An Efficient Index Lattice for XML Query Optimization

James Cheng and Wilfred Ng

Department of Computer Science

The Hong Kong University of Science and Technology

Hong Kong

Email: {csjames, wilfred}@cs.ust.hk

Abstract

Structural indexes of XML data can effectively reduce the search space for the evaluation of path queries over the data. The indexes partition the structural graph of an XML document into equivalent classes of nodes that are then condensed into index nodes. However, structural indexes are inadequate to handle queries with value-based conditions, since equivalent nodes in the same partition become distinguishable by their data contents. In practice, only a small portion of nodes in each partition are relevant for the processing of a value-based condition.

To enhance the applicability of structural indexes, we propose a lattice structure on an XML structural index, which we call the *Structure Index Tree* (*SIT*). The index is defined as partitions of equivalent paths in an XML document, while an element in the lattice, which we call a *SIT-Lattice Element* (*SLE*), is an index of an arbitrary subset of paths in the document. Since paths represent the structure of XML data and each text node is associated with a unique path, we can define an SLE to filter out both irrelevant structures and text nodes. We propose a set of SLE operations and devise efficient techniques to generate SLEs that can be tailored towards query workloads. Our experiments show that SLEs significantly speed up the evaluation of path queries with value-based and aggregation-based conditions. We also demonstrate that SLEs are able to support effective querying over very large XML documents in memory-limited hand-held devices.

1 Introduction

It is well recognized that establishing an efficient index to aid in processing queries on XML data is important. Many existing structural indexes (or structural summaries) on XML or semi-structured data, such as Dataguides [12], 1-index [24], A(k)-indexes [18], D(k)-indexes [7], M(k)-indexes [16], and F&B-index [1, 17], partition nodes in the structure graph of an XML document into classes of equivalent nodes. Each class of equivalent nodes is condensed into a single index node, while the node *id*s are stored in a concise structure, called the *extent*, that is associated with the index node. These indexes can thus effectively prune the search space to speed up the evaluation of structural path queries. However, when value-based conditions are incorporated into the queries, we need also to examine the data value of each node (i.e., character data of an XML element) in an equivalent class. Hence, the indexes become less useful, since normally only a small portion of nodes in an equivalent class match a given value-based condition.

The use of a structural index to process value-based query conditions and structural path expressions is mainly hindered by two factors that are related to the size of the index: (1) huge *structure size* and (2) huge *extent size*. By structure size, we refer to the total number of nodes in the index. By extent size, we refer to, depending on whether we are addressing a node in the index or the index itself, either the number of equivalent nodes represented by the extent of the index node or the sum of the extent sizes of all the nodes in the index.

In this paper, we study the problems arising from these two factors and propose a solution by utilizing a lattice structure [14] defined on a structural index of XML data, called the *Structure Index Tree* (or the *SIT* in short) [8]. The SIT [1] is constructed based on the partitioning of equivalent paths in an XML document, while an element in the lattice is an index of an arbitrary subset of paths in the document. We call the lattice the *SIT-lattice* and its element a SIT-lattice element, or an *SLE* for short. We now explain how the SIT-lattice addresses the two size problems of the existing indexes.

[1] The SIT was first introduced in our preliminary work [8] to aid in efficient evaluation of XPath [10] queries on compressed XML data. The full version of XQzip is under reviewing by another journal.

1.1 The Huge Structure Size Problem

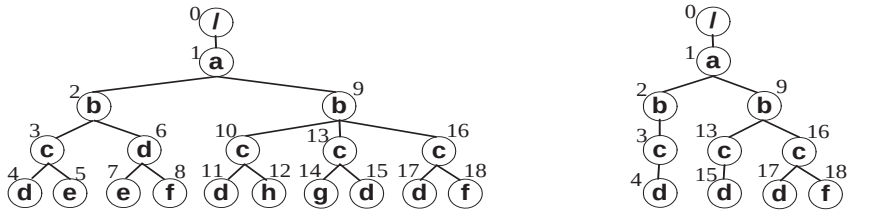
The structure size of most indexes of XML data can approach the size of the base data. For example, the structure size of the 1-index on Treebank [23] and that of the F&B-index on XMark [30] are comparable to the size of the base data. In this case, the indexes are not efficient enough to support the evaluation of structural queries.

Existing indexes, such as A(k)-indexes, D(k)-indexes and M(k)-indexes, are able to reduce their structure size by computing the k-bisimulation [26, 18] for a smaller value of k. A more flexible way of reducing the structure size of an index is the index definition scheme proposed by Kaushik et al. [17]. The index definition scheme reduces the structure size of a covering index by indexing a selected set of tags and *idref*s [1] and by computing the k-bisimulation and restricting the number of iterations of the k-bisimulation computation. These techniques tackle the structure size problem by exploiting a tradeoff between the structure size of the index and the size of the class of queries that the index is able to cover.

We address the structure size problem by considering different combinations of the *root-to-leaf* paths in the SIT. In total, there are 2^n combinations, where n is the number of leaf nodes in the SIT, and each combination constitutes an SLE. Therefore, the structure size of an SLE ranges from as small as the size of a single path to that of the full index, i.e., the SIT, which is the top of the index lattice. Compared with Kaushik et al.'s index definition scheme and other indexing techniques, our proposal of using SLEs is much more flexible and effective, since we select the index of an arbitrary combination of paths that are relevant for query evaluation. We illustrate this by the following example.

