SINBAD: Towards Structure-Independent Querying of Common Neighbors in XML Databases

Ba Quan Truong[1], Sourav S. Bhowmick[1], and Curtis Dyreson[2]

[1] School of Computer Engineering, Nanyang Technological University, Singapore
[2] Department of Computer Science, Utah State University, USA
{bqtruong,assourav}@ntu.edu.sg, curtis.dyreson@usu.edu

Abstract. XML query languages use *directional* path expressions to locate data in an XML data collection. They are tightly coupled to the structure of a data collection, and can fail when evaluated on the *same data* in a *different structure*. This paper extends XPath expressions with a new structure-independent, non-directional axis called the *neighborhood* axis. Given a pair of context nodes, the *neighborhood* axis returns those nodes that are *common* neighbors of the context nodes in *any* direction. Such axis finds its usefulness in structure-independent query formulation as well as supporting relevant results computation in design-independent XML keyword search. We propose an algorithm called SINBAD that exploits the novel notion of *node locality* and small size of XML *structure tree* to efficiently determine the common neighbors of the context nodes. Our empirical study demonstrates that SINBAD, built on top of an existing *path materialization*-based relational storage scheme, has promising query performance.

1 Introduction

A wealth of existing literature has extensively studied evaluation of various navigational axes in XPath expressions [6]. A key common feature of these axes is that they are all *directional* in nature. That is, they locate nodes in a fixed direction relative to a context node (*e.g.,* the descendent axis corresponds to the "down" direction). Unfortunately, queries that rely on directional axes become dependent on the data being in the specified direction, even though data has no "natural" direction and can be organized in different hierarchies. Users who are unfamiliar with a document structure or are knowledgeable about a structure which subsequently changes will sometimes formulate *unsatisfiable queries*, which are queries that fail to produce desired results. These queries are difficult to debug since they run to completion and produce a result, though not the desired one.

As an example, consider the XML document in Figure 1(a). It contains league information organized by teams. Each team consists of a set of players. Suppose that a basketball commentator, John, wishes to find the *common* team of a player, *Hill*, and a manager, *Antoni*. John can issue any one of the following XPath queries to retrieve desired information: Q_1: `//player[name='Hill']/ancestor::team[/descendant ::manager[name='Antoni']]` or Q_2: `//manager[name='Antoni']/ancestor::team[/descendant::player[name='Hill']]`.

To correctly formulate the aforementioned queries, John has to know something about the hierarchical structure of the XML data. For instance, he must know that a

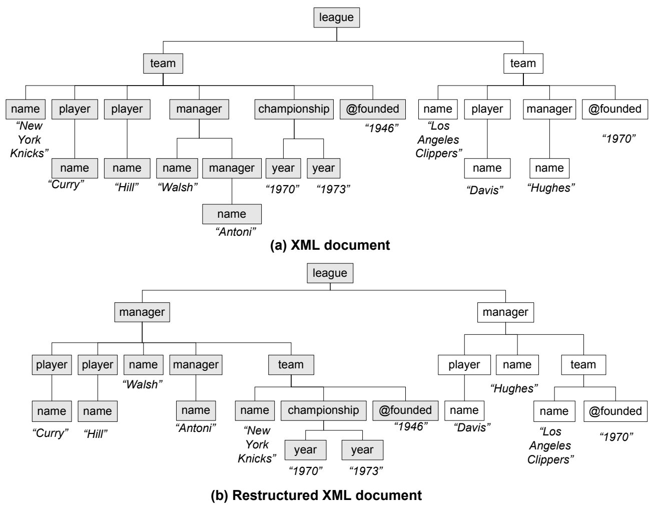


Fig. 1. XML documents

team element is an ancestor of player and manager elements. Furthermore, the team subtree also includes information related to the team. But if John misunderstands the structure or if the structure changes over time then this partial knowledge may not be useful anymore for formulating satisfiable queries as demonstrated below.

Assume that the XML document in Figure 1(a) is now reorganized to the structure depicted in Figure 1(b). Specifically, the document in Figure 1(b) has *same data* but the *structural relationships* of the elements are different. Now the league information is organized according to head managers instead of teams. Both documents contain the same data and same element labels but they have different hierarchical relationships. Such structural change is real because database administrators may revise the design over time to address issues such as redundancy, space overhead, performance, and usability [4,12]. Unfortunately, due to the lack of non-directional axes in XPath, for some queries different path expressions are needed to query each hierarchy. Consequently, Q_1 and Q_2 may become unsatisfiable on the document in Figure 1(b) as the team element is no more an ancestor of player or manager elements.

Note that it is unrealistic to expect users to be "structure-aware" as it does not scale with increasing structural heterogeneity. Is it possible to retrieve the above information using a single query without being aware of the underlying structural heterogeneities of elements? Ideally, such a query technique should work even if the document structure is reorganized. In order to answer this question affirmatively, in this paper we propose a new *non-directional* XPath axis called neighborhood axis, which enables us to locates all *common nodes* of two context nodes in *any* direction.

Specifically, the XPath language is extended with a *non-directional locator*, called the neighborhood axis, to support non-directional exploitation of XML data. The proposed axis allows a user to formulate precise queries knowing only the labels of nodes and unaware of the exact hierarchy. Informally, given *two* context nodes, the `neighborhood` axis returns those nodes that are common nodes to these context nodes. For example, reconsider the query posed by John. The relevant `team` node must be related to both the `player` node containing *Hill* and the `manager` node containing *Antoni*. Accordingly, John can reformulate his query using the `neighborhood` axis as follows: Q_3: `//player[name ='Hill']/neighborhood{//manager[name ='Antoni']}::team`.

Note that Q_3 will retrieve the same information when it is evaluated over Figure 1(b) as well. More importantly, a user does not need to be aware of the structural relationship between the context and test nodes. In this case, John only needs to know that a team could employ a player and a manager (real-world employment relationship). He does not need to know the relative hierarchical relationship among them (e.g., `team` is ancestor or descendant of `manager`) in the document.

The `neighborhood` axis has practical significance in at least two applications. Firstly, it can complement classical approach to query XML data by enabling users to formulate *structure-independent* queries to seek common nodes of a pair of context nodes. Note that classical XPath axes fail to formulate such structure-independent queries. Secondly, it can provide a framework to support *design-independent* XML keyword search [11] by finding relevant nodes that are semantically related to a set of nodes containing matching query keywords. These nodes can be returned with the result set in order to ensure that the results of XML keyword search are informative.

