Retrieving Meaningful Relaxed Tightest Fragments for XML Keyword Search

Kong Lingbo, Rémi Gilleron, Aurélien Lemay

Retrieving Meaningful Relaxed Tightest Fragments for XML Keyword Search*

Lingbo Kong†, Rémi Gilleron, Aurélien Lemay
Mostrare, INRIA Futurs,
Villeneuve d'Ascq, Lille, 59650 FRANCE
mLinking@gmail.com, {remi.gilleron, aurelien.lemay}@univ-lille3.fr

ABSTRACT

Adapting keyword search to XML data has been attractive recently, generalized as XML keyword search (XKS). One of its key tasks is to return the meaningful fragments as the result. [1] is the latest work following this trend, and it focuses on returning the fragments rooted at SLCA (Smallest LCA – Lowest Common Ancestor) nodes. To guarantee that the fragments only contain interesting nodes, [1] proposes a contributor-based filtering mechanism in its MaxMatch algorithm. However, the filtering mechanism is not sufficient. It will commit the false positive problem (discarding interesting nodes) and the redundancy problem (keeping uninteresting nodes).

In this paper, our interest is to propose a framework of retrieving meaningful fragments rooted at not only the SLCA nodes, but all LCA nodes. We begin by introducing the concept of Relaxed Tightest Fragment (RTF) as the basic result type. Then we propose a new filtering mechanism to overcome those two problems in MaxMatch. Its kernel is the concept of valid contributor, which helps to distinguish the interesting children of a node. The new filtering mechanism is then to prune the nodes in a RTF which are not valid contributors to their parents. Based on the valid contributor concept, our ValidRTF algorithm not only overcomes those two problems in MaxMatch, but also satisfies the axiomatic properties deduced in [1] that an XKS technique should satisfy. We compare ValidRTF with MaxMatch on real and synthetic XML data. The result verifies our claims, and shows the effectiveness of our valid-contributor-based filtering mechanism.

1. INTRODUCTION

XML is rapidly emerging as the *de facto* standard for data representation and exchange of Web applications, such as Digital Library, Web service, and Electronic business. An XML data is usually modeled as an XML tree $T = (r, V, E, \Sigma, \lambda)$, where V is the node set, r is a special node in V as the root, and E is the edge set. For each node $v \in V$, λ is the function to assign a "label" l in the string set Σ to v, that is $\lambda : v \in V \rightarrow l \in \Sigma$. There are two kinds of nodes in V according to the fact whether the node contains "value" or not. Figure 1(a) is an XML instance, in which every node has a node name (label), and every leaf node also contains the value (text)[1]. The integer sequence beside each node is the Dewey code [2], such as "0.2.0.1". For an XML tree T, the content C_v of a node v is the word set implied in v's label, text and attributes. Given a keyword query $Q = \{w_1, \ldots, w_k\}$, the node v is a keyword node if $C_v \cap Q \neq \emptyset$ where $\emptyset$ is the empty set.

With the widespread use of XML, adapting keyword search to XML data has become attractive, generalized as XML keyword search (XKS), because keyword search provides a simple and user-friendly query interface to access XML data in web and scientific applications. As a result, XKS has recently attracted more and more research interests [3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 1]. Its basic task is to return meaningful fragments, each of which contains all the interesting keywords.

Based on the LCA concept from graph theory [13, 14, 15, 16], the popular solution for this task is to first locate the LCA nodes (including the popular SLCA nodes), and then retrieve the fragments rooted at those LCA nodes. The semantics of LCA here is to return the special nodes in the given XML tree T. The subtrees rooted at those special nodes contain all the keywords in the keyword query $Q = \{w_1, \ldots, w_k\}$ after *excluding* the LCA nodes and their keyword nodes which are descendants of those special nodes (refer to [4] for more details). As for the SLCA nodes, they are part of those LCA nodes, however, they have no descendant node which also contains all keywords (refer to [7] for more details). Following the above consideration, the fragment related to a SLCA node u can be described as the SLCA-based fragment, which consists of all the keyword nodes $v_1, \ldots, v_m$ of the query Q, and all the nodes on every path from u to $v_i (1 \leq i \leq m)$. However, there is at least one explicit weakness for the above methods, that is, the nodes in a fragment corresponding to a LCA node may not be all interesting. We use Example 1 to illustrate this.

Example 1 (Illustration for LCA related concepts) *The illustration is based on two XML instances in* Figure 1(a) *and* Figure 1(b):(1) (*derived from* [1]). Figure 1(b):(2) *presents several keyword queries. We first demonstrate the concepts of LCA and SLCA. Then we illustrate the deficiency of returning only the LCA nodes as result.*

*This work was partially supported by the Webcontent project ANR-05-RNTL02014 funded by french national research agency.

†The work was done while the first author was visiting the mostrare INRIA project in Lille.

[1]This is different from the XML model in [1], in which there is an independent node for each text value.

EDBT 2009, March 24–26, 2009, Saint Petersburg, Russia.

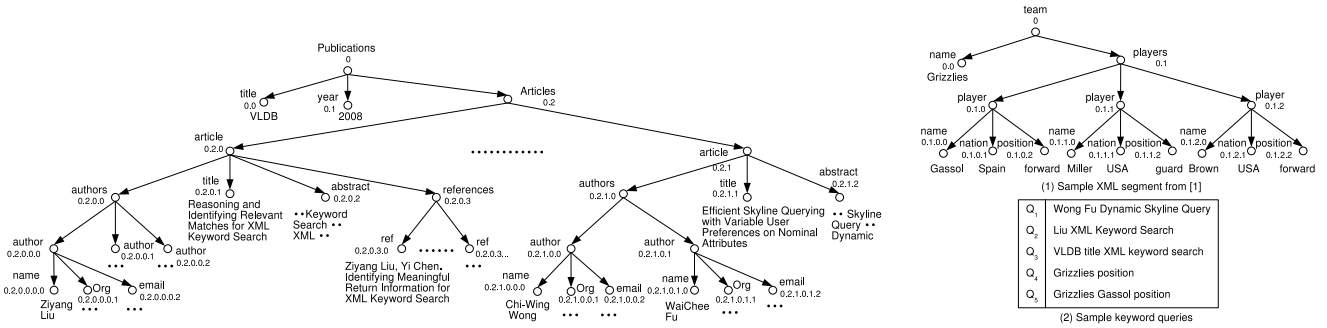


Figure 1: (a): An XML instance; (b):(1) The XML segment used in [1]; (b):(2) Sample keyword queries

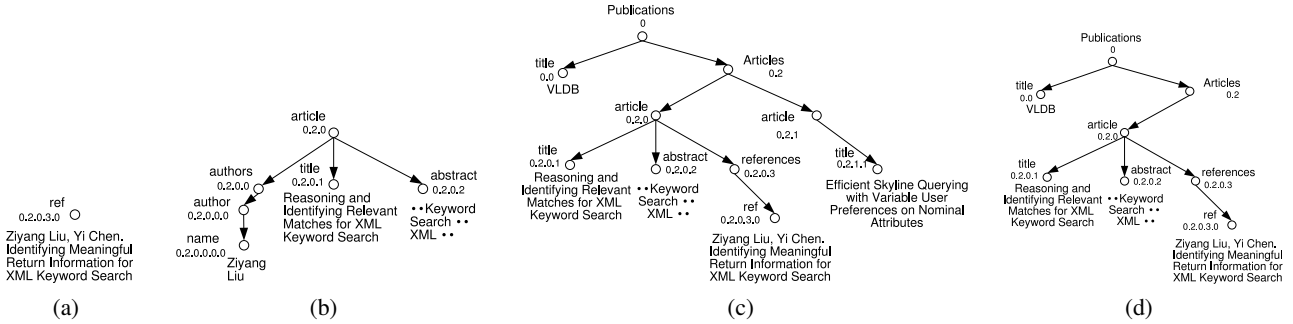


Figure 2: (a): The SLCA-based fragment for Q_2; (b): The LCA related fragment for Q_2; (c): The raw fragment for Q_3; (d): The meaningful fragment for Q_3

[SLCA v.s LCA] *When running Q_2 on* Figure 1(a)*, the SLCA node is* "0.2.0.3.0 (ref)"[2] *according to* [7]. Figure 2(a) *illustrates the SLCA-based fragment* (*similar with the match concept in* [1]) *corresponding to the node* "0.2.0.3.0 (ref)".

According to [4, 11, 12]*, the LCA node* "0.2.0 (article)" *is also interesting for Q_2, because its related fragment shown in* Figure 2(b) *also contains all the keywords in Q_2. However, this LCA node is discarded by the SLCA concept because it is an ancestor of the SLCA node* "0.2.0.2.0 (ref)".

The query result proposed by [1] *has the same semantics as the SLCA-based fragment here, which is only applicable to the SLCA related fragments. It is not applicable to the more general LCA related fragments.*

[Returning only LCA/SLCA nodes] *When running Q_3 on* Figure 1(a)*, the only LCA node* (*also the SLCA node*) *is the root of the XML data* – "0 (Publications)". *It is clear that returning the whole subtree rooted at* "0 (Publications)" *is redundant.*

Sometimes, it is redundant too even we only return the LCA node and the related nodes. Figure 2(c) *demonstrates the fragment rooted at the LCA node* "0 (Publications)" *for Q_3. We can see that the keyword node* "0.2.1.1 (title)" *is not interesting to the keyword query Q_3, because it is not about the research on XML keyword search.*

Different from the above work, several methods [5, 8, 10, 11, 1] are proposed recently to return the meaningful fragments with interesting nodes, instead of only the LCA/SLCA nodes. And how to prune the redundant nodes in the fragments becomes a challenge for XKS. Among those proposals, [1] deserves attention, because it not only deduces the properties that an XKS technique should satisfy, but also proposes a concrete algorithm following those properties. The intuitive and non-trivial axiomatic properties are (1) *data monotonicity*: if a new node is added to the data, the number of query results should be (non-strictly) monotonically increasing; (2) *query monotonicity*: if a keyword is added to the keyword query, the number of query results should be (non-strictly) monotonically decreasing; (3) *data consistency*: after a data insertion, each additional subtree which becomes (part of) a query result should contain the newly inserted node; (4) *query consistency*: if a new keyword is added to the query, each additional subtree which becomes (part of) a query result should contain at least one match to this keyword.