Example 1.1 Consider a full index, I, of an XML document, as shown in Figure 1(a). Suppose that we are only interested in the information of the elements "`d`" and "`f`" that are the children of "`c`" but not the siblings of "`h`". To evaluate a query of this information, our method uses the XPath 2.0 union expression [3], "`//c[`*not* `h]/(d | f)`", to specify an SLE and extract it from I, as depicted in Figure 1(b). With Kaushik et al.'s method, the minimal coverage is to select only the elements "`c`", "`d`", "`f`" and "`h`" and then check a "`c`" element by examining if it has a child, "`h`". However, this is bound to be less efficient, since

not only extra processing of the predicate is needed, but the resulting index also includes nodes such as "$\texttt{c}_{10}$", "$\texttt{d}_{6}$", "$\texttt{d}_{11}$", "$\texttt{f}_{8}$" and "$\texttt{h}_{12}$" which are irrelevant in the evaluation of a query of the required information.



(a) An Index, I (b) A Lattice Element of I

Figure 1: A Full XML Index and Its Lattice Element

1.2 The Huge Extent Size Problem

A critical problem with indexes defined based on node-equivalence partitioning is that the structure size of the index is reduced at the expense of a corresponding increase in the extent size of the index nodes. For example, consider that an XML document has 10,000 "a" elements and an A(k_L)-index that condenses the 10,000 nodes into 10 nodes, each having an extent size of 1,000. If an A(k_s)-Index, for some $k_s < k_L$, further condenses the 10 nodes into a single node, then the extent size of this single node will be increased to 10,000. Although the reduction in the structure size (from 10 nodes to 1 node) accelerates the evaluation of structural queries, such as "`//a`", for a value-based query condition, such as "`//x[a = ''some value'']`", we have to match `''some value''` with the data value of each of the 10,000 "a" elements, even though in practice there are usually few matches.

Our SIT-lattice is a well-defined structure that allows us to select only the relevant subset of nodes from the extent of an index node, since the SLE can select an arbitrary subset of paths from an XML document. We illustrate this idea of using the SLEs to accelerate query evaluation by the following example.

Example 1.2 Consider an XML document tree in Figure 2, where the attached integer of each node is its *node id*. Suppose we are only interested in the information related to the elements, "g", "h" and "i", that are descendants of a "c" element that has an "id" attribute of type "A". To evaluate queries that retrieve data of these elements, such as "`//c[@id =`

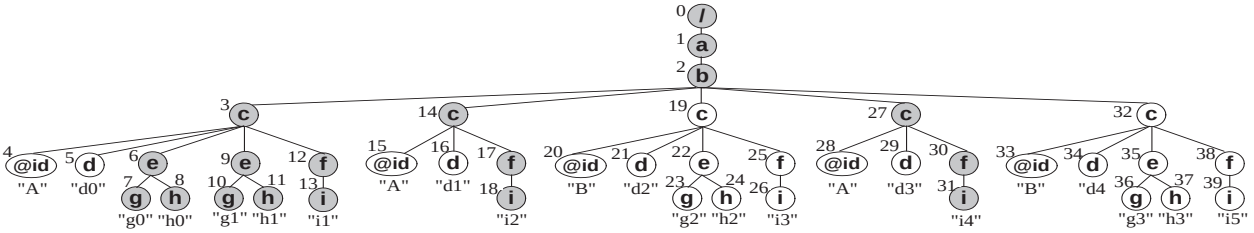


Figure 2: An XML Document Tree

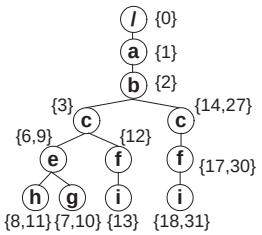


Figure 3: A Sample SIT-Lattice Element (SLE)

`‘‘A’’]//h`", we need only to access the shaded nodes in Figure 2. As we mentioned before, with the SLE, we can select a combination of paths in an XML document and the resultant SLE is an index for the selected XML data. The SLE selected for our example is shown in Figure 3, which is an index of the shaded nodes in Figure 2. Processing queries using this smaller index of relevant data is obviously more efficient than using the full index. Moreover, the SLE also pre-computes the common predicate "`[@id = ‘‘A’’]`" of the query workload.

In Figure 3, we can further combine the two equivalent paths, ⟨`c`, `f`, `i`⟩, into one; however, the collapsed index does not cover branching path expressions. For example, the "`f`" elements are not distinguishable by the query "`//c[e]/f`" with the two paths combined, but are distinguishable with the SLE in Figure 3. In fact, as we will show later, the main factor that accelerates query evaluation is the reduction in the extent size, rather than the further reduction in the structure size obtained by the coalescence of the two paths.

Another practical problem arising from the huge extent size is that in most cases the extents are too large to be loaded in the main memory of a machine. If we store the extents in a relational database then it incurs substantial disk I/O, resulting in degraded query performance. Our method can partition the full index into a set of SLEs, each of which can fit in the main memory. This approach is feasible in practice, since we usually access only a portion of the full index at a certain time.

In this paper, we make the following contributions.

- We propose a novel lattice structure on the SIT [8], which is a structural index for XML data. The lattice elements can effectively filter out irrelevant elements to accelerate query evaluation. To the best of our knowledge, our method is the first to address the problem of both the structure size and the extent size of an index on XML data.
- We specify an SLE in XPath [3]. We also present a set of heuristic rules to aid in SLE specification and three strategies to partition a full index to enable querying large XML datasets in memory-limited devices. We devise a set of lattice operations, such as union, intersection, subtraction and extraction, to obtain useful SLEs efficiently.
- We evaluate the SLEs on several benchmark datasets and a comprehensive set of queries. The results show that significant performance improvement is obtained and that using SLEs can efficiently query large XML datasets in a pocket-PC. We also show that compared with Kaushik et al.'s index definition scheme [17], the SLEs are much easier and less costly to construct and more effective in controlling both the structure size and the extent size of the XML index.

