMatch]; \text{append} \ v \text{ into nextNodeList} )</td>
</tr>
<tr>
<td>12: ( \text{else} )</td>
</tr>
<tr>
<td>13: for each ( i ) in ( \text{node}_\text{seq} ) do</td>
</tr>
<tr>
<td>14: is_contributor $\leftarrow$ true</td>
</tr>
<tr>
<td>15: for ( j \leftarrow i + 1; j &lt; 2^w; j++ ) do</td>
</tr>
<tr>
<td>16: ( \text{if} \ \text{dMatchList}[j].\text{size} &gt; 0 \text{ and AND}(i, j) = i \text{ then is_contributor } \leftarrow$ false; break;</td>
</tr>
<tr>
<td>17: ( \text{if} \ \text{is_contributor} = \text{true} ) then</td>
</tr>
<tr>
<td>18: ( v \leftarrow \text{get the first node from} \ \text{dMatchList}[i]; \text{append} \ v \text{ into nextNodeList;} )</td>
</tr>
<tr>
<td>19: ( \text{curNodeList} \leftarrow \text{nextNodeList}; \text{nextNodeList} \leftarrow \text{null}; )</td>
</tr>
</tbody>
</table>

Fig. 11. Procedure PruneMatch.
After processing all nodes in curNodeList, we will replace it with nextNodeList, and the procedure will stop when no nodes are left in curNodeList.

5.3 Time and Space Complexity

We now analyze the time complexity of the LevelPrune algorithm (Fig. 6). As before, $|M|$ denotes the total frequencies of query keywords, $w$ denotes the number of query keywords, $d$ is the maximum depth of the XML tree, and $D$ is the size of the XML tree. First, step 1 takes $O(|M|)$ to retrieve matches and step 2 takes $O(d|M|)$ to compute the SLCA nodes. In step 3, very match needs an LCA computation, which takes $O(d)$ time, to determine the group assignment. Thus, it takes $O(d|M|)$ time. In the final step, procedure PruneMatch can be divided into two parts. The first part invokes procedure FindNextChild to create nodes and update $dmatch-range$. For node creation, each node takes $O(w)$ to find its child with the smallest encoding, so it takes $O(\min\{|D|, d|M|\} \cdot w)$ time. As to $dmatch-range$ maintenance, each level of nodes needs to scan the structure $kwMatch$ once, so it takes $O(d|M|)$ time. The second part determines each produced node if it is a contributor, which takes $O(\min\{|D|, d|M|\} \cdot 2^w)$ time. Therefore, the time complexity for pruneMatch is $O(\min\{|D|, d|M|\} \cdot w)+O(d|M|)+O(\min\{|D|, d|M|\} \cdot 2^w) = O(\min\{|D|, d|M|\} \cdot 2^w)$. It is the same as that of MaxMatch.

As to the space complexity, LevelPrune first retrieves all matches and represents them in a two dimensional array, which requires $O(d|M|)$ space. After identifying all SLCA's, it constructs the temporary tree level by level. For each iteration, two node lists are represented and take $O(|M|)$. A single $dMatchList$ is maintained and reused for each iteration, which takes $O(2^w + |M|)$ since there are $2^w$ components in $dMatchList$, and the maximum number of nodes represented within the list is $|M|$. Therefore, the total space complexity is $O(d|M| + 2^w)$, which is less than that of MaxMatch.

6. PHYSICAL REPRESENTATIONS OF LEVELPRUNE+

Algorithm LevelPrune produces the intermediate tree level by level from top to bottom. In this section, we discuss the physical representations which are especially designed for such processing to improve its performance. The corresponding algorithm is called LevelPrune+.

6.1 Array Layout

As discussed in Section 5, Function FindNextChild accesses the structure $kwMatch$ to determine the child node $n_c$ with the smallest encoding for a given node $n$. Suppose the Dewey encoding of $n$ has $d$ components. We have to access all the component values represented at the $(d + 1)$th column in $kwMatch$. Since $kwMatch$ is an integer array with two dimensions and most programming languages adopt the row-major layout scheme, the values represented at the same column are scattered in memory. For example, the $kwMatch_2$ structure which corresponds to the query keyword POSITION will be represented in the memory as in Fig. 12 (a). To solve this problem, we implement the column-major layout scheme, as shown in Fig. 12 (b), so that the values represented at the
same level will be continuously stored in memory. Due to this characteristic of spatial locality, we can expedite the retrieval process against \( kwMatch \) arrays.

