JCSE Journal of Computing Science and Engineering

Regular Paper

Journal of Computing Science and Engineering,
Vol. 6, No. 2, June 2012, pp. 127-140

Fast Result Enumeration for Keyword Queries on XML Data

Junfeng Zhou* and Ziyang Chen
School of Information Science and Engineering, Yanshan University, Qinhuangdao, China
zhoujf@ysu.edu.cn, zycheng@ysu.edu.cn

Xian Tang
School of Economics and Management, Yanshan University, Qinhuangdao, China **txianz@ysu.edu.cn**

Zhifeng Bao, TokWang Ling
School of Computing, National University of Singapore, Singapore
baozhife@comp.nus.edu.sg, lingtw@comp.nus.edu.sg

Abstract
In this paper, we focus on efficient construction of tightest matched subtree (TMSubtree) results, for keyword queries on extensible markup language (XML) data, based on smallest lowest common ancestor (SLCA) semantics. Here, "matched" means that all nodes in a returned subtree satisfy the constraint that the set of distinct keywords of the subtree rooted at each node is not subsumed by that of any of its sibling nodes, while "tightest" means that no two subtrees rooted at two sibling nodes can contain the same set of keywords. Assume that d is the depth of a given TMSubtree, m is the number of keywords of a given query Q. We proved that if $d \le m$, a matched subtree result has at most $2m!$ nodes; otherwise, the size of a matched subtree result is bounded by $(d - m + 2)m!$. Based on this theoretical result, we propose a pipelined algorithm to construct TMSubtree results *without* rescanning all node labels. Experiments verify the benefits of our algorithm in aiding keyword search over XML data.

Category: Smart and intelligent computing

Keywords: XML; Keyword search; Result enumeration

I. INTRODUCTION

Over the past few years, keyword search on extensible markup language (XML) data has been a hot research issue, along with the steady increase of XML-based applications [1-12]. As with the importance of effectiveness in keyword search, efficiency is also a key factor in the success of keyword search.

Typically, an XML document is modeled as a node-labeled tree T. For a given keyword query Q, each result t is a subtree of T containing each keyword of Q at least once, where the root node of t should satisfy a certain semantics, such as smallest lowest common ancestor (SLCA) [4], exclusive lowest common ancestor (ELCA) [2, 3, 9], valuable lowest common ancestor (VLCA) [11] or meaningful lowest common ancestor (MLCA) [5]. Based on the set of qualified root nodes, there are three kinds of subtree results: 1) complete subtree (CSubtree), which is a subtree t_v^C rooted at a node v that is excerpted from the original XML tree without pruning any informa-

Open Access http://dx.doi.org/10.5626/JCSE.2012.6.2.127 http://jcse.kiise.org

Received 21 May 2012, **Accepted** 22 May 2012
*Corresponding Author

† An earlier version of this paper has been published at DASFAA 2012.

 pISSN: 1976-4677 eISSN: 2093-8020

tion [2, 4]; 2) path subtree (PSubtree), which is a subtree t_v^P that consists of paths from v to all of its descendants, each of which contains at least one input keyword [13]; and 3) matched subtree (MSubtree), which is a subtree t_v^M rooted at v satisfying the constraints of *monotonicity* and *consistency* [7, 8]. Let $S_{v'} = \{k_1, k_2, ...k_{m'}\}$ be the set of distinct keywords contained in the subtree rooted at a node v', $S_{v'} \subseteq Q$. Intuitively, a subtree t_v is an MSubtree if and only if for any descendant node v' of v, there does not exist a sibling node u' of v', such that $S_{v'} \subset S_{u'}$, which we call the constraint of "*keywords subsumption*". Obviously, for a given CSubtree t_v^C, t_v^P can be got by removing from t_v^C all nodes that do not contain any keyword of the given query in their subtrees, and, according to [8], t_v^M can be got by removing from t_v^P all of the nodes that do not satisfy the constraint of keywords subsumption.

EXAMPLE 1. To find for "CS" laboratory all papers that are written by "Tom" and published in "DASFAA" about "XML" from the XML document D in Fig. 1, we may submit a query Q = {CS, Tom, DASFAA, XML} to complete this task. Obviously, the qualified SLCA node is the root node with Dewey [10] label "1". Therefore, the CSubtree for Q is D itself, the PSubtree is R_1, while the MSubtree is $R2$.

From Example 1 we know that a CSubtree, e.g., D, may be incomprehensible for users, since it could be as large as the document itself, while a PSubtree could make users feel frustrated, since it may contain too much irrelevant information, e.g., although each leaf node of $R1$ directly contains at least one keyword of Q, the three papers with Dewey labels "1.2.3, 1.3.2, 1.3.3" have nothing to do with "XML". In fact, from Fig. 1 we can easily know that for "CS" laboratory, the paper written by "Tom" and published in "DASFAA" about "XML" is the node with Dewey label "1.2.2". According to Fig. 1, we know that the keyword sets for nodes 1.1, 1.2, and 1.3 are $S_{1.1}$ = {CS}, $S_{1.2}$ = {Tom, DASFAA, XML}, and $S_{1.3}$ = {DASFAA}, respectively. According to the constraint of keywords subsumption, all nodes in the subtree rooted at node 1.3 should be pruned, since $S_{1.3} \subset S_{1.2}$. Similarly, the keyword sets for node 1.2.1, 1.2.2, and 1.2.3 are $S_{1.2.1}$ = {Tom}, $S_{1.2.2}$ = {Tom, DASFAA, XML}, and $S_{1.2.3}$ = {Tom, DASFAA}, respectively. According to the constraint of keywords subsumption, since $S_{1.2.1} \subset S_{1.2.2}$ and $S_{1.2.3} \subset S_{1.2.2}$, all nodes in the subtrees rooted at node 1.2.1 and 1.2.3 should be removed. After that, we get the MSubtree, i.e., R_2, which contains all necessary information after removing nodes that do not satisfy the constraint of keywords subsumption, and is more self-explanatory and compact.

However, most existing methods [3, 4, 6, 9, 14] that address efficiency focus on computing qualified root nodes, such as SLCA or ELCA nodes, as efficient as possible. In fact, constructing subtree results is not a trivial task. Existing methods [7, 8] need to firstly *scan* all node labels to compute qualified SLCA/ELCA results, and then *rescan* all node labels to construct the initial subtrees. After that, they need to buffer these subtrees in memory, and apply the constraint of keywords subsumption on each node of these subtrees, to prune nodes with keyword sets subsumed by that of their sibling nodes, which is inefficient in time and space.

Table 1. Comparison of the running time for smallest lowest common ancestor (SLCA) computation and subtree construction

Query	Algorithm	t_1 (ms)	t_2 (ms)	$t_1 / (t_1+t_2)$ (%)
Q1	MaxMatch-IL	4.06	147.54	2.8
	MaxMatch-IMS	10.02		6.8
	MaxMatch-HS	3.52		2.4
Q2	MaxMatch-IL	25.78	6,442	0.4
	MaxMatch-IMS	0.18		0.003
	MaxMatch-HS	4.88		0.076

t_1 is the time of indexed lookup (IL) [4], incremental multiway SLCA (IMS) [6], and hash search (HS) [12] for SLCA computation, t_2 is the time of the MaxMatch algorithm [8] for computing matched subtree results.

EXAMPLE 2. Take the MaxMatch algorithm [8] for example. It firstly gets the set of SLCA nodes based on either indexed lookup (IL) [4], incremental multiway SLCA (IMS) [6] or hash search (HS) [12] algorithm, then computes all matched subtree results. We ran the MaxMatch algorithm for query $Q1$ = {school, gender, education, takano, province} and $Q2$ = {incategory, text, bidder, data} on the XMark (http://monetdb.cwi.nl/xml) dataset of 582 MB. We found that no matter which one of IL, IMS, and HS is adopted, the time of SLCA computation is less than 7% and 1% of the overall running time for Q1 and Q2, respectively, as shown in Table 1.