Paper Outline. We define the SIT-lattice in Section 2. We propose a syntax and a set of rules to specify an SLE, and then present the algorithms to construct an SLE in Sections 3 and 4, respectively. We evaluate the performance of the SLEs in Section 5 and discuss some related work in Section 6. Finally, we give our concluding remarks in Section 7.

2 An Index Lattice

In this section, we introduce the notion of a lattice structure on the *Structure Index Tree* (*SIT*) [8], which we call the *SIT-lattice*. We first describe the SIT and then formally define the SIT-lattice.

2.1 The XML Structure Index Tree (SIT)

The SIT is an index defined on the structure of XML data. We model an XML document as a tree[2], called the *structure tree*, $T = (V_T, E_T, \texttt{root}_T)$, where V_T and E_T are the set of

[2]We do not consider *idref*s [5] in this paper since they are comparatively rare in practice.

tree nodes and edges in T, respectively, and $\texttt{root}_T$ is the unique root of T. Each edge in E_T specifies the parent-child relationship of two nodes. Each tree node, $v \in V_T$, is defined as $v = (lid, nid, ext)$, where $v.lid$ is the unique identifier of the element/attribute label generated by a hash function; $v.nid$ is the unique node identifer assigned to v according to the document order; and ext denotes the *extent* associated with v, which contains the *nid*s of the set of equivalent nodes that are coalesced into v. We set $v.ext = \{v.nid\}$, which is later to be combined with the *ext*s of other equivalent nodes to obtain the SIT.

Each v is identified by the $(v.lid, v.nid)$ pair and the identity of $\texttt{root}_T$ is uniquely assigned to be $(0, 0)$. In addition, if v has n children $(\beta_1, \ldots, \beta_n)$, their order is specified as: (1) $\beta_1.lid \leq \beta_2.lid \leq \cdots \leq \beta_n.lid$; and (2) if $\beta_i.lid = \beta_{i+1}.lid$, then $\beta_i.nid < \beta_{i+1}.nid$. This node ordering accelerates node selection in T by an approximate factor of 2, since we match the nodes by their *lid*s and, on average, we only need to search half of the children of a node in T. As an example, Figure 4 shows the structure tree of the XML document in Figure 2.

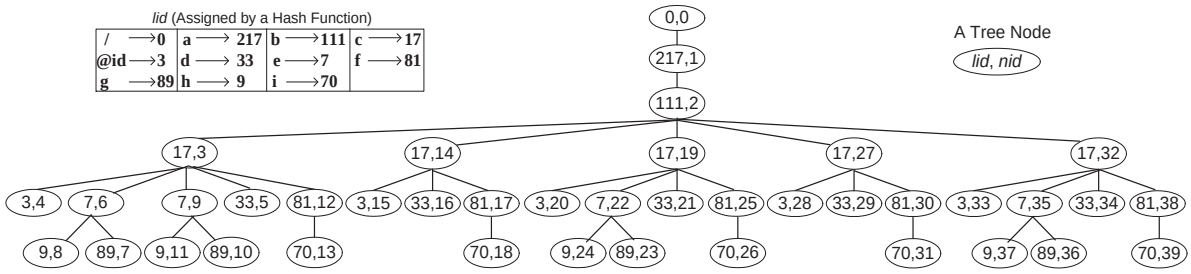


Figure 4: The Structure Tree of the XML document presented in Figure 2

Each text node in an XML document is attached to a unique path in the structure tree, which is defined by "$p = v_i \ldots v_j$", where v_j is a leaf node. We also define a path ordering as follows.

Definition 2.1 (Path Ordering) We define *path ordering*, $\prec$, as follows. Given two paths, $p_1 = u_0 \ldots u_m$ and $p_2 = v_0 \ldots v_n$, $p_1 \prec p_2$ if one of the following two conditions holds:

1. There exists some i, where $0 \leq i < min(m, n)$, such that $u_i.nid = v_i.nid$ and $u_{i+1}.nid \neq v_{i+1}.nid$, and

   1.1 $u_{i+1}.lid < v_{i+1}.lid$; or

   1.2 $u_{i+1}.lid = v_{i+1}.lid$ and $u_{i+1}.nid < v_{i+1}.nid$.

2. $u_i.nid = v_i.nid$, for $0 \leq i \leq m$ and $m = n$.

We also say that $p_1 \preceq p_2$ (or $p_2 \preceq p_1$) if the second condition in Definition 2.1 holds. While $p_1 \preceq p_2$ implies that p_1 and p_2 are the same path if they are in the same structure tree, they are different paths if they exist in two distinct structure trees. With this path ordering, we can specify a structure tree (or a structure subtree), T, as the set of all its paths ordered by $\prec$ as follows: $T = p_0 \prec \cdots \prec p_n$.

To eliminate duplicate structures in a structure tree, we introduce the notion of *SIT-equivalence*, which is employed to merge duplicate paths and subtrees to obtain the SIT.