### 6.2 New Encoding Scheme

Function `FindNextChild` also needs to access the structure \( kwMatch \) to determine the \( \text{dmatch-range} \) for a certain node \( n \). Specifically, we have to determine which encoding represents the descendant nodes of \( n \). Recall that the descendant nodes are represented in contiguous spaces in \( kwMatch \), and the component values of their encodings tend to occur repeatedly. To avoid examining all encodings one by one, we apply a new encoding scheme to directly represent the number of occurrences of component values in \( kwMatch \).

Suppose each component of the new \( kwMatch \) array is denoted as \( new(i,j) \), and the component value of the original \( kwMatch \) array is denoted as \( M(i,j) \). The rule of assigning new encodings is as follows:

\[
new(i,j) = \begin{cases} 
-k & \text{if } M(i-1,j) = M(i,j) = \ldots = M(i+k-1,j) \text{ and } M(i-2,j) \\
& \neq M(i-1,j) \text{ and } M(i+k-1,j) \neq M(i+k,j) \\
M(i,j) & \text{otherwise}
\end{cases}
\]

That is, if a value \( v \) which continuously occurs from the \((i-1)\)th row till \((i+k-1)\)th row at the \(j\)th column, \( new(i,j) \) will be assigned the value \(-k\). We use the minus sign to distinguish it from the normal encoding, and the value \( k \) represents the number of occurrences in the following rows.

**Example 6:** As depicted in the \( kwMatch \) array of Fig. 12 (b), the value 1 occurs from \([0,0]\) till \([3,0]\). Therefore, \( new(1,0) = -(3-0) = -3 \). The complete \( kwMatch_2 \) array adopting the new encoding function is depicted in Fig. 13 (a). When determining the \( \text{dmatch-range} \) of node 1 concerning the 2nd query keyword, we only need to retrieve the values of \( kwMatch_2[0,0] \) and \( kwMatch_2[1,0] \), and can infer that \( I.DR[4] \) and \( I.DR[5] \) have the values 0 and 3, respectively.
6.3 Index Compression

Based on the new encoding scheme, the component values following a negative number can be inferred. Therefore, we represent the index by removing those inferable components to make the index smaller.

Example 7: The compressed index corresponding to the \( \text{kwMatch}_2 \) array based on the new encoding scheme is represented in Fig. 13 (b). Note that it requires twelve bytes for storage instead of sixteen bytes. Although the compression rate is only 75% in this example, empirically it is usually over 50%, as will be shown later in Section 7.

Algorithm LevelPrune\(^+\) differs from LevelPrune in coping with these physical representations. Specifically, after retrieving those matches from the compressed index, this algorithm will restore them as the structure in Fig. 12 (b), and also base on the column-major scheme to retrieve the required element from the array.

7. EXPERIMENTS

In this section, we design a series of experiments based on different datasets and varying types of queries to evaluate the time and space performance of the proposed algorithms. They are also contrasted with those of other representative works to show the pros and cons of different approaches.

Table 1. Data set.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Data Size</th>
<th>No of Nodes</th>
<th>Max/Avg Depth</th>
<th>Index Size</th>
<th>Compressed Size</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseball</td>
<td>10.1MB</td>
<td>516K</td>
<td>8/7.46</td>
<td>16.1MB</td>
<td>7.01MB</td>
<td>43.54%</td>
</tr>
<tr>
<td>Mondial</td>
<td>17.2MB</td>
<td>514K</td>
<td>7/5.2</td>
<td>33.2MB</td>
<td>13.4MB</td>
<td>40.36%</td>
</tr>
<tr>
<td>SwissProt</td>
<td>109MB</td>
<td>4990k</td>
<td>6/4.02</td>
<td>110MB</td>
<td>82.2MB</td>
<td>74.73%</td>
</tr>
<tr>
<td>DBLP</td>
<td>820MB</td>
<td>38000K</td>
<td>6/3.4</td>
<td>1587MB</td>
<td>928MB</td>
<td>58.48%</td>
</tr>
<tr>
<td>XMark</td>
<td>582MB</td>
<td>8350K</td>
<td>12/5.5</td>
<td>3038MB</td>
<td>1663MB</td>
<td>54.74%</td>
</tr>
</tbody>
</table>