We propose a novel and generic algorithm called SINBAD (**S**tructure **I**ndependent commo**N** neigh**B**ors **A**bstraction proce**D**ure) to evaluate `neighborhood` axis by exploiting the notion of *node locality*. Informally, given a context node c, the *locality* of c is a set of nodes that are *semantically related* to c (detailed in Section 3). The intuition behind node locality is that users (queries) are typically interested in nodes within the locality and rarely refer to nodes outside of the locality. As we shall see later, the evaluation of `neighborhood` axis is equivalent to finding the *intersection region* of two node localities.

In summary, this paper makes three main contributions. First, we extend classical XPath query language with a non-directional `neighborhood` axis in Section 4. Secondly, in Section 5 we present a novel algorithm called SINBAD to evaluate neighborhood axis queries by exploiting the notion of node locality. Thirdly, through an experimental study on synthetic and real data sets, in Section 6, we show that our approach can retrieve common neighbors efficiently .

2 Related Work

Our objective to flexibly issue XML queries independent of the structure is shared by several recent papers. [3] presents a semantic search engine for XML. The search relies on an interconnection relationship to decide whether nodes are semantically related. Two nodes are interconnected if and only if the path between them contains no other node that has the same label as the two nodes. [7] proposes a schema-free XQuery, facilitated by a *Meaningful Lowest Common Ancestor Structure* (MLCAS) operation. Unlike `neighborhood` axis, these approaches do not retrieve common neighbors of two context nodes.

Recently, several XML keyword search techniques [8, 9, 13] have been proposed to offer more user-friendly solution for retrieving relevant results. Essentially, these approaches return variants of the subtree rooted at the lowest common ancestor (*e.g.,* VLCA, SLCA) of all the keywords. Due to the lack of expressivity and inherent ambiguity of keyword search, several techniques have also been developed to infer and retrieve relevant results for a search query [8, 9, 11]. Our work differs from the keyword search paradigm in the following ways. First, we retrieve nodes based on common locality of a pair of context nodes and not the entire LCA-variant of all the keywords. Note that LCA and its variants make use of some common ancestors of the context nodes and therefore rely on the hierarchical relationships. Consequently, these techniques are not structure-independent. Secondly, as a neighborhood query is an extension of conventional XPath query, it can impose more complex predicates compared to keyword search queries. Furthermore, it does not suffer from expressivity and ambiguity issues similar to keyword search.

More germane to this work is our previous efforts in [1, 15]. In [15], we extended the XPath language with a *symmetric* locator, called the `closest` axis, which locates nodes that are *closest* to a context node. Here *closest* is measured by the distance from the context node in *any* direction in the XML tree. In [1], we proposed `rank-distance` axis, which is a more generic non-directional axis compared to the closest axis. Specifically, given a context node and two parameters α and β, the `rank-distance` axis returns those nodes that are ranked between α and β in terms of closeness from the context node. Not only it can find closest node(s) (by setting α and β to one) but also nodes that are further away from the context node. In contrast, here we focus on a new axis, called `neighborhood`, which computes common neighbors of two context nodes.

Note that common neighbors cannot be computed using closest axis. For example, reconsider the query in Section 1 for finding the common team of *Hill* and *Antoni*. At first glance, it may seem that this query may be expressed as follows: Q_4: `//player[name='Hill']/closest::team[closest::manager[name='Antoni']]`. Unfortunately, Q_4 returns empty result set. The fragment `//player[name='Hill']/closest::team` will return the team of Hill (which is *New York Knicks*). Hence, when the context node is at this `team` node, the closest `manager` node is, unfortunately, not *Antoni* but manager *Walsh*. Note that we cannot use `rank-distance` axis to select *Antoni* here as it demands knowledge of structural relationship between manager nodes (*Antoni* is a second-level manager) from the user in order to assign appropriate values to the parameters α and β.

3 Node Locality

In this section, we introduce the notion of *node locality* that we shall be using to define neighborhood axis. We begin by briefly introducing the XML data model considered in this paper.

We model XML documents as ordered, labeled trees as follows. A tree is a tuple $(\mathcal{N}, \mathcal{E}, \Sigma, \mathbb{L}, \mathbb{F}, \mathbb{T}, \mathcal{S})$, where (a) $\mathcal{N}$ is the node set. $r \in \mathcal{N}$ is a special node called the root of the tree, (b) let O be the domain of ordinals. Then $\mathcal{E} \subseteq O \times \mathcal{N} \times \mathcal{N}$ is the edge set such that (i) each edge has an ordinal $o_i \in O$ to represent ordering among the children; (ii) there is a path between every pair of nodes; (iii) there is no cycle among the edges; and (iv) every edge has a single incoming edge, except r, which has no incoming edge, (c) Σ is an *alphabet* of labels and text values, (d) $\mathbb{L} : \mathcal{N} \rightarrow \Sigma$ is a *label function* that maps each node to its label, (e) $\mathbb{F} : \mathcal{N} \rightarrow \Sigma \cup \{\epsilon\}$ is a value function that maps a node to its value, in which $\mathbb{F}(n) = \epsilon$ if node n has an empty value, and (f) $\mathbb{T} : \mathcal{N} \rightarrow \mathcal{S}$ is a *type function* that maps each node to a *type* within the *type set* $\mathcal{S}$.

This simple model, which is sufficient for this paper, ignores comments, attributes, processing instructions and namespaces. The model distinguishes between *labels* and *types*. The *label function* maps each node to its label, that is, its element tag. The *type* function specifies the type of each node, where two nodes with the same label could have different types. The type could be defined in various ways, we assume only that each node has a known type. In this paper, the type of a node $n \in \mathcal{N}$ is defined as the *root-to-node path* of n (*i.e.,* the concatenation of the labels on the path from the root to n). For example, suppose that there exist `name` nodes in subtrees rooted at `team` and `player` nodes. Then the path from the root to a team `name` node and a player `name` node differs; therefore they are of different types.

3.1 Intuition

Given a context node, the *node locality* (locality for brevity) is the set of nodes that are *semantically related* to the context node. A node within the locality is called a *local node*. For example, the filled nodes in Figure 1(a) depict the locality of the first `team` node (*New York Knicks*). For instance, the `league` node describes Knicks' league. The two `player` nodes are Knicks' players. The `name` node *Walsh* is Knicks' head manager. Notably, the context node itself is also within the locality.

A key characteristic of node locality is that it is *structure-independent*. That is, when the document structure changes[1], the locality does not change. For instance, all local nodes of team *New York Knicks* in Figure 1(a) are also local nodes of *New York Knicks* in Figure 1(b). Observe that when the document structure changes, the position of all `player` nodes change but the locality of the `team` node still contains these two `player` nodes.

