The VLDB Journal (2010) 19:503–529
DOI 10.1007/s00778-009-0177-7

REGULAR PAPER

Structural consistency: enabling XML keyword search to eliminate spurious results consistently

Ki-Hoon Lee · Kyu-Young Whang · Wook-Shin Han · Min-Soo Kim

Received: 31 October 2008 / Revised: 18 December 2009 / Accepted: 21 December 2009 / Published online: 26 February 2010

Abstract XML keyword search is a user-friendly way to query XML data using only keywords. In XML keyword search, to achieve high precision without sacrificing recall, it is important to remove *spurious* results not intended by the user. Efforts to eliminate spurious results have enjoyed some success using the concepts of LCA or its variants, SLCA and MLCA. However, existing methods still could find many spurious results. The fundamental cause for the occurrence of spurious results is that the existing methods try to eliminate spurious results locally without global examination of all the query results and, accordingly, some spurious results are not consistently eliminated. In this paper, we propose a novel keyword search method that removes spurious results consistently by exploiting the new concept of structural consistency. We define *structural consistency* as a property that is preserved if there is no query result having an ancestor-descendant relationship *at the schema level* with any other query results. A naive solution to obtain structural consistency would be to compute all the LCAs (or variants) and then to remove spurious results according to structural consistency. Obviously, this approach would always be slower than existing LCA-based ones. To speed up structural consistency checking, we must be able to examine the query results at the schema level without generating all the LCAs. However, this is a challenging problem since the schema-level query results do not homomorphically map to the instance-level query results, causing serious false dismissal. We present a comprehensive and practical solution to this problem and formally prove that this solution preserves structural consistency at the schema level without incurring false dismissal. We also propose a relevance-feedback-based solution for the problem where our method has low recall, which occurs when it is not the user's intention to find more specific results. This solution has been prototyped in a full-fledged object-relational DBMS Odysseus developed at KAIST. Experimental results using real and synthetic data sets show that, compared with the state-of-the-art methods, our solution significantly (1) improves precision while providing comparable recall for most queries and (2) enhances the query performance by removing spurious results early.

Keywords XML · Keyword search · Spurious results · Structural consistency · Structural summary · Odysseus DBMS

K.-H. Lee · K.-Y. Whang (✉) · M.-S. Kim
Department of Computer Science,
Korea Advanced Institute of Science and Technology (KAIST),
Daejeon, South Korea
e-mail: kywhang@mozart.kaist.ac.kr

K.-H. Lee
e-mail: khlee@mozart.kaist.ac.kr

M.-S. Kim
e-mail: mskim@mozart.kaist.ac.kr

W.-S. Han
Department of Computer Engineering,
Kyungpook National University, Daegu, South Korea
e-mail: wshan@knu.ac.kr

1 Introduction

As XML becomes the standard for data representation and exchange on the Internet, querying XML data has become an important issue [27]. Research work in this area can be classified into two categories: the structured query approach and the keyword query approach [27]. Both approaches have tradeoffs. The structured query approach specifies the precise structure of the desired results using a structured query language such as XPath and XQuery. However, it is hard to formulate queries without prior knowledge about structured

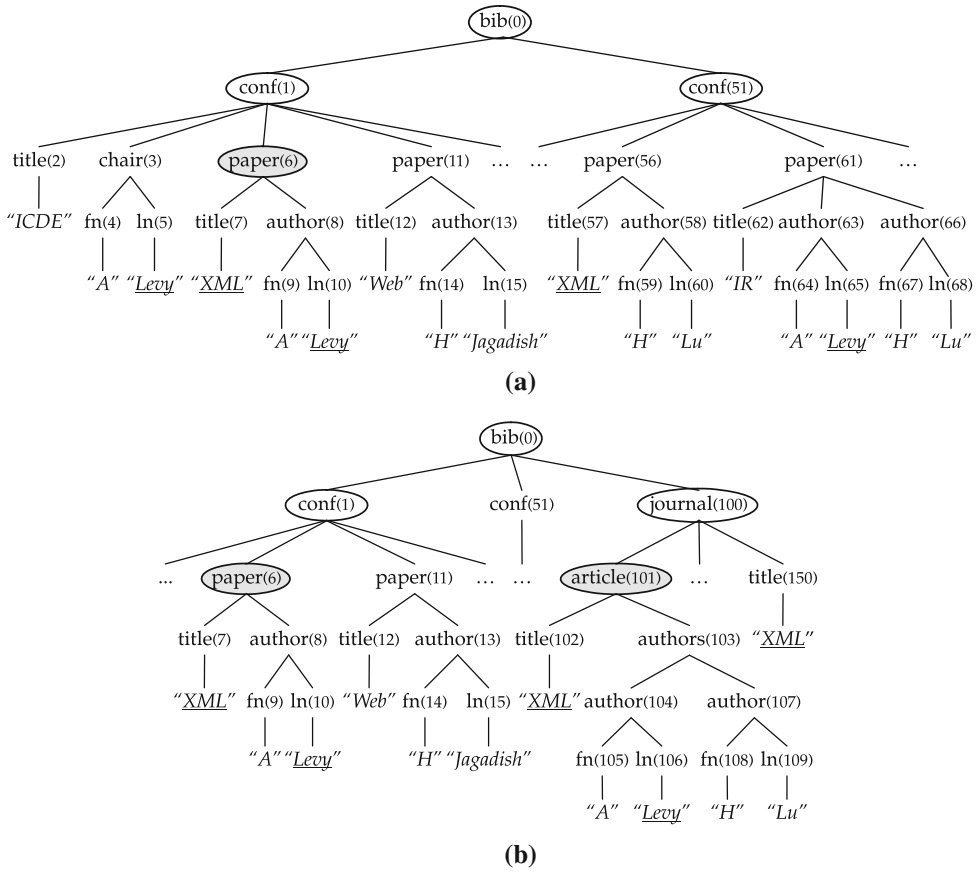


Fig. 1 Querying XML data. **a** XML data on conference publications. **b** XML data on conference and journal publications

query languages or without knowing the schema of the XML data. The keyword query, on the other hand, can overcome this problem by requiring only keywords rather than specific structure information. This approach, however, might not deliver precise results since it does not contain precise structures.

In the structured query, the user's query intention can be expressed as either a single structured query or multiple structured queries, depending on the heterogeneity of the underlying XML data. If there is only one structure matching the user's intention at the schema level, that intention can be expressed in a single structured query. However, if there are multiple structures matching the user's intention, multiple structured queries for those structures must be composed.

Example 1 The XML data in Fig. 1a represent bibliographic data on conference publications. Suppose that a user intends to find the publications of "Levy" on "XML". This query can be stated as a single structured query, Q_1; in the keyword query, it is represented as "XML Levy". The query result is {paper(6)}. Here, we denote the subtree rooted at node p as p in the same way as is done by Xu and Papakonstantinou [45].

Q_1: /bib/conf/paper["XML"]["Levy"][1]

Example 2 The XML data in Fig. 1b represent bibliographic data on conference and journal publications. Here, the subtree rooted at conf(1) is the same as in Fig. 1a. Since there are two structures matching the user's intention, one for conference papers and the other for journal articles, a union of multiple structured queries, Q_2, must be used to find the desired results despite the same query intention as in Example 1. Note that we still use the same keyword query as in Example 1. The query results are {paper(6), article(101)}.

Q_2: /bib/conf/paper["XML"]["Levy"] union
/bib/journal/article["XML"]["Levy"]

In the keyword search, a user wants to have high recall and high precision [5]. A naive way to achieve high recall (100%) in XML keyword search would be to return the root of an XML document. However, with this approach, the user

[1] For ease of exposition, we denote the predicate that checks whether a keyword w is contained in an element e as e["w"] instead of e[contains(., "w")] that uses the contains function in the XPath standard.

would suffer from very low precision due to a large amount of spurious results not intended by the user.

Efforts to eliminate spurious results [11,15,27,45] have enjoyed some success using the concepts of LCA or its variants, SLCA [45] and MLCA [27]. For a keyword query $Q = \{w_1, w_2, \ldots, w_m\}$, an LCA is the common ancestor node of nodes $n_1, n_2, \ldots, n_m$, where n_i is a node directly containing w_i ($1 \leq i \leq m$). It is located farthest from the root node. The SLCA method, a refinement of the LCA method, finds LCAs that do not contain other LCAs. For example, if we use the LCA method to find the results in Fig. 1a, {bib(0), conf(1), paper(6), conf(51)} are retrieved. With the SLCA method, {paper(6), conf(51)} are retrieved. As shown here, existing methods for XML keyword search still could find many *spurious* results (e.g., {bib(0), conf(1), conf(51)}), i.e., those that are not intended by the user. Here, following the common practice [11,25,27], we define *correct* results of a keyword query as those returned by structured queries (such as Q_1) corresponding to the keyword query, which are formulated according to the schema of the underlying XML data. In the real data set (DBLP), spurious results such as conf(51) can include huge subtrees having thousands of nodes. This serious problem of low precision in the-state-of-art methods not only overburdens the user with filtering numerous spurious results, but also degrades the performance of the system due to unnecessary computation. For instance, if we issue a keyword query "XML Levy" over the DBLP data set, we obtain 388,066 nodes using the SLCA method, among which only 69 nodes (precision $= \frac{69}{388{,}066} \approx 0.02\%$) are correct results.

The fundamental cause for the occurrence of spurious results is that the existing methods try to eliminate spurious results locally without global examination of all the query results. For instance, in Example 1, the LCA method finds a correct result {paper(6)}, but also finds spurious results {bib(0), conf(1), conf(51)}. With the SLCA method, we can eliminate two spurious results {bib(0), conf(1)} since they contain other LCAs. However, conf(51) still remains since it is not an ancestor of paper(6). This is *inconsistent* since both conf(1) and conf(51) are spurious results having an identical result structure. Here, we define the *result structure*[2] of a query result qr as a (schema-level) twig pattern composed of the label path [14] from the root of the XML data to the root qr_{root} of qr (simply, the *incoming label path*) and the ancestor-descendant edges from qr_{root} to query keywords. In the result structure of a query result qr, denoted by $rs(qr)$, the node corresponding to qr_{root} is marked as the *query result node* [34] and is distinguished from other nodes by placing it in a box. Figure 2 shows rs(conf(51)) and rs(paper(6)).

We observe that, if two query results have an ancestor-descendant relationship *at the schema level*, the ancestor is spurious. We call this phenomenon *structural anomaly*. Here, a query result qr_1 is an ancestor of a query result qr_2 at the schema level if and only if the incoming label path of $rs(qr_1)$ is a proper prefix of that of $rs(qr_2)$. By examining the query results at the schema level, we can remove spurious results having the same result structure consistently. For example, in Fig. 1a, the query results of the SLCA method are {paper(6), conf(51)}, and the incoming label path of rs(conf(51)) is a proper prefix of that of rs(paper(6)) as in Fig. 2. Hence, conf(51), which has the same result structure as conf(1), is spurious.

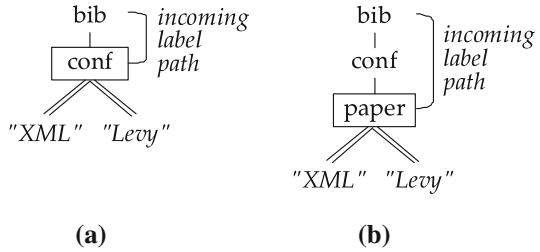


Fig. 2 The result structures of query results. **a** rs(conf(51)). **b** rs(paper(6))

We argue that, to improve precision, there should be no structural anomaly in the query results. We call this property *structural consistency* (to be defined more formally in Sect. 3.1). Otherwise, we are bound to retrieve inconsistent spurious results.

In this paper, we resolve structural anomalies by exploiting the notion of the smallest result structure. The *smallest result structure* is defined to be a result structure whose incoming label path is not a proper prefix of those of any other result structures. We then remove the query result whose structure is not the same as a smallest result structure, thereby obtaining structural consistency. For example, the smallest result structure of {paper(6), conf(51)} is rs(paper(6)) in Fig. 2b since the incoming label path of rs(paper(6)) is not a prefix of that of rs(conf(51)). Thus, conf(51) is removed.

A naive instance-level approach to obtain structural consistency would be to compute all the LCAs (or variants) and then to remove spurious results according to structural consistency. Obviously, this approach would always be slower than existing LCA-based ones. To speed up structural consistency checking, we must examine the query results at the schema level without generating all the LCAs.

The challenging issue here is "How do we formally guarantee that the schema-level approach produces the same query results as the instance-level approach does?" That is, if we blindly find SLCAs at the schema level and compute answers using the SLCAs, we may encounter a *false dismissal* problem (to be elaborated in more detail in Sect. 3.2.2). For example, an empty result can be obtained even though query results corresponding to smallest result structures

[2] Intuitively, the result structure is the schema of a query result (an instance).

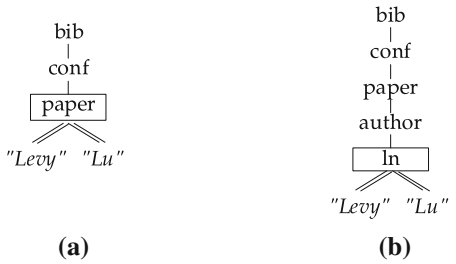


Fig. 3 An example of false dismissal. **a** The smallest structure. **b** The twig pattern obtained from the schema-level SLCA

exist as in Example 3. We may also encounter *phantom schema-level SLCAs* (to be defined in Sect. 3.2.2), which incurs structural anomaly. These problems occur because the schema-level SLCAs do not homomorphically map to the instance-level SLCAs. As a solution to these problems, we introduce the concept of *iterative kth-ancestor generalization*, which iteratively finds the kth ancestors of SLCAs at the schema level and removes phantom schema-level SLCAs. Through iterative kth-ancestor generalization, the schema-level definition of structural consistency becomes equivalent to the instance-level one, and we formally prove this equivalence in Theorem 1 of Sect. 3.2.4.

Example 3 Consider a keyword query $Q = \{\text{“Levy”}, \text{“Lu”}\}$ issued on the XML data in Fig. 1a. In the XML data in Fig. 1a, we see that there is a query result, paper(61), corresponding to the smallest result structure shown in Fig. 3a. However, there is no query result corresponding to the twig pattern shown in Fig. 3b that is obtained from the schema-level SLCA (we will formally define the schema-level SLCA in Sect. 3.2.1).

The contributions of this paper are as follows: (1) we formally propose new notions of structural consistency and structural anomaly, (2) we formally analyze the relationship between the set of schema-level SLCAs and the set of instance-level SLCAs, and then, propose an efficient algorithm that resolves structural anomaly at the schema level using the relationship analyzed (we call this algorithm *schema-level structural anomaly resolution*), (3) we formally prove in Theorem 1 that this algorithm preserves structural consistency as is originally defined at the instance-level without incurring false dismissal, (4) we propose a relevance-feedback base solution for the problem where our method has low recall, which occurs when it is not the user's intention to find more specific results, (5) we propose an efficient algorithm that simultaneously evaluates the multiple XPath queries generated by our method, (6) we have prototyped this algorithm in a full-fledged object-relational DBMS Odysseus [43], (7) we perform extensive experiments using real and synthetic data sets. The results show that we can significantly reduce spurious results compared with the existing methods by exploiting structural consistency. Furthermore, the experimental results show that our schema-level algorithm significantly improves the query performance over the existing ones.

The rest of this paper is organized as follows. Section 2 describes the XML data model, schema of XML data, query models, and quality measure of XML keyword search. Section 3 proposes the concept of structural consistency and schema-level structural anomaly resolution. Section 4 presents the implementation of schema-level structural anomaly resolution. Section 5 reviews existing work, and Sect. 6 presents the experimental results. Finally, Sect. 7 presents our conclusions.

2 Background

2.1 XML data model

We model XML data as a labeled tree [11,27,30,45] where a node represents an element, attribute, or value, and an edge represents the parent–child relationship between two nodes. Every element or attribute node has a *label* and a unique *id*, and each id is assigned a preorder number. A node that has a label l and an id i is denoted as $l(i)$. Definition 1 defines the label path of a node, and Definition 2 the node path.

Definition 1 [14] The *label path* of a node o is defined as a sequence of node labels $l_1, l_2, \ldots, l_m$ from the root to the node o, and is denoted as $l_1, l_2, \ldots, l_m$.

Definition 2 [34] The *node path* of a node o is defined as a sequence of node identifiers $n_1, n_2, \ldots, n_m$ from the root to the node o, and is denoted as $n_1, n_2, \ldots, n_m$. We denote the ith id of a node path $node\_path$ as $node\_path[i]$. We note that the ids $n_1, n_2, \ldots, n_m$ have an ascending order since each $n_i (1 \leq i \leq m)$ is assigned a preorder number.