Definition 2.2 (SIT-equivalence) Two paths, $p_1 = u_0 \ldots u_m$ and $p_2 = v_0 \ldots v_n$, are *SIT-equivalent*, if $u_i.lid = v_i.lid$ for $0 \leq i \leq m$ and $m = n$. Two subtrees, $T_1 = p_{10} \prec \cdots \prec p_{1m'}$ and $T_2 = p_{20} \prec \cdots \prec p_{2n'}$, are *SIT-equivalent*, if (1) the roots of T_1 and T_2 are siblings and (2) p_{1i} and p_{2i} are SIT-equivalent for $0 \leq i \leq m'$ and $m' = n'$.

The following example helps illustrate the concepts of branch ordering and SIT-equivalence.

Example 2.1 Given $p_1 =$ "$(0,0)\ldots(3,4)$", $p_2 =$ "$(0,0)\ldots(9,8)$" and $p_3 =$ "$(0,0)\ldots(3,15)$" in Figure 4, and $p_4 =$ "$(0,0)\ldots(3,15)$" in Figure 5, we have $p_1 \prec p_2 \prec p_3$, $p_3 \preceq p_4$ and $p_4 \preceq p_3$. The subtrees rooted at the nodes (17,14) and (17,27) in Figure 4 are SIT-equivalent, since every pair of corresponding paths in these two subtrees are SIT-equivalent. The subtrees rooted at the nodes (17,19) and (17,32) are also SIT-equivalent.

Since the structures of SIT-equivalent subtrees are duplicate, we define a tree-merge operation, *merge*(T_1, T_2), as shown in Procedure 2.1, to eliminate the redundant tree structures by merging the SIT-equivalent subtrees, T_1 and T_2.

Procedure 2.1 *merge*(T_1, T_2)

/* T_1 and T_2 are SIT-equivalent subtrees and T_1.`root`.*nid* < T_2.`root`.*nid* */

begin

1. Depth-first traverse T_1 and T_2 in parallel in pre-order:
2. **for each** pair of nodes, $u \in T_1$ and $v \in T_2$, visited, **do**
3. $u.ext := u.ext \cup v.ext$;
4. Delete v and its incoming edge;
5. **return** T_1; /* T_2 has been merged into T_1 */

end

It is obvious that the precondition of the *merge* operation, i.e., whether T_1 and T_2 are SIT-equivalent, can also be verified by a pre-order traversal of T_1 and T_2 in parallel and comparing the *lid*s of each pair of nodes visited.

To give an example of the *merge* operation, if we apply the *merge* operation to the SIT-equivalent subtrees rooted at the nodes (17,14) and (17,27) in Figure 4, the resultant merged subtree is the subtree rooted at (17,14) in Figure 5.

We now give the semantic definition of the SIT.

Definition 2.3 (Structure Index Tree) Given a structure tree, T. The Structure Index Tree (SIT) of T is a relaxed form of T that allows more than one element in the extent of a tree node such that no two distinct subtrees in T are SIT-equivalent.

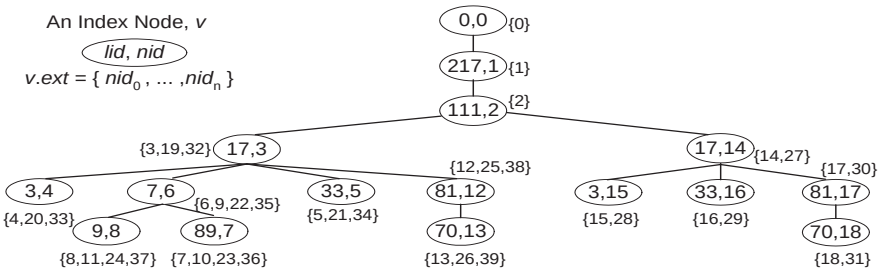


Figure 5: The SIT of the XML Document in Figure 2

It is trivial that T can be viewed as the SIT of T if T has no SIT-equivalent subtrees (i.e. the SIT whose extents are singletons). Otherwise, the SIT of T can be obtained by applying *merge* iteratively on T until no SIT-equivalent subtrees can be found. For example, the tree shown in Figure 5 is the SIT obtained from the the structure tree shown in Figure 4. Note that all SIT-equivalent subtrees in Figure 4 are merged into a corresponding SIT-equivalent subtree in the SIT and that no two subtrees in the SIT is SIT-equivalent.

The operation *merge*(T_1,T_2) merges T_2 into T_1 only if T_1 precedes T_2 in document order, and vice versa. Therefore, the SIT is unique even if the *merge* operation is randomly applied on any two subtrees of a set of SIT-equivalent subtrees in the structure tree. In practice, however, we apply the *merge* operation in post-order, as outlined in Procedure 2.2, so that we can build the SIT for an XML document in a single parse of the document (c.f. [8]).

Procedure 2.2 *buildSIT*(T)

begin
1. Depth-first traverse the structure tree T in post-order:
2. **for each** node, v, visited, **do**
3. **if** v has some preceding-sibling, u, such that the subtrees, T_u and T_v, rooted at u and v respectively, are SIT-equivalent, **then**
4. Apply *merge*(T_u, T_v);
5. **return** T;

end

2.2 The SIT-Lattice

In this subsection, we define the SIT-Lattice on the SIT and introduce a set of efficient SIT-lattice operations.

Given a set of paths, $P = \{p_0, \ldots, p_k\}$, in the SIT, we define the path-join operation *join*, as shown in Procedure 2.3, which joins the paths in P one by one to obtain a partial structure tree.