3.2 Defining Node Locality

Based on the aforementioned discussion, it is evident that a key issue associated with node locality is the identification of local nodes for a context node. In [15], we introduced the notion of locality as follows. A node n whose $\mathbb{T}(n) = t_n$ is *local* to the context node c whose $\mathbb{T}(c) = t_c$ if, among all pairs of nodes with type t_n and type t_c respectively, the *distance*[2] of n to c is minimum. That is, n is local to c *iff* $Distance(n, c) = min\{Distance(n', c')|c', n' \in \mathcal{N}, \mathbb{T}(n') = \mathbb{T}(n), \mathbb{T}(c') = \mathbb{T}(c)\}$.

[1] In this paper, we assume that the original and modified documents must have *same* content, *same* element labels, and real-world semantic relationships are maintained in both documents.

[2] The *distance* between nodes u and c is the number of edges in the unique, simple undirected path between u and c.

Note that based on this definition we can identify all the local nodes of the `team` node in Figures 1(a) and 1(b).

Although the aforementioned definition of local nodes works for many cases, for certain scenario it may fail to identify the local nodes correctly. Let us illustrate this by modifying the documents in Figure 1. Assume that there exists a `predecessor` node with value *San Diego Clippers* as a fourth child of the second `team` node in Figure 1(a). Similarly, the `predecessor` node is added as the second child of the second `team` node in Figure 1(b). Let us now consider the context node to be the `championship` node. Regardless of the structure of the XML document, the locality of a championship should include the team, the managers, the players, the league and the predecessor. Observe that the aforementioned definition of node locality now identifies the `predecessor` node (*San Diego Clippers*) as one of its local node. Semantically, *San Diego Clippers* is the `predecessor` node of *Los Angeles Clippers* and is not related to the `championship` node of *New York Knicks*. Hence, the locality of Knicks' `championship` node should exclude this `predecessor` node.

The reason the locality definition of [15] fails is because both `championship` and `predecessor` are *optional* nodes in this example. In fact, there is *only one* `championship` node and *only one* `predecessor` node. Therefore, they are clearly at the minimum distance of the pair of their types. Consequently, the `predecessor` node is always local to the `championship` node. In the following, we present a novel definition of node locality that addresses this limitation.

We first introduce some terminology to facilitate our exposition. For each type $t \in \mathcal{S}$ where $\mathcal{S}$ is the set of all types in the XML document $\mathbb{D}$, the *sub-type set* of t, denoted as S_t, is a subset of $\mathcal{S}$ including the types of all child nodes of all nodes with type t in $\mathbb{D}$. That is, $S_t = \{t' \in \mathcal{S} | \exists n, n' \in \mathcal{N}, n = Parent(n'), \mathbb{T}(n) = t, \mathbb{T}(n') = t'\}$. For example, considering the type `team`. There are two teams in the modified version of Figure 1(a). The child nodes of the first team (*New York Knicks*) are of types `name`, `player`, `manager`, `championship` and `@founded`. The child nodes of second one (*Los Angeles Clippers*) are of types `name`, `player`, `manager`, `predecessor` and `@founded`. Therefore, $S_{team} = \{name, player, manager, championship, predecessor, @founded\}$. Note that S_t can be computed while parsing the XML document if schema is not available. Otherwise, it can be computed from its schema/DTD.

In an XML document, a node n whose type is t is called a *full node*, denoted as $FullNode(n)$, if for all types in S_t, n has at least one child of that type. That is, $\forall t' \in S_t, \exists n' \in \mathcal{N}, n = Parent(n') \wedge \mathbb{T}(n') = t'$. If we denote the set of all types of all child nodes of n as T_n, then the above definition is equivalent to: $FullNode(n) = \texttt{true} \iff S_t \subseteq T_n$. Moreover, it is obvious that $T_n \subseteq S_t$. Therefore, $FullNode(n) = \texttt{true} \iff S_t = T_n$. For example, since S_{player} consists of only player's `name` and all nodes of type `player` in the modified document of Figure 1(a) has a child node with type player's `name`, all of them are full nodes. On the other hand, both `team` nodes are not full.

An XML document $\mathbb{D}$ is considered *full* (denoted by $Full(\mathbb{D})$) *iff* all of its nodes are full. That is, $\forall n \in \mathcal{N}$, $FullNode(n) = \texttt{true}$. Clearly in a full document, there are no optional nodes. Consequently, in a full document we can use minimum distance to

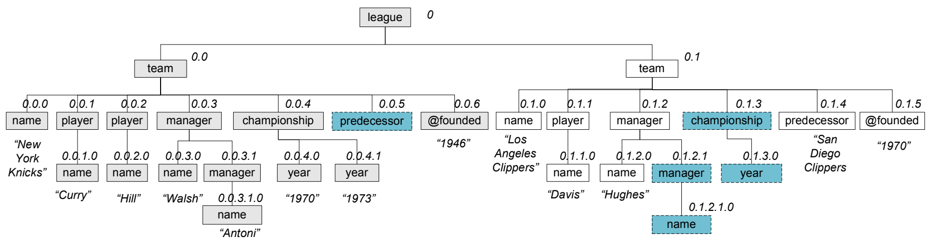

league 0
team 0.0
team 0.1
0.0.0 name "New York Knicks"
0.0.1 player 0.0.1.0 name "Curry"
0.0.2 player 0.0.2.0 name "Hill"
0.0.3 manager 0.0.3.0 name "Walsh" 0.0.3.1 manager 0.0.3.1.0 name "Antoni"
0.0.4 championship 0.0.4.0 year "1970" 0.0.4.1 year "1973"
0.0.5 predecessor
0.0.6 @founded "1946"
0.1.0 name "Los Angeles Clippers"
0.1.1 player 0.1.1.0 name "Davis"
0.1.2 manager 0.1.2.0 name "Hughes" 0.1.2.1 manager 0.1.2.1.0 name
0.1.3 championship 0.1.3.0 year
0.1.4 predecessor "San Diego Clippers"
0.1.5 @founded "1970"

Fig. 2. Full document of `predecessor`-enhanced version of Figure 1(a)

identify node locality. That is, a node n whose $\mathbb{T}(n) = t$ is local to the context node c if, among all nodes with type t, the distance of n to c is minimum.