2.2 Schema of XML data

Although DTD or XML Schema are used as the schema of XML data, XML data often do not have them [12]. For schemaless XML data, we can derive a schema from XML data using the DataGuide [14].[3] The DataGuide is a labeled tree that has every unique label path of XML data. In a DataGuide, a node represents the label of an element (or attribute), and an edge represents the parent–child relationship between two nodes. A node in a DataGuide is uniquely identified by its

[3] Recently, Bex et al. [7] have proposed algorithms for the inference of XML Schema Definitions, but we use the DataGuide since it takes linear time to create and has sufficient power for checking structural consistency. If a DTD or XML Schema are given along with XML data, we can exploit the given schema.

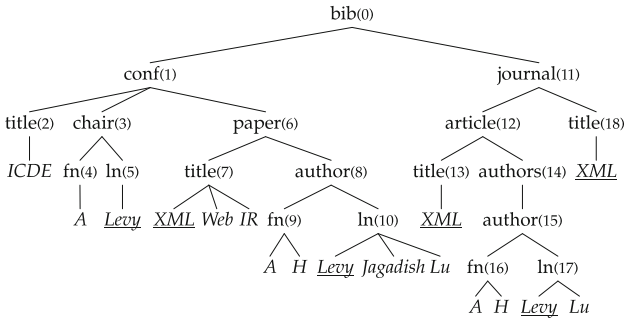


Fig. 4 An example DataGuide$^+$

label path. In this paper, we augment the DataGuide with keywords contained in value nodes to support keyword queries at the schema level. We call the augmented DataGuide *DataGuide*$^+$ and use it as the schema. Every non-value node in a DataGuide$^+$ is assigned a preorder number.[4] Hereafter, we call a node of the DataGuide$^+$ a *schema node* to distinguish it from a node of XML data, which we call an *instance node*. For ease of explanation, we may refer to a schema node by its label path.

Example 4 Figure 4 shows the DataGuide$^+$ for the XML data in Fig. 1b. Every unique label path of the XML data appears exactly once in the DataGuide$^+$. For example, in the XML data, the label path "bib.conf.paper.author" appears twice, and so does "bib.journal.article.authors.author". In contrast, in the DataGuide$^+$, each label path appears only once.

2.3 Query models

2.3.1 Keyword query

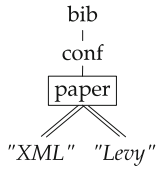
We model a keyword query as a set of keywords [30]. As in the literature [6,19–21,30,31,45], each query keyword may match (1) labels of elements or attributes or (2) keywords contained in value nodes of the XML data.

2.3.2 XPath query

We consider a subset of XPath that uses the child ("/") and descendant ("//") axes and predicates ("[]"). We model a query that belongs to this set as a twig pattern [10]. In the twig pattern a node, called a *query node* [10], represents a label (or a value), and an edge represents the parent–child or ancestor–descendant relationship between two nodes. One node of the twig pattern is marked as the *query result node* [34] and is distinguished from other nodes by placing it in a box. A query node that has more than one child node is called a *branching query node* [34]. A leaf node of the twig pattern is called a *leaf query node*.

[4] We can use other numbering schemes without loss of generality. For example, to handle schema evolution, we can use *Compact Dynamic Quaternary String* (*CDQS*) encoding [24], which allows for updates without the original nodes having to be renumbered.

Fig. 5 An example twig pattern



Example 5 Figure 5 shows an example twig pattern that represents the XPath query Q_1. In Fig. 5, paper is the query result node and, at the same time, the branching query node. Keywords are located in leaf query nodes "XML" and "Levy".

Q_1: /bib/conf/paper["XML"]["Levy"]

2.4 Quality metrics of XML keyword search

As quality metrics for keyword queries, we use precision and recall, which have been widely used in the field of information retrieval (IR). Formula (1) shows the definitions of precision and recall [5]. Here, R is the set of nodes relevant to the query (i.e., desired results) in the database, and A is the set of nodes retrieved as the answer to the query (i.e., actual query results). Precision is the fraction of the retrieved nodes (i.e., A) that are relevant, and recall is the fraction of the relevant nodes (i.e., R) that have been retrieved. The search quality is good when both precision and recall are close to 1.0 [5].

$$\text{precision} = \frac{|R \cap A|}{|A|}, \quad \text{recall} = \frac{|R \cap A|}{|R|} \tag{1}$$

3 Structural consistency

In this section, we formally define the notions of structural consistency and structural anomaly in XML keyword search. We also propose an efficient algorithm that resolves structural anomaly at the schema level.

3.1 The concept

We first define the *result structure* of a query result in Definition 3. Here, a *query result* is a subtree rooted at an SLCA in the XML data. We define *structural containment* and *structural equivalence* of result structures in Definition 4. We then define the *structural consistency* and the *structural anomaly* in Definition 5.

Definition 3 The *result structure* of a query result qr, denoted as $rs(qr)$, is a (schema-level) twig pattern composed of the label path from the root of XML data to the

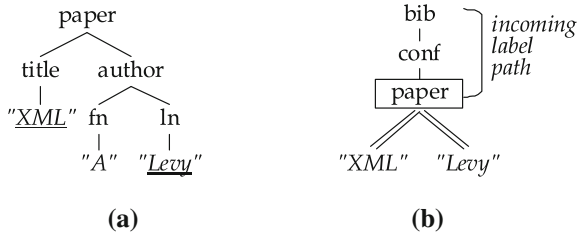


Fig. 6 The result structure of a query result paper(6). **a** A query result paper(6). **b** rs(paper(6))

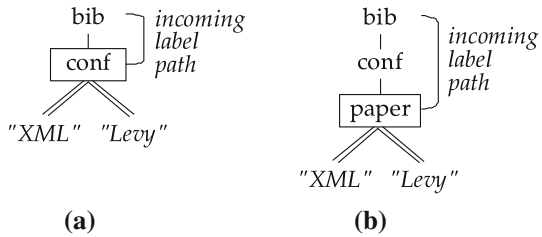


Fig. 7 The result structures of query results causing structural anomaly. **a** rs(conf(51)). **b** rs(paper(6))

root qr_{root} of qr (simply, the *incoming label path*) and the ancestor–descendant edges from qr_{root} to query keywords. In the result structure $rs(qr)$, the node corresponding to qr_{root} is marked as the query result node.

In Definition 3, we note that the incoming label path information is sufficient to define the structural consistency, but we attach query keywords to find query results corresponding to the result structure in query processing.

Example 6 Suppose that a keyword query $Q = \{$"XML", "Levy"$\}$ is issued on the XML data in Fig. 1a. Figure 6 shows a query result paper(6) and its result structure. Note that a query result is a subtree of XML data (i.e., an instance), and its result structure is a twig pattern (i.e., a part of schema).

Definition 4 Given a keyword query Q and the set of query results $QR = \{qr_1, qr_2, \ldots, qr_m\}$ of Q, the result structure $rs(qr_i)$ *structurally contains* the result structure $rs(qr_j)$, as denoted by $rs(qr_i) \prec rs(qr_j)$, if and only if the incoming label path of $rs(qr_i)$ is a proper prefix of that of $rs(qr_j)$. $rs(qr_i)$ and $rs(qr_j)$ are *structurally equivalent*, as denoted by $rs(qr_i) \equiv rs(qr_j)$, if and only if their incoming label paths are identical. We define $rs(qr_i) \preceq rs(qr_j)$ as $rs(qr_i) \prec rs(qr_j)$ or $rs(qr_i) \equiv rs(qr_j)$.

Definition 5 Given a keyword query Q and the set of query results $QR = \{qr_1, qr_2, \ldots, qr_m\}$ of Q, *structural consistency* is a property where the following condition is satisfied for QR: $(\forall qr_i \in QR)\,((\neg\exists qr_j \in QR)(rs(qr_i) \prec rs(qr_j)))$. *Structural anomaly* is a property where structural consistency is violated, i.e., $(\exists qr_i \exists qr_j \in QR)\,(rs(qr_i) \prec rs(qr_j))$.

Example 7 Suppose that a keyword query $Q = \{$"XML", "Levy"$\}$ is issued on the XML data in Fig. 1a, and that a set of query results $QR = \{$conf(51), paper(6)$\}$ is obtained. Figure 7 shows their result structures. We see that rs(conf(51)) $\prec rs$(paper(6)). Thus, QR has structural anomaly.

We resolve structural anomaly, thereby preserving structural consistency, by removing query results whose structure is not the same as a *smallest result structure* as defined in Definition 6. By enforcing structural consistency, we can remove spurious results having the same result structure consistently.

Algorithm 1 ***Naive Structural Anomaly Resolution***

Input: (1) a keyword query Q, (2) XML data D
Output: the set QR of query results of Q preserving structural consistency
Algorithm:
Step 1. Compute the set QR of SLCAs of Q on D
Step 2. Find the set SRS of smallest result structures of QR
  2.1 For each $qr_i \in QR$, obtain $rs(qr_i)$ and add it to SRS
  2.2 Remove all $srs_k \in SRS$ from SRS such that $(\exists srs_j \in SRS)(srs_k \prec srs_j)$
Step 3. Remove all $qr_i \in QR$ such that $(\neg\exists srs_j \in SRS)(rs(qr_i) \equiv srs_j)$
Step 4. Return QR

Fig. 8 A naive algorithm for resolving structural anomaly

Definition 6 Given a keyword query Q and the set of query results $QR = \{qr_1, qr_2, \ldots, qr_m\}$ of Q, *the set of smallest result structures* of QR is $\{rs(qr_i) | qr_i \in QR \wedge (\neg\exists qr_j \in QR)(rs(qr_i) \prec rs(qr_j))\}$

In Definition 6, "smallest" refers to the resulting subtrees since resulting subtrees are smaller if their incoming label paths are longer.

Lemma 1 *Given a keyword query Q, the set of query results $QR = \{qr_1, qr_2, \ldots, qr_m\}$ of Q, and the set of smallest result structures $SRS = \{srs_1, srs_2, \ldots, srs_n\}$ of QR, structural consistency holds for QR if the following condition is satisfied for QR : $(\forall qr_i \in QR)((\exists srs_j \in SRS)(rs(qr_i) \equiv srs_j))$.*

Proof It is straightforward from the definition of the smallest result structure. □

Figure 8 shows a naive algorithm that resolves structural anomaly at the instance level. The algorithm consists of the following four steps: (1) computing all the SLCAs, (2) finding smallest result structures of the SLCAs, (3) removing SLCAs whose result structures are not smallest result structures, and (4) returning the set of SLCAs preserving structural consistency.

3.2 Schema-level structural anomaly resolution

Obviously, the naive algorithm would always be slower than existing SLCA-based algorithms. We propose an efficient algorithm, called *schema-level structural anomaly resolution*, that resolves structural anomaly at the schema level. In this algorithm, we first find smallest result structures at the schema level. We then compute only those query results that correspond to the smallest result structures by evaluating structured queries constructed from the smallest result structures. We prove in Sect. 3.2.4 that we can find the smallest result structures using the schema without incurring false dismissal. To do that we first define the *schema-level SLCA* in Sect. 3.2.1. We then formally analyze the relationship between the set of schema-level SLCAs and the set of instance-level SLCAs in Sect. 3.2.2. Through analysis, we show that simple query evaluation using the schema-level SLCAs cannot obtain the same query results as the instance-level algorithm does. In Sect. 3.2.3, we present a solution for this problem, which we call *iterative kth-ancestor generalization*. In Sect. 3.2.4, we present a novel algorithm that resolves structural anomaly at the schema level using the schema-level SLCAs and iterative kth-ancestor generalization. We finally prove in Theorem 1 that the schema-level and the instance-level algorithms produce an equivalent set of query results that preserve structural consistency.

3.2.1 Schema-level SLCA

We first define the *schema-level LCA* in Definition 7 and then define the set of schema-level SLCAs in Definition 8. In contrast, we call SLCAs in the XML data *instance-level SLCAs*. Hereafter, *ancestor*(s_a, s) denotes that node s_a is an ancestor of node s, and *ancestor-or-self*(s_a, s) denotes that *ancestor*(s_a, s) or $s_a = s$.

Definition 7 Let G be a DataGuide$^+$ and S be the set of all schema nodes in G. For n schema nodes $s_1, s_2, \ldots, s_n \in S$, $s_a \in S$ is the *schema-level LCA* of these n schema nodes if and only if the following conditions are satisfied: (1) $(\forall 1 \leq i \leq n)$ (*ancestor-or-self* s_a, s_i, (2) $\neg\exists s_b \in S$ (*ancestors*$_a$, $s_b \wedge \forall 1 \leq i \leq n$ (*ancestor-or-self* (s_b, s_i))). The schema-level LCA s_a for $s_1, s_2, \ldots, s_n$ is denoted as $LCA(s_1, s_2, \ldots, s_n)$.

We note that, in Definition 7, the LCA is defined for n schema nodes; in Definition 8, the LCA_SET is defined for m sets of schema nodes. Given a keyword query $Q = \{w_1, w_2, \ldots, w_m\}$ and a DataGuide$^+$ G, $S_i (1 \leq i \leq m)$ denotes the set of schema nodes directly containing w_i in G.

Definition 8 Given a keyword query $Q = \{w_1, w_2, \ldots, w_m\}$ and the set S of all schema nodes in a DataGuide$^+$ G, *the set of schema-level SLCAs* $SLCA\_SET(S_1, S_2, \ldots, S_n) = \{s_a | (s_a \in LCA\_SET(S_1, S_2, \ldots, S_n)) \wedge (\neg\exists s_b \in LCA\_SET(S_1, S_2, \ldots, S_n))(ancestor(s_a, s_b))\}$ where $LCA\_SET(S_1, S_2, \ldots, S_m) = \{s_a | (s_a \in S) \wedge (\exists s_1 \in S_1, \exists s_2 \in S_2, \ldots, \exists s_m \in S_m)(s_a = LCA(s_1, s_2, \ldots, s_m))\}$.

Example 8 Suppose that a keyword query $Q = \{$"XML", "Levy"$\}$ is issued on the XML data in Fig. 1b. In the DataGuide$^+$ in Fig. 4, the set of schema-level LCAs is {"bib", "bib.conf", "bib.conf.paper", "bib.journal", "bib.journal.article"}, and the set of schema-level SLCAs is {"bib.conf.paper", "bib.journal.article"} since these schema nodes do not contain other schema-level LCAs.

3.2.2 The relationship between the set of schema-level SLCAs and the set of instance-level SLCAs

To explain the relationship between the set of schema-level SLCAs and the set of instance-level SLCAs, we first define the *schema structure* of a schema node in Definition 9. Since both the schema structure of a schema node and the result structure of a query result are defined as twig patterns, we will use the same notions of structural equivalence and structural containment for schema structures.

Definition 9 The *schema structure* of a schema node s, denoted as $ss(s)$, is a twig pattern composed of the incoming label path from the root of DataGuide$^+$ to s and the ancestor–descendant edges from s to query keywords. In the schema structure $ss(s)$, the node corresponding to s is marked as the query result node.

Given a keyword query, the set SS of schema structures of schema-level SLCAs is largely equivalent to the set SRS of smallest result structures of instance-level SLCAs. However, there exist cases where SS and SRS are not equivalent since the schema loses some instance-level information by storing only unique label paths of the instance nodes. For example, in the XML data in Fig. 1a, "Levy" and "Lu" appear in the instance nodes with the label path "bib.conf.paper. author.ln", but they appear in different instance nodes, ln(65) and ln(68). Nonetheless, in the DataGuide$^+$ in Fig. 4, they appear in the same schema node with the label path "bib.conf. paper.author.ln" since their label paths are the same. Thus, in effect, the schema loses the information that "Levy" and "Lu" appear in different instance nodes with the same label path.

There are two cases where SRS and SS are not equivalent: case 1 for some $ss_j \in SS$, there exists an $srs_i \in SRS$ such that $srs_i \prec ss_j$, and case 2 for some $ss_j \in SS$, there exists no $srs_i \in SRS$ such that $srs_i \preceq ss_j$. We note that $ss_j \prec srs_i$ does not hold according to the definition of the schema-level SLCA. In case 1, if we compute query results corresponding to ss_j, we will miss query results corresponding to srs_i, i.e., we will incur *false dismissal*. Example 9 shows an instance of false dismissal. In Sect. 3.2.3, we

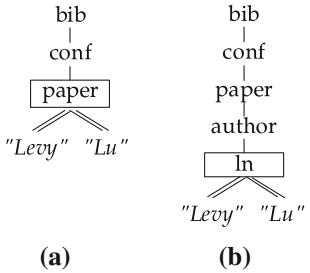


Fig. 9 An example of false dismissal. **a** srs_i. **b** ss_j

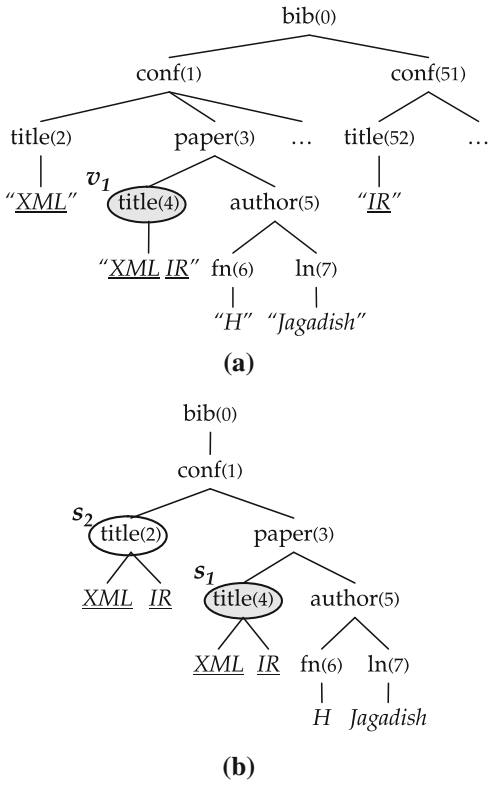


Fig. 10 An example of a phantom schema structure. **a** XML data. **b** The DataGuide$^+$ for the XML data in (**a**)

propose a solution to this problem, which we call *iterative kth-ancestor generalization*. In case 2, if we blindly apply iterative kth-ancestor generalization for ss_j, we could end up with incurring structural anomaly. We call $ss_j \in SS$ such that $(\neg\exists srs_i \in SRS)(srs_i \preceq ss_j)$ a *phantom schema structure*. Example 10 shows an example of the phantom schema structure. In the next section, we will provide a solution to eliminate phantom schema structures.