From Example 2 we know that, given a set of SLCA nodes, the operation of computing matched subtree results will dominate the overall performance of the MaxMatch algorithm, thus should deserve being paid more attention. As illustrated by [7], an MSubtree could still contain redundant information; e.g., the four conference nodes, i.e., 1.2.2.3, 1.2.3.3, 1.3.2.3, and 1.3.3.3, of D in Fig. 1 are the same as each other according to their content, and for keyword query {CS, conference}, returning only one of them is enough. However, the MSubtree result contains all of these conference nodes, because all of them satisfy the constraint of keywords subsumption.

In this paper, we focus on constructing tightest matched subtree (TMSubtree) results according to SLCA semantics. Intuitively, a TMSubtree is an MSubtree after removing redundant information, and it can be generated from the corresponding PSubtree by removing all nodes that do not satisfy the constraint of keywords subsumption, and just keeping one node for a set of sibling nodes that

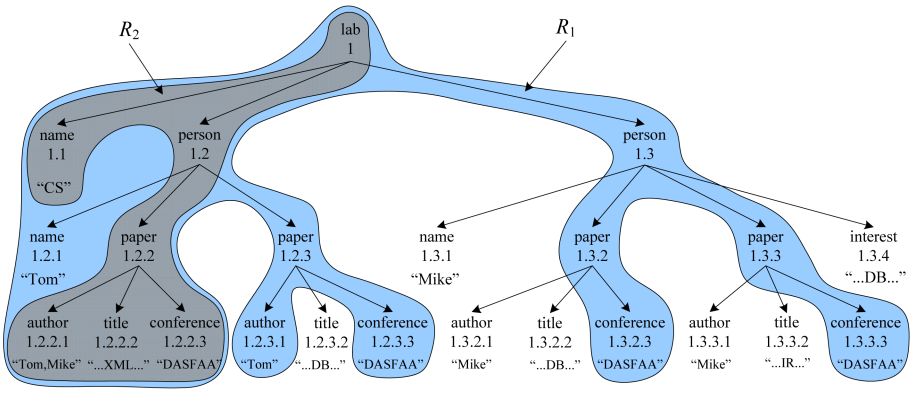


Fig. 1. A sample extensible markup language document D.

have the same keyword set. Assume that d is the depth of a given TMSubtree, m is the number of keywords of the given query Q. We proved that if $d \le m$, then a TMSubtree has at most $2m!$ nodes; otherwise, the number of nodes of a TMSubtree is bounded by $(d - m + 2)\ m!$. Based on this theoretical result, we propose a pipelined algorithm to compute TMSubtrees *without* rescanning all node labels. Our algorithm sequentially processes all node labels in document order, and immediately outputs each TMSubtree once it is found. Compared with the MaxMatch algorithm [8], our method reduces the space complexity from $O(d\sum_1^m|L_i|)$ to $O(d \cdot max\{2m!, (d - m + 2) \cdot m!\})$, where L_i is the inverted Dewey label list of keyword k_i.

The rest of the paper is organized as follows. In Section II, we introduce background knowledge, and discuss the related work. In Section III, we give an in-depth analysis of the MaxMatch algorithm [8], then define the tightest matched subtree (TMSubtree) and discuss its properties, and finally, present our algorithm on computing all TMSubtree results. In Section IV, we present the experimental results, and in Section V, we conclude our paper.

II. BACKGROUND AND RELATED WORK

We model an XML document as a node labeled ordered tree, where nodes represent elements or attributes, while edges represent direct nesting relationships between nodes in the tree. Fig. 1 is a sample XML document. We say a node v directly contains a keyword k, if k appears in the node name or attribute name, or k appears in the text value of v.

A Dewey label of node v is a concatenation of its parent's label and its local order. The last component is the local order of v among its siblings, whereas the sequence of components before the last one is called the parent label. In Fig. 1, the Dewey label of each node is marked as the sequence of components separated by ".". For a Dewey label A: $a_1.a_2...a_n$, we denote the number of components of A as $|A|$, and the i^{th} component as $A[i]$. As each Dewey label [10] consists of a sequence of components representing the path from the document root to the node it represents, the Dewey labeling scheme is a natural choice of state-of-the-art algorithms [2, 4, 6, 15, 16] for keyword query processing on XML data. The positional relationships between two nodes include document order ($\prec_d$), equivalence (=), ancestor-descendant (AD, $\prec_a$), parent-child (PC, $\prec_p$), ancestor-or-self ($\preceq_a$) and Sibling relationship. $u \prec_d v$ means that u is located before v in document order, $u \prec_a v$ means that u is an ancestor node of v, and $u \prec_p v$ denotes that u is the parent node of v. If u and v represent the same node, we have $u = v$, and both $u \preceq_d v$ and $u \preceq_a v$ hold. In the following discussion, we do not differentiate between a node and its label, if without ambiguity.

For a given query $Q = \{k_1, k_2, ..., k_m\}$ and an XML document D, we use L_i to denote the inverted Dewey label list of k_i, of which all labels are sorted in document order. Let $LCA(v_1, v_2, ..., v_m)$ be the lowest common ancestor (LCA) of nodes $v_1, v_2, ..., v_m$, the LCAs of Q on D are defined as $LCA(Q) = \{v | v = LCA(v_1, v_2, ..., v_m), v_i \in L_i (1 \le i \le m)\}$; e.g., the LCAs of $Q = \{XML, Tom\}$ on D in Fig. 1 are nodes 1.2 and 1.2.2.

In the past few years, researchers have proposed many LCA-based semantics [1, 2, 4, 5, 11], among which SLCA [4, 6] is one of the most widely adopted semantics. Compared with LCA, SLCA defines a subset of $LCA(Q)$, of which no LCA in the subset is the ancestor of any other LCA, which can be formally defined as $SLCASet = SLCA(Q) = \{v | v \in LCA(Q)$ and $\nexists v' \in LCA(Q)$, such that $v \prec_a v'\}$. In Fig. 1, although 1.2 and 1.2.2 are LCAs of $Q = \{XML, Tom\}$, only 1.2.2 is an SLCA node for Q, because 1.2 is an ancestor of 1.2.2.

Based on the set of matched SLCA nodes, there are three kinds of subtree results: 1) CSubtree [2, 4]; 2) PSubtree [13]; and 3) MSubtree, which is a subtree rooted at v satisfying the constraints of *monotonicity* and *consistency* [7, 8], which can be further interpreted by the changing of data and query, respectively. *Data monotonicity* means

that if we add a new node to the data, the number of query results should be (non-strictly) monotonically increasing. *Query monotonicity* means that if we add a keyword to the query, then the number of query results should be (non-strictly) monotonically decreasing. *Data consistency* means that after a data insertion, each additional subtree that becomes (part of) a query result should contain the newly inserted node. *Query consistency* means that if we add a new keyword to the query, then each additional subtree that becomes (part of) a query result should contain at least one match to this keyword. [8] has proved that if all nodes of a subtree t_v satisfy the constraint of "*keywords subsumption*", then t_v must satisfy the constraints of *monotonicity* and *consistency*, that is, t_v is an MSubtree. According to Example 1, we know that compared with CSubtrees and PSubtrees, MSubtrees contain all necessary information after removing nodes that do not satisfy the constraint of keywords subsumption, and are more self-explanatory and compact.

To construct MSubtrees, the existing method [8] needs to firstly *scan* all node labels to compute qualified SLCA nodes, then *rescan* all node labels to construct the initial subtrees. After that, they need to buffer these subtrees in memory and apply the constraint of keywords subsumption on each node of these subtrees to prune nodes with keyword sets subsumed by that of their sibling nodes, which is inefficient in time and space.

Considering that an MSubtree may still contain redundant information (discussed in Section I), in this paper, we focus on efficiently constructing TMSubtree results based on SLCA semantics. Intuitively, a TMSubtree is an MSubtree, after removing redundant information. Constructing TMSubtree results based on ELCA semantics [7] is similar, and therefore omitted for reasons of limited space.

III. RESULT ENUMERATION

A. Insight into the MaxMatch Algorithm

The MaxMatch algorithm [8] returns MSubtree results that are rooted at SLCA nodes, and satisfy the constraint of "keywords subsumption". For a given query $Q = \{k_1,...,k_m\}$ and an XML document D, supposing that $L_1(L_m)$ is the Dewey label list of occurrence of the least (most) frequent keyword of Q, d is the depth of D. As shown in Algorithm 1, the MaxMatch algorithm works in three steps to produce all MSubtree results.