Procedure 2.3 *join*(L, p)

/* L=$p_0 \prec \cdots \prec p_{k-1}$ and $p_{k-1} \prec p_k$, where p_{k-1}=$u_0 \ldots u_m$ and p_k=$v_0 \ldots v_n$ */

begin
1. **for each** $0 \leq i \leq m$ **do**
2. **if** ($u_i.nid = v_i.nid$) **then**
3. $u_i.ext := u_i.ext \cup v_i.ext$;
4. Delete v_i and its outgoing edge, if any;
5. **else**
6. Connect $v_i \ldots v_n$ to T such that v_i is the last child of u_{i-1};
7. **return** L;
8. **return** L;

end

We can apply *join* on a (meaningful) set of selected paths to obtain a tree, which we call a *partial tree*, as defined in Definition 2.4.

Definition 2.4 (Partial Tree) Let $P = \{p_0, p_1, \ldots, p_k\}$ be a set of paths in the SIT and, without loss of generality, assume that $p_0 \prec p_1 \prec \cdots \prec p_k$. A *Partial Tree*, L, over P, is a tree constructed as follows: L = *join*($\cdots$ *join*(L', p_1),..., p_k), where L' is the initial tree that consists of only one path, p_0.

Example 2.2 If we apply the *join* operation to the three paths, p_0, p_1 and p_2, in Figure 6, we obtain the partial tree, L = *join*(*join*(p_0, p_1), p_2). Note that the paths are joined together by the SIT-equivalent portions of the paths.

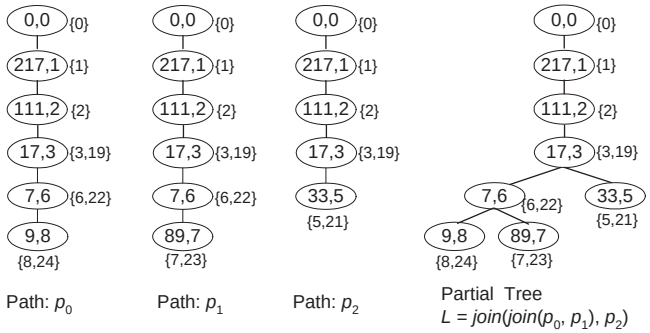


Figure 6: A Partial Tree Constructed by Joining Three Paths

Each path, p, in the SIT is the concise representation of a set of SIT-equivalent paths, P_T, in the structure tree, T. However, in most cases, only a subset of P_T is useful for the evaluation of a given query workload. We define a *partial index path*, p^p, of p to represent a subset, P_t, of P_T.

Definition 2.5 (Partial Index Path) Let $p = u_0 \dots u_n$ be a path in the SIT and P_t be any non-empty set of paths in the structure tree T, such that $\forall p_t \in P_t$, where $p_t = v_0 \dots v_n$, $v_n.nid \in u_n.ext$. A *partial index path*, p^p, of p is defined by $p^p = w_0 \dots w_n$, such that $w_i.nid = u_i.nid$, $w_i.lid = u_i.lid$, and $w_i.ext = \bigcup_{p_t \in P_t}\{p_t.v_i.nid\}$, for $0 \leq i \leq n$.

Example 2.3 The three paths in Figure 6 are partial index paths of their corresponding SIT-equivalent paths in the SIT in Figure 5. For example, p_0 is a partial path of the path "$(0,0)\dots(9,8)$" shown in Figure 5, while it represents the two paths, "$(0,0)\dots(9,8)$" and "$(0,0)\dots(9,24)$", shown in Figure 4.

Each index path has $(2^{|P_T|}-1)$ partial index path for the corresponding $(2^{|P_T|}-1)$ non-empty subsets of P_T. Similarly, there is also a bijection from the set of all partial trees to the power-set of the set of paths in T, since any arbitrary subset of paths in T can be represented by a set of partial index paths in the SIT that can be formed into a partial tree. Therefore, we can define as many as 2^n partial trees, where n is the number of root-to-leaf paths in T. Since each text node is associated with a unique path, we can define a partial tree for any subset of relevant paths in an XML document to accelerate query evaluation. More importantly, we can define a lattice structure [14] on the set of all partial trees. From the index lattice, we can effectively select more efficient lattice elements served as indexes to

accelerate query evaluation. We also define a set of lattice operations, which includes union, intersection, subtraction and extraction, for efficient construction of useful lattice elements.

Theorem 2.4 The set of all partial trees defined over a SIT is a lattice.

Proof. Let $\mathcal{L}$ be the set of all partial trees over a SIT. For all $L_x, L_y \in \mathcal{L}$, let P_x and P_y be the set of paths in L_x and L_y, respectively. Let $p_x = u_0 \dots u_m$ and $p_y = v_0 \dots v_n$, where $p_x \in P_x$ and $p_y \in P_y$.

1. $\mathcal{L}$ is a *non-empty ordered set*: we define a binary relation $\leq$ on $\mathcal{L}$: $L_x \leq L_y$ if, for all p_x, there exists some p_y such that $u_i.ext \subseteq v_i.ext$, for $0 \leq i \leq m$ and $m = n$. It is straightforward to see that $\leq$ on $\mathcal{L}$ is a *partial order*.
2. The *least upper bound* of L_x and L_y, $L_x \vee L_y$, and the *greatest lower bound* of L_x and L_y, $L_x \wedge L_y$, exist:
   - $P_{L_x \vee L_y} = P_x \cup P_y$.
   - Let $p = w_0 \dots w_m$, where $w_i.ext = u_i.ext \cap v_i.ext$, $w_i.nid = u_i.nid$ and $w_i.lid = u_i.lid$, for $0 \leq i \leq m$. $P_{L_x \wedge L_y} = \{p\text{: } p_x \preceq p_y \text{ and } w_m.ext \neq \emptyset\}$. $\square$