After the axiomatic work, [1] further proposes the MaxMatch algorithm, which satisfies all those properties, and returns the filtered fragments rooted at the SLCA nodes after pruning the uninteresting nodes. Its kernel is the filtering mechanism based on the concept of contributor, which helps to filter the nodes in each SLCA-based fragment. The contributor filtering mechanism requires that every node n in the SLCA-based fragment should satisfy the following condition: n does not have a sibling n_2 satisfying $dMatch(n) \subset dMatch(n_2)$, where the function $dMatch(n)$ represents all the keywords contained in the subtree rooted at the node n.

Even though the contributor-based MaxMatch satisfies all those four properties, however, it has the false positive problem which

[2]Because of its uniqueness, the Dewey code is used to represent the corresponding node in this paper.

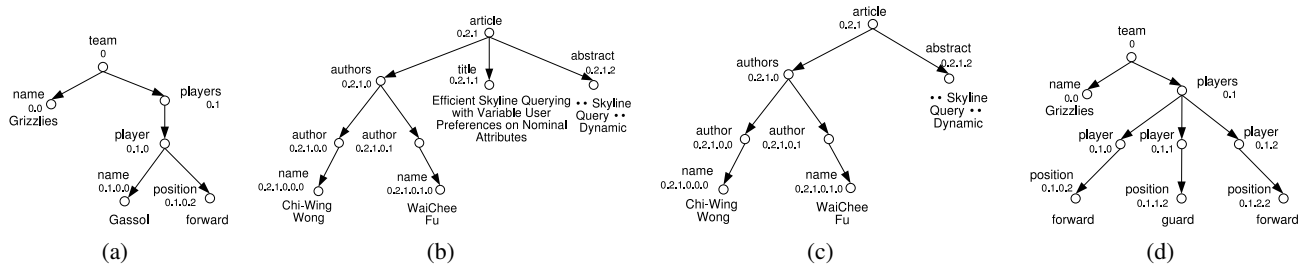


Figure 3: (a): The MaxMatch result for Q_5 on the XML segment of Figure 1(b):(1)**; (b): The SLCA-based fragment for Q_1 on the XML instance in Figure 1(a); (c): The MaxMatch result for Q_1 after pruning the node "0.2.1.1 (title)" in Figure 3(b); (d): The MaxMatch result for Q_4 on the XML segment of** Figure 1(b):(1)

means it will discard interesting nodes, and the redundancy problem which means it can not prune all the uninteresting nodes. The key is that the condition of $\nexists n_2, dMatch(n) \subset dMatch(n_2)$ is not sufficient to prune only the uninteresting nodes. Example 2 illustrates these deficiencies.

Example 2 (Illustration for the contributor mechanism) *We first demonstrate a positive example of the contributor-based filtering mechanism, then illustrate the two problems of it.*

[Positive example] *As running Q_5 on* Figure 1(b):(1)*, the $dMatch$ of the node* "0.1.0" *is* {Gassol, position}*, and its two siblings are* "0.1.1" *and* "0.1.2"*, whose $dMatch$s are both* {position}*. Since $dMatch(0.1.1) = dMatch(0.1.2) \subset dMatch(0.1.0)$, the two nodes* "0.1.1" *and* "0.1.2" *are discarded by MaxMatch, and the final result is shown in* Figure 3(a)*, which clearly presents the information of the player* "Gassol" *in the team* "Grizzlies".

[False positive problem] *When doing the keyword query Q_1 on the XML instance in* Figure 1(a)*, there is only one SLCA node,* "0.2.1 (article)"*. The related keyword nodes are the node* "0.2.1.0.0.0 (name)" *with keyword* "Wong"*, the node* "0.2.1.0.1.0 (name)" *with keyword* "Fu"*, the node* "0.2.1.1 (title)" *with keywords* "Skyline Query" *and the node* "0.2.1.2 (abstract)" *with keywords* "Dynamic Skyline Query"*. The basic SLCA-based fragment is shown in* Figure 3(b)*.*

When using contributor concept to filter the nodes in Figure 3(b)*, it is clear that the node* "0.2.1.1 (title") *is a sibling of the node* "0.2.1.2 (abstract)"*, and its keyword set* {Skyline, Query} *is covered by the latter's keyword set* {Dynamic, Skyline, Query}*. Consequently the keyword node* "0.2.1.1 (title)" *is discarded, since it is not a contributor to Q_1. So the final result of Q_1 on the XML instance of* Figure 1(a) *is the fragment shown in* Figure 3(c)*.*

However, intuitively, the keyword node "0.2.1.1 (title)" *and its value* "Efficient Skyline Querying with Variable User Preferences on Nominal Attributes" *should not be discarded, because it is the* "title" *of the corresponding paper.*

[Redundancy problem] *Here we directly borrow the XML instance used in* [1]*, shown in* Figure 1(b):(1)*. The keyword query is* "Grizzlies position" *(Q_4 in* Figure 1(b):(2)*), which is used to collect all the positions in the team* "Grizzlies"*. Now the $dMatch$s of the three nodes* "0.1.0"*,* "0.1.1" *and* "0.1.2" *are the same as* {position}*. Consequently the fragment retrieved by MaxMatch based on contributor contains all the position nodes, even the values of some of them are same. The fragment is shown in* Figure 3(d)*. We can see that there are two position nodes which have the same value –* "forward".

From Example 2 we can see that the contributor-based filtering mechanism is not sufficient to filter out all the uninteresting nodes in a SLCA-based fragment. Besides, only focusing on the SLCA related fragments is not enough for XKS either. The papers [4, 17, 8, 18, 11, 12] confirm that the LCA nodes other than the SLCA nodes are also significant to XKS.

Thus, our interest in this paper is to improve the filtering mechanism in [1], and then to extend the work to all the interesting LCA-based fragments, not just the SLCA-based fragments. The first task in this paper is to formalize the description of the basic interesting fragments for the given keyword query. We propose the Relaxed Tightest Fragment (RTF) concept to represent the basic result for the XML keyword search, which considers the interesting LCA node and all its related keyword nodes together. Then we propose a new filtering mechanism in order to overcome the false positive problem and the redundancy problem in the contributor-based filtering mechanism in MaxMatch. The kernel of our filtering mechanism is the concept of *valid contributor*, which filters the children of a node by taking into account not only the children's labels but also their contents. The contributions of this paper can be concluded as follows:

- We propose the Relaxed Tightest Fragment (RTF) concept to represent the basic result for the XML keyword search. A RFT in an XML data for the given keyword query encloses the basic interesting information with respect to the query.
- We propose the concept of valid contributor to filter the nodes in a RTF so as to overcome the false positive problem and the redundancy problem in contributor concept. By taking into account the label and the content of a node, the children of a node could be classified as valid contributors and non-contributors. The filtering mechanism is to make sure that all the nodes in the RTF are valid contributors to their parents. After the filtering, we get a more concise and reasonable RTF.
- We implement the valid-contributor-based filtering mechanism in ValidRTF, and compare it with MaxMatch on both real and synthetic datasets. The results verify the efficiency and effectiveness of our proposal.

The rest of this paper is organized as follows. Section 2 presents the formal description of the RTF. The idea of our valid contributor is discussed in Section 3. Section 4 illustrates the ValidRTF algorithm, which returns the meaningful RTFs for the given keyword

query after filtering the nodes based on the valid contributor concept. Section 5 illustrates the experimental result. Section 6 briefly reviews the related work of XML keyword search. Section 7 is the conclusion.

2. RELAXED TIGHTEST FRAGMENT

In this section, we introduce the concept of Relaxed Tightest Fragment (RTF), which is used in this paper as the basic result type for XKS. It is also based on the LCA concept, but a RTF can also determine all the related nodes (no more and no less) in the XML data.

Following the examples in Section 1, we can learn that the essence behind retrieving the fragments for the given keyword query is a partitioning mechanism of the keyword nodes with the help of LCA concept, so that each partition satisfies the following conditions. (1) each partition covers all the keywords; (2) the LCA node of a partition is unique among the LCA nodes of all the partitions; (3) the intersection of any two partitions is empty; (4) if there is a subset of a partition that also covers the query, its LCA node should be the same as that of the whole partition; (5) any keyword node in a partition with LCA node a can not compose a partition with any other keyword nodes so that the new LCA node of the new partition is lower than a in the XML tree; (6) there is no other keyword node outside a partition p so that the LCA node of the combination of p and that node is the same as the LCA node of p.

For example, the fragments in Figure 2(a), 2(b), 2(c), 3(b) and 3(d) are ideal basic results for corresponding keyword queries, each of which contains all the interesting nodes related to the corresponding keyword query. They all satisfy the above six requirements. As for Figure 2(d), it is not an ideal basic result for Q_3 according to (6). The reason is that after integrating the outside node "0.2.1.1" into Figure 2(d), the LCA node does not change. The fragments in Figure 3(a) and 3(c) are not ideal either because of the similar reason.

The above conditions can be categorized into three kinds of requirements for the basic result of XKS: the keyword requirement [(1)], the uniqueness requirement [(2), (3)] and the completeness requirement [(4), (5), (6)]. It is straightforward to understand the first two requirements. As for the third, it requires that a valid partition should contain all the related keyword nodes, no more and no less. However, it is not convenient to express them in words. It is necessary to formalize them in a more precise manner. This is the motivation of our RTF here. Our first step is to enumerate all the partitions of the keyword nodes, each of which contains all the given keywords. This is the concept of extended keyword node combination set ($\mathcal{EC}_Q^T$) in Definition 1. Then we confine them by adding additional conditions to get the RTF concept which satisfies the above requirements. The RTF concept is shown in Definition 2.