Step 1 (line 1): MaxMatch finds from the m inverted Dewey label lists the set of SLCA nodes, i.e., *SLCASet*, by calling the IL algorithm [4]. The cost of this step is $O(md|L_1| \log |L_m|)$. In this step, all Dewey labels are processed once. Note that any algorithm for SLCA computation can be used in this step.

Step 2 (line 2): MaxMatch calls function *groupMatches* to construct the set of groups, i.e., *groupSet*. As shown in

Algorithm 1 MaxMatch (Q) /*$Q=\{k_1,\ldots,k_m\}$*/

```
1 SLCASet ← findSLCA(L1, L2,…, Lm)
2 groupSet ← groupMatches(L1, L2,…, Lm, SLCASet)
3 foreach group g ∈ groupSet do
4     pruneMatches(g)
5 endfor
```

Function groupMatches ($L_1, L_2,\ldots, L_m$, *SLCASet*)

```
1 L ← merge(L1, L2,…, Lm)
2 sequentially scan each Dewey label s ∈ SLCASet to construct
  groupSet, where each group g_s corresponds to an SLCA node s
3 sequentially scan each Dewey label n ∈ L, and put n to group g_s
  if s ∈ SLCASet satisfies that s ⪯_a n
4 return groupSet
```

Procedure pruneMatches (group g)

```
1 process all Dewey labels of g to construct a PSubtree t
2 foreach node n of t do/*traversing t in depth-first order*/
3     if n satisfies the constraint of keywords subsumption then
4         output n
5     end if
6 end for
```

groupMatches, it needs to firstly merge the m lists into a single list with cost $O(\log m\sum_1^m|L_i|)$, then sequentially rescan all labels and insert each one to a certain group (if possible), with cost $O(d\sum_1^m|L_i|)$. $d\sum_1^m|L_i|$ in this step, all Dewey labels are processed twice to construct the set of groups.

Step 3 (lines 3-5): For each group g, MaxMatch firstly constructs the PSubtree, then traverses it to prune redundant information. The overall cost of Step 3 is $O(min\{|D|, d\sum_1^m|L_i|\}\cdot 2^m)$, where 2^m is the cost of checking whether the set of distinct keywords of node v is subsumed by that of its sibling nodes; if not, then v is a node that satisfies the constraint of keywords subsumption.

Therefore, the time complexity of Algorithm 1 is $O(max\{min\{|D|, d\sum_1^m|L_i|\}\cdot 2^m, md|L_1|log|L_m|\})$. Moreover, as the MaxMatch algorithm needs to buffer all groups in memory before step 3, its space complexity is $O(d\sum_1^m|L_i|)$.

B. The Tightest Matched Subtree

DEFINITION 1. (TMSubtree) *For an XML tree D and a keyword query Q, let $S_v \subseteq Q$ be the set of distinct keywords that appear in the subtree rooted at v, $S_v \neq \Phi$. A subtree t is a TMSubtree iff t's root node is an SLCA node, and each node v of t satisfies that for each sibling node v' of v, $S_v \not\subset S_{v'}$, and for each set of sibling nodes $\{v_1, v_2, \ldots, v_n\}$ satisfying $S_{v1} = S_{v2} = \ldots = S_{vn}$, only one of them is kept for presentation.*

Intuitively, a TMSubtree is an MSubtree with redundant information being removed. It can be generated from the corresponding PSubtree by removing all nodes that do not satisfy the constraint of keywords subsumption, and just keeping one node for a set of sibling nodes that have the same keyword set.

DEFINITION 2. (Maximum TMSubtree) *Let t be a*

TMSubtree of Q, v a node of t, $S_v \subseteq Q$ the set of distinct keywords that appear in the subtree rooted at v, v.level the level value of v in t. We say t is a maximum TMSubtree if it satisfies the following conditions:

1) *v has $|S_v|$ child nodes $v_1, v_2, \ldots, v_{|S_v|}$,*

2) *if $|S_v| \geq 2 \wedge v.level < d$, then for any two child nodes v_i, $v_j (1 \leq i \neq j \leq |S_v|)$, $|S_{vi}| = |S_{vj}| = |S_v| - 1 \wedge |S_{vi} \cap S_{vj}| = |S_v| - 2$,*

3) *if $|S_v| = 1 \wedge v.level < d$, then v has one child node v_1 and $S_v = S_{v1}$.*

LEMMA 1. Given a maximum TMSubtree *t*, *t* is not a TMSubtree anymore, after inserting any keyword node into *t* without increasing *t*'s depth.

Proof. Suppose that *t* is not a maximum matched subtree result, then there must exist a non-leaf node *v* of *t*, such that we can insert a node v_c into *t* as a child node of *v*; v_c satisfies that $S_{vc} \subseteq S_v$. Obviously, there are four kinds of relationships between S_{vc} and S_v:

1) $|S_{vc}| = |S_v| = 1$. In this case, *v* has one child node that contains the same keyword as *v*. Obviously, v_c cannot be inserted into *t,* according to Definition 1.

2) $S_{vc} = S_v \wedge |S_v| \geq 2$. Since *v* has $|S_v|$ child nodes and each one contains $|S_v|-1$ keywords, for each child node v_i $(1 \leq i \leq |S_v|)$ of *v*, we have $S_{vi} \subset S_v = S_{vc}$. According to Definition 1, all existing child nodes of *v* should be removed if v_c is inserted into *t*, thus v_c cannot be inserted into *t* as a child node of *v,* in such a case.

3) $|S_{vc}| = |S_v| - 1 \wedge |S_v| \geq 2$. According to condition 2, all existing child nodes of *v* contain all possible combinations of keywords in S_v, thus there must exist a child node v_{ci} of *v*, such that $S_{vc} = S_{vci}$, which contradicts Definition 1, thus v_c cannot be inserted into *t* in this case.

4) if $|S_{vc}| < |S_v| - 1 \wedge |S_v| \geq 2$. According to condition 2, all existing child nodes of *v* contain all possible combinations of keywords in S_v, thus there must be a child node v_{ci} of *v*, such that $S_{vc} \subset S_{vci}$, which also contradicts Definition 1, thus v_c cannot be inserted into *t* in such a case.

In summary, if *t* is a maximum TMSubtree result, no other keyword node can be inserted into *t*, such that *t* is still a TMSubtree, without increasing *t*'s depth.

THEOREM 1. Given a keyword query $Q = \{k_1, k_2, ..., k_m\}$ and one of its TMSubtree *t* of depth *d*, if $d \leq m$, then *t* has at most $2m!$ nodes; otherwise, the number of nodes of *t* is bounded by $(d - m + 2)\, m!$.

Proof. Assume that *t* is a maximum TMSubtree, obviously, the number of nodes at 1^{st} level of *t* is $1 = C_m^m$.

For the 2^{nd} level, since $S_{t.root} = Q$, *t.root* has $|S_{t.root}| = C_m^{m-1}$ child nodes, of which each one contains $|S_{t.root}| - 1$ distinct keywords.

Thus we have Formula 1 to compute the number of nodes at the i^{th} level.

$$N(i) = N(i-1) \cdot C_{m-i+2}^{m-i+1} = P_m^{i-1},\ 1 \leq i \leq m \qquad (1)$$

If $d \leq m$, the total number of nodes in *t* is

$$N = \sum_{i=1}^{m} P_m^{i-1} = m! \sum_{i=1}^{m} \frac{1}{(m-i+1)!} < 2m! \qquad (2)$$

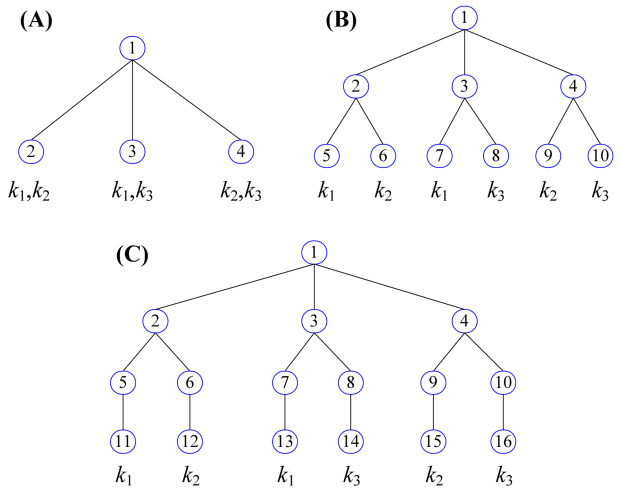


Fig. 2. Illustration of three possible TMSubtrees for keyword query $Q = \{k_1, k_2, k_3\}$.