7.1 Experimental Setup

In this paper, we propose Algorithm LevelPrune to construct the smallest necessary intermediate tree level by level, and design Algorithm LevelPrune\(^+\) to further improve the space utilization. The algorithms used for comparison are the MaxMatch algorithm [14] and the SingleProbe algorithm [12], which are state-of-the-art in literature for the problem of contributor identifications. We implement all algorithms in Microsoft Visual C++. AB-tree index, which is designed to efficiently retrieve all the matches of a given keyword, is also created based on the Oracle Berkeley DB\(^4\). We apply five datasets with different characteristics, such as large or small, real or artificial, flat or deep, to perform experiments, which are Baseball.xml\(^5\), Mondial\(^6\), SwissProt\(^7\), a subset of DBLP\(^7\).

\(^4\) http://www.oracle.com/technetwork/database/
\(^5\) http://www.ibiblio.org/xml/books/biblegold/examples/baseball/
\(^6\) http://www.cs.washington.edu/research/xmldatasets/
\(^7\) http://dblp.uni-trier.de/xml/
and XMark. For Baseball and Mondial datasets, we duplicate their original contents 10 times to make them larger. The relevant statistic data for all datasets are listed in Table 1. In that table, we also contrast the sizes of the original indices and the compressed indices. We can see that the new encoding scheme significantly compresses the indices of all datasets, and the size reduction is about a half or even over a half.

All experiments are performed on a personal computer with Intel Quad-Core 3.4 GHz CPU, 16GB memory, and the Microsoft Windows 7 operating system. We apply the hot cache strategies for evaluation. Specifically, for each query, we execute three times and calculate the average processing time of the last two. Besides, recall that the smallest necessary intermediate tree created by our approach includes nodes $v_c$ and $v_{pc}$, while MaxMatch additionally creates nodes $v_{pmc}$. We define the following equation to denote the ratio of the sizes of the intermediate trees constructed by two approaches, which will be used later to examine the performances of both systems:

$$\text{contribution rate} = \frac{|v_c| + |v_{pmc}|}{|v_{pc}| + |v_{pmc}|}.$$

Table 2. Test queries I.

<table>
<thead>
<tr>
<th>The Baseball dataset</th>
<th>The Mondial dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Keywords</td>
</tr>
<tr>
<td>TB1</td>
<td>Jim, Abbott, Outfield</td>
</tr>
<tr>
<td>TB2</td>
<td>Jim, Abbott, James, Baldwin, Starting Pitcher</td>
</tr>
<tr>
<td>TB3</td>
<td>PLAYER, Abbott, Baldwin</td>
</tr>
<tr>
<td>TB4</td>
<td>Tigers, Starting Pitcher, SURNAME</td>
</tr>
<tr>
<td>TB5</td>
<td>Tigers, Starting Pitcher, Outfield, SURNAME</td>
</tr>
<tr>
<td>TB6</td>
<td>Cordero, First Base</td>
</tr>
<tr>
<td>TB7</td>
<td>Cordero, First Base, Tigers</td>
</tr>
<tr>
<td>TB8</td>
<td>1998 Abbott TEAM</td>
</tr>
</tbody>
</table>