Definition 1 (Extended Keyword Node Combination Set) *Given an XML tree T, and the keyword query $Q = \{w_1, \ldots, w_k\}$, D_i records all the keyword nodes with respect to the keyword w_i. The set V_i contains all the valid keyword node subsets of D_i, which means $V_i = 2^{D_i} - \{\emptyset\}$, where 2^{D_i} is the power set of D_i, and $\emptyset$ is the empty set.*

The extended keyword node combination set $\mathcal{EC}_Q^T$ on all D_i $(1 \le i \le k)$ is defined as $\{V | V = \bigcup_{i=1}^{k} v_i$ where $v_i \in V_i\}$.

Without loss generality, $\mathcal{EC}_{Q,j}^T$ represents an element of $\mathcal{EC}_Q^T$, where $1 \le j \le \prod_{i=1}^{k} \left(2^{|D_i|} - 1\right)$ if $\bigcap_{i=1}^{k} D_i = \emptyset$. $\mathcal{EC}_{Q,j}^T|_i$ is the keyword node set in $\mathcal{EC}_{Q,j}^T$ corresponding to the keyword w_i.

The basic idea of $\mathcal{EC}_Q^T$ is to enumerate all the partitions of the corresponding keyword node sets. Each partition $\mathcal{EC}_{Q,j}^T$ naturally satisfies the keyword requirement, namely $C_{\mathcal{EC}_{Q,j}^T} \cap Q = Q$ where $C_{\mathcal{EC}_{Q,j}^T} = \bigcup_{v \in \mathcal{EC}_{Q,j}^T} C_v$. Besides, the definition of $\mathcal{EC}_Q^T$ also covers the situation that many keyword nodes containing the same keyword occur in some $\mathcal{EC}_{Q,j}^T$. This property provides a platform for the later RTF concept that we can filter $\mathcal{EC}_Q^T$ so that the related nodes with respect to a LCA node can be retrieved totally. Example 3 expresses these properties.

Example 3 (Illustration of $\mathcal{EC}_Q^T$) *We run the keyword query $Q =$* "Liu Keyword" *on the XML data in* Figure 1(a). *We take* "Liu" *as w_1, and* "keyword" *as w_2. The keyword node set corresponding to w_1 is $D_1 =$* {"name (0.2.0.0.0.0)", "ref (0.2.0.3.0)"}*; while $D_2 =$* {"title (0.2.0.1)", "ref (0.2.0.3.0)", "abstract (0.2.0.2)"} *is the keyword node set for w_2. We use "r" to stand for the node* "ref (0.2.0.3.0)"*, "n" for* "name (0.2.0.0.0.0)"*, "a" for* "abstract (0.2.0.2)"*, and "t" for* "title (0.2.0.1)" *respectively, so as to keep the following descriptions simple.*

To achieve $\mathcal{EC}_Q^T$, we should first compute the valid keyword node subsets from D_1 and D_2. Following Definition 1*, they are $V_1 = \{\{n\}, \{r\}, \{n, r\}\}$; while $V_2 = \{\{t\}, \{r\}, \{a\}, \{t, r\}, \{t, a\}, \{r, a\}, \{t, r, a\}\}$ respectively.*

Based on V_1 and V_2, we can get the $\mathcal{EC}_Q^T$ as $\{\{r\}, \{n, t\}, \{n, r\}, \{n, a\}, \{r, a\}, \{r, t\}, \{n, t, r\}, \{n, t, a\}, \{r, t, a\}, \{n, r, a\}, \{n, t, r, a\}\}$. The number of subsets is 11 (not $(2^2 - 1) \times (2^3 - 1) = 21$), because $D_1 \bigcap D_2 = \{r\} \neq \emptyset$.

From Example 3 it is clear that each partition in $\mathcal{EC}_Q^T$ naturally contains all the given keywords in Q. Now the next task is to filter the partitions in $\mathcal{EC}_Q^T$ so that the residual partitions satisfy the other two requirements – the uniqueness requirement and the completeness requirement, that is the LCA nodes of the residual partitions are the same as those LCA nodes computed according to [4, 12], and each partition contains all the related keyword nodes. We present the RTF concept in Definition 2 for this task, which provides the rules to filter $\mathcal{EC}_Q^T$.

Definition 2 (Relaxed Tightest Fragment) *Given an XML tree T, and the keyword query $Q = \{w_1, \ldots, w_k\}$, D_i records all the keyword nodes with respect to the keyword w_i, V_i is the set of the valid keyword node subsets of D_i, and $\mathcal{EC}_Q^T$ is the extended keyword node combination set on all D_i $(1 \le i \le k)$.*

For each element $\mathcal{EC}_{Q,j}^T$ in $\mathcal{EC}_Q^T$, $LCA(\mathcal{EC}_{Q,j}^T)$ is the unique LCA node based on all the nodes in $\mathcal{EC}_{Q,j}^T$, and $I(\mathcal{EC}_{Q,j}^T)$ records all the nodes determined by the keyword node set $\mathcal{EC}_{Q,j}^T$ and its LCA node, which is defined as $I(\mathcal{EC}_{Q,j}^T) = I(LCA(\mathcal{EC}_{Q,j}^T), \mathcal{EC}_{Q,j}^T) = \bigcup_{v \in \mathcal{EC}_{Q,j}^T} I(LCA(\mathcal{EC}_{Q,j}^T), v)$ [3]. $V_j|_i$ stands for the valid subsets of $\mathcal{EC}_{Q,j}^T|_i$, namely $V_j|_i = 2^{\mathcal{EC}_{Q,j}^T|_i} - \{\emptyset\}$.

[3]The function $I(u, v)$ stands for the path node set from the node u to the node v if there exists a path between u and v in T.

The $I(\mathcal{EC}^T_{Q,j})$ is a RTF if the $\mathcal{EC}^T_{Q,j}$ and its $LCA(\mathcal{EC}^T_{Q,j})$ satisfy all the following three conditions:

- $\nexists S_i \in V_j|_i (1 \leq i \leq k)$, $LCA\left(\mathcal{EC}^T_{Q,j}\right) \neq LCA\left(\bigcup_{i=1}^{k} S_i\right)$.
- $\forall \mathcal{EC}^T_{Q,j}|_i \in \mathcal{EC}^T_{Q,j}$, $\nexists V' \in (V_i - \mathcal{EC}^T_{Q,j}|_i) \wedge \mathcal{EC}^T_{Q,j}|_i \subset V'$, $LCA\left(\mathcal{EC}^T_{Q,j}\right) = LCA\left(\mathcal{EC}^T_{Q,j}|_1, \ldots V' \ldots, \mathcal{EC}^T_{Q,j}|_k\right)$.
- $\nexists V' \in V_j|_i$, $LCA(\mathcal{EC}^T_{Q,j}) \prec_a LCA(S_1, \ldots, V', \ldots, S_k)$ *where* $S_i \in V_i$ $(1 \leq i \leq k)$, *and* $u \prec_a v$ *represents that the node* u *is an ancestor of the node* v *in the XML tree.*

The first rule is used to ensure that there is no other keyword node subset in $\mathcal{EC}^T_{Q,j}$, which also covers the given keyword query, but whose LCA node is different from $LCA(\mathcal{EC}^T_{Q,j})$. The second rule is to make sure that the keyword node set $\mathcal{EC}^T_{Q,j}$ is the maximum set in $\mathcal{EC}^T_Q$ whose LCA node is $LCA(\mathcal{EC}^T_{Q,j})$. These two rules guarantee that the RTFs satisfying them conform to the completeness requirement, because they imply that the RTF corresponding to a LCA node contains all the related keyword nodes, no more and no less. As for the third rule, it means that any keyword node in $\mathcal{EC}^T_{Q,j}$ can not be included in other keyword node set whose LCA node is a descendant of $LCA(\mathcal{EC}^T_{Q,j})$. This rule helps to ensure that the RTF is unique, because it implies that any keyword node should belong to a unique partition. Together with Definition 1, we can see that the RTFs determined by Definition 2 satisfy not only the keyword requirement but also the uniqueness and the completeness requirements. Besides, it is easy to further distinguish the SLCA-related RTFs. We use Example 4 to illustrate the concept of RTF.

Example 4 (Illustration of RTF concept) *Following* Example 3*, there are only two keyword combination sets in $\mathcal{EC}^T_Q$ satisfying the RTF concept, $\{r\}$ and $\{n, t, a\}$. All the other partitions do not satisfy all those three conditions.*

For the partitions of $\{n, r\}$, $\{r, a\}$, $\{r, t\}$, $\{r, t, a\}$, $\{n, r, a\}$ and $\{n, t, r, a\}$, they could not be RTFs because they conflict both the first and the third conditions. It is clear that their LCA nodes are all the node "0.2.0 (article)"*, and $\{r\}$ is their common subset. Based on* Figure 1(a) *we can see that the node r* "0.2.0.3.0 (ref)" *itself contains all the keywords and is a LCA node. It is clear that it is a descendant of the node* "0.2.0 (article)" *– the common LCA node of those partitions.*

As for the partitions of $\{n, t\}$ and $\{n, a\}$, they are not RTFs either because they do not satisfy the second condition of Definition 2*. From* Figure 1(a) *we can see that they both cover all the keywords and their LCA nodes are same – the node* "0.2.0 (article)"*. When adding the node a to $\{n, t\}$, and the node t to $\{n, a\}$, the new partition is $\{n, t, a\}$, which also covers all the keyword nodes and whose LCA node is also the node* "0.2.0 (article)".