If $d > m$, each node at the m^{th} level of *t* contains only one keyword, and all levels greater than *m* contain the same number of nodes as that of the m^{th} level. According to Formula 1, we know that $N(m) = m!$, thus the total number of nodes in *t* is

$$N = \sum_{i=1}^{m} P_m^{i-1} + (d-m)m! < (d-m+2)m! \qquad (3)$$

Therefore if $d \leq m$, a TMSubtree *t* has at most $2m!$ nodes, otherwise, the number of nodes of *t* is bounded by $(d - m + 2) \cdot m!$.

EXAMPLE 3. Given a keyword query $Q = \{k_1, k_2, k_3\}$, Fig. 2 shows three subtree results; according to Definition 1, they are all TMSubtrees. Obviously, by fixing their depths, no other node with any kind of combination of k_1 to k_3 can be inserted into these TMSubtree results, according to Lemma 1, that is, they are all maximum TMSubtrees. The TMSubtree in Fig. 2a has 4 nodes. Since its depth is 2 and is less than the number of keywords, i.e., 3, it satisfies Theorem 1, since $4 < 2 \times 3! = 12$. The TMSubtree in Fig. 2b is another maximum TMSubtree with 10 nodes, and also satisfies Theorem 1, since $10 < 2 \times 3! = 12$. Fig. 2c is also a maximum TMSubtree with 16 nodes. Since the depth of the TMSubtree is 4 and is greater than the number of keywords of *Q*, it still satisfies Theorem 1, since $16 < (4 - 3 + 2) \times 3! = 18$.

C. The Algorithm

Compared with the MaxMatch algorithm that produces all subtree results in three steps, the basic idea of our method is directly constructing all subtree results in the procedure of processing all Dewey labels. The benefits of our method lie in two aspects: 1) the buffered data in memory is largely reduced; 2) each Dewey label is visited only once. The first benefit comes from Theorem 1,

which guarantees that our method does not need to buffer huge volumes of data in memory as [8] does; the second benefit is based on our algorithm.

In our algorithm, for a given keyword query $Q = \{k_1, k_2, ..., k_m\}$, each keyword k_i corresponds to a list L_i of Dewey labels sorted in document order, and L_i is associated with a cursor C_i pointing to some Dewey label of L_i. C_i can move to the Dewey label (if any) next to it by using *advance*(C_i). Initially, each cursor C_i points to the first Dewey label of L_i.

As shown in Algorithm 2, our method sequentially scans all Dewey labels in document order. The main procedure is very simple: for all nodes that have not been visited yet, in each iteration, it firstly chooses the currently minimum Dewey label, by calling the selectMinLabel function (line 3), then processes it, by calling the pushStack procedure (line 4), and finally, it moves C_k forwardly to the next element in L_k (line 5). After all Dewey labels are processed, our algorithm pops all in-stack elements (line 7-9), then outputs the last TMSubtree result, to terminate the processing (line 10).

During the processing, our algorithm uses a stack S to temporarily maintain all components of a Dewey label, where each stack element e denotes a component of a Dewey label, which corresponds to a node in the XML tree. e is associated with two variables: the first one is a binary bitstring indicating which keyword is contained in the subtree rooted at e; the second is a set of pointers pointing to its child nodes, which is used to maintain intermediate subtrees.

The innovation of our method lies in that our method immediately outputs each TMSubtree result t_v when finding v is a qualified SLCA node, which makes it more efficient in time and space. Specifically, in each iteration (line 2 to 6 of Algorithm 2), our method selects the currently minimum Dewey label C_k in line 3, then pushes all components of C_k into S in line 4. The pushStack procedure firstly pops from S all stack elements that are not the common prefix of C_k and the label represented by the current stack elements, then pushes all C_k's components that are not in the stack into S. To pop out an element from S, the pushStack procedure will call popStack procedure to complete this task. The popStack procedure is a little more tricky. It firstly checks whether the popped element v is an LCA node. If v is not an LCA node, popStack firstly transfers the value of v's bitstring to its parent node in S (line 14), then inserts subtree t_v into $t_{top(S)}$ in line 15. In lines 16 to 22, popStack will delete all possible redundant subtrees, by checking the subsumption relationship between the keyword set of v and that of its sibling nodes. If v is an LCA node (line 3) and located after the previous LCA node u in document order (line 4), it means that u is an SLCA node if it is not an ancestor of v (line 5), then popStack directly outputs the TMSubtree result rooted at u in line 6. The subtree rooted at u is then deleted (line 8), and u points to v in line 9. If v is an LCA node but located

Algorithm 2 mergeMatching(Q)

```
1  u ← 1 /* u is the root node initially*/
2  while(∃i(¬eof(L_i))) do
3     C_k ← selectMinLabel(Q)
4     pushStack(S , C_k) /*S is the stack*/
5     advance(C_k)
6  endwhile
7  while(¬isEmpty(S)) do
8     popStack(S)
9  endwhile
10 output the subtree rooted at u
```

Function selectMinLabel(Q)

```
1  C_min ← C_1
2  foreach(2 ≤ i ≤ m) do
3     if(C_i ≺_d C_min) then C_min ← C_i
4  endfor
5  return C_min
```

Procedure pushStack(S, C_k)

```
1  n ← the length of the LCA of C_k and the Dewey label in S
2  while(|S|>n) do
3     popStack(S)
4  endwhile
5  foreach(|S|<i≤|C_k|) do
6     pushStack(S)
7  endfor
8  top(S).bit ← top(S).bit OR 1<<(k-1)
```

Procedure popStack(S)

```
/*Suppose that e_1.e_2…e_n is a Dewey label of a node, and all the n
components are in the stack S. In this procedure, v denotes the last
component e_n popped from S if it is used for bit operation; otherwise, it represents Dewey label e_1.e_2…e_n or the node itself*/
1  flag ← ~(~0<<m)
2  v ← pop(S) /*v denotes the Dewey label consists of all components
                of S*/
3  if((v.bit AND flag)=flag) then
4     if(u ≺_d v) then
5        if(u ⊀_a v) then
6           output the subtree rooted at u
7        end if
8        delete the subtree rooted at u
9        u ← v
10    else
11       delete the subtree rooted at v
12    end if
13 else
14    top(S).bit ← top(S).bit OR v.bit
15    add subtree rooted at v to the subtree rooted at top(S)
16    foreach(sibling node v' of v) do
17       if((v'.bit AND v.bit) = v.bit) then
18          delete the subtree rooted at v
19       else if ((v'.bit AND v.bit) = v'.bit) then
20          delete the subtree rooted at v'
21       end if
22    endfor
23 endif
```

Function eof(L_i)

```
1 if (all Dewey labels of L_i are processed) then return TRUE
2 else return FALSE
3 end if
```

before u, it means that v is not an SLCA node, thus we directly delete the subtree rooted at v (line 11).