Example 9 Consider a keyword query $Q = \{\text{"Levy"}, \text{"Lu"}\}$ issued on the XML data in Fig. 1a. Figure 9a, b shows $srs_i \in SRS$ and $ss_j \in SS$, respectively. Here, $srs_i \prec ss_j$. In the XML data in Fig. 1a, we see that there is a query result corresponding to srs_i, paper(61), but there is no query result corresponding to ss_j.

Example 10 Suppose that a keyword query $Q = \{\text{"XML"}, \text{"IR"}\}$ is used. In the XML data in Fig. 10a, $SRS = \{rs(v_1)\}$. In the DataGuide$^+$ in Fig. 10b, $SS = \{ss(s_1), ss(s_2)\}$. Thus, we do not have an $srs\ rs(v_2)$ such that $rs(v_2) \preceq ss(s_2)$, and $ss(s_2)$ is a phantom schema structure. In this case, if we applied kth-ancestor generalization to s_2, we would find conf(1) in Fig. 10a as a result, which causes structural anomaly because $rs(\text{conf(1)}) \prec rs(v_1)$.

We now formally state the relationship between SRS and SS, which will be used in iterative kth-ancestor generalization.

Lemma 2 *Given a keyword query Q, for all $srs_i \in SRS$, there exists $ss_j \in SS$ such that $srs_i \preceq ss_j$.*

Proof See Appendix A. □

We can obtain $srs_i \in SRS$ by computing the set QR_j of the query results corresponding to $ss_j \in SS$. If QR_j is non-empty, then we have obtained $srs_i in SRS$ such that $srs_i \equiv ss_j$. If QR_j is empty, we can obtain $srs_i \in SRS$ such that $srs_i \prec ss_j$ by applying iterative kth-ancestor generalization.

3.2.3 Iterative kth-ancestor generalization

In this section, we present *iterative kth-ancestor generalization* to solve the problems of false dismissal and phantom schema structures. We iteratively find a kth-ancestor s_a of the schema-level SLCA s such that $ss(s_a) \equiv srs \in SRS$ where $srs \prec ss(s)$. We define the *kth-ancestor* in Definition 10.

Definition 10 Given two nodes, s_a and s, s_a is the *kth-ancestor* of s if s_a is an ancestor of s and $depth(s) = depth(s_a) + k$, where $depth(s)$ is the length of the path from the root to s.

Example 11 We can obtain $srs_i \in SRS$ in Fig. 9a by finding the second-ancestor of the schema-level SLCA in Fig. 9b.

Lemma 3 *Given a keyword query Q, suppose that $srs_i \in SRS$ structurally contains $ss(s) \in SS$, i.e., $srs_i \prec ss(s)$. Then, there must exist a kth-ancestor $s_a (1 \leq k \leq depth(s))$ of s such that $ss(s_a) \equiv srs_i \in SRS$.*

Proof See Appendix B. □

In iterative kth-ancestor generalization, we iteratively find the kth-ancestor s_a of the schema-level SLCA s from the parent of s (i.e., $k = 1$) until the set of the query results corresponding to $ss(s_a)$ is non-empty. Here, obtaining non-empty results indicates that $srs \in SRS$ has been found. Thus, we solve the false dismissal problem.

To eliminate phantom schema structures during iterative kth-ancestor generalization, we need to iteratively check

structural consistency. Initially, there is no structural anomaly for the set of schema-level SLCAs. As schema-level SLCAs are generalized, structural anomaly can be incurred by their ancestors in the schema. Then, computing query results corresponding to the kth-ancestor incurring structural anomaly in the schema will incur structural anomaly in the instances. For example, in Fig. 10b, the schema structure of the first-ancestor of s_2, $ss(\mathsf{conf(1)})$, structurally contains the schema structure $ss(s_1)$ of the schema-level SLCA s_1. In this case, if we compute query results corresponding to $ss(\mathsf{conf(1)})$, we obtain conf(1) in Fig. 10a. Here, $rs(\mathsf{conf(1)}) \prec rs(v_1)$ causing structural anomaly. Thus, we iteratively remove ancestors incurring structural anomaly and stop applying generalization for them. That is, we remove phantom schema structures.

We note that one $srs_i \in SRS$ can structurally contain multiple schema structures $ss(s_1), ss(s_2), \ldots, ss(s_n) \in SS$. In such cases, if we blindly generalize all the schema-level SLCAs $s_1, s_2, \ldots, s_n$, we obtain duplicate query results corresponding to srs_i. Thus, we must generalize only one schema-level SLCA for srs_i. This constraint is also enforced by iteratively checking structural consistency. Suppose that $s_1, s_2, \ldots, s_n$ are being generalized to srs_i in this order. It is clear that $s_j (1 \leq j \leq n-1)$ will be removed since s_j, when sufficiently generalized, must become the ancestor of s_n. Therefore, we can guarantee that only one schema-level SLCA, s_n, is generalized.

3.2.4 Putting it altogether

Figure 11 shows an enhanced algorithm that resolves structural anomaly at the schema-level using the schema-level SLCAs and iterative kth-ancestor generalization. This algorithm produces the same query results as the instance-level algorithm in Fig. 8 does. We will present the detailed query processing method of this algorithm in Sect. 4. Step 1 finds the set of schema-level SLCAs $S_{\text{unmarked}} = \{s_1, s_2, \ldots, s_m\}$, and Step 2 computes the set of the query results corresponding to $ss(s_i) (1 \leq i \leq m)$ by evaluating the XPath query that represents $ss(s_i)$. Here, we convert $ss(s_i)$ to an XPath query to make our method run on top of *any* query evaluation engine that supports XPath. Step 3 applies iterative kth-ancestor generalization for $s_i \in S_{\text{unmarked}}$. In Step 3.2.1.1, we check whether an $srs \in SRS$ such that $srs \equiv ss(s_i)$ has been found by examining whether QR_i is non-empty. If it has, in Step 3.2.1.1.1, we move such s_i to S_{marked}. If not, in Step 3.2.1.2.1, we obtain the parent of s_i using the parent(s_i) function. In Step 3.2.1.2.2.1, we remove s_i, which incurs structural anomaly, from S_{unmarked}.

Example 12 Suppose that a keyword query $Q = \{$"XML", "IR"$\}$ is used to query the XML data in Fig. 10a. In Step 1, $S_{\text{unmarked}} = \{s_1, s_2\}$. In Step 2, the set QR_1 of the query results corresponding to $ss(s_1)$ is non-empty ($\{\mathsf{title(4)}\}$), but

Algorithm 2 ***Schema-level Structural Anomaly Resolution***

Input: (1) a keyword query Q, (2) XML Data D,
(3) the DataGuide$^+$ G for D
Output: the set QR of query results of Q preserving structural consistency
Algorithm:
Step 1. Find the set of schema-level SLCAs $S_{unmarked}=\{s_1, s_2, ..., s_m\}$ of Q on G
Step 2. Compute the set QR_i of the query results in D corresponding to $ss(s_i)$ $(1 \leq i \leq m)$ using the query processing method in Section 4
Step 3. Apply iterative kth-ancestor generalization
3.1 $QR := \{\}$; $S_{marked} := \{\}$ /* initialize */
3.2 Repeat until $S_{unmarked} \neq \{\}$
3.2.1 For each $s_i \in S_{unmarked}$
/* check if an $srs \in SRS$ such that $srs \equiv ss(s_i)$ has been found */
3.2.1.1 If $QR_i \neq \{\}$ Then
/* an $srs \equiv ss(s_i)$ has been found */
3.2.1.1.1 Move s_i from $S_{unmarked}$ to S_{marked}
/* add the query results corresponding to srs to QR */
3.2.1.1.2 $QR := QR \cup QR_i$
3.2.1.2 Else /* $QR_i = \{\}$ */
/* generalize s_i */
3.2.1.2.1 $s_i :=$ parent(s_i)
/* check structural consistency */
3.2.1.2.2 If $(\exists s_k \in S_{marked})(ss(s_i) \prec ss(s_k)) \vee (\exists s_l \in S_{unmarked})(ss(s_i) \prec ss(s_l))$ Then
/* s_i incurs structural anomaly */
3.2.1.2.2.1 Remove s_i from $S_{unmarked}$
3.2.1.2.3 Else
3.2.1.2.3.1 Compute the set QR_i of the query results in D corresponding to $ss(s_i)$ using the query processing method in Section 4

Fig. 11 The algorithm for resolving structural anomaly at the schema-level

QR_2 for $ss(s_2)$ is empty. In Step 3.2.1.1, since $QR_1 \neq \{\}$, we move s_1 from S_{unmarked} to S_{marked} and add QR_1 to the set QR of query results. Hence, $S_{\text{unmarked}} = \{s_2\}$, $S_{\text{marked}} = \{s_1\}$, and $QR = \{\mathsf{title(4)}\}$. In Step 3.2.1.2, since $QR_2 = \{\}$, we generalize s_2. Now s_2 incurs structural anomaly since $(\exists s_1 \in S_{\text{marked}})(ss(s_2) \prec ss(s_1))$. Thus, we remove s_2 from S_{unmarked}. Now, $S_{\text{unmarked}} = \{\}$, and we end the iteration.

In Step 3, even if we process s_2 first, we can obtain the correct result without a problem. In Step 3.2.1.2.2, s_2 incurs structural anomaly since $(\exists s_1 \in S_{\text{unmarked}})(ss(s_2) \prec ss(s_1))$. Thus, we remove s_2 from S_{unmarked} obtaining $S_{\text{unmarked}} = \{s_1\}$ and $S_{\text{marked}} = \{\}$. Now we move s_1 from S_{unmarked} to S_{marked}, add QR_1 to QR, and end the iteration.

Theorem 1 *The Schema-level Structural Anomaly Resolution algorithm produces the same query results as the instance-level algorithm in Fig. 8 does.*

Proof By Lemma 2, for every $srs_i \in SRS$, there exists $ss(s_j) \in SS$ such that (1) $srs_i \equiv ss(s_j)$ or (2) $srs_i \prec ss(s_j)$. For case 1, we can obtain $srs_i \in SRS$ by computing the query results corresponding to $ss(s_j)$ (Step 2). For case 2, we can obtain $srs_i \in SRS$ by applying iterative kth-ancestor generalization according to Lemma 3 (Step 3). In this case, even if generalization is stopped for s_j because of incurring structural anomaly, we are still able to obtain $srs_i \in SRS$ since there always exists a schema-level SLCA s_n such that $ss(s_j) \prec ss(s_n)$—which is exactly what caused the structural anomaly—and we can find srs_i by generalizing s_n. Finally, $ss(s_j) \in SS$ such that $(\neg\exists srs_i \in SRS)(srs_i \preceq ss(s_j))$, i.e., the phantom schema structure, is always removed since the kth-ancestor s_a of s_j must eventually incur structural anomaly when s_j is generalized to the root node. Otherwise, we contradict the assumption $(\neg\exists srs_i \in SRS)(srs_i \preceq ss(s_j))$ since it must be that $srs_i \equiv ss(s_a)$ at the root node. □

We now analyze the complexity of our schema-level algorithm. Given a keyword query $Q = \{w_1, w_2, \ldots, w_n\}$, the worst case time complexity of the schema-level algorithm is $O(|S_1|d\sum_{i=2}^{n} log|S_i| + dC_{\text{XPath}})$ where $S_i (1 \leq i \leq n)$ is the set of schema nodes directly containing the query keyword w_i in the DataGuide$^+$, d the maximum depth of the XML data, and C_{XPath} the cost of XPath query evaluation, which will be presented in Sect. 4.2.2. Here, $O(|S_1|d\sum_{i=2}^{n} log|S_i|)$ [45] is the cost of computing schema-level SLCAs using the algorithm of Xu and Papakonstantinou [45], and $O(dC_{\text{XPath}})$ is the cost of iterative kth-ancestor generalization since, in the worst case, generalization can be applied until one of the schema-level SLCAs reaches the root node.

Compared with the existing instance-level SLCA algorithm [45], the schema-level algorithm is generally more efficient since it avoids unnecessary computation of spurious results by removing them early at the schema-level. The additional overheads of the schema-level algorithm are the computation of schema-level SLCAs and iterative kth-ancestor generalization. However, those overheads are small in practice. First, the cost of the schema-level SLCA computation tends to be very small since the schema is generally several orders of magnitude smaller than the XML data [4]. Second, the cost of iterative kth-ancestor generalization is negligible since the generalization occurs only occasionally and is usually applied only once or twice (according to our experiments in Sect. 6, the cost of iterative kth-ancestor generalization is less than 10% of the total query processing cost). In the worst case, however, our schema-level algorithm could be about twice slower than the instance-level SLCA algorithm. The reasons are as follows. First, when the schema is as large as the XML data, the overhead of schema-level SLCA computation would be almost the same as the cost of the instance-level SLCA computation. Second, after obtaining the schema-level SLCAs, we compute query results that correspond to the schema-level SLCAs by evaluating the XPath queries. This query evaluation could also be as expensive as the instance-level SLCA computation if there exist few spurious results since then our method loses the benefit over existing SLCA-based methods of avoiding unnecessary computation of spurious results through early removal (see the experimental results of QD_1 and QD_5 in Fig. 23c and QX_1 and QX_8 in Fig. 27c of Sect. 6).

3.3 A relevance-feedback-based solution for the low recall problem

When users intend to find more general results (although this is relatively rare), which we regard as spurious results, our method can have lower recall than existing methods. For example, suppose that a user intends to find a conference on "XML" where "Levy" is the chair. If there is at least one paper about "XML" authored by "Levy", our method does not retrieve the desired conference. We call this problem the *low recall problem*.

The fundamental cause for this problem is the inherent ambiguity in keyword search, i.e., the actual intention of the user is unknown. We can solve this problem by exploiting the user's relevance feedback. Relevance feedback is an important way of enhancing search quality using relevance information provided by the user [16,36]. The solution is as follows. The initial query results are presented to the user, and the user gives feedback if desired results are not retrieved[5] (this kind of relevance feedback can be easily implemented using a user-friendly GUI, and users just need to click a button). This feedback is sent to the system, and the system generalizes the smallest result structure and finds results again (we can repeat this feedback process until all the desired results are retrieved). For example, our method does not retrieve the desired conference if there is at least one paper about "XML" authored by "Levy". Since the desired result has not been retrieved, the user sends feedback to the system, and the system now finds conferences containing "XML" and "Levy" by generalizing the smallest result structure. Then, the user can obtain the desired result. When there are multiple smallest result structures, we can allow the user to choose which smallest result structure he wants to generalize. To do this, we need to group the query results for each smallest result structure and show each group to the user.

We implement this relevance-feedback-based solution by modifying Algorithm 2. In Step 3.2.1.1 of Algorithm 2, we check whether the set QR_i of the query results corresponding to a schema-level SLCA s_i is non-empty. If QR_i is empty,

[5] We assume that the user has enough knowledge about XML to understand the structure of the results. Thus, the user is able to give feedback on the structure of the results.

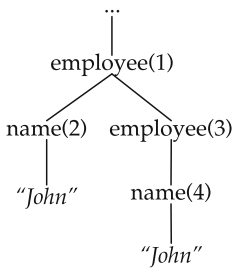


Fig. 12 XML data having a recursive schema

we generalize s_i in Step 3.2.1.2.1 by finding the parent of s_i. We implement relevance feedback by modifying Step 3.2.1.1 such that s_i should be generalized even if QR_i is non-empty when the user's relevance feedback is received.

The reason why relevance feedback is possible is that we process queries at the schema level. The schema-level processing makes the relevance-feedback mechanism feasible since users just need to give feedback on a small number of schema-level SLCAs. However, it is hard to apply to instance-level methods since the number of instance-level SLCAs is generally much larger than that of schema-level SLCAs.[6] Furthermore, it is not clear how we can receive the relevance feedback and generalize the results in the instance-level SLCA algorithm [45].