Following the Definition 2 and Example 4, it is clear that the fragments in Figure 2(a) and 2(b) are the RTFs for Q_2, and Figure 2(a) also satisfies the SLCA semantics. Figure 2(c) is the unique RTF for Q_3. Figure 3(b) is the unique RTF for Q_1, and Figure 3(d) is the unique RTF for Q_4. We can see that the RTF concept is applied uniformly across the SLCA and the LCA semantics, and it is easy to distinguish the SLCA related RTFs from the total RTFs for the given keyword query. All these properties guarantee that the RTF concept is appropriate to represent the basic result for XKS.

After retrieving the RTFs for the given keyword query, the next task is to prune the uninteresting nodes in a RTF so as to make sure that all the residual nodes are meaningful to the query. We denote the pruned RTF as meaningful RTF. For instance, we should find an appropriate filtering mechanism so as to get the meaningful RTF of Figure 2(d) from the raw RTF of Figure 2(c), keep the RTF of Figure 3(b) as the final RTF, and prune the duplicate forward "position" node in the RTF of Figure 3(d). From the discussion in Section 1 we have learnt that the contributor-based filtering mechanism could not be used for these RTFs of Figure 2(c), 3(b) and 3(d). In view of this, we further propose the concept of valid contributor to cope with this problem.

3. VALID CONTRIBUTOR FOR MEANINGFUL RTF

In this section, we introduce the concept of valid contributor. which can overcome the false positive problem and the redundancy problem of the contributor function in [1].

From Example 2 we learn that both the false positive problem and the redundancy problem are caused by the insufficient contributor function for MaxMatch algorithm in [1]. The key to overcome the two problems is to ameliorate the filtering condition to cope with the following two questions: (1) how to keep interesting node n when $dMatch(n) \subset dMatch(n_2)$, and (2) how to filter out the uninteresting node n when $dMatch(n) = dMatch(n_2)$.

Fortunately, Example 2 also reminds us of some hints to cope with those problems, *i.e.*, the label and the content information of the keyword node can help to overcome those problems. For instance in Figure 3(b) for the query Q_1, the node "0.2.1.1" should not be discarded even though for Q_1 $dMatch(0.2.1.1)$ = {query, skyline} $\subset$ $dMatch(0.2.1.2)$ = {dynamic, query, skyline}, because its label "title" is different from the label "abstract" of the node "0.2.1.2". For Figure 3(d) for Q_4, we have $dMatch(0.1.0)$ = $dMatch(0.1.2)$ = {position}, but we should discard one of them because their position values are same – "forward".

The above observations remind us that, intuitively, the label and the content information of the keyword nodes can help to overcome the two problems. That is, if the label of a child is unique among its siblings, this child should not be discarded even though its $dMatch$ is a subset of one of its siblings' $dMatch$s. And when two siblings have same label, if their values are same, one of them should be discarded even though their $dMatch$s are same. To formalize the above observations, we first present the concepts of tree content set and tree keyword set of a node in Definition 3, which collect the interesting words and the keywords contained in the subtree rooted at a node.

Definition 3 (Tree Content Set and Tree Keyword Set of a node) *Given an XML tree T, and the keyword query $Q = \{w_1, \ldots, w_k\}$, R is a RTF in T, which contains all the keywords in Q. The tree content set TC_v of a node v in R is the content union of all the keyword nodes rooted at the node v, that is $TC_v = \bigcup_{v' \in T_v} C_{v'}$, where T_v represents the subtree rooted at the node v, and $v' \in T_v$ stands for a keyword node in T_v. The tree keyword set of the node v is defined simply as $TC_v \cap Q$, denoted as TK_v*[4].

Based on Definition 3, we further present the concept of valid con-

[4]Clearly the concept TK_v here is the same as the $dMatch(v)$ in [1].

tributor in Definition 4, which helps to distinguish the valid contributors from the children of a node. Our filtering mechanism for a RTF then can be described as following: *every node in a RTF* (except for its root) *should be a valid contributor to its parent*.

Definition 4 (Valid Contributor) *Given an XML tree T, and the keyword query $Q = \{w_1, \ldots, w_k\}$, R is a RTF in T. u, v are two nodes in R, and u is the parent of v. The child v is a valid contributor of u if either of the following two conditions holds:*

1. *v is the unique child of u with label $\lambda(v)$;*
2. *v has several siblings $v_1, \ldots, v_m$ $(m \geq 1)$ with same label as $\lambda(v)$, but the following conditions hold:*
   (a) $\nexists v_i, TK_v \subset TK_{v_i}$;
   (b) $\forall v_i \wedge TK_v = TK_{v_i}, TC_v \neq TC_{v_i}$.

From Definition 4, the rule 1 aims to overcome the false positive problem. As for the rule 2, there are two subrules, in which 2.(a) inherits the positive property of the contributor mechanism, and 2.(b) is used to overcome the redundancy problem. We illustrate these in Example 5.

Example 5 (Illustration for the valid contributor) *We first illustrate that our valid-contributor-based filtering function also returns the positive example of contributor-based filtering illustrated in* Example 2. *Then we demonstrate how our proposal overcomes the false positive problem and the redundancy problem committed by the contributor-based filtering mechanism.*

[Covering the positive example] *As running Q_5 on* Figure 1(b):(1)*, the node* "0.1 (players)" *has three children with same label* "player" – "0.1.0"*,* "0.1.1" *and* "0.1.2"*. Their tree keyword sets are* {Gassol, position} *for* "0.1.0"*, and* {position} *for* "0.1.1" *and* "0.1.2"*. According to the rule* 2.(a) *in* Definition 4*, only the node* "0.1.0" *and its descendants are kept in the meaningful RTF, the same as the fragment in* Figure 3(a)*.*

[Overcoming the false positive problem] *Back to the RTF in* Figure 3(b)*, we can see that the node* "0.2.1 (article)" *has three children* – "0.2.1.0 (authors)"*,* "0.2.1.1 (title)"*,* "0.2.1.2 (abstract)"*. Since their labels are different from each other, all of them are valid contributors to their parent* "0.2.1 (article)"*. So all of them are kept in the final meaningful RTF, even though the tree keyword set* {query, skyline} *of the node* "0.2.1.0 (authors)" *is a subset of the tree keyword set* {dynamic, query, skyline} *of its sibling* "0.2.1.2"*.*

So, for Q_1 on the XML instance of Figure 1(a)*, the result by our valid-contributor-based filtering mechanism is the fragment in* Figure 3(b)*, not that in* Figure 3(c) *by the contributor-based mechanism.*

[Overcoming the redundancy problem] *Now we continue our illustration based on the RTF in* Figure 3(d) *of doing Q_4 on the XML segment of* Figure 1(b):(1)*, in which the MaxMatch will commit the redundancy problem. When using our valid contributor to filter those nodes, however, the final result will only contain one* "forward" *position node.*

According to Definition 3*, the tree keyword sets of the three nodes* – "0.1.0 (player)"*,* "0.1.1 (player)" *and* "0.1.2 (player)" *are all* {position}*. While their tree content sets are $TC_{0.1.0} =$* {position, forward}*, $TC_{0.1.1} =$* {position, guard} *and $TC_{0.1.2} =$* {position, forward} *respectively. Since there are only two distinct valid contents* – {position, forward} *and* {position, guard}*, the filtering mechanism based on the valid contributor will discard one of the nodes with valid content* {position, forward}*. Consequently the final result contains all the position information* – {forward, guard}*, and keeps itself concise without the redundant nodes.*

Finally, we briefly illustrate how to get the meaningful RTF in Figure 2(d) *for the query Q_3. According to* Definition 2*, the RTF for Q_3 on* Figure 1(a) *is the fragment in* Figure 2(c)*. When using our valid contributor to prune it, there are two nodes that we should pay attention to, i.e.,* "0.2 (Articles)" *and* "0.2.0 (article)"*. The former has two children* – "0.2.0 (article)" *and* "0.2.1 (article)"*, whose tree keyword sets are* {title, XML, keyword, search} *and* {title} *respectively. According to the rule* 2.(a) *of* Definition 4*, only the node* "0.2.0 (article)" *is the valid contributor of its parent* "0.2 (Articles)"*. As for the node* "0.2.0 (article)"*, its three children* – "0.2.0.1 (title)"*,* "0.2.0.2 (abstract)" *and* "0.2.0.3 (references)"*, are all valid contributors following the rule* 1 *of* Definition 4*, because their labels are different. So, for Q_3, the result by our valid-contributor-based filtering mechanism is the fragment in* Figure 2(d)*.*

From Example 5 we can see that the valid contributor not only inherits the good property of the contributor, but also can overcome both the false positive problem and the redundancy problem of the contributor. These properties of our valid contributor guarantee that it has better capability than the contributor to filter the RTFs. This is also verified by the later experiments in Section 5.

4. VALIDRTF

In this section, we introduce the algorithm of ValidRTF. We first describe the data structure of a node in Section 4.1, which facilitates the required computations. Then we demonstrate the ValidRTF algorithm in Section 4.2 following a running example. Finally the analysis of ValidRTF is concluded in Section 4.3.

4.1 The node data structure

The basis of our ValidRTF is the data structure of a node designed here. From Definition 4 and Example 5, there are three requirements we should take into account to design the node data structure, namely (1) we need a label item to store the children with the same label; for each label item, (2) how to examine if there is a superset for the given keyword set in a set of keyword sets, and (3) how to check if there is an equal set for the given word set in a set of word sets. Clearly (2) corresponds to the condition 2.(a), and (3) corresponds to the condition 2.(b) in Definition 4 respectively.