Example 4. Consider the XML document D in Fig. 1 and query Q = {Mike, DASFAA, DB}. The inverted Dewey label lists for keywords of Q are shown in Fig. 3b. The status of processing these labels is shown in Fig. 3a1-a14. In this example, we use "001" ("010" or "100") to indicate that "Mike" ("DASFAA" or "DB") is contained in a subtree rooted at some node. After 1.2.2.1 is pushed into stack, the status is shown in Fig. 3a1, where the bitstring of the top element of S is "001", indicating that node 1.2.2.1 contains "Mike". The second pushed label is 1.2.2.3, and the status is shown in Fig. 3a2. Note that after an element is popped out from the stack, its bitstring is transferred to its parent in the stack. The next two labels are processed similarly. Before 1.3.1 is pushed into the stack, we can see from Fig. 3a5 that the bitstring of the top element in S is "111", which means that node 1.2 contains all keywords. After 1.3.1 is pushed into stack, the subtree rooted at 1.2 is temporarily buffered in memory. As shown in Fig. 3a10, before 1.3.3.1 is pushed into stack, the last component of 1.3.2 will be popped out from the stack. Since the bitstring of the current top element in S is "111" (Fig. 3a10), we know that 1.3.2 is an LCA node. According to line 5 of popStack procedure, we know that the previous LCA, i.e., 1.2, is an SLCA node, thus we output the matched subtree result rooted at 1.2. After that, the subtree rooted at 1.3.2 will be temporarily buffered in memory. When the last component of 1.3 is popped out from S (Fig. 3a14), according to line 3 of popStack procedure, we know that 1.3 is an LCA node. According to line 4 of popStack procedure, we know that the previous LCA node, i.e., 1.3.2, is located after 1.3 in document order, thus we know that 1.3 is not an SLCA node immediately, and delete the subtree rooted at 1.3 in line 11 of popStack procedure. Finally, we output the TMSubtree rooted at 1.3.2 in line 10 of Algorithm 2. Therefore for Q, the two TMSubtree results are rooted at 1.2 and 1.3.2, respectively.

As shown in Algorithm 2, for a given keyword query $Q = \{k_1, k_2, \ldots, k_m\}$ and an XML document D of depth d, our method just needs to sequentially scan all labels in the m inverted label lists *once*, therefore the overall I/O cost of Algorithm 2 is $O(d\sum_1^m|L_i|)$.

Now we analyze the time complexity of our algorithm. Since our algorithm needs to process all components of each involved Dewey label of the given keyword query $Q = \{k_1, k_2, \ldots, k_m\}$, the total number of components processed in our method is bounded by $d\sum_1^m|L_i|$, and the cost of processing these components is $dm\sum_1^m|L_i|$. During processing, each one of the $d\sum_1^m|L_i|$ components will be inserted into a subtree and deleted from the same subtree just once, and the cost of both inserting and deleting a component is $O(1)$. When inserting a subtree into another subtree, the operation of checking the subsumption relationship between the two keyword sets of two sibling nodes will be executed at most m times, according to Definition 2. Therefore, the overall time complexity is $O(dm^2\sum_1^m|L_i|)$.

Since our method is executed in a pipelined way, at any time it just needs to maintain at most d subtrees, where each one is not greater than a maximum TMSubtree. According to Theorem 1, each matched subtree result contains at most $max\{2m!,(d-m+2)m!\}$ nodes. Therefore, the space complexity of our method is $O(d{\cdot}max\{2m!, (d-m+2)m!\})$. Since d and m are very small in practice, the size of these subtrees buffered in memory is very small.

Note that to output the name of nodes in a TMSubtree result, existing methods may either store all path infor-

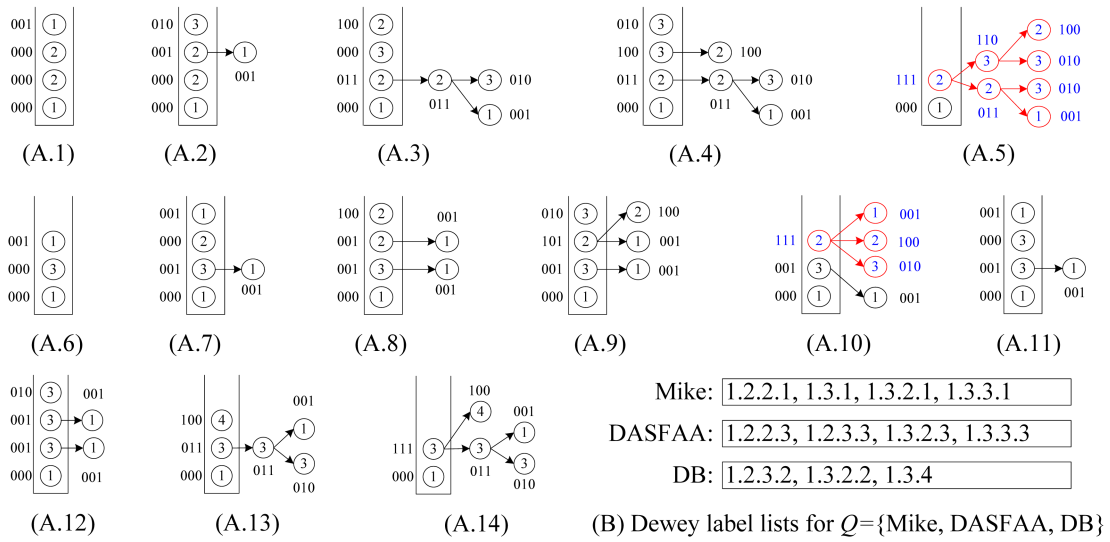


Fig. 3. Running status for Q = {Mike, DASFAA, DB}.

mation in advance, by suffering from huge storage space [1], or use the extended Dewey labels [17], by affording additional cost on computing the name of each node, according to predefined rules. In contrast, our method maintains another hash mapping between each path ID and the path information; the total number of index entries is the number of nodes in the dataguide index [18] of the XML tree, which is very small in practice. To derive

Table 2. Statistics of keywords used in our experiment

Keyword	tissue	baboon	necklace	arizona	cabbage	hooks	shocks	patients	cognition	villages
$\|L_i\|$	384	725	200	451	366	461	596	382	495	829
Keyword	male	takano	order	school	check	education	female	province	privacy	gender
$\|L_i\|$	18,441	17,129	16,797	23,561	36,304	35,257	19,902	33,520	31,232	34,065
Keyword	bidder	listitem	keyword	bold	text	time	data	emph	incategory	increase
$\|L_i\|$	299,018	304,969	352,121	368,544	535,268	313,398	457,232	350,560	411,575	304,752

Table 3. Queries on XMark dataset

ID	Keywords	$\sum_1^m \|L_i\|$	$\|L_{max}\|$	$\|L_{min}\|$	N_s	R_s (%)	Freq.
Q1	village, hooks	1,290	829	461	9	1.95	Low
Q2	baboon, patients, Arizona	1,575	742	382	1	0.26	
Q3	cabbage, tissue, shocks, baboon	2,088	742	366	9	2.46	
Q4	shocks, necklace, cognition, cabbage, tissue	2,041	596	200	9	4.5	
Q5	female, order	36,594	19,894	16,700	570	3.41	Med
Q6	privacy, check, male	85,960	36,300	18,428	29	0.16	
Q7	takano, province, school, gender	108,187	34,061	17,129	107	0.62	
Q8	school, gender, education, takano, province	143,444	35,257	17,129	107	0.62	
Q9	bold, increase	674,824	370,118	304,706	34,136	11.2	High
Q10	data, listitem, emph	1,112,760	457,231	304,969	43,777	14.35	
Q11	incategory, text, bidder, data	1,696,631	528,807	299,018	1	0.0003	
Q12	bidder, data,keyword, incategory, text	2,048,752	528,807	299,018	1	0.0003	
Q13	text, tissue	529,191	528,807	384	384	100	Random
Q14	takano, province	50,649	33,520	17,129	1,803	10.526	
Q15	incategory, cabbage	411,941	411,575	366	224	61.2	
Q16	check, bidder	335,318	299,018	36,300	1,922	5.2948	
Q17	baboon, patients	1,124	742	382	9	2.356	
Q18	tissue, shocks, order	17,680	16,700	384	9	2.344	
Q19	province, bold, increase	708,344	370,118	33,520	427	1.27	
Q20	cabbage, male, female	38,688	19,894	366	9	2.459	
Q21	listitem, emph, Arizona	655,980	350,560	451	1	0.22	
Q22	patients, school, gender	57,920	34,061	382	9	2.356	
Q23	patients, school, gender, text	586,727	528,807	382	9	2.356	
Q24	bold, increase, hooks, takno	692,414	370,118	461	6	1.3	
Q25	male, female, keyword, incategory	802,018	411,575	18,428	69	0.374	
Q26	emph, Arizona, villages, education	387,097	350,560	451	1	0.22	
Q27	check, bidder, data, baboon	793,291	457,231	742	1	0.13	
Q28	school, gender, time, baboon, patients	371,980	313,318	382	9	2.356	
Q29	tissue, shocks, order, province, bold	421,318	370,118	384	9	2.344	
Q30	femal, keyword, incategory, cabbage, male	802,384	411,575	366	9	2.459	
Q31	arizona, villages, education, listitme, emph	692,066	350,560	451	1	0.22	
Q32	bidder, data, necklace, cognition, check	793,244	457,231	200	1	0.5	

$\sum_1^m |L_i|$: the sum of the lengths of all keyword inverted lists, $|L_{max}|$($|L_{min}|$): the length of the longest (shortest) keyword inverted list, N_s: the number of qualified SLCA results, $R_s = N_s/|L_{min}|$: the results selectivity.

for each node its name on a path, we maintain in each Dewey label a path ID after the last component, thus we can get the name of each node on a path in constant time.