We can handle XML data having a recursive schema using the same technique. Figure 12 shows recursive XML data where the parent–child relationship between two employees represents the supervisor–supervisee relationship. Suppose that the query is "John employee" and the user intends to find all employees whose name is "John". In this case, our method (and also LCA and MLCA) finds only employee(3), resulting in low recall. We can also resolve this problem by generalizing the smallest result structure via relevance feedback.

The low recall problem may also be handled by ranking in a spirit similar to the work of Amer-Yahia et al. [3]. Enabling users to exploit partial knowledge of the schema in user queries [11,27,47] can also help us to disambiguate user's intention. We leave these issues for future work.

3.4 Search quality comparisons with earlier methods

In this section, we summarize search quality comparisons with earlier methods, SLCA [45], MLCA [27] (a variant of LCA), XSEarch [11], CVLCA [25], and XReal [6]. XSEarch and CVLCA are based on a heuristic called *interconnection* relationship. According to the heuristic, two nodes are considered to be semantically related if and only if there are no two distinct nodes with the same label on the path between these two nodes (excluding the two nodes themselves). Li et al. [27] have pointed out that the heuristic could retrieve spurious results and have shown that MLCA is generally superior to the heuristic. XReal infers the user's intention using the statistics of the underlying XML data.

Since keyword queries are inherently ambiguous, the desired results of a keyword query depend on the user's intention. The user may want to find (1) more specific results or (2) more general (as opposed to specific) results. For example, for a keyword query "XML Levy", the user may want to find either (1) papers about "XML" authored by "Levy" or (2) conferences on "XML" where "Levy" is the chair.

When the user's intention is to find more specific results, the precision values of our method are higher than or equal to those of existing methods since our method is able to eliminate more spurious results (i.e., general results) than existing methods by enforcing structural consistency. In addition, the recall values of our method and those of existing methods are the same since our method finds all the specific results, i.e., the query results that correspond to smallest result structures, as existing methods do.

Example 13 Suppose that a keyword query $Q = \{$"XML", "Levy", "Lu"$\}$ is issued on the XML data in Fig. 13. The user wants to find papers about "XML" authored by "Levy" and "Lu", and the desired result is paper(2). SLCA, XSEarch, and CVLCA find not only paper(2) but also spurious (i.e., general) results conf(10) and conf(17). MLCA can eliminate conf(10) since in the subtree rooted at conf(10), title(12) and title(15) are the nodes that contain "XML", and speaker(13) is the node that contains "Levy" and the LCA of title(15) and speaker(13), i.e., conf(10), contains the LCA of title(12) and speaker(13), i.e., keynote(11). XReal retrieves {conf(10), conf(17)} with the ranking since it infers conf as the desired node type[7] based on the XML document frequency [6]. Our method can eliminate all the spurious results by enforcing structural consistency. Thus, compared with LCA, MLCA, XSEarch, CVLCA, and XReal, our method improves precision without sacrificing recall.

When the user's intention is to find more general results, our method can have lower recall than existing methods, and we can solve this problem using relevance feedback.

[6] According to our experiments in Sect. 6, the average number of instance-level LCAs is two orders of magnitude larger than that of schema-level LCAs. Although it is infeasible in practice to give feedback on a large number of instance-level SLCAs, we have manually done experiments on the effect of relevance feedback in existing methods in Sect. 6.

[7] Since the highest confidence value (2.67) is significantly higher than the second highest value (1.41), XReal chooses the one with the highest confidence, conf, as the desired node type and retrieves only conf nodes. We compute the confidence values using Equation (6) in Bao et al. [6].

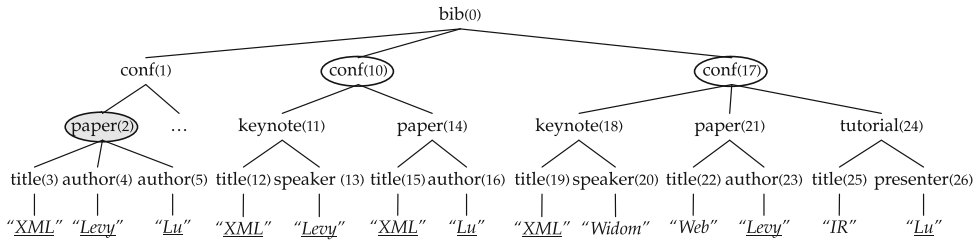


Fig. 13 The case where structural consistency shows high precision

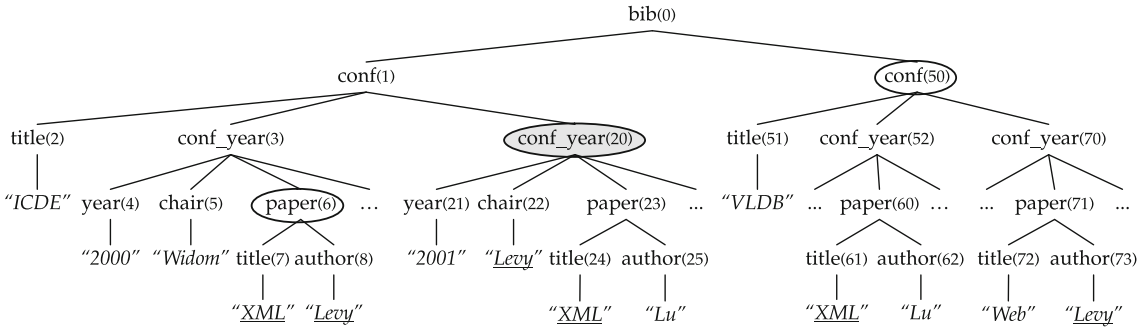


Fig. 14 The case where structural consistency shows low recall

The recall values of our method with relevance feedback are higher than or equal to those of existing methods since we can eventually obtain the desired results via generalization. In the worst case, however, the precision values of our method with relevance feedback could be lower than those of existing methods since it may find more spurious results during generalization as we see in Example 14. We note that the worst case is quite rare in practice.[8]

Example 14 Suppose that a keyword query $Q = \{$"XML", "Levy"$\}$ is issued on the XML data in Fig. 14 to find conferences on "XML" where "Levy" is the chair. The desired result is conf_year(20). SLCA and MLCA find {paper(6), conf_year(20), conf(50)}. XSEarch and CVLCA find {paper(6), conf_year(20)}. XReal finds {conf_year(3), conf_year(20)}. Here, paper(6), conf_ year(3), and conf(50) are spurious results. Our method initially finds only {paper(6)}, and thus, the recall of our method is 0. Using relevance feedback, our method obtains {conf_year(3), conf_year(20)} through generalization, and thus, the recall becomes 1.0. During generalization, our method finds a spurious result conf_year(3), but the precision value of our method is higher than those of SLCA and MLCA since the subtree rooted at conf(50) is much bigger than that of conf_year(3). However, if we remove the subtree rooted at conf(50) from the XML data (this is the worst case of our method), the precision value of our method can be lower than those of SLCA and MLCA (see Figs. 26a, 28a in Sect. 6.2). Compared with XSEarch and CVLCA, the precision value of our method is lower since our method finds conf_year(3). Compared with XReal, the precision value of our method is lower since our method finds paper(6).

[8] To find one, we had to test more than one hundred queries that are structurally similar to that shown in Example 14 against the NASA and XMark data sets in Sect. 6. We were not able to find a similar query in the DBLP data set since its structure is simpler than those of the NASA and XMark data sets.

4 Implementation

In this section, we describe the implementation details of the schema-level structural anomaly resolution. Section 4.1 presents the index structures used in the query processing. Section 4.2 presents the query processing method.

4.1 Index structures

To speed up query processing, we use indexes for the DataGuide$^+$ and XML data. We use an inverted index for a DataGuide$^+$, which we call the *schema index*, to efficiently compute the schema-level LCAs. We use an inverted index for XML data, which we call the *instance index*, to efficiently evaluate XPath queries. Inverted indexes have been used in many XML query processing methods [10,15,27,34]. We also use a table called *LabelPath* [34] to store all the label paths occurring in the DataGuide$^+$.

Table 1 summarizes the notation to be used for explaining the index structures. In Table 1, if a schema (or an instance) node s is a value node, we use $parent(s)$ instead of s as a

Table 1 Summary of notation

Symbols	Definitions
$snode\_id(s)$	The id of a schema node s
$label\_path(s)$	The label path of a schema (or an instance) node s
$label\_path\_id(s)$	The id of $label\_path(s) = snode\_id(s)$
$numeric\_label\_path(s)$	$label\_path(s)$ represented as a sequence of $snode_id$s rather than labels ($numeric\_label\_path(s)[i]$ denotes the ith id)
$inode\_id(o)$	The id of an instance node o
$node\_path(o)$	The node path of an instance node o

label_path_id	label_path
6	bib.conf.paper
7	bib.conf.paper.title
10	bib.conf.paper.author.ln
…	…
12	bib.journal.article
13	bib.journal.article.title
17	bib.journal.article.authors.author.ln
...	...

Fig. 15 An example LabelPath table

parameter for all functions since value nodes themselves do not have ids.

A LabelPath table consists of tuples of the form $\langle label\_path\_id, label\_path\rangle$, where $label\_path$ is the label path of a schema node s, and $label\_path\_id$ is the same as the id of s. A B$^+$-tree index is created on the $label\_path\_id$ column, and an inverted index on the $label\_path$ column.

Example 15 Figure 15 shows the LabelPath table for the DataGuide$^+$ in Fig. 4. In the DataGuide$^+$, the label path of the schema node having the id of 6 is "bib.conf.paper".

The schema index stores a list of postings for each unique value (or label) that appears in the DataGuide$^+$. The posting of a schema node s has the form $\langle snode\_id(s), numeric\_label\_path(s)\rangle$. $numeric\_label\_path(s)$ is used to find the ancestor nodes of s. Postings in a posting list are stored in ascending order of $snode\_id(s)$.

Example 16 Figure 16 shows the schema index for the DataGuide$^+$ in Fig. 4. Let s be the schema node with the value = "Jagadish" in Fig. 4. Then, $snode\_id(s) = 10$ and $numeric\_label\_path(s) = 0.1.6.8.10$. Thus, a posting $\langle 10, 0.1.6.8.10\rangle$ is stored in the posting list of "Jagadish".

The instance index stores a list of postings for each unique keyword (or label) that appears in XML data. The posting of an instance node o has the form $\langle inode\_id(o), node\_path(o), numeric\_label\_path(o)\rangle$. $node\_path(o)$ is used to find the ancestor nodes of o, and $numeric\_label\_path(o)$ is used to find the label path of o. Postings in a posting list are stored in ascending order of $inode\_id(o)$. We create a B$^+$-tree index, which is called a *subindex* [42,43], on each posting list of the instance index in the same way as was done by Guo et al. [15] and Whang et al. [42,43]. The key of a subindex is $inode\_id(o)$.

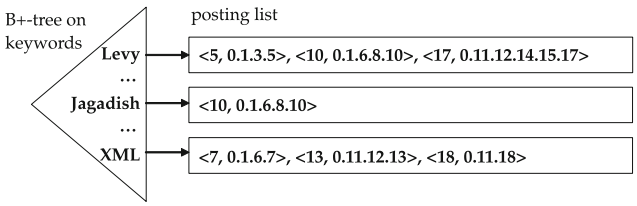


Fig. 16 An example schema index

Example 17 Figure 1b. Then, $inode\_id(o)$=15, $node\_path(o)$= 0.1.11.13.15, and $label\_path(o)$ = "bib.conf.paper.author.ln". Since $numeric\_label\_path(o) = 0.1.6.8.10$ for $label\_path(o)$ in the DataGuide$^+$ in Fig. 4, a posting $\langle 15, 0.1.11.13.15, 0.1.6.8.10\rangle$ is stored in the posting list of "Jagadish".

4.2 Query processing method

The query processing method consists of the following two steps. The first step presented in Sect. 4.2.1 translates a given keyword query Q into multiple XPath queries corresponding to the schema-level SLCAs. The second step presented in Sect. 4.2.2 evaluates the XPath queries obtained in the first step.

4.2.1 Query translation

We first compute schema-level SLCAs (or their ancestors) and then generate XPath queries specifying their schema structures. Figure 18 shows the algorithm *Query translation*, which consists of the following two steps.

In Step 1, we compute the set S of schema-level SLCAs using the GetSLCA function that implements the SLCA searching algorithm of Xu and Papakonstantinou [45]. They use this function to compute instance-level SLCAs, but we use it here to compute schema-level ones. For each schema-level SLCA $sslca_i$, we add the $snode\_id$ of $sslca_i$ to S. In iterative kth-ancestor generalization, the algorithm is modified to find ancestors of the schema-level SLCAs.

In Step 2, we generate an XPath query xpq_i for each schema-level SLCA with the $snode\_id$ $s_i \in S$. In the XPath query generated from s_i, s_i becomes the query result node and, at the same time, the branching query node since s_i is a schema-level SLCA of all the query keywords; query

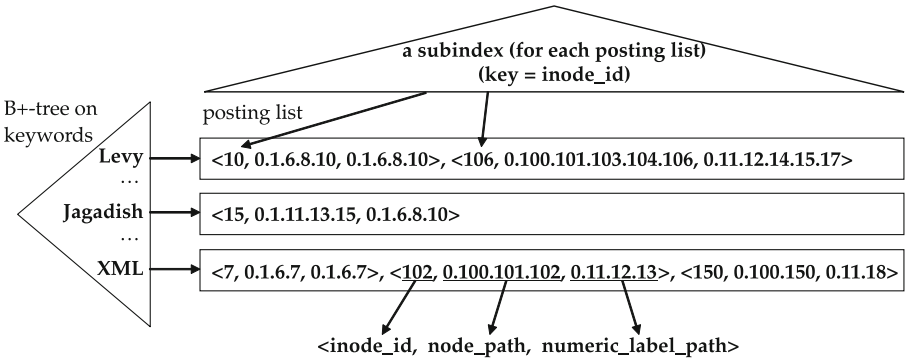


Fig. 17 An example instance index

Algorithm 3 ***Query Translation***

Input: (1) a keyword query $Q = \{w_1, \ldots, w_n\}$, (2) the schema index, (3) the LabelPath table
Output: the set *XPQ* of XPath queries
Algorithm:
Step 1. Compute a set S of schema-level SLCAs
  1.1 $S := \{\}$ /* initialize */
  1.2 Obtain posting lists $L_1, \ldots, L_n$ of $w_1, \ldots, w_n$ from the schema index
  1.3 $T :=$ GetSLCA$(L_1, \ldots, L_n)$
  1.4 For each schema-level SLCA $sslca_i \in T$
    1.4.1 Add $snode\_id(sslca_i)$ to S
Step 2. Generate the set *XPQ* of XPath queries
  2.1 $XPQ := \{\}$ /* initialize */
  2.2 For each schema-level SLCA $s_i \in S$
    2.2.1 Obtain the label path lp_i of s_i from the LabelPath table
    2.2.2 $xpq_i :=$ MakeXPathQueryString$(lp_i, w_1, \ldots, w_n)$
    2.2.3 $XPQ := XPQ \cup \{xpq_i\}$
  2.3 Return *XPQ*
Function ***MakeXPathQuerySting***
Input: (1) a label path lp, (2) query keywords $w_1, \ldots, w_n$
Output: an XPath query *xpq*
Step 1. Convert "." in lp into "/"
Step 2. For each query keyword w_j $(1 \le j \le n)$, create a predicate $expr_j$
  2.1 If w_j is a label and is not the last label of lp, $expr_j := w_j$
  2.2 If w_j is a value, $expr_j :=$ contains(., "w_j")
Step 3. $xpq := lp[expr_1] \ldots [expr_n]$
Step 4. Return *xpq*

Fig. 18 The query translation algorithm

keywords that are descendants of s_i become the leaf query nodes. Here, we first obtain the label path lp_i of s_i by searching the LabelPath table using $snode\_id(s_i)$. We then make the query string of xpq_i by calling the MakeXPathQueryString function with lp_i and the query keywords. In Step 2.1 of the MakeXPathQueryString function, we do not create a predicate when w_i is the last label of lp. It means that w_i is the label of the schema-level SLCA. Since it is a part of lp already, a predicate for it is not needed.

Example 18 We translate a keyword query "XML Levy" on the XML data in Fig. 1b into XPath queries xpq_1 and

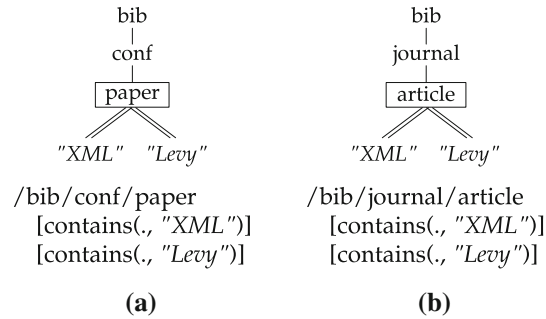


Fig. 19 (The XPath queries generated from "XML Levy". **a** xpq_1. **b** xpq_2

xpq_2 in Fig. 19 as follows. In Step 1, we first obtain the posting lists L_1, L_2 of "XML", "Levy" by searching the schema index in Fig. 16. We then compute the set T of $numeric\_label\_path$'s of schema-level SLCAs for L_1 and L_2 by evaluating GetSLCA (L_1, L_2). Here, $T = \{$"0.1.6", "0.11.12"$\}$. For each $sslca_i \in T$, we add $snode\_id(sslca_i)$ to S. Thus, $S = \{6, 12\}$ in Fig. 4. In Step 2, for the schema-level SLCA with the $snode\_id$ $s_1 = 6 \in S$, we first obtain the label path "bib.conf.paper" of s_1 from the LabelPath table in Fig. 15. We note that the $label\_path\_id = s_1 = 6$. We then create predicates for "XML" and "Levy". The predicates are "[contains(., "XML")]" and "[contains(., "Levy")]". Finally, we generate the XPath query xpq_1 by concatenating the label path and the predicates. We similarly generate the XPath query xpq_2 for the schema-level SLCA with the $snode\_id$ $s_2 = 12$.