Definition 1. ***[Node Locality]*** *The locality of a node c in an* XML *document* $\mathbb{D}$, *denoted as* $Locality(c)$, *is a set of nodes in the full document* $Full(\mathbb{D})$ *in which each node* n *satisfies the following conditions: (a)* n *is in* $Full(\mathbb{D})$; *(b)* $\forall n'$ *in* $Full(\mathbb{D})$, $\mathbb{T}(n') = \mathbb{T}(n) \implies Distance(c, n') \geq Distance(c, n)$.

Next, we shall discuss how to convert any XML document $\mathbb{D}$ to a corresponding full document $Full(\mathbb{D})$. It is achieved by adding *ghost nodes* to $\mathbb{D}$. For each node n with type t and each type $t' \in S_t$ that n has no child nodes of type t', a new node n' with type t' is conceptually added to $\mathbb{D}$ as a child node of n. n' is called a *ghost node* since it does not actually exists in $\mathbb{D}$. For example, in the modified version of Figure 1(a), a `predecessor` node (ghost node) is added as the child of the first `team` node and a `championship` node is added as the child of the second `team` (*Los Angeles Clippers*). Notably, since $S_{championship}$ has type `year`, the `championship` node also has a ghost node `year`. Note that only one `year` node is sufficient to make the `championship` node full. Similarly, an assistant `manager` node (with accompanying `name` node) is added as the child of the second `manager` node (*Hughes*). Figure 2 depicts the full document (ghost nodes are shown in dotted blue rectangles). Note that no value nodes are added in the transformation as a full document does not require value nodes.

We can now compute locality of a node correctly using Definition 1. For instance, the `predecessor` node is no longer optional. Therefore, the `predecessor` node (*San Diego Clippers*) is now excluded from the locality of Knicks' `championship` node. Instead, the `predecessor` node with minimum distance to the context node is now the corresponding ghost node, which can be filtered out in the final results.

Definition 1 offers a straightforward method to compute the locality of a node c in document $\mathbb{D}$ by converting $\mathbb{D}$ to $Full(\mathbb{D})$ and finding the nodes in $Full(\mathbb{D})$ with minimal distance to c. However, this naive method has two drawbacks. First, $Full(\mathbb{D})$ is less space-efficient than $\mathbb{D}$ and may require updates every time $\mathbb{D}$ is updated. Second, locating the nodes with minimal distance to c requires traversal all nodes in $Full(\mathbb{D})$ at least once which is expensive when $Full(\mathbb{D})$ is large. In Section 5, we shall address these drawbacks by adopting an efficient strategy to evaluate the locality of c without transforming $\mathbb{D}$ to $Full(\mathbb{D})$ and with minimal node traversal.

4 Neighborhood Axis

The `neighborhood` axis is used to select *common* nodes of two context nodes. Informally, a node that is common to two nodes is semantically related to these nodes even when the document structure changes. Recall that the locality of a node is the set of all related (local) nodes to the context node. Therefore, the common nodes selected by the `neighborhood` axis should be in the locality of *both* input nodes. Observe that since node locality is structure-independent, the common locality of the two context nodes are identical even when structure of the document changes. For example, if John is interested in the common team of *Hill* and *Antoni*, the result should be the only `team` node in the common locality (team *New York Knicks*). On the other hand, if John asks for common `name` nodes associated to *Hill* and *Antoni*, then this query is ambiguous. In this case, the `neighborhood` axis should return all `name` nodes in the common locality of these two nodes.

Definition 2. *Let c_1 and c_2 be two context nodes and ℓ be the name test of the step. The* ***neighborhood*** *nodes of c_1 and c_2 with label ℓ, denoted as $neighborhood(c_1, c_2, \ell)$, is a list of nodes $[n_1, n_2, \dots, n_j]$ where:*

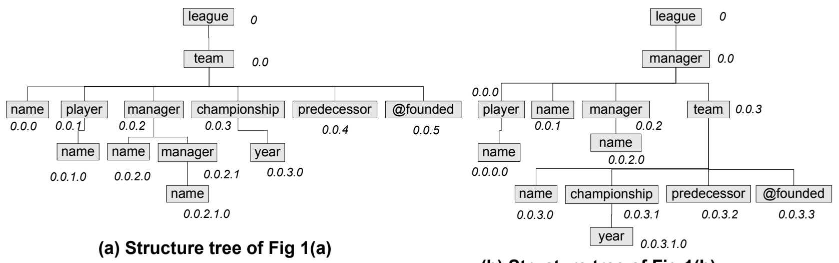
- $n_1, n_2, \dots, n_j \in N$ *and* $j \geq 1$
- $\forall n_i \in neighborhood(c_1, c_2, \ell), \mathbb{L}(n_i) = \ell$
- $\forall n_i \in neighborhood(c_1, c_2, \ell), n_i \in Locality(c_1) \wedge n_i \in Locality(c_2)$
- $\forall p, q, 1 \leq p < q \leq j$, n_p *precedes* n_q *in document order*

The syntax for expressing `neighborhood` axis should consist of two input nodes (context nodes). One of them is the context node specified by the previous step. We refer to it as *left* context node. The other input node is a parameter (can be expressed as path expression), which we refer to as *right* context node. Hence, the BNF rules for `neighborhood` axis is as follow. First, the `neighborhood` is added into "$NonDirectionalStep$"[3]. Next, additional rules are specified to describe the `neighborhood` axis.

$$\begin{aligned} NonDirectionalStep &::= ClosestStep | RankDistanceStep | NeighborhoodStep \\ NeighborhoodStep &::= NeighborhoodAxis \quad NodeTest \\ NeighborhoodAxis &::= \texttt{"neighborhood""\{"} PathExpression \texttt{"\}"} \quad \texttt{"::"} \end{aligned}$$

Reconsider Figure 1 to find the common team of players *Hill* and *Curry*. Then, this query can be formulated as follows: Q_5: `//player [name='Hill']/neighborhood{//player[name='Curry']}::team`. Observe that the query consists of three parts: (a) `//player[name='Hill']` is used to select the player *Hill* (left context node). (b) `//player[name='Curry']` is used to select the player *Curry* (right context node). (c) A `neighborhood` step with name test of `team` is used to find the common team of the two players. Observe that for both documents in Figure 1, Q_5 will return the `team` node whose name is "*New York Knicks*".

[3] We have introduced two additional non-directional axes, namely `closest` and `rank-distance`, in [15] and [1], respectively.