To support all of them, we design the data structure of a node as shown in Figure 4(a), which comprises two parts: (1) the information of the node itself in "Self Info" frame; and (2) the information of its children in "Children Info" frame. The former stores the basic information of the node itself, including its Dewey code(*dewey*), label (*label*), keyword list (*kList*, which corresponds to the tree keyword set of the node, and is the same as the list used in MaxMatch) and the content ID (*cID*, which represents the tree content set, and only records the word pair (min, max) of the tree content set of the node. The min and max are the smallest and the largest word in the tree content set according to the lexical order.). The latter stores the information of a node's children, maintained in a list *chlList* according to their distinct labels. For each distinct label, there is an item, which stores *counter* (the number of the children with that distinct label), *chkList* (records the sorted distinct keyword list numbers), *chcIDList* (stores the cIDs of the children), and

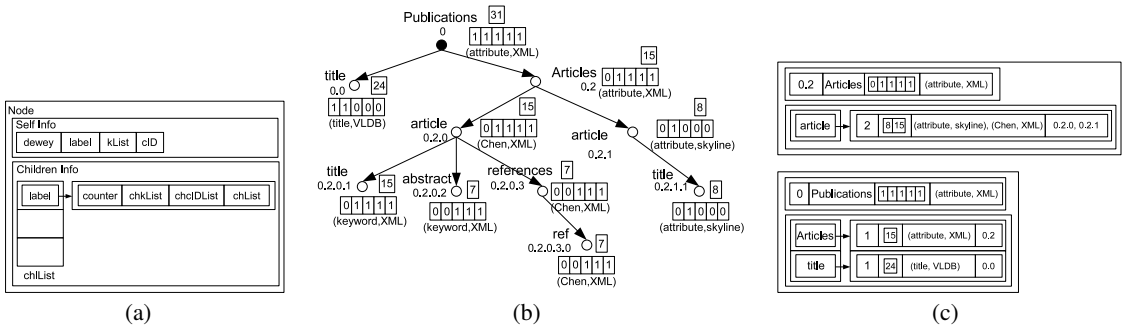


Figure 4: (a): The node data structure designed for our ValidRTF algorithm; (b): The sketch of the RTF in Figure 2(c) after the constructing step in pruneRTF; (c): Two node instances filled according to the node data structure defined in Section 4.1.

chList (records the references to those children).

From the above description we can see that the requirement (1) is supported by storing the children in *chlList* according to their labels.

As for the requirement (2), it is supported by *kList* in "Self Info" and *chkList* in a label item together. The computation for requirement (2) is the same as that of $dMatch(n) \subset dMatch(n_2)$ in MaxMatch. For example, if the query Q_3= "VLDB title XML keyword search", and the *kList* of a node v is $\boxed{0}\boxed{1}\boxed{1}\boxed{1}\boxed{1}$, it means the tree keyword set of the node v contains the keyword "title XML keyword search". We can infer that its key number is then $0 * 2^4 + 1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 1 * 2^0 = 15$. If the node u is the parent of v, v_1 is u's another child with same label $\lambda(v)$ as v, and its *kList* is like $\boxed{0}\boxed{0}\boxed{1}\boxed{1}\boxed{1}$ (its key number is then 7), which means v_1 contains keywords "XML keyword search". Then the item in *chlList* corresponding to $\lambda(v)$ has *chkList* as $\boxed{7}\boxed{15}$. If we want to check if there is a sibling of the node v_1 whose keyword set covers v_1's, we only need to compare v_1's key number 7 with those numbers that are larger than 7. Since $15 > 7$ and $(7 \texttt{ AND } 15) = true$, we can directly infer that there is a sibling of v_1, which has same label as v_1's and whose keyword set covers v_1's.

As for the computation for requirement (3), generally we need to compare all the tree content sets of a node's children to find out if a child's content is duplicate or not. However, from our knowledge the computation following this idea is expensive. Hence, we take an approximate method, *i.e.*, we construct a feature for each content set, and compare the features to find if there is duplicate content or not. We choose a word pair *cID* =(min, max) as the feature for a tree content set of a node, which corresponds to the smallest and the largest word in the set (sorted according to the lexical order). For instance, the sorted tree content set of the node "0.2.0.1" in Figure 1(a) could be {keyword, match, relevant, search, XML}, therefore, its *cId* is (keyword, XML). Once all the *cId*s of the children with same label are sorted in *chcIDList* of their parent's label item, it is easy to find out if there is any duplicate content by comparing the *cID*s. If two children have the same *cID*, we conceive that their corresponding content sets are same.

4.2 ValidRTF algorithm

The *pseudo code* of ValidRTF is shown in Algorithm 1 with four stages like MaxMatch. The first is *getKeywordNodes*, which retrieves all the keyword nodes in the XML data, and dispatches them into corresponding node set D_i according to the keyword w_i in Q. The retrieved keyword node sets are transferred to the second stage – *getLCA*, which is directly the Indexed Stack algorithm of [12], and returns all the LCA nodes in *LCAs*. The LCA nodes in *LCAs* are stored following the pre-order relationship of their Dewey codes[5]. The third stage is *getRTF*, which takes the LCA nodes and the keyword nodes as input, and collects all the related keyword nodes for each LCA node. The retrieved Dewey codes of the keyword nodes for each LCA node are sorted based on the pre-order relationship. Finally, the *pruneRTF* procedure filters out the uninteresting nodes of each RTF based on our valid contributor.

Next we illustrate each stage of the algorithm using Q_3 as a running example on the XML instance of Figure 1(a). Its result is the whole fragment in Figure 2(d), which lists all the papers published in "VLDB" 2008 on "XML keyword search". We briefly demonstrate the first three procedures in Example 6, because they are similar with those in MaxMatch except for the LCA nodes and the update of the node information. Then we pay much attention to the pruning procedure, especially the pruning based on the condition 1 and the condition 2.(b) in Definition 4.

Example 6 (The first three procedures of ValidRTF) *When processing the keyword query Q_3 on the XML instance in* Figure 1(a)*,* GETKEYWORDNODES *returns three node sets – $D_1 = \{0.0\}$ for keyword* "VLDB"*, $D_2 = \{0.0, 0.2.0.1, 0.2.1.1\}$ for keyword* "title" *and $D_3 = D_4 = D_5 = \{0.2.0.1, 0.2.0.2, 0.2.0.3.0\}$ for keywords* "XML"*,* "keyword" *and* "search" *respectively.*

GETLCA *computes the LCA nodes based on the above five keyword node sets, and there is only one LCA node (also the SLCA node) –* "0 (Publications)"*.*

When running GETRTF*, the related keyword nodes of each LCA node are collected, and stored using RTF data structure. For instance, the RTF R for LCA node* "0 (Publications)" *comprises three parts. $R.a$ records the Dewey code of the LCA node, and here it is 0. $R.knodes = \{0.0, 0.2.0.1, 0.2.0.2, 0.2.0.3.0, 0.2.1.1\}$ stores all the keyword nodes, and each of them is filled with necessary information following the discussion in Section 4.1. The elements in $R.knodes$ are sorted according to the pre-order relationship of*

[5]Similarly to [4, 7, 12], each node v in the XML tree T is assigned a Dewey code d_v that is compatible with preorder numbering, in the sense that if a node v_1 precedes a node v_2 in the preorder left-to-right depth-first traversal of the tree then $d_{v_1} < d_{v_2}$. And the relationship between every two different Dewey code following that sense is denoted as the pre-order relation between those two Dewey codes.

Algorithm 1 VALIDRTF ALGORITHM

VALIDRTF (T, Q)

Input: The XML data T and the keyword query $Q = \{w_1, \ldots, w_k\}$
Output: All the meaningful RTFs containing only reasonable nodes

1. $D_i \leftarrow$ getKeywordNodes(T, Q);
2. LCAs $\leftarrow$ getLCA$(D_1, \ldots, D_k)$;
   /* The getLCA function is directly the IndexedStack function from [12], which returns all the interesting LCA nodes. The nodes in LCAs are sorted according to the pre-order relationship of their Dewey codes */
3. RTFs $\leftarrow$ getRTF(LCAs, $D_1, \ldots, D_k$);
4. **For** (each RTF R in RTFs)
5. pruneRTF(R)

GETRTF (LCAs, $D_1, \ldots, D_k$)

1. **For** (each LCA node a in LCAs)
2. **Construct** a RTF R for a: $R.a \leftarrow a$, $R.knodes \leftarrow \{\}$, and $R.r$ is the root node for the tree used in later PRUNERTF;
3. **For** (each node d in $D_1, \ldots, D_k$)
4. **Locate** the last RTF R in LCAs whose $R.a$ is the ancestor of or the same as d;
5. **If** $R.knodes$ contains d
6. **Update** the information of the corresponding keyword node in $R.knodes$;
7. **Else**
8. **Add** d into $R.knodes$;

PRUNERTF (R)

/* CONSTRUCTING STEP */
1. $i \leftarrow |R.knodes| - 1$;
2. $start \leftarrow R.a$;
3. **While** $(i \geq 0)$ **do**
4. $n_i \leftarrow R.knodes$'s i^{th} keyword node;
5. **For** (each ancestor x of n_i on the path from n_i to $start$)
6. **Construct** a node X following the node data structure described in Section 4.1, and fill X based on the information in x;
7. **If** (the tree $R.r$ contains a node with same Dewey code as X)
8. **Update** the corresponding node using X;
9. **Else**
10. **Add** X into $R.r$;
    /* The following two lines are added to guarantee that the node information of n_i could be totally transferred to all its ancestors. */
11. **If** $(start \neq R.a)$
12. **Update** the nodes from $start$ to $R.a$ using the information of the node corresponding to $start$;
13. $i \leftarrow i - 1$;
14. **If** $(i < 0)$ break;
15. **Else** $start \leftarrow LCA(n_i, n_{i+1})$;

/* PRUNING STEP */
16. **Breadth-first traverse** the tree, and **For** (each node n)
17. **If** ($n.chlList$ is not *null*)
18. **For** (each label item $n.chlList[j]$)
19. **If** $(n.chlList[j].counter! = 1)$
20. $usedKNums \leftarrow \{\}$, $usedCIDs \leftarrow \{\}$;
21. **For** (each ch in $n.chlList[j].chList$)
22. **If** $(knum(ch.kList) \in usedKNums)$
23. **Keep** ch when $ch.cID \notin usedCIDs$;
24. **Else If** $(knum(ch.kList)$ is not covered by any element in $n.chlList[j].chkList$ which is larger than it)
25. **Keep** ch, **add** $knum(ch.kList)$ into $usedKNums$ and $ch.cID$ into $usedCIDs$;
26. **Else Keep** that child node in $n.chlList[j]$.

their Dewey codes. $R.r$ is a node filled only with dewey = 0, which is used to store the root of the tree that corresponds to the RTF.