4.2.2 Query evaluation

The set of XPath queries obtained in the query translation step can be evaluated with *any* existing XPath engine. In this section, we propose an efficient algorithm that simultaneously evaluates the specific set of XPath queries generated by our method.

In general, there are multiple structures matching the user's query intention, and thus, multiple XPath queries for those structures are generated from a keyword query. The

result of the keyword query is the union of the results of these XPath queries. As explained in Sect. 4.2.1, an XPath query xpq_i generated from a schema-level SLCA s_i has one branching node, i.e., s_i, and the label path of s_i is the path from the root node to s_i. Query keywords that are descendants of s_i become the leaf query nodes of xpq_i. The query xpq_i finds the instance nodes that have the label path of s_i and that contain all the query keywords (this is common to all xpq_i's). We exploit this commonality for efficient simultaneous computation of multiple queries.

There has been a lot of work on XPath evaluation, but most of the work focuses on answering one query at a time. Some research efforts [9,29,48] have been done on answering multiple queries simultaneously, but they are not optimized for the specific set of XPath queries that are generated by our method. Bruno et al. [9] and Zhang et al. [48] only handle linear XPath queries. Liu et al. [29] handle XPath queries with branches. This method is not suitable for the specific set of XPath queries because of the following reasons. They combine multiple queries into a single structure, called *supertwig query*, to exploit query commonalities. They only consider the scenario where query commonalities exist in the top parts—the parts close to the root node—of multiple original queries. However, in the specific set of XPath queries, much of the query commonalities exist in the bottom parts of the original queries, which consist of query keywords. Little query commonalities exist in the top parts since each query has a unique path from the root node to the branching node. Thus, in the worst case, the cost of the method is almost the same as that of processing one query at a time. In contrast, our algorithm simultaneously evaluates all the queries in this specific set by exploiting the query commonalities existing in the bottom parts of the original queries.

Since the queries in this specific set share the same query keywords that appear in the original keyword query, we can simultaneously evaluate all the queries by joining the posting lists of the query keywords. We obtain the posting lists from the instance index introduced in Sect. 4.1. Suppose that XPath queries $xpq_1, xpq_2, \ldots, xpq_m$ are obtained from a keyword query $Q = \{w_1, w_2, \ldots, w_n\}$. We perform an index nested-loop join over the posting lists $L_j (1 \leq j \leq n)$ of query keywords w_j. For each posting in the outer-most posting list L_1, we identify the query to be evaluated from among $xpq_i (1 \leq i \leq m)$. Thus, we simultaneously evaluate different queries while we are scanning L_1. As explained in Sect. 4.1, the posting of an instance node o has the form $\langle inode\_id(o), node\_path(o), numeric\_label\_path(o)\rangle$ where $inode\_id(o)$ is the node id of o, $node\_path(o)$ the node path of o, and $numeric\_label\_path(o)$ the label path of o that is represented as a sequence of integer ids rather than labels. $node\_path(o)$ contains the ids of the ancestor nodes of o in the ascending order, and its last id is $inode\_id(o)$. A posting list is sorted in the ascending order of $inode\_id(o)$. Hereafter, we refer to an instance node o by its posting for ease of exposition. For each posting o_{1a} in L_1, we find the query to be evaluated using $numeric\_label\_path(o_{1a})$. For $xpq_i (1 \leq i \leq m)$, if the path p_i from the root node to the branching node of xpq_i is a prefix of the label path of o_{1a}, xpq_i must be the query that we need to evaluate for o_{1a} since xpq_i finds the instance nodes that have the label path p_i and that contain all the query keywords. Here, o_{1a} matches the query keyword w_1 since o_{1a} is a posting of w_1. We note that at most one xpq_i is found since each query has a unique branching node. We compute the results only for the postings in L_1 that have the corresponding XPath query to be evaluated. Thus, we avoid unnecessary computation of spurious results. We note that, in contrast, the SLCA algorithm [45] computes SLCAs for all postings in L_1 incurring unnecessary computation.

We now explain how we evaluate xpq_i. Let d_i be the depth of the branching node of xpq_i from the root node, and $node\_path(o_{1a})[d_i]$ be the d_ith id of $node\_path(o_{1a})$. We need to check if the instance node o with the id $node\_path(o_{1a})[d_i]$ contains all the query keywords $w_j (1 \leq j \leq n)$. Here, o corresponds to the query result since the branching node is the query result node in xpq_i. o clearly contains w_1 since o is an ancestor of o_{1a}. o contains $w_j (2 \leq j \leq n)$ if there exists $o_{jb} \in L_j$ for each L_j such that $node\_path(o_{jb})$ and $node\_path(o_{1a})$ have the same prefix from the root node to d_i. Since we assign a unique preorder id to each node in the XML data tree, $node\_path(o_{jb})$ and $node\_path(o_{1a})$ have the same prefix from the root node to d_i if $node\_path(o_{jb})[d_i] = node\_path(o_{1a})[d_i]$. Let k be $node\_path(o_{1a})[d_i]$, which is $inode\_id(o)$. To check the existence of $o_{jb} \in L_j$ such that $node\_path(o_{jb})[d_i] = k$, we utilize the subindex on L_j whose key is $inode\_id$ of the posting in L_j, exploiting Lemmas 4 and 5. Here, we do not need to find all $o_{jb} \in L_j$ such that $node\_path(o_{jb})[d_i] = k$ since we only need to check if o—which corresponds to the query result—contains w_j. By Lemmas 4 and 5, to check the existence of $o_{jb} \in L_j$ such that $node\_path(o_{jb})[d_i] = k$, we only need to find a posting o_{jb} such that $inode\_id(o_{jb})$ is the smallest id that is greater than or equal to k in L_j and check whether $node\_path(o_{jb})[d_i] = k$. In summary, we simultaneously evaluate all the queries $xpq_i (1 \leq i \leq m)$ through one scan of L_1 and an index nested-loop join over the posting lists $L_j (1 \leq j \leq n)$.

Lemma 4 *$inode\_id(o_{jb}) \geq k$ if $node\_path(o_{jb})[d_i] = k$.*

Proof It is straightforward since we assign a preorder id to each node. □

Lemma 5 *Let $inode\_id(o_{jb})$ be the smallest id that is greater than or equal to k in L_j. If $node\_path(o_{jb})[d_i] \neq k$, then there is no $o_{jb'} \in L_j$ such that $node\_path(o_{jb'})[d_i] = k$.*

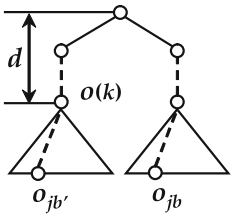


Fig. 20 An example of XML data tree for the proof of Lemma 5

Proof Suppose that there exists $o_{jb'} \in L_j$ such that $node\_path(o_{jb'})[d_i] = k$. Then, as we see in Fig. 20, $o_{jb'}$ must be in the subtree rooted at $o(k)$, and o_{jb} must be in the right subtree of $o(k)$. Thus, $inode\_id(o_{jb}) > inode\_id(o_{jb'}) \geq k$. This contradicts the assumption that $inode\_id(o_{jb})$ is the smallest id that is greater than or equal to k in L_j. □

Our algorithm uses the idea of XIR [34] that exploits the schema information—more precisely, the label path—for XPath query processing. XIR decomposes a given XPath query into linear XPath queries. A *linear XPath query*, which is also known as a *linear path expression* [34], is an XPath query without branches. It then finds a set of result node paths by processing each linear XPath query, and performs prefix match join between the sets of result node paths. Here, the *prefix match join* [34] identifies the prefix (a subpath from the root to the branching node) of a node path on one side and finds the matching node paths having the same prefix on the other side of the join. In contrast to XIR, our algorithm simultaneously evaluates multiple XPath queries using the instance index without computing the result node paths a priori for each linear XPath query. In this sense, our algorithm is completely different from XIR.

Figure 21 shows the query evaluation algorithm, which consists of the following two steps.

In Step 1, we obtain necessary information for query evaluation from the XPath queries. For each XPath query $xpq_i (1 \leq i \leq m)$, we first obtain the depth d_i of the branching node from the root node (simply, the *branching depth*). We then obtain the id $label\_path\_id_i$ of the label path from the root node to the branching node using the LabelPath table.

In Step 2, we compute the results of the XPath queries. We first obtain the posting lists of the query keywords. We then scan the outer-most posting list L_1 and perform an index nested-loop join over the posting lists $L_j (1 \leq j \leq n)$. For each posting $o_{1a} \in L_1$, we find the query xpq_i to be evaluated in Step 2.3.1. If found, we do the inner loop step to check whether the node with the id $node\_path(o_{1a})[d_i]$ contains all the query keywords in Step 2.3.2.1. For each posting list $L_j (2 \leq j \leq n)$, we check the existence of $o_{jb} \in L_j$ such that $node\_path(o_{jb})[d_i] = node\_path(o_{1a})[d_i]$, by calling the FindMatchingPosting function in Step 2.3.2.1.1. The FindMatchingPosting function finds such a posting using the subindex created on the posting list L_j based on Lemmas 4 and 5. If a posting is found for every posting list $L_j (2 \leq j \leq n)$, we return $node\_path(o_{1a})[d_i]$ as the result of xpq_i.

Algorithm 4 ***Query Evaluation***

Input: (1) a set of XPath queries $\{xpq_1, \ldots, xpq_m\}$ having the same query keywords $w_1, \ldots, w_n$,
(2) the LabelPath table, (3) the instance index
Output: the results of the XPath queries
Algorithm:
Step 1. For each XPath query xpq_i $(1 \leq i \leq m)$
1.1 d_i := the depth of the branching node from the root node
1.2 $label\_path\_id_i$:= the id of the label path from the root node to the branching node
Step 2. Perform an index nested-loop join
2.1 R := { } /* initialize */
2.2 Obtain the posting lists $L_1, \ldots, L_n$ of $w_1, \ldots, w_n$ from the instance index
/* outer loop */
2.3 For each posting $o_{1a} \in L_1$
/* find xpq_i to be evaluated */
2.3.1 For i = 1 to m, find xpq_i such that
$numeric\_label\_path(o_{1a})[d_i] = label\_path\_id_i$
/* Note that at most one xpq_i is found since each query has a unique branching node */
2.3.2 If xpq_i is found
/* inner loop */
2.3.2.1 For each posting list L_j $(2 \leq j \leq n)$
2.3.2.1.1 Check the existence of a posting $o_{jb} \in L_j$ such that $node\_path(o_{jb})[d_i] = node\_path(o_{1a})[d_i]$ by calling the function *FindMatchingPosting*
2.3.2.2 If a posting is found for every posting list L_j $(2 \leq j \leq n)$
2.3.2.2.1 Add $node\_path(o_{1a})[d_i]$ to R
2.4 Return R
Function ***FindMatchingPosting***
Input: (1) d_i, (2) $node\_path(o_{1a})[d_i]$, (3) L_j
Output: a posting o_{jb}
Step 1. $k := node\_path(o_{1a})[d_i]$
/* Check the existence of a posting $o_{jb} \in L_j$ such that $node\_path(o_{jb})[d_i] = k$ using the subindex and exploiting Lemmas 4 and 5 */
Step 2. Find a posting $o_{jb} \in L_j$ such that $inode\_id(o_{jb})$ is the smallest id that is greater than or equal to k using the subindex created on L_j
Step 3. If $node\_path(o_{jb})[d_i] = k$, return o_{jb}

Fig. 21 The query evaluation algorithm

Given a set of XPath queries $\{xpq_1, xpq_2, \ldots, xpq_m\}$ having the same query keywords $\{w_1, w_2, \ldots w_n\}$, the worst case time complexity C_{XPath} of the query evaluation algorithm is $O(|L_1|(m + \sum_{j=2}^{n} log|L_j|))$ where L_j $(1 \leq j \leq n)$ is the posting list of w_j. For each posting in L_1, we find the query to be evaluated from among the m queries and one posting from each of the other $n - 1$ posting lists. Finding a posting in L_j using the subindex costs $O(log|L_j|)$.

We now compare the performance of our algorithmWe now compare the performance of our algorithm with that of the

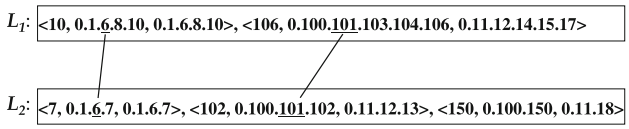


Fig. 22 An example of Algorithm 4

instance-level SLCA algorithm [45]. The worst case complexity of the SLCA algorithm is $O(|L_1|d\sum_{j=2}^{n} log|L_j|)$ [45] where d is the maximum depth of the XML data. In practice, d of the SLCA algorithm and m of our algorithm are small and do not affect performance significantly. Thus, the "worst case" performance of the two algorithms is almost the same. The critical benefit of our algorithm over the SLCA algorithm is that we avoid unnecessary computation of spurious results by only computing the results of the XPath queries obtained from schema-level SLCAs. This effect comes from the fact that we compute the results only for the postings in L_1 that have the corresponding XPath query to be evaluated (in Step 2.3.2) while the SLCA algorithm computes SLCAs for all postings in L_1.

Example 19 We evaluate the XPath queries xpq_1 and xpq_2 in Fig. 19 as follows. In Step 1, the branching depth $d_i = 3$ for $xpq_i (i = 1, 2)$. Since, in the LabelPath table in Fig. 15, the id of the label path "bib.conf.paper" is 6 and that of "bib.journal.article" is 12, $label\_path\_id_1 = 6$ and $label\_path\_id_2 = 12$. In Step 2, we first obtain the posting lists L_1, L_2 of the query keywords "Levy", "XML" as shown in Fig. 22. For the posting $\langle inode\_id(o_{1a}),\ node\_path(o_{1a}),\ numeric\_label\_path(o_{1a})\rangle = \langle 10,\ 0.1.6.8.10,\ 0.1.6.8.10\rangle \in L_1$, $numeric\_label\_path(o_{1a})[d_1] = label\_path\_id_1$, or equivalently, "0.1.6.8.10"[3]=6. That is, "bib.conf.paper" of xpq_1 is a prefix of the label path "bib.conf. paper.author.ln" that corresponds to $numeric\_label\_path(o_{1a})$. Thus, xpq_1 is the query to be evaluated, and we do the inner loop step. We find a posting in L_2 such that $node\_path(o_{2b})[d_1] = node\_path(o_{1a})[d_1]$="0.1.6.8.10"[3]=6 using the subindex created on L_2. Since there is a posting $\langle 7,\ 0.1.6.7,\ 0.1.6.7\rangle \in L_2$ such that "0.1.6.7"[3]=6, we return 6, which is the node id of paper(6) in Fig. 1b, as the result of xpq_1. For the posting $\langle 106,\ 0.100.101.103.104.106,\ 0.11.12.14.15.17\rangle \in L_1$, we can similarly find the result article(101) of xpq_2.

5 Related work

There has been a lot of work on keyword search in relational databases [1,8,17,18,28,32], which inspired XML keyword search. However, the work on relational databases is not directly applicable to XML since the schema of XML data cannot always be mapped to a rigid relational schema [15] due to the semi-structured and heterogeneous nature of XML. Our approach provides novel notions and algorithms that are suitable for the semi-structured and heterogeneous nature of XML and eliminates spurious results by exploiting the hierarchical nature of XML.

Extensive research has been done on XML keyword search. Under the assumption that smaller subtrees are more relevant to the query, most of the existing methods find the smallest subtrees containing all the query keywords based on the concepts of the LCA or its variants. Schmidt et al. [37] have introduced the notion of the LCA, and Guo et al. [15] have defined a subset of LCAs and proposed an efficient ranking method for the subtrees rooted at the nodes in this set. Xu and Papakonstantinou [46] have studied the properties of LCAs to accelerate the computation. Hristidis et al. [19] have focused on computing the whole subtrees rooted at LCAs. Xu and Papakonstantinou [45] have proposed the concept of the SLCA and presented algorithms for finding SLCAs efficiently. Sun et al. [38] have proposed a method that processes keyword queries involving boolean operators AND and OR under the SLCA semantics. Li et al. [27] have proposed the concept of *Meaningful LCA* (*MLCA*), a concept similar to that of SLCA, and incorporated MLCA search in XQuery. Cohen et al. [11] have attempted to find meaningful results based on a heuristic called *interconnection* relationship, and Li et al. [25] have presented an efficient algorithm for the heuristic.