We call this lattice defined over the SIT the *SIT-lattice* and an element in the SIT-lattice, i.e., a partial tree of the SIT, a *SIT-lattice element* or simply an *SLE*. Therefore, the *maximum SLE* is the SIT and the *minimum SLE* is an empty tree. The least upper bound and the greatest lower bound of two SLEs, L_x and L_y, i.e. $(L_x \vee L_y)$ and $(L_x \wedge L_y)$, are also referred to as the *union* and the *intersection* of L_x and L_y, respectively. To allow more flexible construction of useful SLEs to aid query evaluation, we introduce two more SIT-lattice operations, *subtraction* and *extraction*. The subtraction of two SLEs, $(L_x - L_y)$, is the index of the set of paths $P = (P_x - P_y)$, where P_x and P_y are the set of paths indexed by L_x and L_y respectively. We say L_x is an extraction of L_y if $L_x \leq L_y$. We give efficient algorithms for these four SIT-lattice operations in Section 4.

Example 2.4 Figure 7 shows two SLEs, L_x and L_y, and their union $(L_x \vee L_y)$, intersection $(L_x \wedge L_y)$ and subtraction $(L_x - L_y)$. All the five SLEs are extractions of the SIT in Figure 5, while $(L_x - L_y)$ is an extraction of L_x and $(L_x \wedge L_y)$ is an extraction of L_x (or L_y), which in turn is an extraction of $(L_x \vee L_y)$.

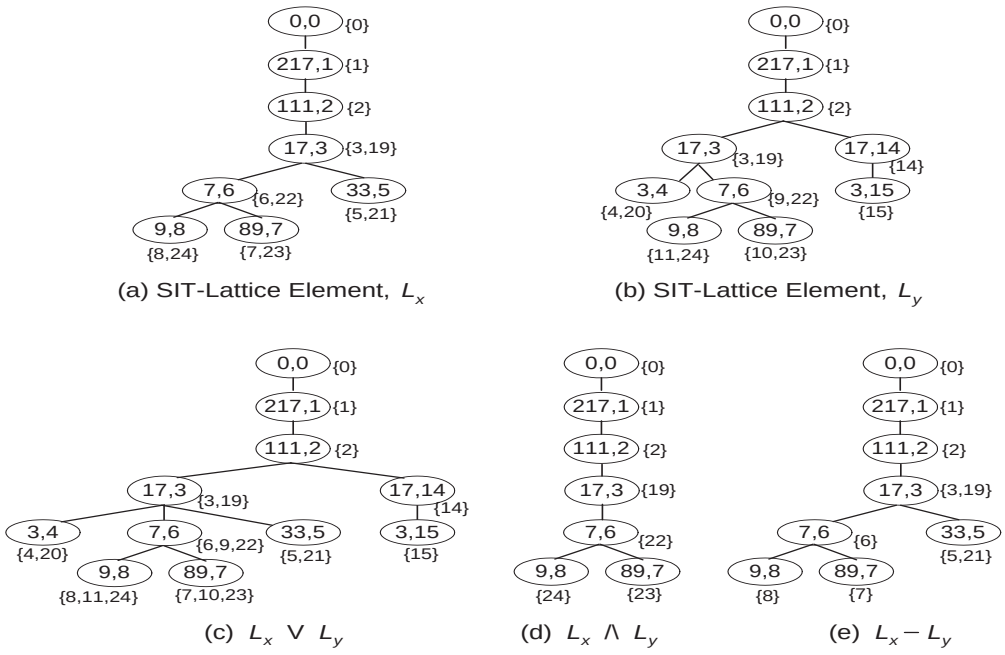


Figure 7: SIT-lattice Elements and Operations

3 SIT-Lattice Element Specification

In this section, we discuss the syntax to specify a SIT-Lattice Element (SLE) and a set of heuristic rules to aid the specification of a more desirable SLE. We also describe three partition strategies to support querying large XML data in memory-limited devices.

3.1 An XPath Specification of SLE

We use XPath [3] as the syntax to specify an SLE. Recall that an SLE is an index of a set of paths and each path is uniquely identified by its leaf node. Given a set of leaf nodes, we can construct a structure tree by tracing their paths. Therefore, an SLE can be specified using an XPath query, such as "`//a[b = ''B'' and avg(.//c) > 10]/(d | e | f)`". We will see in Section 4 (Extraction) how to construct an SLE from its XPath specification.

Intuitively, we can regard an SLE as a materialized view [11, 13, 15] on XML data. In data warehouse applications, the user can define a materialized view for pre-fetching a fragment of data so that queries covered by this fragment can be efficiently processed. The SLE shares a similar spirit in the context of XML by allowing the user to specify an SLE at any time, such that the set of queries asked subsequently can be processed more efficiently using the SLE instead of using the full index. For example, we can specify an SLE as "`//data[type`

= ‘‘A’’]” to filter out non-“Type A” data, so that any subsequent queries of the form “//data[type = ‘‘A’’]$\cdots$” can be more efficiently processed. The SLE can also help an administrator restrict the access privilege of “Type A” users to only “Type A” data.

3.2 Heuristic Selection Rules

The problem of specifying an SLE, L, to cover a given set of queries, Q, is equivalent to checking whether the set of nodes selected by L is a superset of the union of the set of nodes selected by $q \in Q$. We call this problem the SLE containment problem.