IV. EXPERIMENTAL EVALUATION

A. Experimental Setup

Our experiments were implemented on a PC with Pentium(R) Dual-Core E7500 2.93 GHz CPU, 2 GB memory, 500 GB IDE hard disk, and Windows XP professional as the operating system.

The algorithms used for comparison include the MaxMatch algorithm (the MaxMatch algorithm is used to output TMSubtrees in our experiment) [8] and our mergeMatching algorithm. MaxMatch was implemented based on IL [4], IMS [6] and HS [12] to test the impacts of different SLCA computation algorithms on the overall performance, and is denoted as MaxMatch-IL, MaxMatch-IMS and MaxMatch-HS, respectively. All these algorithms and our mergeMatching algorithm were implemented using Microsoft VC++. All results are the average time, by executing each algorithm 100 times on hot cache.

B. Datasets and Queries

We use XMark dataset for our experiments, because it possesses a complex schema, which can test the features of different algorithms in a more comprehensive way. The size of the dataset is 582 MB; it contains 8.35 million nodes, and the maximum depth and average depth of the XML tree are 12 and 5.5, respectively.

We have selected 30 keywords, which are classified into three categories, according to the length of their inverted Dewey label lists (the $|L_i|$ line in Table 2): 1) low frequency (100-1,000); 2) median frequency (10,000-40,000); and 3) high frequency (300,000-600,000).

Based on these keywords, we generated four groups of queries, as shown in Table 3: 1) four queries (Q1 to Q4) with 2, 3, 4, 5 keywords of low frequency; 2) four queries (Q5 to Q8) of median frequency; 3) four queries (Q9 to Q12) of high frequency; and 4) 20 queries (Q13 to Q32) with keywords of random frequency.

C. Evaluation Metrics

The metrics used for evaluating these algorithms include: 1) the number of buffered nodes, which is used to compare the space cost of these algorithms, 2) running time on SLCA and subtree result computation, and 3) scalability.

For a given query, we define the *result selectivity* as the size of the results over the size of the shortest inverted list. The second to last column of Table 3 is the result selectivity of each query.

D. Performance Comparison and Analysis

1) Comparison of the Number of Buffered Nodes

To make comparison of space cost for different algorithms, we have collected the statistics of the number of buffered nodes for the MaxMatch and mergeMatching algorithms. Table 4 shows a comparison of the number of buffered nodes for each query, from which we know that the number of buffered nodes for mergeMatching is

Table 4. Comparison of the number of buffered nodes for each query

Query	MaxMatch	mergeMatching
Q1	1,290	28
Q2	6,979	18
Q3	2,088	44
Q4	2,041	61
Q5	1,171	39
Q6	6,206	76
Q7	17,693	62
Q8	19,715	113
Q9	243,461	50
Q10	465,991	72
Q11	2,826,470	18
Q12	3,218,018	18
Q13	768	12
Q14	3,608	42
Q15	1,096	20
Q16	12,134	27
Q17	1,124	27
Q18	17,680	38
Q19	3,779	51
Q20	6,707	40
Q21	1,441,330	15
Q22	10,025	39
Q23	538,832	39
Q24	231,594	57
Q25	920	62
Q26	1,331,246	18
Q27	17,123	13
Q28	324,085	83
Q29	389,656	92
Q30	623,179	49
Q31	1,540,236	30
Q32	669,062	28

much less than that of MaxMatch. The reason lies in that MaxMatch needs to firstly construct *all* path subtrees and buffer them in memory, while mergeMatching just needs to buffer at most *d* TMSubtrees in memory, where *d* is the depth of the given XML tree.

2) Comparison of SLCA Computation:

To get all matched subtree results, the first critical step is computing all qualified SLCA nodes. Fig. 4 shows the comparison of different algorithms on SLCA computation.

From Fig. 4 we have the following observations: 1) mergeMatching is beaten by HS, IL and IMS for almost all queries. This is because mergeMatching sequentially processes all involved Dewey labels. Its performance is dominated by the number of involved Dewey labels, while IL and IMS are more flexible in utilizing the positional relationship to prune useless keyword nodes; HS only sequentially processes each Dewey label of the shortest inverted Dewey label list L_1, and its performance is dominated by the length of L_1. 2) HS is more efficient than IL and IMS when there exists huge difference between the set of involved Dewey label lists, such as Q13, Q15, Q17 to Q32, while it is beaten by IMS when the result selectivity is low, such as Q11 and Q12. This is because HS needs to process all Dewey labels of the shortest list, while IMS can wisely skip many useless nodes. 3) IL can beat IMS for some queries, while it can also be beaten by IMS for other queries, especially when the result selectivity is low, such as Q6, Q11 and Q12. This is because the IL algorithm computes the SLCA results by processing two lists each time from the shortest to the longest, while the IMS algorithm computes each potential SLCA by taking one node from each Dewey label list in each iteration. IMS could be the best choice when all nodes of the set of lists are not uniformly distributed; IL could perform best when all nodes are uniformly distributed, and there exists huge difference in lengths of the set of lists.

Table 5. Comparison of the running time for the MaxMatch algorithm based of different methods on smallest lowest common ancesto computation (ms)

Query	MaxMatch-HS	MaxMatch-IL	MaxMatch-IMS
Q1	2.34	2.35	2.3
Q2	2.3	2.35	2.36
Q3	3.1	3.15	3.35
Q4	2.703	3.15	2.9
Q5	4.71	7.45	5.21
Q6	15.11	17.15	15.8
Q7	97.12	105.45	103.2
Q8	149.3	151.6	158.3
Q9	460.6	479.7	452.4
Q10	3,021.7	3,135.95	3,037
Q11	6,446.2	6,468	6,443.7
Q12	9,067	9,594	9,057
Q13	25.13	25.75	26.7
Q14	29.4	14.05	17.21
Q15	11.4	11.75	11.7
Q16	32.57	28.9	27.47
Q17	5.3	5.45	5.6
Q18	93.81	94.5	94.21
Q19	28.1	34.4	30.1
Q20	38.33	39.05	38.79
Q21	1,909	1,910.2	1910
Q22	78.3	78.9	78.47
Q23	4,047.63	4,048.45	4,048.2
Q24	1,651.1	1,651.55	1,652.4
Q25	26.17	28.1	28.1
Q26	1,344.2	1,344.55	1,344.5
Q27	20.2	20.3	20.7
Q28	1,429.3	1,429.7	1,429.8
Q29	2,944.7	2,945.35	2,945.9
Q30	2,204.7	2,204.7	2,204.7
Q31	2,073.3	2,073.45	2,074.2
Q32	1,531.7	1,532	1,531

HS: hash search, IL: indexed lookup, IMS: incremental multiway smallest lowest common ancestor.