Liu and Chen [30] have pioneered a novel method for inferring *return nodes* for XML keyword search. They have proposed a system called *XSeek*, which infers desirable return nodes by recognizing entities in the XML data. Huang et al. [21] have addressed the important problem of generating effective snippets (i.e., summaries) for XML search results. Liu and Chen [31] have proposed properties to find relevant nodes that match query keywords in the subtree rooted at each SLCA. These schemes on generating return nodes are orthogonal to and can be incorporated into our method as we see in Sect. 6.

Several research efforts [11,27,47] have been made to enable users to exploit partial knowledge of the schema in user queries. The query models used in those methods are commonly called *labeled keyword search* [47], which allows the user to annotate query keywords with labels. For example, in labeled keyword search, "XML Levy" is expressed as "title:XML author:Levy". Using this partial schema information, labeled keyword search can retrieve more meaningful results than simple keyword search that specifies only keywords. The search quality of labeled keyword search relies on the correctness of the labels in a given query [27]. However, a casual user is unlikely to have perfect knowledge of those labels [27]. Our method does not have this problem since it uses the simple keyword search model.

Yu and Jagadish [47] have proposed novel schema-based matching methods for *labeled keyword search* and

Meaningful Summary Query (schema-aware query). They contrast with our framework that supports schema-free keyword search. They use the schema of XML data to define the matching semantics. In contrast, our method uses the schema to efficiently resolve structural anomaly instead.

Most recently, Bao et al. [6] have proposed a probabilistic framework for inferring user's intention and ranking the query results. They compute the confidence level of each candidate *node type*, which is defined as a label path, using the statistics of the underlying XML data and use it to infer the user's intention. The method of Bao et al. processes queries at the instance level and additionally uses the schema to improve search quality. In contrast, our method, being primarily at the schema level, improves not only search quality using the schema but also search performance by processing queries at the schema level.

Besides, there has been extensive work done by W3C to define a full-text extension of XQuery [40], which has today many implementations such as GalaTex [13]. Amer-Yahia et al. [2] have presented efficient evaluation algorithms for full-text XQuery queries, and Pradhan [35] has demonstrated several optimization techniques. In this paper, our focus is to effectively and efficiently support "schema-free" XML keyword search where users only need to specify keywords as opposed to the full-text extension of XQuery where users must specify structure information as well as keywords according to the XQuery grammar.

There has been a lot of work on ranking schemes [1,6,8,15,17,18,20,26,28,41] for keyword search over XML, RDF, or relational databases. The ranking schemes and the concept of structural consistency can complement each other to help users find relevant results. For example, enforcing structural consistency could be too restrictive for certain applications, i.e., some query results eliminated by structural consistency may be relevant to the query. In this case, we can exploit structural consistency as one of the ranking criteria that measures the *meaningfulness* [47] of the results rather than as a criterion for removing spurious results as has similarly been suggested by Yu and Jagadish [47].

6 Experimental evaluation

6.1 Experimental setup

The goal of the experiments is to verify the advantage of our method in terms of search quality and search performance. As for *search quality*, we compare our method with SLCA [45] and MLCA [27] as they are the state-of-the-art methods; we exclude XSEarch [11] from the comparison since Li et al. [27] have shown that MLCA is generally superior to XSEarch. As for *search performance*, we compare our method with SLCA, excluding MLCA from the comparison, since Xu and Papakonstantinou [45] have shown that the SLCA searching algorithm generally shows superior performance over the MLCA searching algorithm. In addition, we compare the index creation time and index size of our method with those of the SLCA method to show that an extra schema index for efficient structural consistency checking incurs negligible overhead to overall system performance. We use precision and recall as the measure for search quality. Following the common practice [11,25,27], we define the *desired results of a keyword query* as those returned by structured queries (XPath queries) corresponding to the keyword query, which are formulated by the users who participated in the experiments. We use the wall clock time as the measure for search performance and index creation, and the number of pages allocated for the index size.

Independent of the query processing method, we need to specify which output (i.e., return nodes) generation strategies [30] to use: *Subtree Return*, *Path Return*, *Subtree-Entity Return*, and *Path-Entity Return*. Subtree Return outputs the whole subtree rooted at each query result. Path Return outputs the paths from the root of each query result to the query keywords. Subtree-Entity Return and Path-Entity Return first find the lowest entity ancestor-or-self node of each query result, and then, output the subtree rooted at the node and the paths from the node to the query keywords, respectively. In the same way as was done by Liu and Chen [30], if a node with label l_1 has a one-to-many relationship with nodes with label l_2, we consider the nodes with label l_2 as entities. According to Liu and Chen [30], Path Return usually has higher precision but lower recall than Subtree Return since it returns only paths. The strategies with entities generally have higher precision and recall than the ones without entities.

We present experimental results using the output strategies with entities since these strategies show superior search quality over those without. We note that this superiority has also been verified in all the experiments we performed. Thus, we omit experimental results for the output strategies without entities. For complete experimental results including other output strategies, please refer to our technical report [23]. Hereafter, "SC" denotes our method; "S-E" a method with Subtree-Entity Return; "P-E" a method with Path-Entity Return; and "-U" denotes a method with relevance feedback. For example, SC-S-E-U denotes our method with Subtree-Entity Return and with relevance feedback.

We have performed experiments using three real data sets and one synthetic data set. The first one is the DBLP data set [33]. We use the same schema used in the experiments by Xu and Papakonstantinou [45], that groups the DBLP data set first by journal/conference names, and then, by years. The second one is the SIGMOD Record data set [33]. The third one is the NASA data set [33], which consists of astronomical data. It has a complex and recursive schema and allows a wider variety of queries than the DBLP and SIGMOD Record

data sets. The fourth and synthetic one is the XMark benchmark data set available at the XMark web site [44]. These data sets have been extensively used in the existing work on XML keyword search [11,15,19,25,27,30,37,38,45,47]. Table 2 shows statistics of these data sets. We see that the size of the schema is significantly smaller than that of the XML data.

6.1.1 Experiment 1

To compare search performance and analyze the relationship between search performance and precision/recall in a controlled setting, we have generated the queries in Table 3 for the DBLP, NASA, and XMark data sets.[9] To show the cases where our method has low precision or recall, which are seldom, we add the following queries: QD_6, QD_7, QX_6, QX_7, $QN_4 \sim QN_7$. We also include QD_8, QX_8, QN_8 to test the case where users specify very long queries containing 9–13 keywords. We run each query in Table 3 ten times and measure precision, recall, and the average wall clock time. Since how the underlying XML data are stored highly affects the query result construction time, which is not our focus, we only access the root node r of each query result and report the number of the descendant nodes of r for the Subtree-Entity Return when measuring the wall clock time of query performance.

6.1.2 Experiment 2

To compare search performance for a real set of user queries, we have obtained 200 queries[10] for each of the real data sets (a total of 600 queries)—the DBLP, SIGMOD Record, and NASA data sets—from ten graduate students majoring in databases (but not involved in this project) for this purpose. We measure the wall clock time for all the queries.

6.1.3 Experiment 3

To show the superiority of the query evaluation algorithm presented in Sect. 4.2.2, we compare search performance of our method that uses the algorithm and the one that uses XIR [34], which does not process multiple XPath queries simultaneously. We measure the wall clock time for the 600 queries used in Experiment 2.

6.1.4 Experiment 4

To compare search quality for real sets of user queries, we measure precision and recall for the 600 queries used in Experiment 2.

6.1.5 Experiment 5

To compare the index creation time[11] and index size, we measure the wall clock time and the number of pages allocated.

6.1.6 Experiment 6

To test the scalability of our method, we generate XMark data sets by varying the size from 1 to 4 GB and from 100 MB to 10 GB. We measure the wall clock time for queries QX_2, QX_3, QX_4, and QX_8.

All the experiments are conducted on SUN Ultra 60 workstation with UltraSPARC-II 450MHz CPU and 512 MB of main memory. We implement all the methods on ODYSSEUS ORDBMS [43], which supports the inverted index. The page size for data and indexes is set to be 4,096 bytes. We use the Indexed Lookup Eager algorithm [45] as the SLCA searching algorithm since it generally shows superior performance over other algorithms. Finally, all the methods are implemented using C++.

6.2 Experimental results

6.2.1 Experiment 1

Figure 23 shows the precision, recall, and wall clock time for the queries $QD_1 \sim QD_8$ in Table 3 over the DBLP data set. SC-S-E (SC-P-E) improves the query performance by up to 2.4 times (2.5 times) over SLCA-S-E (SLCA-P-E). The reason for the improvement is that our method eliminates spurious results early by enforcing structural consistency at the schema-level. We note that the recall values of our method and SLCA are the same. The improvement becomes more marked when the precision of SLCA is low, i.e., when the number of spurious results is high. For example, in Fig. 23a, the precision of SLCA for QD_4 is lower than that for QD_3, and thus, in Fig. 23c, the query processing time for QD_4 is higher than that for QD_3, while those of our method are hardly changed. However, if the precision of SLCA is high, i.e., when there are few spurious results, for a specific query, our method could be marginally slower than SLCA due to the overhead of XPath query evaluation and iterative kth-ancestor generalization. For example, in Fig. 23c, our method is about 10% slower than SLCA for QD_1 and QD_5.

[9] For the XMark data set, the XMark benchmark queries are not used since the queries are expressed in XQuery and has complex semantics such as path expressions, join, aggregation, grouping, and ordering. Since keyword queries have inherently limited expressive power, it is not feasible to rewrite all the benchmark queries into keyword queries. For some queries that do not have complex semantics and can easily be converted into keyword queries, e.g., QX_4 and QX_7, we exploit them.

[10] For the list of queries, please refer to http://dblab.kaist.ac.kr/~drlee/sc.html.

[11] In the index creation time, the time for XML document parsing, keyword extraction, and data loading is excluded.

Table 2 Data statistics

Data set	Size (MB)	No. of instance nodes (excl. value nodes)	No. of distinct keywords	No. of schema nodes (excl. keywords)	Average depth
SIGMOD Record	0.5	15,263	5,652	12	5
DBLP	127	3,736,406	572,062	145	3
NASA	23	530,528	48,430	110	6
XMark	111	2,048,193	127,905	548	5

Table 3 Query sets

ID	Query
DBLP data set	
QD_1	"flexibility"
QD_2	"scheduling management"
QD_3	"quality analysis data"
QD_4	"rule programming object system"
QD_5	"Levy J Jagadish H"
QD_6	"flexibility message scheme"
QD_7	"ICDE XML Jagadish"
QD_8	"distributed data base systems performance analysis Michael Stonebraker John Woodfill"
NASA data set	
QN_1	"astroObjects"
QN_2	"Michael magnitude"
QN_3	"photometry galactic cluster Astron"
QN_4	"pleiades dataset"
QN_5	"PAZh components"
QN_6	"pleiades journal"
QN_7	"textFile name"
QN_8	"accurate positions of 502 stars Eichhorn Googe Murphy Lukac"
XMark data set	
QX_1	"Zurich"
QX_2	"Arizona Mehrdad edu"
QX_3	"Takano sun com mailto"
QX_4	"homepage name"
QX_5	"Helena 96"
QX_6	"mehrdad takano net"
QX_7	"person id person0 name"
QX_8	"harpreet mahony nodak edu 99 lazaro st el svalbard and jan mayen island"

In Fig. 23a, our method shows low precision for QD_6 and QD_7. For QD_6, there is a conference paper on "flexibility message scheme" in the database, but no journal article. In this case, our method finds spurious journal nodes through generalization, resulting in low precision. For QD_7, the user wants to find "ICDE" papers about "XML" authored by "Jagadish", but our method and SLCA return the whole subtree rooted at "ICDE" conference node (or the paths from the conference node to the query keywords), resulting in the same low precision. Even for such queries, the precision of our method is higher than or equal to that of SLCA since our method is able to eliminate more spurious results than

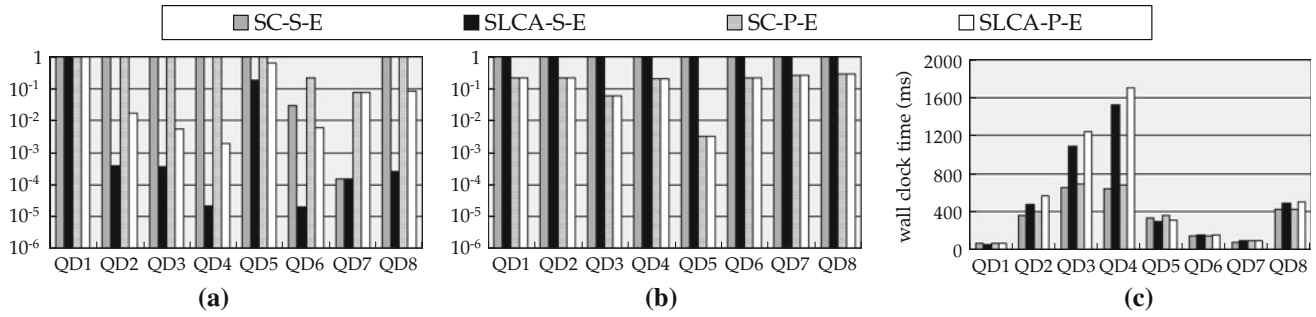


Fig. 23 Precision (**a**), recall (**b**), and wall clock time (**c**) of queries in Table 3 for the DBLP data set

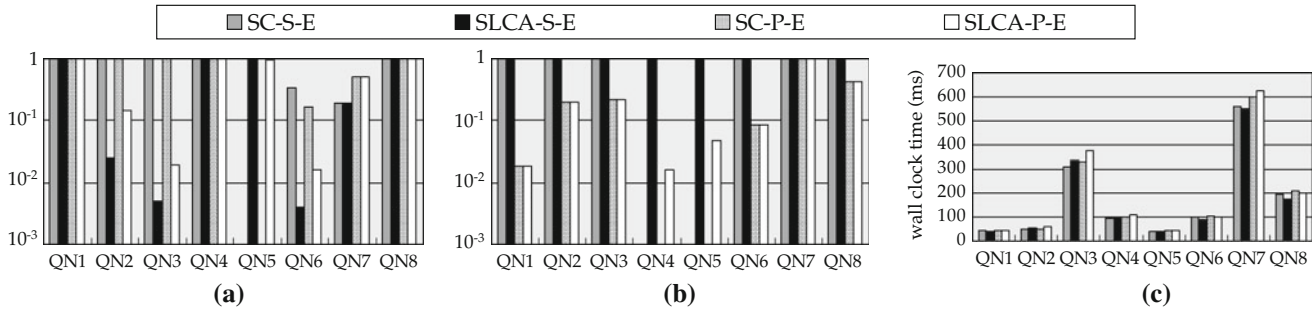


Fig. 24 Precision (**a**), recall (**b**), and wall clock time (**c**) of queries in Table 3 for the NASA data set

SLCA. For example, for QD_6, our method does not find spurious

conf nodes since there is a paper on "flexibility message scheme", but SLCA does.

The reason why the SLCA method often has very low precision is that it often finds more spurious SLCA nodes than correct ones. For example, there are only five publications of "Levy" on "XML" in the DBLP data set, but the SLCA method finds 50 SLCAs for the query "XML Levy", 45 of which are spurious conf nodes. Furthermore, conf nodes typically include huge subtrees having thousands of nodes. Thus, the number of retrieved nodes that are spurious becomes very large leading to very low precision. The Subtree-Entity Return (S-E) has even lower precision because this strategy returns the whole subtree rooted at each query result, and the number of all nodes in the subtree is counted as the number of retrieved nodes.

Figure 24 shows the precision, recall, and wall clock time for the NASA data set, having a tendency similar to that of the DBLP data set except QN_4 and QN_5.