After collecting all the related nodes for each RTF, the next stage is PRUNERTF, which has two steps – the constructing step and the pruning step. The former is to construct a tree, in which each node is filled following the node data structure defined in Section 4.1. The latter is to prune the tree by using our valid-contributor-based mechanism so as to filter out the uninteresting nodes in each RTF. Example 1 demonstrates this procedure.

Example 7 (pruneRTF) *There are two steps in* PRUNERTF.

[Constructing step] *In essence, the task of the constructing step in* PRUNERTF *is to collect the information of the nodes in a RTF, and keep the relations among them for the later pruning step. It begins with each keyword node. After collecting the necessary information following the definition of the node data structure in* Section 4.1*, it then transfers the information of the keyword node to its ancestors. We use* Figure 4(b) *to demonstrate this.*

Compared with the RTF in Figure 2(c)*, the sketch keeps the label and Dewey code for each node, and collects the necessary information from its children. The information contains three fields, which correspond to its kList, key number and cID. The kList and cID of a node absorb the kLists and cIDs of its children (if it has). For instance, the kList information of the node* "0.2 (Articles)" *is* $\boxed{0}\boxed{1}\boxed{1}\boxed{1}\boxed{1}$*, which means the subtree rooted at* "0.2" *contains the keywords* "title XML keyword search"*. Its key number is then* $\boxed{15}$*, and its cID =* (attribute, XML)*.*

Figure 4(c) *illustrates two concrete nodes* – "0.2 (Articles)" [top] *and* "0 (Publications)" [bottom]*, which are filled with the data according to the description of the node data structure. Since there are only two distinct labels among its children, the node* "0" *has two label items as shown in the bottom of* Figure 4(c)*. Besides, since its children's cIDs satisfy* "attribute" $\prec$ "title" $\prec$ "VLDB" $\prec$ "XML" *according to the lexical order, the cID of* "0" *is* (attribute, XML)*, which, we conceive, covers the content of its children*[6]*. As for the node* "0.2"*, it has only one label item even though it has two children, because their labels are same* – "article"*. Its cID is* (attribute, XML) *too, as its children's cIDs satisfy* "attribute" $\prec$ "Chen" $\prec$ "skyline" $\prec$ "XML".

[Pruning step] *After collecting the necessary information for each node in a RTF, the pruning is relatively simpler. According to* Algorithm 1*, the pruning step visits each node during the breadth-first traversing of the RTF, and distinguishes the valid contributors from its children following the rules defined in* Definition 4.

The node "0" *is the first. From the bottom of* Figure 4(c)*, we can see that the two labels of its two children* "0.0" *and* "0.2" *are distinct. Consequently they are both valid contributors of their parent* "0"*, and are kept in the result according to* line 26*.*

For the node "0.2"*, we can see from the top of* Figure 4(c) *that its two children have same label* – "article"*. Since the counter of the corresponding label item is 2, we continue to check each child according to* line 21*.*

(1) *For the child* "0.2.0"*, the $usedKNums$ is empty and we continue to check if its $kList$ is covered by other child* (line 24)*. Its $kList$ is* $\boxed{0}\boxed{1}\boxed{1}\boxed{1}\boxed{1}$*, and the key number is 15. Since 15 is the largest in chkList, the child* "0.2.0" *is a valid contributor of its parent* "0.2"*. Now the $usedKNums$ is* {15}*, and the $usedCIDs$ is* {(Chen, XML)} *according to* line 25*.*

(2) *As for the child* "0.2.1"*, its $kList$ is* $\boxed{0}\boxed{1}\boxed{0}\boxed{0}\boxed{0}$*. From* (1) *we can see that $usedKNums$ does not contain its key number 8. Similarly we continue to check if its $kList$ is covered by other child or not. From the top of* Figure 4(c)*, we can see that 15 in $chkList$ is larger than 8 and covers the latter. Therefore the child* "0.2.1" *is not a valid contributor of its parent* "0.2"*.*

[6] As mentioned in Section 4.1, the cID of a node is an approximate representation of its tree content set. It is possible that two different content sets have the same cIDs. However, the well-performed property of the XML data guarantees that this is applicable.

4.3 Analysis

Now we present the analysis of our ValidRTF algorithm. There are four aspects we are interested in.

(1). After getting all the interesting LCA nodes by using the Indexed Stack algorithm of [12], the *getRTF* procedure can retrieve all the basic RTFs.

Since the Indexed Stack *algorithm from* [12] *returns all the interesting LCA nodes in an XML data for the given keyword query, we only need to illustrate that the nodes collected by* getRTF *for each LCA node satisfy the requirements of a RTF determined by* Definition 2.

Following the description of getRTF*, it is clear that each keyword node is dispatched to the last LCA node in the sorted LCA node set, which is the ancestor of or equal to the keyword node. Consequently, after* getRTF*, each keyword node set corresponding to an interesting LCA node a satisfies the following conditions*: (a) *the content union of the keyword nodes with regard to a covers all the keywords in the query*[7]*,* (b) *the node a is the unique common LCA node of those keyword nodes*[8]*, and* (c) *there is no subset of those keyword nodes whose content union covers the keyword query, and whose LCA node a' is not a*[9]*. We can see that the keyword nodes for each LCA node satisfy the keyword requirement, the uniqueness requirement and the completeness requirement together. Therefore, each keyword node set collected by* getRTF *corresponds to a basic RTF with respect to that LCA node.*

(2). Our valid-contributor-based ValidRTF algorithm satisfies the four required properties for an XKS technique deduced by [1].

According to Section 1*, the essence of those requirements is that an XKS technique should be able to control the result when changing the XML data or the keyword query. Following the description of the* Algorithm 1 *and the discussion in the above* (1)*, the* getLCA *and the* getRTF *guarantee that the changes of the XML data and the keyword query can be caught by our* ValidRTF*. This is because* getLCA *retrieves all the LCA nodes, and* getRTF *collects all the related keyword nodes for each LCA node once the keyword query and the XML data are fixed.*

(3). The concept of our valid contributor can overcome the false positive problem and the redundancy problem committed by the contributor in [1].

This is intuitively expressed in Example 5.

(4). Our ValidRTF has competent performance as the revised MaxMatch for RTFs[10].

Following the description of the Algorithm 1 *we can see that the first three stages in our* ValidRTF *and the revised MaxMatch are similar. As for the pruning stage, the* pruneRTF *in our* ValidRTF *first dispatches the children of a node according to their unique labels. Since the processing for the children with unique labels is simple, the performance of* pruneRTF *is mainly determined by the processing for the children with the same label. There are two situations – checking if a child's keyword set is covered and if there is duplicate content among its siblings. From* Example 7 *we can see that the former is more complex. Consequently the complexity of* pruneRTF *can be represented by the cost of checking if a child's keyword set is covered. However, this part in* pruneRTF *is similar to $\nexists n_2, dMatch(n) \subset dMatch(n_2)$ in* pruneMatches*. Therefore our* ValidRTF *has competent performance as that of the revised Maxmatch. This is also verified by our experiments shown in* Section 5.

5. EXPERIMENTS

In this section, we present the experimental result to illustrate the efficiency and the effectiveness of our ValidRTF. We implement our ValidRTF and the revised MaxMatch, then compare them on both real (DBLP[19]) and synthetic datasets (generated by XMark [20]). We only choose MaxMatch (we still keep MaxMatch as the name of its revised version for simplicity) in our experiments, because it is the only algorithm which satisfies the axiomatic properties according to [1].

5.1 Configurations

The DBLP dataset is "dblp20040213" (197.6MB) from [19]. The three XMark datasets are standard (111.1MB), data1 (334.9MB) and data2 (669.6MB). The XMark standard data is directly downloaded from [20], while the other two XMark data are generated by XMark with the size limitations. When shredding XML data into the relational tables (refer to Section 5.2), we record the frequency of the interesting words, and choose the following keywords to compose our keyword queries.

• **Keywords for DBLP**: **k**eyword (90), **s**imilarity (1242), **r**ecognition (6447), **a**lgorithm (14181), **d**ata (25840), **p**robabilistic (2284), **x**ml (2121), d**y**namic (7281), si**g**mod (3983), **t**ree (3549), **q**uery (3560), aut**o**mata (3337), patter**n** (6513), retrieva**l** (5111), e**f**ficient (8279), **u**nderstanding (1450), sear**c**hing (4618), **v**ldb (2313), **h**enry (1322), se**m**antics (3694)

• **Keywords for XMark series**: **p**article (12, 33, 69), domin**a**tor (56, 150, 285), **t**hreshold (123, 405, 804), **c**hronicle (426, 1286, 2568), **m**ethod (552, 1667, 3356), **s**trings (615, 1847, 3620), **u**njust (1000, 3044, 6150), **i**nvention (1546, 4715, 9404), **e**gypt (2064, 5255, 12466), **l**eon (2519, 7647, 15210), pre**v**entions (66216, 199365, 397672), **d**escription (11681, 35168, 70230), **o**rder (12705, 38141, 76271)

The integer(s) behind each keyword is(are) the frequency(ies) of corresponding keyword in the XML data. For example, "**k**eyword (90)"s means that there are 90 "keyword" words in DBLP. While "**p**article (12, 33, 69)" means that there are 12, 33 and 69 "particle" words in XMark standard, data1 and data2 respectively.