(a) Structure tree of Fig 1(a) **(b) Structure tree of Fig 1(b)**

Fig. 3. Structure trees of modified versions of the XML documents in Figure 1

The `neighborhood` axis can also be used with a predicate. Suppose John now wishes to find the teammates of player *Hill*. Intuitively, a teammate is a player in the same team. Therefore, the XPath for this query can be expressed as follows: Q_6: `//player[neighborhood{//player[name='Hill']}::team]`. This query will return the `player` node whose name is "*Curry*" in both Figures 1(a) and 1(b).

5 Evaluation of Neighborhood Axis

In this section, we present a generic algorithm called SINBAD for evaluating neighborhood axis by exploiting node locality information. We begin by briefly introducing the terminology that we shall subsequently to describe the algorithm.

We denote the root-to-node path of a node n in an XML tree as $Path(n)$. That is, $Path(n)$ is a concatenation of the labels on the path from the root to n. Observe that $Path(n)$ is equivalent to the type of n ($\mathbb{T}(n)$). In the sequel, we shall use these two concepts interchangeably. Next we define the notion of *structure tree*. Given an XML document $\mathbb{D}$, the *structure tree* of $\mathbb{D}$, denoted as $\mathbb{S}_D$, is a DataGuide structural summary [5] representing all unique paths in $\mathbb{D}$. That is, each unique path p in $\mathbb{D}$ is represented in $\mathbb{S}_D$ by a node whose path from the root node to this node is p. Further, every unique label path of $\mathbb{D}$ is described exactly once, regardless of the number of times it appears in $\mathbb{D}$. The structure tree encodes no path that does not appear in $\mathbb{D}$. Note that a structure tree can be computed in linear time for tree-structured data [5]. Figure 3 depicts the structure trees of the modified `predecessor`-enhanced documents in Figure 1 (the nodes are encoded with their Dewey labels). Intuitively, a document $\mathbb{D}$ and its full document version $Full(\mathbb{D})$ share a common structure tree.

Lemma 1. *Given a document $\mathbb{D}$, the structure trees of $\mathbb{D}$ and $Full(\mathbb{D})$ are identical.*

Proof. (Sketch) According to the definition of full documents, for any type $t \in \mathcal{S}$, its subtype set is identical in $\mathbb{D}$ and in $Full(\mathbb{D})$. In the structure tree, each type t corresponds to a path p and each subtype of t corresponds to a child of p. Thus, in both $\mathbb{D}$ and $Full(\mathbb{D})$, the children list of all nodes in the structure tree are identical. Hence, their structure trees are identical. ∎

For example, Figure 3(a) depicts the structure tree of both the document in Figure 1(a) and its full document in Figure 2.

Given two paths p_1 and p_2 in $\mathbb{S}_D$, the *path distance* between p_1 and p_2, denoted as $Distance(p_1, p_2)$, is the length of path connecting the nodes represented by p_1 and p_2. The *level* of p_1, denoted as $Level(p_1)$, is the length of p_1. The LCA of p_1 and p_2, denoted as $LCA(p_1, p_2)$, is the longest common prefix of p_1 and p_2. Finally, the LCA *level* of p_1 and p_2, denoted as $LCALevel(p_1, p_2)$, is the level of the $LCA(p_1, p_2)$. That is, $LCALevel(p_1, p_2) = Level(LCA(p_1, p_2))$.

5.1 Evaluation of Locality

In this section, we present a set of properties that shall be exploited by Algorithm SINBAD (Section 5.2) to efficiently check whether a node is in the locality of a context node.

Lemma 2. *Let n_1 and n_2 be two nodes in a document $\mathbb{D}$ and $p_1 = Path(n_1)$ and $p_2 = Path(n_2)$ in structure tree $\mathbb{S}_D$. Then,*

$$Distance(n_1, n_2) = Level(n_1) + Level(n_2) - 2 \times LCALevel(n_1, n_2)$$
$$Distance(p_1, p_2) = Level(p_1) + Level(p_2) - 2 \times LCALevel(p_1, p_2)$$

Proof. (Sketch) The path connecting two nodes n_1 and n_2 in a tree is unique and this path must pass through the $LCA(n_1, n_2)$. Therefore, we can divide this path into two parts: one from n_1 to $LCA(n_1, n_2)$ and another from $LCA(n_1, n_2)$ to n_2. The length of the path from n_1 to $LCA(n_1, n_2)$ is equal to $Level(n_1) - LCALevel(n_1, n_2)$ while the length of the path from $LCA(n_1, n_2)$ to n_2 is $Level(n_2) - LCALevel(n_1, n_2)$. Hence, $Distance(n_1, n_2)$ is equal to the sum of these two subparts and is equal to $Level(n_1)$ + $Level(n_2) - 2 \times LCALevel(n_1, n_2)$. The proof is similar for $Distance(p_1, p_2)$. ∎

Theorem 1. *Let n_1 and n_2 be two nodes in a document $\mathbb{D}$. Let $p_1 = Path(n_1)$ and $p_2 = Path(n_2)$ be two paths in $\mathbb{S}_D$. Then, (i) $LCALevel(n_1, n_2) \leq LCALevel(p_1, p_2)$ and (ii) $Distance(n_1, n_2) \geq Distance(p_1, p_2)$.*

Proof. (Sketch) From Lemma 2, it is clear that (i) and (ii) are equivalent (notice that $Level(n) = Level(path(n)) \forall n$). Thus, we only need to prove (i).

Let n be $LCA(n_1, n_2)$ and $p = Path(n)$. Since n is an ancestor of both n_1 and n_2, p is a prefix of both p_1 and p_2. Therefore, p is an ancestor of both p_1 and p_2. Based on the definition of LCA level, we have: $Level(p) \leq LCALevel(p_1, p_2)$. Therefore: $LCALevel(n_1, n_2) = Level(n) = Level(p) \leq LCALevel(p_1, p_2)$. ∎

For example, in Figure 2, let n_1 and n_2 be nodes whose Dewey codes are `0.0.1` and `0.1.4`, respectively. Then $LCALevel(n_1, n_2) = 1$ and $Distance(n_1, n_2) = 4$. In Figure 3(a), the paths of n_1 and n_2 (p_1, p_2) are nodes whose Dewey codes are `0.0.1` and `0.0.4`, respectively. Their LCA level is 2 and their distance is 2. Hence, $LCALevel(n_1, n_2) < LCALevel(p_1, p_2)$ and $Distance(n_1, n_2) > Distance(p_1, p_2)$.