For QN_4, the recall value of our method, SC-S-E and SC-P-E, is almost 0 (both 1.3×10^{-4} since they find the same para nodes). This is because the user's intention is to find more general results, which we regard as spurious results, i.e., for QN_4 ("pleiades dataset"), the user's intention is to find the subtrees rooted at dataset nodes that contain the keyword "pleiades". However, our method finds only the para nodes (i.e., paragraphs) that are contained in the subtrees rooted at

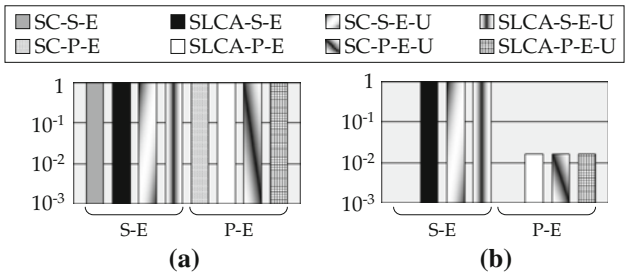


Fig. 25 Precision (**a**) and recall (**b**) of QN_4 with relevance feedback

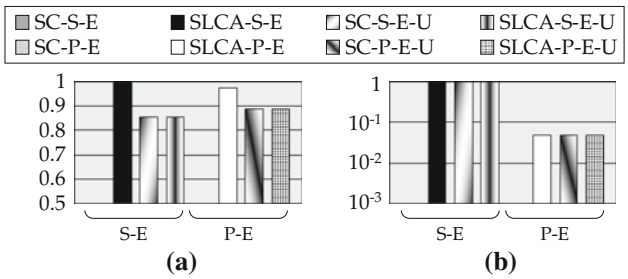


Fig. 26 Precision (**a**) and recall (**b**) of QN_5 with relevance feedback

the dataset nodes. In contrast, the SLCA method finds (1) the para nodes and (2) the dataset nodes that do not have para nodes containing the keywords "pleiades" and "dataset" (we note that the recall value of SLCA-S-E for QN_4 looks perfect in Fig. 24b, but it is not 1.0 since the SLCA method also finds the para nodes as our method does). We can solve this

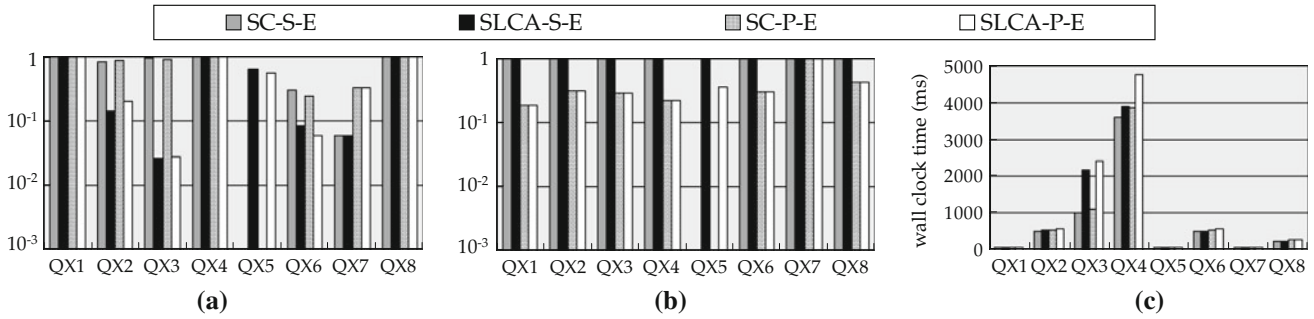


Fig. 27 Precision (**a**), recall (**b**), and wall clock time (**c**) of queries in Table 3 for the XMark data set

low-recall problem using relevance feedback. The result is shown in Fig. 25. Using relevance feedback, we can generalize the para nodes to the dataset nodes and obtain the desired results. With feedback, our method SC-S-E-U (or SC-P-E-U) and the SLCA method SLCA-S-E-U (or SLCA-P-E-U) show the same recall values. For all the methods, the precision values are the same (1.0).

For QN_5 ("PAZh components"), the precision and recall values of our method are both 0 constituting the worst case of our method. Here, the user's intention is to find the subtrees rooted at the dataset nodes that (1) have altname nodes whose value is "PAZh" and (2) contain the keyword "components". However, our method finds holding nodes only since there are holding nodes that contain the keywords "PAZh" and "components". In contrast, the SLCA method finds (1) the holding nodes and (2) the desired dataset nodes. We can also solve this problem by generalizing the holding nodes to the dataset nodes. The result is shown in Fig. 26. In Fig. 26a, the precision values of our method with feedback SC-S-E-U (or SC-P-E-U) are worse than those of the SLCA method without feedback SLCA-S-E (or SLCA-P-E) because we find spurious results during generalization while the SLCA method without feedback does not as explained in Example 14 of Sect. 3.4[12]. However, SC-S-E-U (or SC-P-E-U) shows the same precision values with those of SLCA-S-E-U (or SLCA-P-E-U) since the SLCA method with feedback also finds the same spurious results during generalization. With feedback, our method and the SLCA method show the same recall values.

Figure 27 shows the precision, recall, and wall clock time for the XMark data set, showing a similar tendency to those of the DBLP and NASA data sets. Similar to QN_5 in the NASA dataset, QX_5 constitutes the worst case of our method. Figure 28 shows the results of QX_5 with relevance feedback. With feedback, our method and the SLCA method show the same precision and recall values.

[12] In Example 14, conf_year nodes correspond to dataset nodes; chair to altname; "Levy" to "PAZh"; "XML" to "components"; paper to holding.

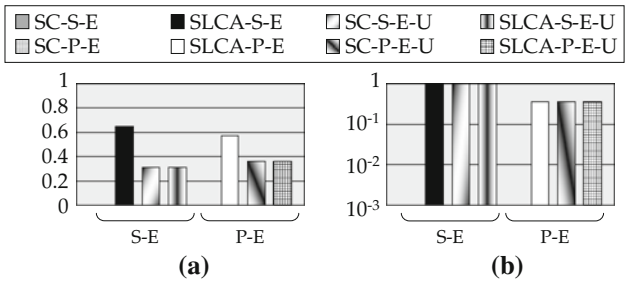


Fig. 28 Precision and recall of QX_5 with relevance feedback

6.2.2 Experiment 2

Figure 29 shows the search performance results for a real set of user queries. The Y-axis represents the fraction of queries for which our algorithm has a given range of performance improvement over the SLCA algorithm. The performance improvement is defined as the wall clock time $T_{\text{SLCA-S-E}}$ of SLCA over the wall clock time $T_{\text{SC-S-E}}$ of SC and denoted as x. For the NASA data set in Fig. 29c, SC-S-E (SC-S-E-U) outperforms SLCA-S-E by more than 10% for 69% (66%) of queries. In contrast, SLCA-S-E outperforms SC-S-E (SC-S-E-U) for only 10% (12%) of queries. Figure 29a, b shows a tendency similar to that of the NASA data set. We omit the results for the Path-Entity Return (P-E) since they show a tendency similar to those of the Subtree-Entity Return (S-E).

6.2.3 Experiment 3

Our method that uses the algorithm presented in Sect. 4.2.2 outperforms the one that uses XIR [34] by 1.8–5.2 times since the algorithm simultaneously evaluates multiple XPath queries while XIR evaluates one query at a time.

6.2.4 Experiment 4

Figures 30 and 31 show the precision (denoted as p) and the recall (denoted as r) of 200 queries over the DBLP data

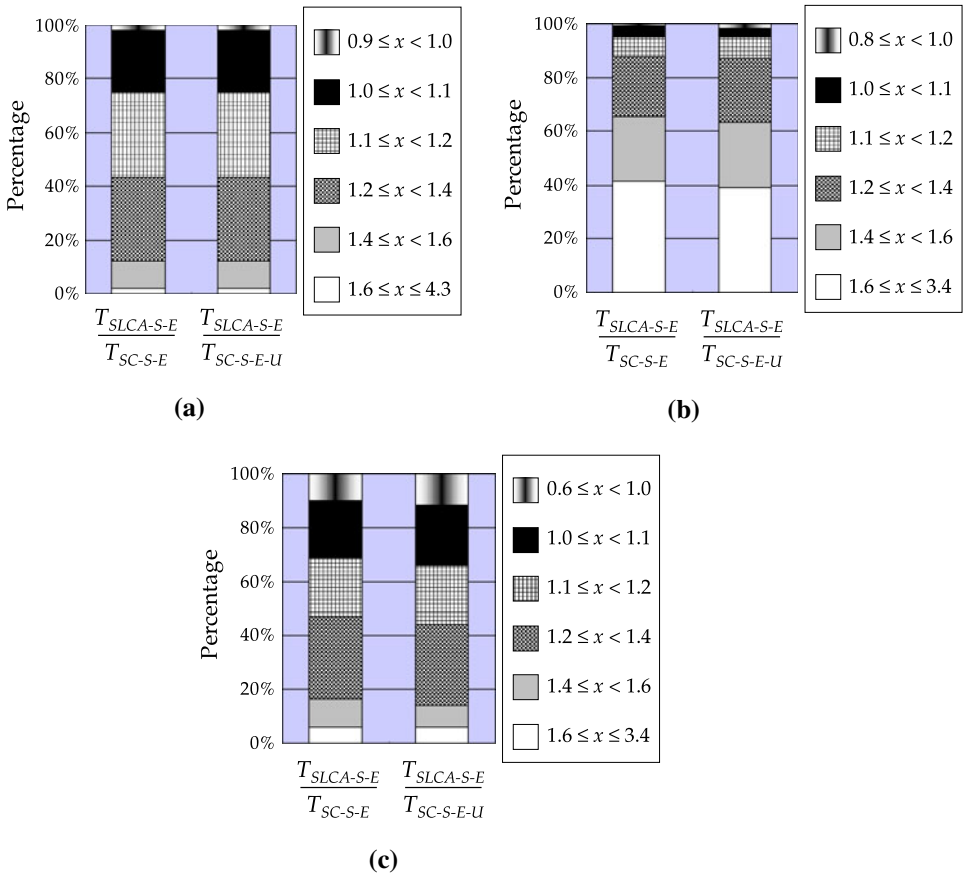


Fig. 29 The search performance results of 600 queries for the DBLP (**a**), SIGMOD Record (**b**), and NASA (**c**) data sets. The Y-axis represents the fraction of queries for which our algorithm has a given range of performance improvement over the SLCA algorithm

set and the SIGMOD Record data set, respectively. The Y-axis of the figures represents the fractions of queries having given precision/recall ranges. MLCA and SLCA often find more spurious nodes than correct ones. For example, for the query "activity recognition", they find 130 results, 122 of which are spurious conf or journal nodes. Thus, for the DBLP data set, the precision of SLCA and MLCA is less than 0.5 for 46–87% of queries! For the SIGMOD Record data set, their precision is less than 0.5 for 23–59% of queries. In contrast, the precision of our method is 1.0 for all queries since it eliminates all the spurious results by enforcing structural consistency. We note that the recall values of our method, MLCA, and SLCA are the same. These results are similar to those of Experiment 1.

In Fig. 31b, SC-S-E, MLCA-S-E, and SLCA-S-E show low recall for about 16% of queries. In this case, the users want the articles of an author, e.g., "Jennifer Widom", but all methods return the author in the articles since the author is the lowest entity containing all the query keywords. With feedback, our method SC-S-E-U (or SC-P-E-U) and existing methods MLCA-S-E-U and SLCA-S-E-U (or MLCA-P-E-U and SLCA-P-E-U) show the same higher recall values as in Fig. 31b. The precision values of our method are still higher than or equal to those of existing methods as in Fig. 31a. The average number of relevance feedbacks provided by the users for the 200 queries on the SIGMOD Record data set is 0.36/query.

Figure 32 shows the precision and the recall of 200 queries over the NASA data set. The precision of SLCA and MLCA is less than 0.5 for 35–56% of queries. In contrast, the precision of our method is less than 0.5 for only 9–10% of queries. Here, our method shows low precision for some queries due to the complex schema of the NASA data set. For example, for the query "radio journal", the user wants to find journal articles on "radio". Our method finds not only correct results but also spurious results such as revision nodes, as SLCA and MLCA do, since there are revision nodes that contain the keywords "radio" and "journal".

In Fig. 32b, for about 9% of queries, the recall values of our method without feedback are lower than those of SLCA and MLCA due to the same reason as in Example 14 of Sect. 3.4. With feedback, our method SC-S-E-U (or SC-P-E-U) and existing methods MLCA-S-E-U and SLCA-S-E-U (or MLCA-P-E-U and SLCA-P-E-U) show the same recall values as in Fig. 32b. The precision values of our method are still higher than or equal to those of existing methods as in Fig. 32a. The average number of relevance feedbacks provided by the users for the 200 queries on the NASA data set is 0.30/query.

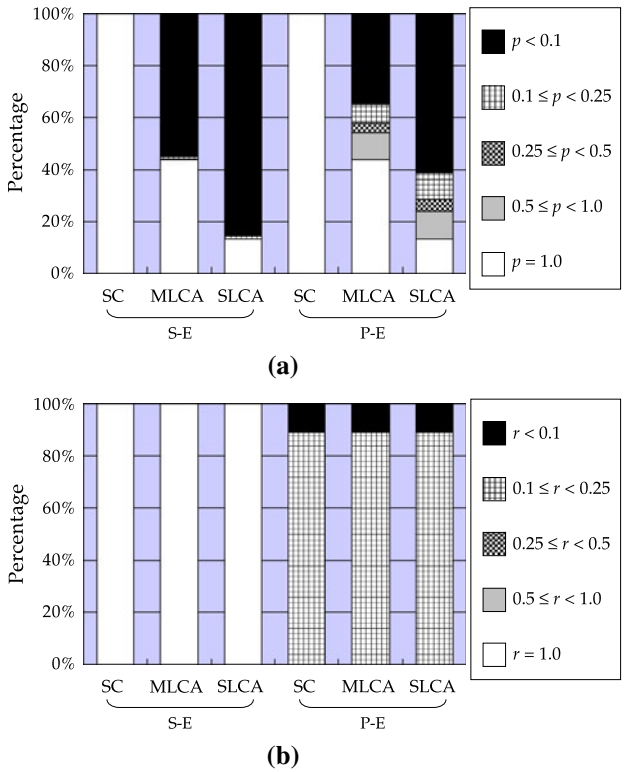


Fig. 30 Precision (**a**) and recall (**b**) of 200 queries for the DBLP data set. The Y-axis represents the fraction of queries having a given precision/recall range

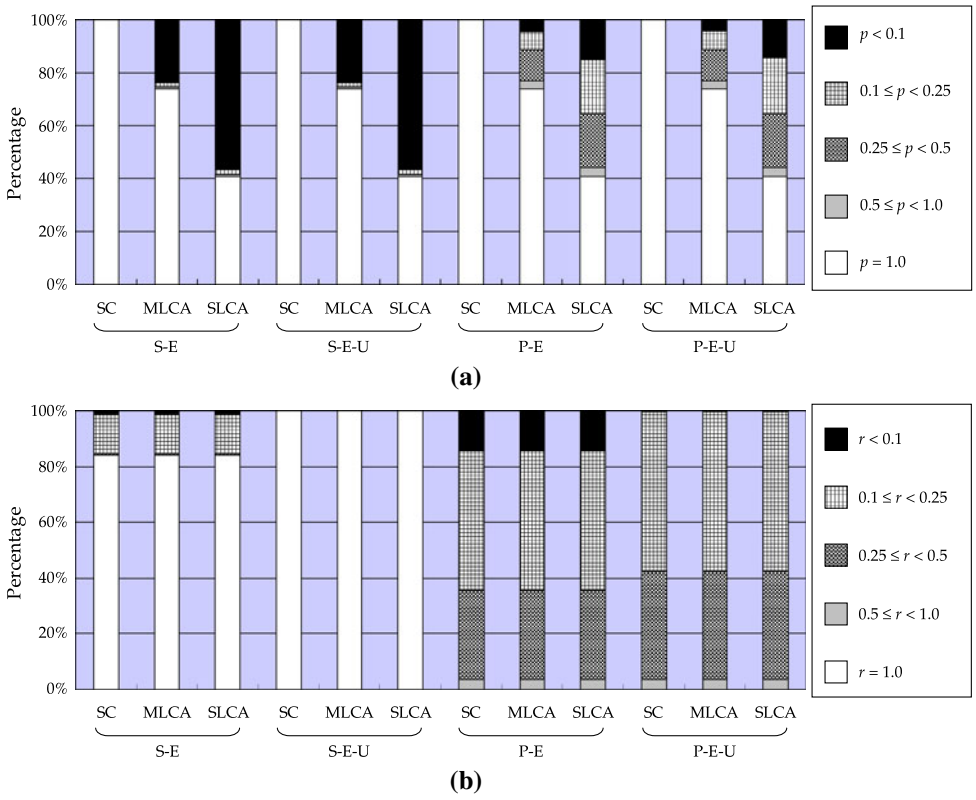
6.2.5 Experiment 5

Figure 33 shows the index creation time and the index size. All methods use an inverted index for XML data and the *Dewey index* [30] to find the lowest entity ancestor of each query result. SC-S-E and SC-P-E additionally use the schema index for efficient structural consistency checking. Thus, the index creation time of SC-S-E and SC-P-E is about 5–7% longer, and the index size is about 5–7% larger than those of SLCA-S-E and SLCA-P-E. This verifies that an extra schema index incurs negligible overhead to overall system performance. We note that the index is bigger than the original data due to the space required for storing id paths from the root to each node. SLCA-based methods have similar space overhead since they also use id paths, i.e., Dewey numbers. We could reduce the space by exploiting the UTF-8 encoding as an efficient way to represent id paths, which was proposed by Tatarinov et al. [39].