7.2 Evaluating Different Types of Meaningful Queries

In the first set of experiments, we apply meaningful queries with different combinations of tag names and values to run the experiments. The queries listed in Table 2 are some of the representative test queries, where TB1-TB8 are designed for the Baseball dataset and queries TM1-TM8 are against the Mondial dataset. The experimental results are depicted in Fig. 14, where MM, SP, LP, and LP’ represent the MaxMatch algorithm, the SingleProbe algorithm, the LevelPrune algorithm, and the LevelPrune' algorithm, respectively. First, observe that LevelPrune is more efficient than MaxMatch for all queries of both datasets. The reason is that LevelPrune can always produce less (or equal) nodes in all cases, and at the same time reduce the times of visiting nodes. In some cases,
such as TM2, the difference is even by an order of magnitude. The main reason is that TM2 has an extremely low contribution rate, which is 4.66% as shown in Table 3. This means that MaxMatch produces a lot more nodes than LevelPrune does, and is therefore particularly slower. Second, note that the performance of SingleProbe varies a lot, since it may need to process a lot of un-outputted nodes in the worst case, but may be quite efficient otherwise. However, LevelPrune is always faster than SingleProbe due to the time saved in processing the intermediate tree. Third, we can see that LevelPrune further outperforms LevelPrune and is the most efficient algorithm among four algorithms. However, since these two datasets are comparatively small, and the I/O time saved by retrieving compressed indices is minor compared with the CPU time, the difference of execution time of the two proposed algorithms is not very obvious.

![Graph showing execution time](image)

Fig. 14. Examining different types of meaningful queries.

<table>
<thead>
<tr>
<th>query</th>
<th>TM1</th>
<th>TM2</th>
<th>TM3</th>
<th>TM4</th>
<th>TM5</th>
<th>TM6</th>
<th>TM7</th>
<th>TM8</th>
</tr>
</thead>
<tbody>
<tr>
<td>ratio</td>
<td>7.97%</td>
<td>4.66%</td>
<td>18.84%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>50.84%</td>
<td>100%</td>
</tr>
</tbody>
</table>

To examine how our approach improves the performance in detail, we divide the total execution time into two parts. The first one, which is called the SLCA time, corresponds to the execution time starting from the first beginning till getting SLCAs; the second one, which is called the Contributor time, corresponds to the rest of the execution time. We compare the performance of LevelPrune and MaxMatch using the Mondial dataset. As depicted in Table 4, the second and the third columns, denoted by MMt and LPt, respectively, represent the total execution time of the two approaches; the fourth column represents the SLCA time, which is the same for both approaches; the next two columns, denoted by MMc and LPC, respectively, represent the Contributor Time of the two algorithms. From the ratio shown in the seventh column, we can see that in the original MaxMatch algorithm, the Contributor time is the major part of the total execution time. This supports our motivation for focusing on improving the efficiency of this part, as stated in Section 1. The last two columns show the ratios of execution time required by two algorithms at different stages. Observe that our approach significantly outperforms MaxMatch when only comparing the Contributor time, whose difference is around 1-2 orders of magnitude. This also leads to our better performance when comparing the total execution time, where the difference can be usually around one order of magnitude.
7.3 Evaluating Queries with Different Input Sizes

Recall that the time complexity of algorithms MaxMatch and LevelPrune is both affected by the number of total matches \((M)\) and the number of query keyword \((w)\). In this subsection, we apply the SwissProt and DBLP datasets to examine how these two factors affect the performance. The test queries, as listed in Table 5, are designed to control the size of input, so they may or may not be meaningful. Specifically, for TS1-TS4 and TD1-TD4, we keep the number of matches the same, and vary the number of query keywords from three to six. For TS5-TS7 and TD5-TD7, we keep the number of query keyword the same, and vary the size of matches from \(10^2\) to \(10^4\).

Table 5. Test queries II.

<table>
<thead>
<tr>
<th>The SwissProt dataset</th>
<th>The DBLP dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>(M)</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>TS1 (10^2)</td>
<td>3</td>
</tr>
<tr>
<td>TS2 (10^2)</td>
<td>4</td>
</tr>
<tr>
<td>TS3 (10^5)</td>
<td>5</td>
</tr>
<tr>
<td>TS4 (10^5)</td>
<td>6</td>
</tr>
<tr>
<td>TS5 (10^2)</td>
<td>3</td>
</tr>
<tr>
<td>TS6 (10^3)</td>
<td>3</td>
</tr>
<tr>
<td>TS7 (10^4)</td>
<td>3</td>
</tr>
</tbody>
</table>

Examine Fig. 15 (a), which shows the experimental result using the SwissProt dataset. Note that the execution time is depicted in log scale. We can see that MaxMatch and SingleProbe are the slowest. On the other hand, LevelPrune and LevelPrune+ are still the most efficient ones. The difference is more significant than before since SwissProt is a larger dataset than Baseball and Mondial.