Theorem 2. *Let c be a node in the full document $Full(\mathbb{D})$ ($Path(c) = p_c$). Then, for any path p_n in $\mathbb{S}_D$, there exists a node $n \in Full(\mathbb{D})$ whose path is p_n such that (i) $LCALevel(c, n) = LCALevel(p_c, p_n)$ and (ii) $Distance(c, n) = Distance(p_c, p_n)$.*

Proof. (Sketch) Following Lemma 2, (i) and (ii) are equivalent. Thus, we only need to prove (i). Let $\ell = LCALevel(p_c, p_n)$. Note that $\ell \leq Level(p_c) = Level(c)$. Let a_ℓ be the ancestor (or self) of c at level ℓ. Choose $a_{\ell+1}$ as a child of a_ℓ whose tag is equal to the tag of p_n at level $\ell + 1$ of the structure tree $\mathbb{S}_D$. Since $\ell = LCALevel(p_c, p_n)$, this tag is not in p_c and therefore, is not the tag of the ancestor of c at level $\ell + 1$. Therefore, the chosen $a_{\ell+1}$ is not the ancestor of c at level $\ell + 1$.

Similarly, $a_{\ell+2}$ is selected as a child of $a_{\ell+1}$ whose tag is the tag of p_n at level $\ell + 2$. Continue this process repeatedly until $k = Level(p_n)$ where a_k is the node n that we need to find. Obviously, $Path(a_k) = p_n$. Moreover, since a_ℓ is a common ancestor at level ℓ of c and a_k and their ancestors at level $\ell+1$ are different (since they have different tags), $a_\ell = LCA(c, a_k)$. Hence, $\ell = LCALevel(c, a_k)$. So, $LCALevel(p_c, p_n) = \ell = LCALevel(c, a_k)$. ∎

For example, consider the full document in Figure 2 and the corresponding structure tree in Figure 3(a). Let c be the node whose Dewey code is `0.0.3`. Hence, p_c is `league.team.manager` which has a Dewey code of `0.0.2` in the structure tree. Let's choose p_n as `league.team.predecessor` (Dewey code is `0.0.4` in Figure 3(a)). In the structure tree, $LCALevel(p_c, p_n) = 2$. Then $p_2 = LCA(p_c, p_n)$ is the path `league.team` representing the type `team`. Observe that the level 2 ancestor, a_2, of c is the node `0.0` (clearly, the path of a_2 is p_2).

The tag at level 3 of p_n is `predecessor`. Let p_3 be the level 3 ancestor of p_n. Then p_3 is `league.team.predecessor`. Since p_3 is a child of p_2 in the structure tree, there exists nodes which have path p_3 and are the child nodes of nodes with path p_2 in the full document. Hence, type `predecessor` is a type in the sub-type set of type `team`. Consequently, a_2 must have a child node with type `predecessor` (path p_3). Let a_3 be this child node of a_2. Then a_3 is the node n we need to find. Observe that a_3 is a ghost node in Figure 2 with Dewey code of `0.0.5` and $LCALevel(c, a_3) = LCALevel(p_c, p_n) = 2$ and $Distance(c, a_3) = Distance(p_c, p_n) = 2$.

Given a context node c whose path is p_c and a set of nodes N in a full document $Full(\mathbb{D})$ whose path is p_n, Theorem 1 shows that $Distance(p_c, p_n)$ is a lower bound of the distance between c and any nodes $n \in N$. On the other hand, Theorem 2 shows that $\exists n \in N, Distance(c, n) = Distance(p_c, p_n)$. Thus, $Distance(p_c, p_n)$ is the minimum distance between c and any nodes in N. Following Definition 1, all nodes $n \in Locality(c)$ must satisfy $Distance(c, n) = Distance(p_c, p_n)$.

Theorem 3. *Let c be the context node in document $\mathbb{D}$. Then a node $n \in Locality(c)$ iff (a) $Distance(c, n) = Distance(p_c, p_n)$ or (b) $LCALevel(c, n) = LCALevel(p_c, p_n)$.*

Proof. (Sketch) Condition (a) is a direct consequence of Definition 1, Theorem 1 and Theorem 2. Condition (b) can be proved by exploiting Lemma 2. Since $Distance(c, n) = Distance(p_c, p_n)$, $Level(c) + Level(n) - 2LCALevel(c, n) = Level(p_c) + Level(p_n) - 2LCALevel(p_c, p_n)$. Hence, $LCALevel(c, n) = LCALevel(p_c, p_n)$. ∎

For example, reconsidering Figure 2. Let c be the node whose Dewey code is `0.0.3` (the head `manager` node). Let us find the `player` nodes that are local to c. Here p_c = `league.team.manager` and p_n = `league.team.player`. Therefore, $LCALevel(p_c, p_n) = 2$. Observe that in Figure 2, there are three nodes having path p_n.

Algorithm 1: The Algorithm SINBAD.

```
Input: A document $\mathbb{D}$, context nodes $c_1$ and $c_2$ in $\mathbb{D}$, name test $\ell$, set $P$ of all paths in $\mathbb{S}_D$
Output: A set of neighborhood nodes $Results$
1  Initialize $NeighborhoodPaths = \infty$ ;
2  Initialize $Results = \infty$ ;
3  for (each $p \in P$) do
4      if ($LastTag(p) == \ell$) then
5          Add $p$ into $NeighborhoodPaths$;
6  for $p \in NeighborhoodPaths$ do
7      $a_1 \leftarrow$ the ancestor (or self) of $c_1$ at level $LCALevel(Path(c_1), p)$;
8      $a_2 \leftarrow$ the ancestor (or self) of $c_2$ at level $LCALevel(Path(c_2), p)$;
9      if $a_1$ is an ancestor-or-self of $a_2$ or $a_2$ is ancestor-or-self of $a_1$ then
10         $a \leftarrow$ the descendant between $a_1$ and $a_2$;
11         Add all descendants-or-self of $a$ in $\mathbb{D}$ whose path is $p$ into $Results$;
12 return $Results$
```

Their Dewey codes are `0.0.1`, `0.0.2`, and `0.1.1`. Their $LCALevel$ with c are 2, 2, and 1, respectively. Hence, according to Theorem 3 only nodes `0.0.1` and `0.0.2` are in the locality of c. Observe that these two nodes represent the players *Curry* and *Hill* who are managed by the head manager *Walsh*.