The containment problem for XPath fragments (c.f. A survey on XPath query containment [31]), that consists of the “*child*” axis and any two of the following three constructs, (1) “*descendant*” axis, (2) predicates, and (3) wildcards, is shown to be in PTIME in [33, 24, 2]. However, the containment problem for the XPath fragment that consists of all three constructs is shown to be co-NP complete [22], while adding the union expression “|” to the fragment makes the containment problem to be in EXPTIME [25].

The SLE containment problem is even harder, since we allow a more rich set of XPath features such as aggregation-based and value-based predicates. Therefore, we establish a set of heuristic rules to aid the specification of an efficient SLE, which is described as follows.

- Given a set of queries, Q, we specify an SLE, L, that covers Q as follows.
  - We specify the primary location path of L, i.e. the location path that remains when all predicates are removed from L, as the combination of the primary location paths of all $q \in Q$, with the common prefix removed. For example, given the three queries, “//a/b/c”, “//a/b/d//e” and “//a/b/d//f”, we can specify an SLE to cover the queries as L = “//a/b/(c | d//(e | f))”, or simply some less-efficient upper bounds of L, such as “//a/b/(c | d)” and “//a/b”.
  - If a location step, s, in L is in the common prefix of a set of queries, $Q' \subseteq Q$, let s_i be the corresponding location step of $q \in Q'$ and P_i be the predicate, if any, of s_i. We define P'_i as follows: (1) for all “p_x *and* p_y” in P_i, either p_x or p_y or “p_x *and* p_y” in P'_i; and (2) for all “p_x *or* p_y” in P_i, “p_x *or* p_y” in P'_i. Note that P'_i

is defined inductively and the basis of the induction is when both p_x and p_y are reduced to an atomic predicate.

The predicates of s are either all or some of the P'_i connected by the "*or*" operator. For example, given the three queries, "`//a/b[[d` *and* `e]` *or* `f]/c`", "`//a/b[e` *or* `f]/d//e`" and "`//a/b[e]/d//f`", we can specify L as "`//a/b[e` *or* `f]/(c | d//(e | f))`".

– If Q can be classified into two disjointed subsets of queries of the forms, "$s_0 \cdots s_i$`[`p_i`]` $\cdots$" and "$s_0 \cdots s_i$ `[`*not* p_i`]` $\cdots$", and L covers Q, then we can extract L' = "$s_0 \cdots s_i$`[`p_i`]`" from L and then construct the complement of L', given by $\overline{L'} = L - L'$, to cover the two disjointed subsets of queries. For example, if L covers "`//a[`p`]/`$*$" and "`//a[`*not* p`]/`$*$", we can extract L' = "`//a[`p`]`" from L to cover the set of queries of the form "`//a[`p`]`$\cdots$" and then construct $\overline{L'}$ to cover the set of queries of the form "`//a[`*not* p`]`$\cdots$".

- Identify frequently imposed predicates from the queries and impose them on the SLE. This allows the predicates to be pre-computed by the SLE.
- If an SLE is to be constructed from some existing SLEs, smaller SLEs should always be preferred to their upper bounds if both of them cover a given set of queries.
- The union and intersection of a set of SLEs covers the union and intersection of the sets of queries covered by each of the SLEs, respectively.

3.3 Partition Strategies

The indexes of real XML data datasets [19, 32, 30, 23] are often too large to be loaded into the main memory of a machine, especially hand-held devices such as pocket-PCs. Apart from extracting an SLE from a large index to reduce the index size, we can also partition a large index into smaller partitions in order to load them into the main memory. We present three index partition strategies: *Horizontal Partition*, *Vertical Partition* and *Hybrid Partition*.

Let $t(v)$ be the subtree (having n children) rooted at a node, v, in the SIT or an SLE, we express $t(v)$ as: $v(t(u_0), \ldots, t(u_i), \ldots, t(u_n))$, where u_i is the i-th child of v. We describe the three partition strategies applied to $t(v)$ as follows.

- Horizontal Partition: each partition is the union of a partial tree of $t(u_i)$ for $0 \leq i \leq n$. To partition $t(v)$ horizontally, we impose predicates on each u_i, such as "`//v/u[`p`]`", where p is usually a range-match predicate. For example, "`//c[.//i >= 3000]`" and "`//c[.//i < 3000]`" partition the SIT in Figure 5 into two SLEs, as shown in Figure 8. Note that $t((17,3))$ and $t((17,14))$ in both SLEs are a partial tree of $t((17,3))$ and $t((17,14))$ in the SIT, respectively.