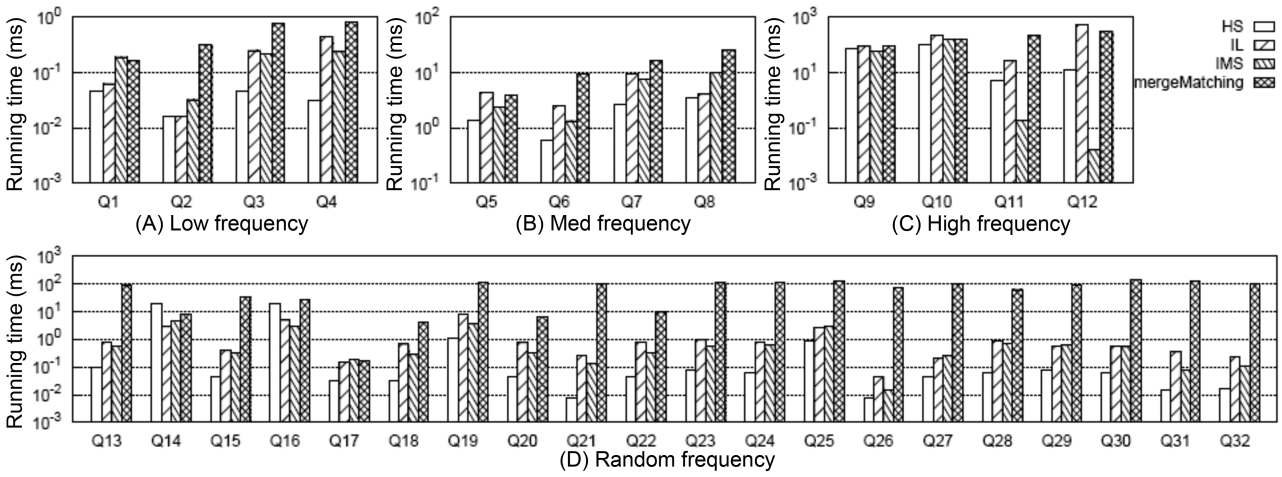


Fig. 4. Comparison of running time on smallest lowest common ancestor (SLCA) computation. HS: hash search, IL: indexed lookup, IMS: incremental multiway SLCA.

3) Impacts of the SLCA Computation on the Overall Performance

To further investigate the impacts of SLCA computation on the overall performance, we implemented the MaxMatch algorithm based on IL, IMS, and HS, which are denoted as MaxMatch-IL, MaxMatch-IMS, and MaxMatch-HS, respectively. As shown in Table 5, the three algorithms always consume similar running time, and in most cases, MaxMatch-HS is a little better than the other two algorithms. The reason lies in that they all share the same procedure in constructing matched subtree results, which usually needs much more time than computing SLCA nodes. Therefore, even though HS performs better than IL and IMS in most cases, their overall performance is similar to each other.

Further, from Fig. 4 and Table 5 we know that no matter how efficient HS, IL, and IMS are, and no matter how much HS performs better than IL and IMS, or vice versa, their performance benefits on SLCA computation do not exist anymore, if compared with the time on constructing matched subtree results based on the MaxMatch algorithm.

4) Comparison of Constructing Subtree Results

Fig. 5 shows the comparison between MaxMatch and mergeMatching on constructing all matched subtree results, from which we know that although mergeMatching is not as efficient as HS, IL, and IMS on SLCA computation, it is much more efficient than MaxMatch on constructing matched subtree results, such as Q1 to Q12, Q17, Q18, Q20 to Q24, Q26, and Q28 to Q32, and the time saved in the second phase is much more than that wasted in the first phase. The reason lies in that mergeMatching processes each Dewey label only once, while MaxMatch needs to firstly scan all Dewey labels once to compute SLCA results, then scan all Dewey labels once more to construct a set of initial groups according to different SLCA nodes. After that, it constructs all path subtree results by processing these Dewey labels again.

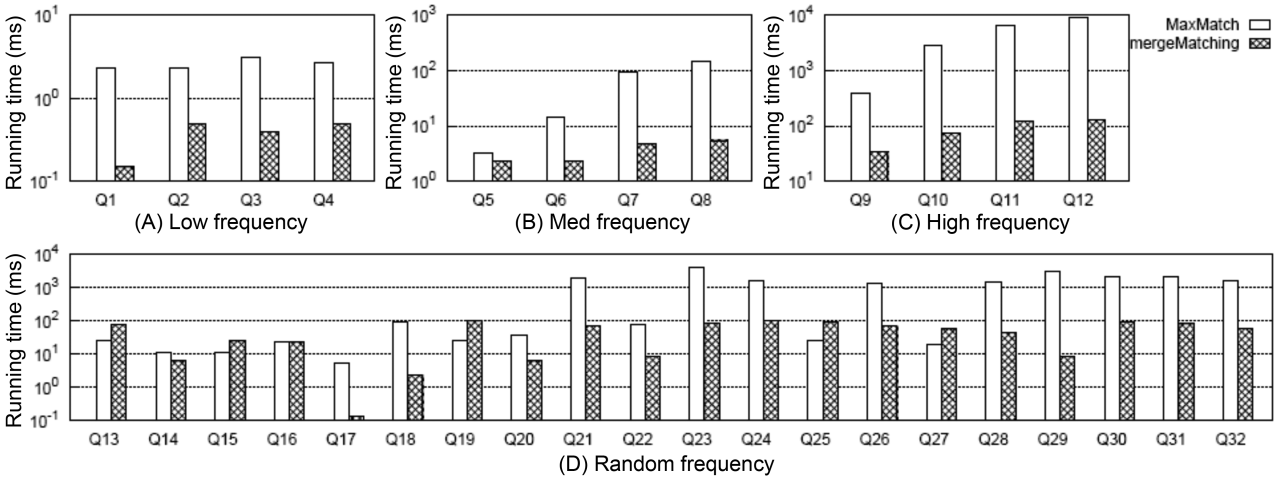


Fig. 5. Comparison of running time on constructing all subtree results.

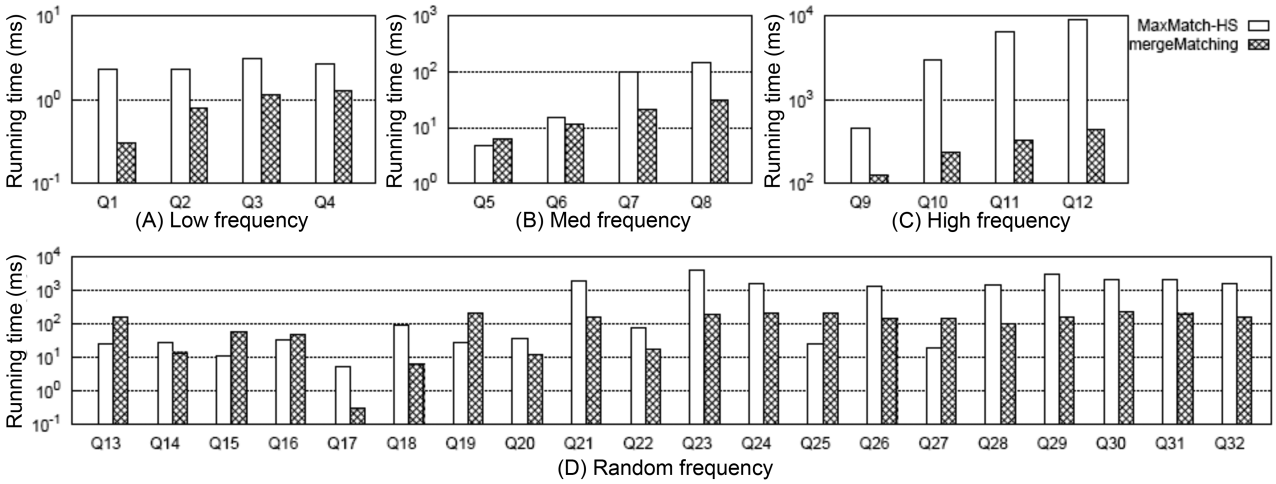


Fig. 6. Comparison of the overall running time for different methods. HS: hash search.