Remark. Theorem 3 offers a very efficient technique to evaluate locality due to three reasons. First, $LCALevel(p_c, p_n)$ can be computed completely from the structure tree $\mathbb{S}_D$ whose size is significantly smaller than $\mathbb{D}$ in most practical cases. $\mathbb{S}_D$ can also be built directly from $\mathbb{D}$ without creating $Full(\mathbb{D})$ (Lemma 1). Second, due to Theorem 1, $LCALevel(c, n) = LCALevel(p_c, p_n) = \ell$ is equivalent with n is a descendant (or self) of the ancestor-or-self node a of c at level ℓ. Notice that if our list of nodes are sorted in document order, the descendant list of a are consecutive and usually small so that traversing them is usually cheap. Third, observe that users are not interested in ghost nodes. Hence, these nodes need to be filtered out in the output. Theorem 3 allows us to accomplish this efficiently *without transforming the original document* $\mathbb{D}$ *to a full document*. We only traverse the descendant list of a in the original document $\mathbb{D}$ so that all result nodes are actual nodes. If a local node n (*i.e.,* descendant of a) in the full document is not returned, it means that n is a ghost node.

For example, let c be the node with Dewey code `0.0` (`team` node). We wish to find the `predecessor` nodes that are local to c. Since p_c = `league.team` and p_n = `league.team.predecessor`, $LCALevel(p_c, p_n) = 2$. The ancestor-or-self of c at level 2 is c itself. In the full document there are two nodes with type p_n (`0.0.5` and `0.1.4`). However, since we only need to traverse the descendant list of node `0.0` in the original document $\mathbb{D}$, neither would be traversed and there does not exist any `predecessor` node that is in the locality of c in the original document (Fig.1). Notably, although node `0.0.5` is descendant of node `0.0` and local to c in the full document, since we traverse only `0.0`'s descendant list in the original document (we do not transform it to a full document), node `0.0.5` would not be returned.

5.2 Algorithm SINBAD

Algorithm 1 outlines the SINBAD algorithm. Most importantly, Algorithm 1 does not take $Full(\mathbb{D})$ as input or require $Full(\mathbb{D})$ in any of its steps. We illustrate the steps with an example. Reconsider the query Q_5 which selects the common team of player *Hill* and manager *Antoni*. The two context nodes in this query are the `player` node (`0.0.2`) and the `manager` node (`0.0.3.1`). Lines 3-5 are used to find all paths whose last tag is the label `team`. In our example, the only such path is `league.team`. For each path $p \in NeighborhoodPaths$, Lines 7-8 are used to find the ancestor of c_1 and c_2 at level $LCALevel(p, path(c_1))$ and $LCALevel(p, path(c_2))$, respectively. According to Theorem 3, all result nodes must be descendants of *both* ancestor nodes a_1 and a_2. Hence, a_1 and a_2 must have ancestor-descendant relation (Line 9). Let a be the descendant node between a_1 and a_2 (Line 10), all results must be descendants (or self) of a (Line 11). Notice that Line 11 finds the descendants of a in $\mathbb{D}$, not $Full(\mathbb{D})$, due to reasons mentioned in Section 5.1. Therefore, Algorithm 1 does *not* require $Full(\mathbb{D})$ or convert $\mathbb{D}$ to $Full(\mathbb{D})$. Specifically, for our example, both a_1 and a_2 would be node `0.0` and the only descendant-or-self nodes with label `team` and descendant of `0.0` are that node itself. It is our only result.

Time Complexity. Let k be the number of paths satisfying the neighborhood path condition (Lines 3-5) and the $Desc(p)$ be the set of descendant-or-self node of node a produced in Lines 10-11. Assume that $LCALevel()$ and ancestor-descendant evaluation could be computed in $O(1)$ time. The time complexity of the algorithm is: $O(|P| + 3k + \sum_{p \in P} |Desc(p)|)$. Notice that both $|P|$ and k are usually very small so that the worst-case complexity is usually dominated by $\sum_{p \in P} |Desc(p)|$. Furthermore, since all nodes in $Desc(p)$ have path p, all nodes in $Results$ are unique and $\sum_{p \in P} |Desc(p)|$ happens to be our result size which is the expected lower bound of our algorithm. Moreover, we can also notice that the input context nodes are only used for LCA computation and ancestor-descendant checking, both can be achieved using only the node identifiers (*e.g.,* Dewey code). Hence, retrieving data for a context node is cheaper than retrieving data for a result node.

6 Performance Study

We present the experiments conducted to evaluate the performance of our proposed axis and report some of the results obtained. Note that SINBAD is independent of any specific storage scheme for XML data. In this paper, we have realized it on top of a *path materialization*-based [6] relational storage scheme called SUCXENT++ [10] in Java JDK1.6. Due to space constraints, we do not discuss the implementation of SQL translation strategy for SINBAD. Note that our implementation *does not* require any invasion of the database kernel. All of our experiments are conducted on an Intel Core 2 Quad CPU 2.66GHz machine running on Windows XP Service Pack 3 with 2GB RAM. The RDBMS used was MS SQL Server 2008 Developer Edition.

Id	Size	No. of Attributes	No. of Leaf Elements	Max Depth
DC10	11MB	15,000	152,673	8
DC100	111MB	150,000	1,520,322	8
DC1000	1111MB	1,500,000	15,215,868	8

(a) XBENCH data sets

Id	Size	No. of Attributes	No. of Leaf Elements	Max Depth
U28	28MB	771,877	422,972	6
U284	284MB	7,791,617	4,230,003	6
U2843	2843MB	77,977,270	42,421,745	6

(b) UniProt data sets

Fig. 4. Datasets

Id	Query	No. of right context nodes			No. of result nodes		
		DC10	DC100	DC1000	DC10	DC100	DC1000
XQ1	//subject/neighborhood{//title}::ISBN	2500	25000	250000	2500	25000	250000
XQ1'	//item[subject][title]//ISBN						
XQ2	//subject/neighborhood{//title}::name	2500	25000	250000	23385	232290	2375909
XQ2'	//item[subject][title]//name						
XQ3	//subject/neighborhood{//author/date_of_birth}::ISBN	6295	62420	625303	2500	25000	250000
XQ3'	//item[subject][//author/date_of_birth]//ISBN						
XQ4	closest::subject/neighborhood{closest::phone_number}::ISBN	2500	25000	250000	2500	25000	250000
XQ4'	//item[subject][publisher//phone_number]//ISBN						

(a) XBench query sets

Id	Query	No. of right context nodes			No. of result nodes		
		U28	U284	U2843	DC10	DC100	DC1000
UQ1	//protein/neighborhood{//feature}::evidence	150255	1382599	13820779	9618	74778	725820
UQ1'	//entry[//protein][//feature]//evidence						
UQ2	//protein/neighborhood{//feature}::reference	150255	1382599	13820779	411665	4084917	40821754
UQ2'	//entry[//protein][//feature]//reference						
UQ3	//protein/neighborhood{//entry/dbReference}::evidence	347754	3601383	36053637	9510	74136	720072
UQ3'	//entry[//protein][dbReference]//evidence						
UQ4	closest::protein/neighborhood{closest::gene}::evidence	13000	137292	1378484	9024	69924	678378
UQ4'	//entry[//protein][gene]//evidence						