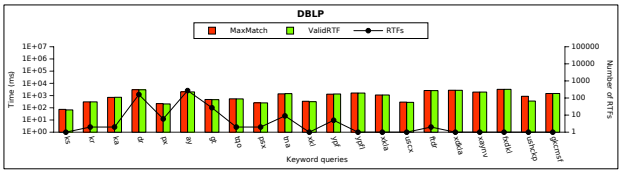
The underlined letter in each keyword is used as the abbreviation of that keyword in the keyword queries. For instance, the "vdo" for XMark series in Figure 5(b)∼5(d) means that the keyword query is "pre**v**entions **d**escription **o**rder". By randomly combining these

[7]This is guaranteed by the semantics of the LCA node computation by the Indexed Stack algorithm, and the collecting mechanism of getRTF which is based on the sorted LCA node set. The former implies that there is a keyword node set that covers the keyword query. While the latter ensures that its computed node set contains all those keyword nodes used to compute the corresponding LCA node.

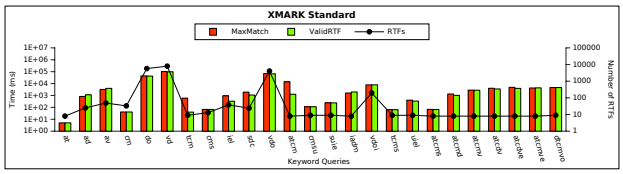
[8]Guaranteed by the property of the Indexed Stack algorithm.

[9]If there is a subset like that, its LCA node must have been figured out by the Indexed Stack algorithm, and the getRTF can dispatch those keyword nodes to that LCA node accordingly.

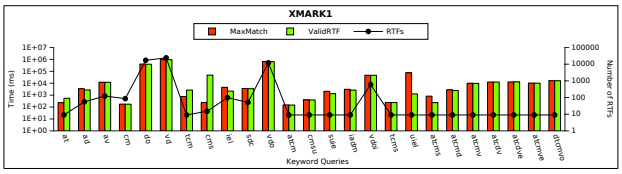
[10]Except for the node data structure, there are mainly two modifications in MaxMatch. (1) The *findSLCA* procedure for SLCA nodes is substituted with the Indexed Stack algorithm for LCA nodes; (2) The codes of line 11 and 12 in pruneRTF are added into pruneMatches of MaxMatch to guarantee that the information of the keyword nodes can be correctly transferred to all its ancestors.



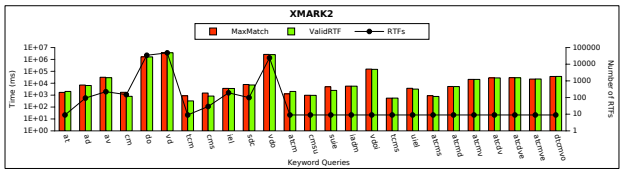
(a) dblp20040213



(b) xmark standard



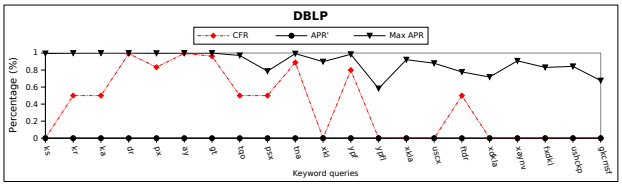
(c) xmark data1



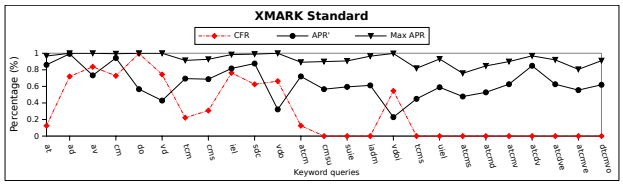
(d) xmark data2

Figure 5: Performance of ValidRTF and MaxMatch on the datasets with different keyword queries.

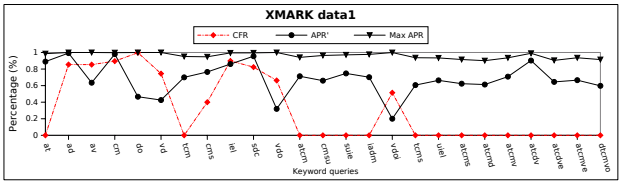
.



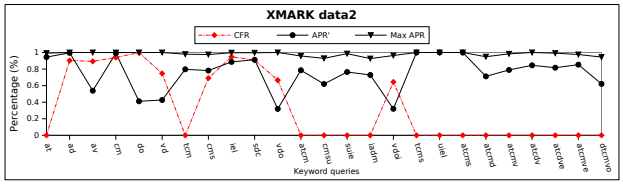
(a) dblp20040213



(b) xmark standard



(c) xmark data1



(d) xmark data2

Figure 6: Ratio of the retrieved results by ValidRTF and MaxMatch

keywords, we construct different keyword queries, which cover different frequency requirements.

To accomplish the performance comparison, we run those keyword queries composed by the above selected keywords in ValidRTF and MaxMatch. The efficiency is illustrated by recording the elapsed time of a keyword query on a dataset. For each dataset, we run ValidRTF and MaxMatch 6 times, and record the average time after discarding the first processing.

As far as the effectiveness is concerned, we use two ratios to illustrate it, namely the common fragment ratio (CFR) and the average pruning ratio (APR). They both can reflect the capability of our valid-contributor-based filtering mechanism. We first present several notions, and then give out the definitions of those two ratios. For a given keyword query Q, the sets A, V and X stand for respectively the LCA nodes computed by getLCA, the meaningful RTFs computed by our ValidRTF, and the fragments computed by MaxMatch. It is clear that the LCA node sets of V and X are the same as A. For a LCA node $a \in A$, we use v_a and x_a to represent the corresponding meaningful RTF in V and the fragment in X. $v_a - x_a$ stands for the nodes further discarded by our ValidRTF. The two ratios are defined as follows.

(1). [CFR] If the node sets in v_a and x_a are same, we conceive that the two fragments are same. We define $V \cap X$ standing for the same fragments in V and X. The CFR for a query Q is defined as $CFR = \frac{|V \cap X|}{|A|}$, which records the ratio of the same result computed by ValidRTF and MaxMatch for the given query. If CFR=1, it means that our ValidRTF and the MaxMatch return same result for that query.

(2) [APR] As for the average pruning ratio, it is defined as $APR = \frac{\sum_{a \in A} \frac{|x_a - v_a|}{|x_a|}}{|V - V \cap X|}$. It is clear that $\frac{|x_a - v_a|}{|x_a|}$ records the ratio of the discarded nodes in x_a, and $|V - V \cap X|$ corresponds to the number of the RTFs which our ValidRTF can further prune. If $APR = 0$, it means that there is no need for our ValidRTF to prune the RTFs computed by MaxMatch.

In short, for a given query, if the CFR is smaller than 1, it means that there exist RTFs in the result computed by MaxMatch, which still contain uninteresting nodes and can be filtered by our ValidRTF. The smaller the CFR for a query is, the more RTFs our ValidRTF needs to prune in the result retrieved by MaxMatch for that query. As for the APR, the larger the APR for a query is, the more uninteresting nodes our ValidRTF discards from the RTFs retrieved by MaxMatch.

5.2 Platform

Our experiments are run on a Dell OPTIPLEX 745, with two Intel Core CPUs (2.4GHz), 2GB memory and 160GB hard disk. The OS is Ubuntu 7.0.4. The algorithms are implemented in Java using Eclipse 3.2. We use Xerces 2.9.0 to parse the XML documents, and it is the stop-word filtering function in Lucence [21]to filter the stop-words [22].

The shredded records are stored in PostgreSQL 8.2.4 with three simple tables – "label (label, ID)", "element (node's label, Dewey, level, label number sequence, content feature)" and "value (node's label, Dewey, attribute, keyword)". The label table records all the distinct labels in the XML data, and their unique number (ID) assigned during the shredding. The element table stores the information of the nodes, in which the field "content feature" corresponds to the node's cID. As for the field "label number sequence", it is used to record the labels of a node's ancestors on the path from it to the root of the XML tree, which is useful to support the collection of the node's information in Algorithm 1[11]. As for the value table, it maintains the information of all the interesting keywords. When given a keyword query Q, we use SQL to search for the interesting keywords in the fields – "node's label", "attribute" and "keyword" of the value table, and collect the information of "label number sequence" from element table. Then our ValidRTF and MaxMatch are run on those keyword nodes.

5.3 Experiment Results

This section presents the result of our experiments.

• **Performance**: Figure 5 shows the performance of ValidRTF and MaxMatch. As described in Section 5.1 , we run the keyword queries on the datasets, and record the elapsed time after retrieving the Dewey codes of the keyword nodes. The results on the dblp, xmark's standard, data1 and data2 are shown as the bars in Figure 5(a)∼5(d) respectively. From those figures we can see that ValidRTF has competent performance as MaxMatch does on both real and synthetic data.

Besides, we also record the number of the RTFs for each query in those figures, shown as "RTFs" lines. They are recorded for the better understanding of the CFR lines in Figure 6. Based on the values of RTFs and CFR for a query, we can learn the number of the further pruned fragments by our ValidRTF after MaxMatch. For instance, the values of RTFs and CFR for the query "keyword recognition (kr)" in dblp data are 2 and 50% respectively. This means that one of the two fragments computed by MaxMatch still contains uninteresting nodes that are pruned further by our ValidRTF.

• **Effectiveness**: Figure 6 illustrates the effectiveness of our valid-contributor-based ValidRTF algorithm. We take the same keyword queries used for the performance experiments, and record the ratios of the results retrieved by ValidRTF and MaxMatch following the description in Section 5.1. The CFR is shown as diamond line in each figure. As for the APR, we split it into two variations. One is the Max APR shown as nabla line, which records the maximum value of $xv_{max} = \max_{a \in A} \left\{ \frac{|x_a - v_a|}{|x_a|} \right\}$. The other is the APR' shown as bullet line, which corresponds to the APR values after discarding the xv_{max}. The reason of this split is to highlight the pruning difference on the regular RTF and the extreme RTF. We find in our experiments that there is always an extreme RTF for each keyword query which usually collects many keyword nodes, and generally corresponds to the root of the XML tree. The xv_a on it is commonly huge, which will mask the APR values for regular RTFs.