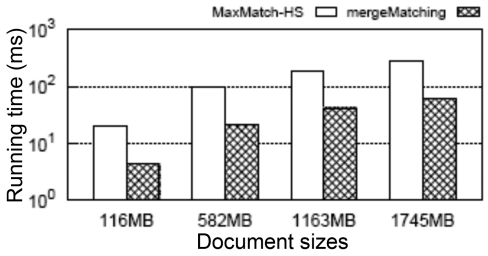


Fig. 7. Comparison of scalability on extensible markup language documents of different size for Q7. HS: hash search.

At last, it needs to traverse all nodes to prune useless nodes.

5) Comparison of the Overall Performance

Fig. 6 shows the comparison of the overall running time between MaxMatch-HS and mergeMatching, from which we know that in most cases, mergeMatching performs much better than MaxMatch-HS. The reason lies in that compared with MaxMatch, mergeMatching does not need to rescan all involved Dewey labels, and does not need to delay prune useless nodes after constructing all path subtree results.

On the contrary, it prunes useless nodes when constructing each matched subtree result, and thus can quickly reduce the number of testing operations of the keywords consumption relationship between sibling nodes.

Besides, we show in Fig. 7 the scalability when executing Q7 on XMark datasets with different sizes (from 116 MB to 1745 MB [15x]). The query time of the MaxMatch-HS and our mergeMatching algorithms grow sublinearly with the increase of the data size. Also, mergeMatching consistently saves about 78% time when compared with MaxMatch-HS. For other queries, we have similar results, which are omitted due to limitations of space.

V. CONCLUSIONS

Considering that TMSubtree is more self-explanatory and compact than CSubtree and PSubtree, but existing methods on subtree result computation need to rescan all Dewey labels, we focus in this paper on efficient construction of TMSubtree results for keyword queries on XML data based on SLCA semantics. We firstly proved the upper bound for the size of a given TMSubtree, that is, it has at most $2m!$ nodes if $d \leq m$; otherwise, its size is bounded by $(d\text{-}m+2) \cdot m!$, where d is the depth of a given TMSubtree, and m is the number of keywords of the given query Q. Then we proposed a pipelined algorithm to accelerate the computation of TMSubtree results, which only needs to sequentially scan all Dewey labels *once* without buffering huge volumes of intermediate results. Because the space complexity of our method is $O(d \cdot max\ \{2m!, (d\text{-}m+2) \cdot m!\})$, and in practice, d and m are very small, the size of the buffered subtrees is very small. The experimental results in Section IV verify the benefits of our algorithm in aiding keyword search over XML data.

ACKNOWLEDGMENTS

This research was partially supported by grants from the Natural Science Foundation of China (No. 61073060, 61040023). Zhifeng Bao was partially supported by a Singapore Ministry of Education Grant No. R252-000-394-112, under the project name of UTab. Xian Tang was partially supported by the Hebei Science and Technology research and development program (No. 11213578).

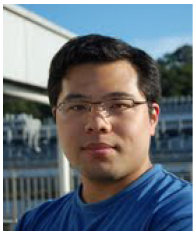
REFERENCES

1. S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv, "XSEarch: a semantic search engine for XML," *Proceedings of the 29th International Conference on Very Large Data Bases*, Berlin, Germany, 2003, pp. 45-56.
2. L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram, "XRANK: ranked keyword search over XML documents," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, San Diego, CA, 2003, pp. 16-27.
3. R. Zhou, C. Liu, and J. Li, "Fast ELCA computation for keyword queries on XML data," *Proceedings of the 13th International Conference on Extending Database Technology*, Lausanne, Switzerland, 2010, pp. 549-560.
4. Y. Xu and Y. Papakonstantinou, "Efficient keyword search for smallest LCAs in XML databases," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Baltimore, MD, 2005, pp. 527-538.
5. Y. Li, C. Yu, and H. V. Jagadish, "Schema-free XQuery," *Proceedings of the 30th International Conference on Very Large Data Base*, Toronto, Canada, 2004, pp. 72-83.
6. C. Sun, C. Y. Chan, and A. K. Coenka, "Multiway SLCA-based keyword search in XML data," *Proceedings of the 16th International Conference on World Wide Web, Banff*, Alberta, Canada, 2007, pp. 1043-1052.
7. L. Kong, R. Gilleron, and A. L. Mostrare, "Retrieving meaningful relaxed tightest fragments for XML keyword search," *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, Saint-Petersburg, Russia, 2009, pp. 815-826.
8. Z. Liu and Y. Cher, "Reasoning and identifying relevant matches for XML keyword search," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, 2008, pp. 921-932.
9. Y. Xu and Y. Papakonstantinou, "Efficient LCA based keyword search in XML data," *Proceedings of the 11th International Conference on Extending Database Technology: Advances in Database Technology*, Nantes, France, 2008, pp. 535-546.
10. I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang, "Storing and querying ordered XML using a relational database system," *Proceedings of the*

ACM SIGMOD International Conference on Management of Data, Madison, WI, 2002, pp. 204-215.
11. G. Li, J. Feng, J. Wang, and L. Zhou, "Effective keyword search for valuable LCAs over XML documents," *Proceedings of the 16th ACM Conference on Information and Knowledge Management*, Lisbon, Portugal, 2007, pp. 31-40.
12. W. Wang, X. Wang, and A. Zhou, "Hash-search: an efficient SLCA-based keyword search algorithm on XML documents," *Proceedings of the 14th International Conference on Database Systems for Advanced Applications*, Brisbane, Queensland, Australia, 2009, pp. 496-510.
13. V. Hristidis, N. Koudas, Y. Papakonstantinou, and D. Srivastava, "Keyword proximity search in XML trees," *IEEE Transactions on Knowledge and Data Engineering*, vol. 18, no. 4, pp. 525-539, 2006.
14. L. F. Chen and Y. Papakonstantinou, "Supporting top-K keyword search in XML," *Proceedings of the 26th International Conference on Data Engineering*, Long Beach, CA, 2010, pp. 689-700.
15. Z. Liu and Y. Chen, "Identifying meaningful return information for XML keyword search," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Beijing, China, 2007, pp. 329-340.
16. Z. Bao, T. W. Ling, B. Chen, and J. Lu, "Effective XML keyword search with relevance oriented ranking," *Proceedings of the 25th IEEE International Conference on Data Engineering*, Shanghai, China, 2009, pp. 517-528.
17. J. Lu, T. W. Ling, C. Y. Chan, and T. Chen, "From region encoding to extended dewey: on efficient processing of XML twig pattern matching," *Proceedings of the 31st International Conference on Very Large Data Bases*, Trondheim, Norway, 2005, pp. 193-204.
18. R. Goldman and J. Widom, "Dataguides: enabling query formulation and optimization in semistructured databases," *Proceedings of the 23rd International Conference on Very Large Data Bases*, Athens, Greece, 1997, pp. 436-445.

Junfeng Zhou

Junfeng Zhou received his PhD degree in computer science from Renmin University of China. He is an associate professor of Yanshan University, and a member of the China Computer Federation. His research interests include XML structured query processing and XML keyword search.



Zhifeng Bao

Zhifeng Bao is a research fellow in the School of Computing, National University of Singapore. He received his PhD degree in the Department of Computer Science, School of Computing, National University of Singapore. His research interests include XML structured query processing, XML keyword search and provenance data management.

Xian Tang

Xian Tang received her PhD degree in computer science from Renmin University of China. She is a lecturer at Yanshan University. Her research interests include XML keyword search and flash databases.

Ziyang Chen

Ziyang Chen received his PhD degree in computer science from Yanshan University of China. He is a professor of Yanshan University and a member of the China Computer Federation. His research interests include relational database and XML keyword search.

Tok Wang Ling

Tok Wang Ling received his PhD degree in Computer Science from the University of Waterloo (Canada). He is a professor of the Department of Computer Science, School of Computing at the National University of Singapore. His research interests include Data Modeling, ER approach, Normalization Theory, Semi-structured Data Model, XML twig pattern query processing, and XML keyword query processing. He has published more than 190 papers, co-authored a book, co-edited a book, and co-edited 9 conference proceedings. He is an ACM Distinguished Scientist, a senior member of IEEE and Singapore Computer Society, and an ER Fellow.