(b) UniProt query sets

Fig. 5. Querysets

Data and Query Sets. We use XBench DCSD [14] as a synthetic data set and Uniprot/KB XML[4] as a real-world data set. We vary the size of XML documents from 10MB to 1GB for XBench DCSD data set. Since the original UNIPROT data is 2.8GB in size (denoted as U2843), we also truncated this document into smaller XML documents of sizes 28MB and 284MB (denoted as U28 and U284, respectively) to study scalability. Figure 4 depicts the characteristics of the data sets. Figure 5 depicts the query sets for XBench DCSD (XQ1–XQ4) and Uniprot/KB data sets (UQ1–UQ4). Note that queries XQ4 and UQ4 showcase usage of `neighborhood` axis with other non-directional axis (*i.e.,* `closest` [15]). We also consider equivalent directional XPath queries (same results set) of XQ1–XQ4 (denoted as XQ1′ - XQ4′) and UQ1–UQ4 (UQ1′ - UQ4′) in order to compare the performance of structure-independent queries with their directional counterparts.

Test Methodology. Appropriate indexes were constructed for SUCXENT++. Prior to our experiments, we ensured that statistics had been collected. The bufferpool of the

[4] Downloaded from `ftp://ftp.uniprot.org/pub/databases/uniprot/current_release/knowledgebase/complete/uniprot_sprot.xml.gz`

Id	DC10	DC100	DC1000
XQ1	119	419	937
XQ1'	126	432	972
XQ2	306	1779	8436
XQ2'	309	1777	8261
XQ3	144	449	1003
XQ3'	157	481	1021
XQ4	117	376	891
XQ4'	128	394	925

(a) XBench

Id	U28	U284	U2843
UQ1	170	390	2285
UQ1'	189	401	2329
UQ2	520	5620	35834
UQ2'	509	5565	35397
UQ3	168	513	2817
UQ3'	180	518	2819
UQ4	171	322	1248
UQ4'	168	328	1272

(b) Uniprot

Fig. 6. Performance results (in msec.)

RDBMS was cleared before each run. Each query was executed six times and the results from the first run were always discarded.

Experimental Results. Figure 6 reports the performance `neighborhood` axis queries. We can make the following observations. Firstly, *the execution time increases sub-linearly with result size*. Notice that XQ1 and UQ1 have identical context node set with XQ2 and UQ2, respectively, but with different result sizes. In particular, the result size of XQ2 is nearly 10 times larger than that of XQ1 and the result size of UQ2 is about 50 times larger than that of UQ1. However, the execution times grows at much slower rate compared to the result size. Secondly, *the execution time increases sub-linearly with the number of right context nodes* (recall from Section 4). For instance, although the numbers of right context nodes of XQ3 and UQ3 are significantly larger than XQ1 and UQ1, respectively, there is not much difference in the corresponding execution times. This is consistent with our discussion in Section 5.2. Thirdly, *our proposed technique is scalable* as the execution time increases linearly to the document size. Notice that for 75% of queries, the execution time on even the largest dataset is less than 3 seconds. Lastly, there are *no significant performance difference between the neighborhood axis queries and their corresponding directional counterparts* (XQ$1'$ - XQ$4'$ and UQ$1'$ - UQ$4'$). This highlights the strength of our approach as users can query in a structure-independent manner without compromising on query performance.

7 Conclusions and Future Work

The quest for structure-independent querying of XML data has become more pressing due to inability of end-users to be aware of structural details of underlying data. In this paper, we present a novel structure-independent, non-directional XPath axis, called the `neighborhood` axis, to locate common neighbors of two context nodes. We proposed an algorithm called SINBAD that exploits the notion of node locality and small size of XML structural summary to efficiently abstract the common nodes of a pair of context nodes. Our empirical study on top of an existing path materialization-based relational storage showed that SINBAD has excellent real-world performance. In future, we plan to extend our approach to yet other non-directional axes, which we believe can be supported using the techniques presented in this paper.

References

1. Bhowmick, S.S., Dyreson, C., et al.: Towards Non-Directional XPath Evaluation in a RDBMS. In: CIKM (2009)
2. Brodianskiy, T., Cohen, S.: Self-Correcting Queries in XML. In: CIKM (2007)
3. Cohen, S., Mamou, J., Kanza, Y., Sagiv, Y.: XSEarch: A Semantic Search Engine for XML. In: VLDB (2003)
4. Curino, C.A., Moon, H.J., Zaniolo, C.: Graceful Database Schema Evolution: The Prism Workbench. In: VLDB (2008)
5. Goldman, R., Widom, J.: DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In: VLDB (1997)
6. Gou, G., Chirkova, R.: Efficiently Querying Large XML Data Repositories: A Survey. IEEE TKDE 19(10) (2007)
7. Li, Y., Yu, C., Jagadish, H.V.: Schema-Free XQuery. In: VLDB (2004)
8. Liu, Z., Chen, Y.: Identifying Meaningful Return Information for XML Keyword Search. In: SIGMOD (2007)
9. Liu, Z., Chen, Y.: Reasoning and Identifying Relevant Matches for XML Keyword Search. In: VLDB (2007)
10. Soh, K.H., Bhowmick, S.S.: Efficient Evaluation of not-Twig Queries in A Tree-Unaware RDBMS. In: Yu, J.X., Kim, M.H., Unland, R. (eds.) DASFAA 2011, Part I. LNCS, vol. 6587, pp. 511–527. Springer, Heidelberg (2011)
11. Termehchy, A., Winslett, M., Chodpathumwan, Y.: How Schema Independent Are Schema Free Query Interfaces? In: ICDE (2011)
12. Velegrakis, Y., Miller, R.J., Popa, L.: Mapping Adaptation Under Evolving Schemas. In: VLDB (2003)
13. Xu, Y., Papakonstantinou, Y.: Efficient Keyword Search for Smallest LCAs in XML Databases. In: SIGMOD (2005)
14. Yao, B., Tamer Özsu, M., Khandelwal, N.: XBench: Benchmark and Performance Testing of XML DBMSs. In: ICDE (2004)
15. Zhang, S., Dyreson, C.: Symmetrically Exploiting XML. In: WWW (2006)