6.2.6 Experiment 6

Figures 34 and 35 show the processing time of queries QX_2, QX_3, QX_4, and QX_8 as the data set size is varied from 1 to 4 GB and from 100 MB to 10 GB. As we can see, the processing time of all methods increases approximately

Fig. 31 Precision (**a**) and recall (**b**) of 200 queries for the SIGMOD Record data set. The Y-axis represents the fraction of queries having a given precision/recall range



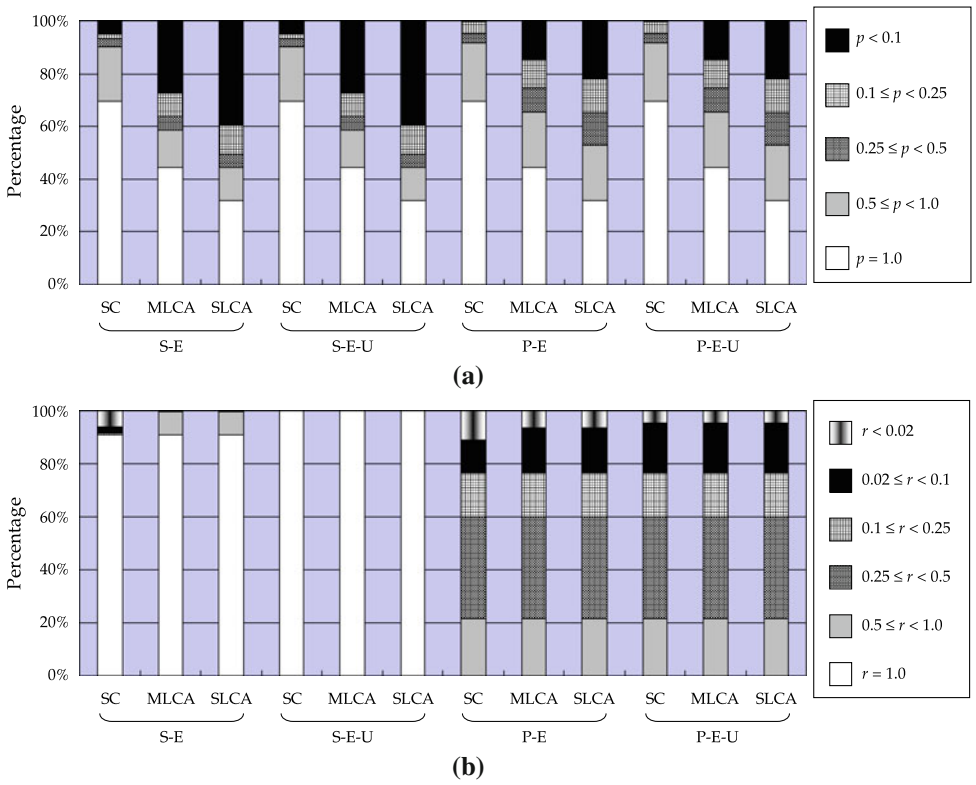


Fig. 32 Precision (**a**) and recall (**b**) of 200 queries for the NASA data set. The Y-axis represents the fraction of queries having a given precision/recall range

linearly when the data set size increases and that our methods are largely superior or comparable to SLCA-based methods.

7 Conclusions

We have proposed a new notion of structural consistency (and structural anomaly) in XML keyword search. By exploiting structural consistency, we can eliminate spurious results having the same result structure consistently. We have introduced the concept of the result structure in Definition 3 and the smallest result structure in Definition 6. We have formally defined the structural anomaly in Definition 5 as a phenomenon where there exist result structures that structurally contain other result structures. We have defined the structural consistency as a property where there is no structural anomaly in the query results.

We have proposed a naive algorithm that resolves structural anomaly at the *instance* level. We have then proposed an advanced algorithm that resolves structural anomaly at the *schema* level. To this end, we have formally analyzed the relationship between the set of schema-level SLCAs and the set of instance-level SLCAs in Lemmas 2–3, identified the discrepancies between them, and proposed the notion of iterative kth-ancestor generalization to resolve the anomalies (false dismissal and phantom schema structures) that are caused by these discrepancies. We have formally proved

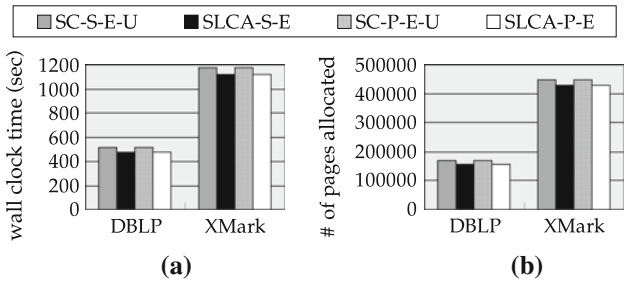


Fig. 33 Index creation time (**a**) and index size (**b**) for the DBLP and XMark data sets

that the proposed algorithms produce the same set of results preserving structural consistency in Theorem 1. We have proposed a solution using relevance feedback for the problem where our method has low recall; this problem occurs when it is not the user's intention to find more specific results. We have provided an efficient algorithm that simultaneously evaluates multiple XPath queries generated by our method. We have implemented our method in a full-fledged object-relational DBMS.

We have performed extensive experiments using real and synthetic data sets. Experimental results show that our method improves precision significantly compared with the existing methods while providing comparable recall for most queries. Experimental results also show that our method improves the query performance over the existing methods significantly by removing spurious results early.

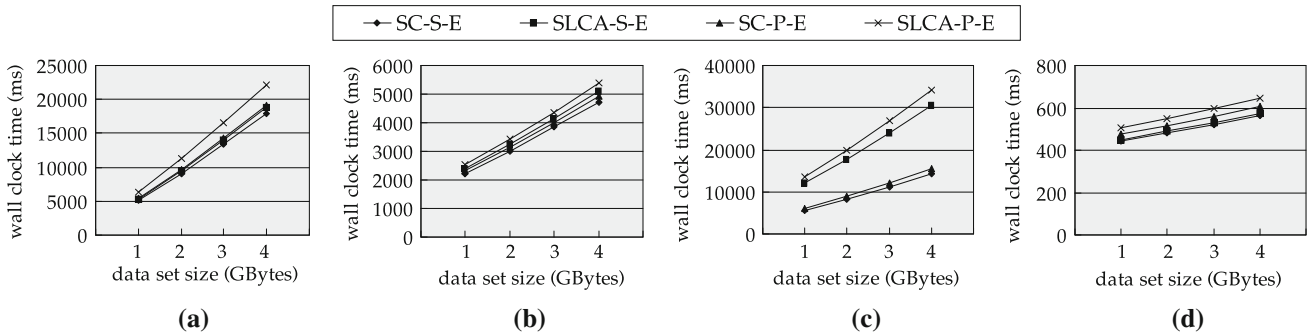


Fig. 34 Query processing time with increasing data set size from 1 to 4 GB in a linear scale. **a** QX_2, **b** QX_3, **c** QX_4, **d** QX_8

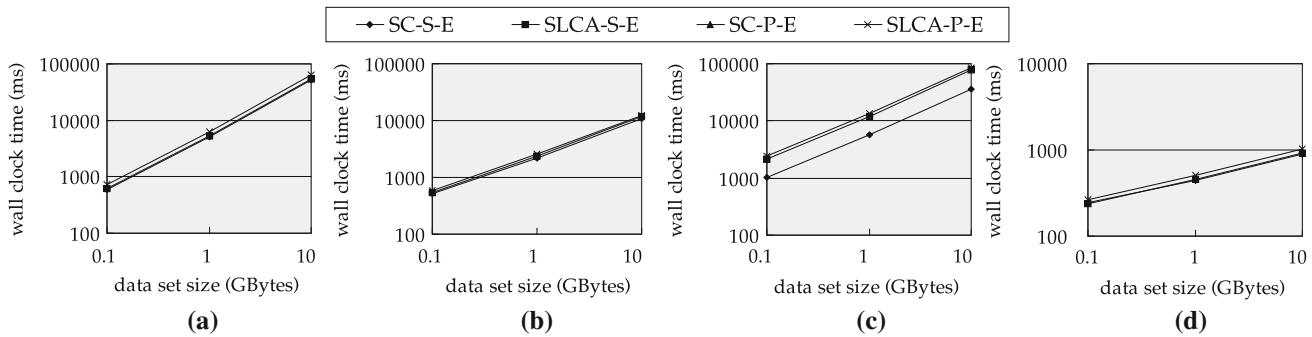


Fig. 35 Query processing time with increasing data set size from 100 MB to 10 GB in a logarithmic scale

Acknowledgments Earlier versions of this paper were presented in the KAIST CS technical report [23], PhD dissertation [22] of Ki-Hoon Lee, and CoRR technical report arXiv:0911.4329. This work was partially supported by the National Research Lab Program (No. 2009-0083120) through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science, and Technology (MEST) and was partially supported by the Engineering Research Center of Excellence Program (No. 2009-0063243) funded by MEST. This work was also partially supported by the Microsoft Research Asia. We deeply appreciate the anonymous reviewers' incisive comments, which helped completeness and readability of the paper.

8 Appendix

8.1 A. Proof of Lemma 2

Let $\{w_1, w_2, \ldots, w_n\}$ be the set of query keywords of Q, and $l_1, l_2, \ldots, l_m$ be the incoming label path of srs_i. We need to show that there always exists a schema-level SLCA s such that $l_1, l_2, \ldots, l_m$ is a prefix of the label path of s. Since srs_i is a smallest result structure of instance-level SLCAs, there exists an instance node v such that $l_1, l_2, \ldots, l_m$ is the label path of v, and $w_1, w_2, \ldots, w_n$ are descendants of v. It follows that there exists a schema node s_a such that $l_1, l_2, \ldots, l_m$ is the label path of s_a and $w_1, w_2, \ldots, w_n$ are descendants of s_a (i.e., $srs_i \equiv ss(s_a)$) since the DataGuide$^+$ has every unique label path of instance nodes. Thus, by the definition of schema-level SLCA, there exists a schema-level SLCA s such that $ss(s_a) \preceq ss(s)$. □

8.2 B. Proof of Lemma 3

Let $ILP(srs_i)$ be the incoming label path of srs_i, and $ILP(ss_j)$ be the incoming label path of ss_j. Since $srs_i \prec ss_j$ $ILP(srs_i)$ is a proper prefix of $ILP(ss_j)$. This implies that there must exist a kth-ancestor $s_a (1 \leq k \leq depth(s))$ of the schema-level SLCA s whose label path is the same as $ILP(srs_i)$. Here, $ss(s_a) \equiv srs_i$ since the label path of s_a is the same as $ILP(srs_i)$. □

References

1. Agrawal, S., Chaudhuri, S., Das, G.: DBXplorer: a system for keyword-based search over relational databases. ICDE (2002)
2. Amer-Yahia, S., Curtmola, E., Deutsch, A.: Flexible and efficient XML search with complex full-text predicates. SIGMOD (2006)
3. Amer-Yahia, S., Koudas, N., Marian, A., Srivastava, D., Toman, D.: Structure and content scoring for XML. VLDB (2005)
4. Arion, A., Bonifati, A., Manolescu, I., Pugliese, A.: Path summaries and path partitioning in modern XML databases. World Wide Web J., **11**(1), (2008)
5. Baeza-Yates, R., Ribeiro-Neto, B.: Modern Information Retrieval. ACM Press, New York (1999)
6. Bao, Z., Ling, T.W., Chen, B., Lu, J.: Effective XML keyword search with relevance oriented ranking. ICDE (2009)

7. Bex, G.J., Neven, F., Vansummeren, S.: Inferring XML schema definitions from XML data. VLDB (2007)
8. Bhalotia, G., Hulgeri, A., Nakhe, C., Chakrabarti, S., Sudarshan, S.: Keyword searching and browsing in databases using BANKS. ICDE (2002)
9. Bruno, N., Gravano, L., Koudas, N., Srivastava, D.: Navigation- vs. index-based XML multi-query processing. ICDE (2003)
10. Bruno, N., Koudas, N., Srivastava, D.: Holistic twig joins: optimal XML pattern matching. SIGMOD (2002)
11. Cohen, S., Mamou, J., Kanza, Y., Sagiv, Y.: "XSEarch: a semantic search engine for XML. VLDB (2003)
12. Deutsch, A., Fernandez, M., Florescu, D., Levy, A., Maier, D., Suciu, D.: Querying XML data. IEEE Data Eng. Bull. **22**(3), (1999)
13. GalaTex: An Implementation of XQuery Full Text. http://www.galaxquery.com/galatex
14. Goldman, R., Widom, J.: DataGuides: enabling query formulation and optimization in semistructured databases. VLDB (1997)
15. Guo, L., Shao, F., Botev, C., Shanmugasundaram, J.: XRANK: ranked keyword search over XML documents. SIGMOD (2003)
16. Hlaoua, L., Boughanem, M., Pinel-Sauvagnat, K.: Combination of evidences in relevance feedback for XML retrieval. CIKM (2007)
17. Hristidis, V., Papakonstantinou, Y.: DISCOVER: keyword search in relational databases. VLDB (2002)
18. Hristidis, V., Gravano, L., Papakonstantinou, Y.: Efficient IR-style keyword search over relational databases. VLDB (2003)
19. Hristidis, V., Koudas, N., Papakonstantinou, Y., Srivastava, D.: Keyword proximity search in XML trees. IEEE Trans. Know. Data Eng. **18**(4), (2006)
20. Hristidis, V., Papakonstantinou, Y., Balmin, A.: Keyword proximity search on XML graphs. ICDE (2003)
21. Huang, Y., Liu, Z., Chen, Y.: Query biased snippet generation in XML search. SIGMOD (2008)
22. Lee, K.: Processing XML keyword queries and structured queries using the structural summary, Ph. D. Dissertation, Computer Science Department, KAIST, Nov. 2008
23. Lee, K., Han, W., Whang, K., Kim, M.: Structural consistency: a correctness criterion in XML keyword search, Technical Report CS-TR-2008-286, Department of Computer Science, KAIST, June 2008
24. Li, C., Ling, T.W., Hu, M.: Efficient updates in dynamic XML data: from binary string to quaternary string. VLDB J. **17**(3), (2008)
25. Li, G., Feng, J., Wang, J., Zhou, L.: Effective keyword search for valuable LCAs over XML documents. CIKM (2007)
26. Li, G., Ooi, B.C., Feng, J., Wang, J., Zhou, L.: EASE: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured Data. SIGMOD (2008)
27. Li, Y., Yu, C., Jagadish, H.V.: Enabling schema-free XQuery with meaningful query focus. VLDB J. **17**(3), (2008)
28. Liu, F., Yu, C.T., Meng, W., Chowdhury, A.: Effective keyword search in relational databases. SIGMOD (2006)
29. Liu, H., Ling, T.W., Yu, T., Wu, J.: Efficient processing of multiple XML twig queries. DEXA (2006)
30. Liu, Z., Chen, Y.: Identifying return information for XML keyword search. SIGMOD (2007)
31. Liu, Z., Chen, Y.: Reasoning and identifying relevant matches for XML keyword search. VLDB (2008)
32. Luo, Y., Lin, X., Wang, W., Zhou, X.: SPARK: Top-k keyword query in relational databases. SIGMOD (2007)
33. Miklau, G.: The XML Data Repository. http://www.cs.washington.edu/research/xmldatasets
34. Park, Y., Whang, K., Lee, B., Han, W.: Efficient evaluation of partial match queries for XML documents using information retrieval techniques. DASFAA (2005)
35. Pradhan, S.: An algebraic query model for effective and efficient retrieval of XML fragments. VLDB (2006)
36. Schenkel, R., Theobald, M.: Feedback-driven structural query expansion for ranked retrieval of XML data. EDBT (2006)
37. Schmidt, A., Kersten, M.L., Windhouwer, M.: Querying XML documents made easy: nearest concept queries. ICDE (2001)
38. Sun, C., Chan, C., Goenka, A.K.: Multiway SLCA-based keyword search in XML data. WWW (2007)
39. Tatarinov, I., Viglas, S., Beyer, K.S., Shanmugasundaram, J., Shekita, E.J., Zhang, C.: Storing and querying ordered XML using a relational database system. SIGMOD (2002)
40. The World Wide Web Consortium, XQuery and XPath Full Text 1.0 (W3C Candidate Recommendation). http://www.w3.org/TR/xquery-full-text, (2008)
41. Tran, T., Rudolph, S., Cimiano, P., Wang, H.: Top-k exploration of query candidates for efficient keyword search on graph-shaped (RDF) data. ICDE (2009)
42. Whang, K., Park, B., Han, W., Lee, Y.: An inverted index storage structure using subindexes and large objects for tight coupling of information retrieval with database management systems, U.S. Patent No. 6,349,308, Feb. 19, 2002, Appl. No. 09/250,487, Feb. 15, 1999
43. Whang, K., Lee, M., Lee, J., Kim, M., Han, W.: Odysseus: A high-performance ORDBMS tightly-coupled with IR features. ICDE (2005) (this paper received the Best Demonstration Award)
44. XMark—An XML Benchmark Project. http://monetdb.cwi.nl/xml
45. Xu, Y., Papakonstantinou, Y.: Efficient keyword search for smallest LCAs in XML databases. SIGMOD (2005)
46. Xu, Y., Papakonstantinou, Y.: Efficient LCA based keyword search in XML Data. EDBT (2008)
47. Yu, C., Jagadish, H.V.: Querying complex structured databases. VLDB (2007)
48. Zhang, B., Geng, Z., Zhou, A.: SIMP: efficient XML structural index for multiple query processing. WAIM (2008)