Figure 6(a) shows the result on dblp data, and Figure 6(b)∼6(d) correspond to the results on xmark series data – standard, data1 and data2. For dblp data, the values of APR' are all zero, which means either there is no other RTF except for the extreme, or our ValidRTF need not further prune the regular RTFs. The reason of this phenomenon is that the dblp data is generated from real applications, and the keywords in it implicitly obey some practical rule, which ensures that the regular RTFs are self-complete. As for the Max APR, all its values are larger than 20%. This means that the extreme fragment after MaxMatch still contains many uninteresting nodes (at least 20%). For CFR, all its values are smaller than 1, even though there are several nodes whose values are close to 1. According to the discussion in Section 5.1, we can see that our ValidRTF is more powerful than MaxMatch to filter out the uninteresting nodes.

As for Figure 6(b)∼6(d), their biggest differences from Figure 6(a) are that the values of their APR' lines are all larger than 0. This means, according to the description of APR' in Section 5.1, that all the fragments retrieved by MaxMatch still contain uninteresting nodes, which are filtered by our ValidRTF. This further verifies that our valid-contributor-based mechanism can prune more uninteresting nodes than the contributor-based mechanism does. As for their Max APR lines, they are all close to 1, which means that there are more uninteresting nodes in those extreme fragments. They all intuitively are caused by the fact that those data are synthetic, and the distribution of the keywords in them is less meaningful. For the CFR lines, they have similar situation like that in Figure 6(a).

In summary, in terms of efficiency, the experimental result shows that ValidRTF has competent performance like MaxMatch does. As far as the effectiveness is concerned, our valid-contributor-based ValidRTF is more effective in pruning the uninteresting nodes than the contributor-based MaxMatch is.

6. RELATED WORK

This section briefly reviews the related work on XML keyword search.

Following the knowledge of Information Retrieval on text data [23], the basic task of adapting keyword search over XML data is clear, *i.e.*, retrieving meaningful fragments (instead of the whole document), each of which contains all the given keywords. To accomplish this task, many ideas are proposed, including the LCA variants [4, 6, 7, 9, 10, 24, 12, 1], interconnected pairs [5], GDMCT (Grouped Distance MCT – Minimum Connecting Tree) [8], MIU (Minimum Information Unit) [17] and biased snippet [25].

Among them, [1] is worth attention. As illustrated in Section 1, it retrospects the related work and deduced several axiomatic properties that an XKS technique should satisfy, namely the *consistency* and *monotonicity* with regard to the query and the XML data. After that, [1] further proposes the MaxMatch algorithm, which uses the contributor-based filtering mechanism to prune the SLCA-based fragment to ensure that the filtered fragment is more meaningful. However, the contributor-based mechanism is not sufficient. It will commit the false positive problem and the redundancy problem. Besides, it is not adequate to focus only on the SLCA related fragments, because the LCA related fragments, whose roots are ances-

[11]For instance in Figure 1(a), the numbers for the labels "Publications", "Articles" and "article" can be 0, 3 and 4 respectively. Then the "label number sequence" of the node "0.2.0" is "0.3.4".

tors of those SLCA nodes, could be still meaningful (conformed by [4, 11, 12, 17]).

Along with the work on retrieving meaningful fragments for XKS, the following three aspects are also interesting. The first is the extension of the keyword query so as to incorporate more information in the keywords, including [5, 26, 27, 28]. The second is about integrating keyword proximity search in a structural query language, such as [29, 30, 32, 31]. The third is the ranking of the retrieved fragments in [4, 5, 33].

Besides the above research concentrating mainly on the XML data (semi-structured data), the research of using keyword search on other kinds of data, such as the structured data (relational data) and the graph data, is also becoming attractive. They usually take the data as a graph, and focus on returning Steiner trees. We refer readers to [34] for the related work. As for [34] itself, by modeling all the data as a graph, it proposes a framework for keyword search on that graph.

7. CONCLUSION

This paper revisits XML keyword search (XKS), and focuses on returning the meaningful fragments for the query. It begins by introducing the Relaxed Tightest Fragment (RTF) to represent the basic result for XKS, which is based on the partitioning of the keyword nodes and the LCA concept, and relates to all the LCA nodes. Then we propose valid contributor concept to prune the basic RTFs to get the meaningful fragments. Compared with the contributor concept in [1], our valid contributor can overcome the false positive problem and the redundancy problem. Finally we implement the above concepts in the ValidRTF algorithm, and compare it with the MaxMatch algorithm after modifying MaxMatch for the RTFs. The experiments on real and synthetic data verify the efficiency and the effectiveness of our ValidRTF.

This work improves the research of [1] mainly in two aspects. First, it extends the processing to all LCA nodes after formalizing the RTF concept. Second, it ameliorates the filtering mechanism to get more meaningful result by the valid contributor concept. It can filter out more uninteresting nodes in the RTFs, while keeping the related interesting nodes. However, the ranking of the retrieved meaningful RTFs is still needed for carrying out the keyword search over XML data, and this is also a part of our future work.

8. REFERENCES

[1] Z. Liu and Y. Chen, "Reasoning and identifying relevant matches for xml keyword search," in *VLDB*, 2008.

[2] I. Tatarinov and S. D. Viglas, "Storing and querying ordered xml using a relational database system," in *SIGMOD*, 2002.

[3] D. Florescu, D. Kossmann, and I. Manolescu, "Integrating keyword search into xml query processing," in *WWW*, 2000.

[4] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram, "Xrank: Ranked keyword search over xml documents," in *SIGMOD*, 2003.

[5] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv, "Xsearch: A semantic search engine for xml," in *VLDB*, 2003.

[6] Y. Li, C. Yu, and H. V. Jagadish, "Schema-free xquery," in *VLDB*, 2004.

[7] Y. Xu and Y. Papakonstantinou, "Efficient keyword search for smallest lcas in xml databases," in *SIGMOD*, 2005.

[8] V. Hristidis, N. Koudas, Y. Papakonstantinou, and D. Srivastava, "Keyword proximity search in xml trees," *IEEE TKDE*, vol. 18, no. 4, pp. 525–539, April 2006.

[9] C. Sun, C. Y. Chan, and A. K. Goenka, "Multiway slca-based keyword search in xml data," in *WWW*, 2007.

[10] Z. Liu and Y. Chen, "Identifying meaningful return information for xml keyword search," in *SIGMOD*, 2007.

[11] G. Li, J. Feng, J. Wang, and L. Zhou, "Effective keyword search for valuable lcas over xml documents," in *CIKM*, 2007.

[12] Y. Xu and Y. Papakonstantinou, "Efficient lca based keyword search in xml data," in *EDBT*, 2008.

[13] D. Harel and R. E. Tarjan, "Fast algorithms for finding nearest common ancestors," *Society for Industrial and Applied Mathematics (SIAM)*, vol. 13, no. 2, pp. 338–355, 1984.

[14] B. Schieber and U. Vishkin, "On finding lowest common ancestors: simplification and parallelization," *Society for Industrial and Applied Mathematics (SIAM)*, vol. 17, no. 6, pp. 1253–1262, 1988.

[15] R. Cole and R. Hariharan, "Dynamic lca queries on trees," in *SODA*, 1999.

[16] M. A. Bender, M. Farach-Colton, et.al, "Lowest common ancestors in trees and directed acyclic graphs," *Journal of Algorithms*, vol. 57, pp. 75–94, 2005.

[17] J. Xu, J. Lu, W. Wang, and B. Shi, "Effective keyword search in xml documents based on miu," in *DASFAA*, 2006.

[18] Y. Xu and Y. Papakonstantinou, "Efficient lca based keyword search in xml data," in *CIKM*, 2007.

[19] "http://www.cs.washington.edu/research/xmldatasets/."

[20] "http://monetdb.cwi.nl/xml/."

[21] "http://lucene.apache.org/."

[22] "www.syger.com/jsc/docs/stopwords/english.htm."

[23] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. Pearson Education Limited, 1999.

[24] Z. Liu and Y. Chen, "Answering keyword queries on xml using materialized views," in *ICDE*, 2008.

[25] Y. Huang, Z. Liu, and Y. Chen, "Query biased snippet generation in xml search," in *SIGMOD*, 2008.

[26] P. Buneman, S. B. Davidson, et.al, "Keys for xml," in *WWW*, 2001.

[27] T. T. Chinenyanga and N. Kushmerick, "Expressive retrieval from xml documents," in *SIGIR*, 2001.

[28] D. Theodoratos and X. Wu, "An original semantics to keyword queries for xml using structural patterns," in *DASFAA*, 2007.

[29] N. Fuhr and K. Großjohann, "Xirql: A query language for information retrieval in xml documents," in *SIGIR*, 2001.

[30] S. Amer-Yahia, C. Botev, and J. Shanmugasundaram, "TexQuery: A fulltext search extension to XQuery," in *WWW*, 2004.

[31] E. Curtmola, S. Amer-Yahia, P. Brown, and M. Fernàndez, "Galatex: A conformant implementation of the xquery fulltext language," in *XIME-P*, 2005.

[32] M. Theobald, R. Schenkel, and G. Weikum, "An efficient and versatile query engine for topx search," in *VLDB*, 2005.

[33] Z. Bao, T. W. Ling, B. Chen, and J. Lu, "Effective xml keyword search with relevance oriented ranking," in *ICDE*, 2009.

[34] G. Li, B. C. Ooi, J. Feng, et.al, "Ease: Efficient and adaptive keyword search on unstructured, semi-structured and structured data," in *SIGMOD*, 2008.