We further compare the execution time for TS1-TS4. Note that it does not increase along with the number of query keywords. The reason is that although the number of matches is kept the same for these four queries, the number of manipulated nodes is not. Take MaxMatch as an example. The number of nodes in the match tree, \(|v|\), is \(115786\) for Query TS1. However, it is only \(31726\) for TS4. Therefore, TS1 requires more processing time than TS4 although the number of query keywords is less. We then compare
the execution time for TS5-TS7. Although the execution time for SingleProbe, LevelPrune, LevelPrune’ all increase along with the number of matches, the execution time of TS5 and TS6 is about the same for MaxMatch. The reason is that the value of $|v|$ is 10032 for query TS5, but it is only 1141 for TS6. Therefore, the number of nodes in the match tree compensates the difference in the number of matches.

The experimental result for the DBLP dataset is shown in Fig. 15 (b). It is basically similar to the result of Fig. 15 (a), and we omit the detailed discussion.

7.4 Evaluating Queries with Different Output Sizes

In this subsection, we apply the DBLP dataset to run the experiments. The test queries are designed to see how the sizes of outputs affect the performance of different approaches.

In the first experiment, we intend to control the number of $v_c$, which are the contributor nodes. We let the query include the root element `dblp` and a number which represents the content of the element `year`. By giving different numbers, we can control the number of matches, which in turn control the number of $v_c$ for this particular set of queries. The test queries are listed in Table 6, and the experimental result is illustrated in Fig. 16 (a). We can see that the execution time for all algorithms all increase along with the number of contributors, which are actually the number of intermediate nodes for all algorithms in this particular case. Observe that LevelPrune and LevelPrune’ are still the fastest, and outperform MaxMatch to a large extent.

| No | $|v_c|$ | Keywords |
|----|-------|----------|
| TD8 | 30004 | dblp, 1987 |
| TD9 | 66196 | dblp, 1987, 1988 |
| TD10 | 93478 | dblp, 1995 |
| TD11 | 125782 | dblp, 1998 |

In the second experiment, we examine the impact of the contribution rate, which is defined in Eq. (2) and represents the size differences of the constructed intermediate trees. We randomly produce 33 queries, and divide them into three groups based on their
contribution rates. The execution time of the same group is then averaged. As shown in Fig. 16 (b), in the leftmost group, which corresponds to the smallest contribution rate, the differences between the execution time of our approaches and that of MaxMatch are the most obvious. Fig. 16 (c) illustrates the results of only comparing the Contributor time, and we can get the same conclusion.

Fig. 16. Effects of output sizes.

8. CONCLUSION

In this paper, we investigate the techniques of efficiently producing meaningful results for XML keyword queries. We first propose the LevelPrune algorithm to eliminate unnecessary node creations and reduce the times of processing intermediate nodes. It is further improved in terms of physical representations and the corresponding algorithm is called LevelPrune+. Extensive experimental results show that both algorithms are more efficient than the state-of-the-art approaches. Moreover, the space required by LevelPrune+ is significantly reduced. In the future, we plan to extend our approach to include some ranking mechanism for processing the cases that users do not require all answers to be outputted.

ACKNOWLEDGEMENT

We thank Mr. Po-Hsien Chien for implementing the algorithms and performing part of the experiments.

REFERENCES

3. Z. Bao, T. W. Ling, B. Chen, and J. Lu, “Effective xml keyword search with relevance oriented ranking,” in Proceedings of International Conference on Data Engi-
neering, 2009, pp. 517-528.

Ya-Hui Chang (張惠) is an Associate Professor of Computer Science Department at National Taiwan Ocean University. She received the B.S. degree in Computer Science and Information Engineering from National Taiwan University, and the M.S. and Ph.D. degree in Computer Science from University of Maryland at College Park, U.S.A. Her research interests include database query processing, XML techniques, graph databases and spatial databases.

Wei-Hsi Lee (李偉錦) is a graduate student of Computer Science Department at National Taiwan Ocean University. He received the BS degree in Computer Science from Ming Chung University, Taoyuan, Taiwan, in 2012. His research interests include XML query optimization, XML query processing, web application and database management.