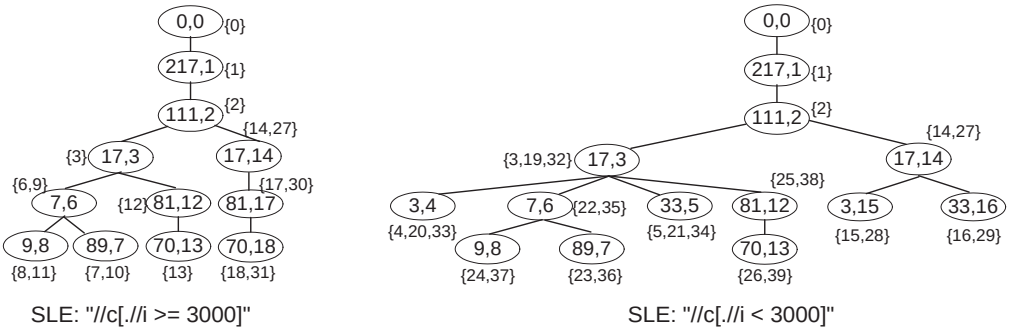


Figure 8: Horizontal Partition of the SIT in Figure 5

- Vertical Partition: each partition is the union of some $t(u_i)$. To partition $t(v)$ vertically, we do not impose predicate on u_i, such as "`//v/u`". For example, "`//c`" vertically partitions the SIT in Figure 5 into two SLEs, one contains the subtree rooted at the node (17,3) and the other the subtree rooted at (17,14).
- Hybrid Partition: we apply Horizontal Partition at one level of an SLE and Vertical Partition at another level, and so on. Hybrid Partition is applied when a simple Horizontal Partition or Vertical Partition of the SLE may not be sufficient to satisfy the memory requirement or other requirements of user applications. For example, "`//c[.//i >= 3000]/e`" is a specification for Hybrid Partition.

4 SIT-lattice Operations

In this section, we propose efficient algorithms for the SIT-lattice operations, union, intersection, subtraction and extraction. The complexities for running these operations are also discussed.

Let L_x and L_y be two SLEs, and u and v be nodes in L_x and L_y respectively. We describe efficient algorithms to perform the union, intersection and subtraction of L_x and L_y.

Union. The idea is to depth-first traverse L_x and L_y in parallel in pre-order and incrementally construct a new tree, T_{new}, which essentially combines the structures of L_x and L_y. We process each traversal step as follows:

- If the depth of u is equal to that of v, then we consider the following two cases:
  - Case 1: If $u.nid = v.nid$, then we create a new node, w, for T_{new}, where $w.ext = u.ext \cup v.ext$, $w.nid = u.nid$ and $w.lid = u.lid$. We then advance the pre-order traversal one step forward in both L_x and L_y in parallel.
  - Case 2: If ($u.lid < v.lid$) or ($u.lid = v.lid$ and $u.nid < v.nid$), we copy the subtree rooted at u and joint it to T_{new} and advance the pre-order traversal one step forward in L_x; vice versa for ($u.lid > v.lid$) or ($u.lid = v.lid$ and $u.nid > v.nid$).
- If the depth of u is greater than that of v, then we copy the subtree rooted at u and join it to T_{new} and advance the pre-order traversal one step forward in L_x only; and vice versa if the depth of v is greater than that of u.

Intersection. The idea is similar to the union operation. We depth-first traverse L_x and L_y in parallel in pre-order by only tracing a pair of nodes, (u, v), if $u.nid = v.nid$. We perform the intersection of the *ext*s of the nodes in a post-order manner, i.e., the *ext*s of the children are always intersected before those of their parents. If u and v are leaf nodes and $u.ext \cap v.ext \neq \emptyset$, we create a new node with its $ext = u.ext \cap v.ext$, and then create its parent node, whose *ext* is the intersection of the *ext*s of the parents of u and v, and so on. The process of the new node creation and the intersection of the *ext* sets continues in a post-order manner until the roots of L_x and L_y are visited.

Constructing the nodes in a bottom-up manner avoids cascading deletion, because L_x and L_y may share some common structures in the upper part of the tree but the intersection of the *ext*s of two nodes in the lower part of the tree may be empty.

Subtraction. To subtract L_y from L_x, we depth-first traverse L_x and L_y in parallel in pre-order by only tracing a pair of nodes, (u, v), where $u.nid = v.nid$. Similar to intersection, we perform the subtraction of the *ext*s of the nodes in a post-order manner. If u and v are leaf nodes and $(u.ext - v.ext) \neq \emptyset$, then $u.ext = (u.ext - v.ext)$. When we finish the subtraction

of the *ext*s of the leaf nodes, we compute the *ext* of their parent according to Definition **??**, and so on for other ancestors until the roots of L_x and L_y are visited.

The complexity of the union, intersection and subtraction operations is all linear to the size of the two SLE operands, since each operation takes a parallel depth-first traversal of the two trees, while it takes linear time, in the size of the *ext*s, to perform the union, intersection and subtraction of the *ext*s of the node pairs.

Extraction. Extraction is the fundamental SIT-lattice operation in the construction of new SLEs, since initially we have only the SIT, i.e., the maximum SLE. From this maximum SLE, we extract smaller SLEs, and then apply other more efficient SIT-lattice operations on them to generate more SLEs. We present an efficient algorithm for the extraction of an SLE from its upper bound elements. The specification of the SLE is evaluated using the upper bound SLE and then we extract the new SLE by tracing the paths of the result nodes obtained from the evaluation. We show the algorithm in Procedure 4.1 and illustrate the construction by Example 4.1, in which we use the SIT as the upper bound SLE.

Procedure 4.1 *extractSLE*(V)
/* V is the set of result nodes obtained by evaluating the SLE specification using an SLE, L. S is a list of triplets of the form, (id, v_{old}, v_{new}), where v_{old} is a node in L and v_{new} is a node in the new SLE to be extracted from L. And "$u :=_{id} v$" denotes "$u.nid := v.nid$ and $u.lid := v.lid$". */

begin
    /* *INITIALIZATION* */
1. **for each** $v \in V$ **do**
2.   Create a new S element, $s := (v.nid, v', v)$; /* v' is a node in L, where $v'.nid$=$v.nid$ */
3.   **if** (S is empty or $v.nid < S.head.id$) **then**
4.   Insert s as $S.head$;
5.   **else** /* $v.nid > S.head.id$ */
6.   Insert s after $S.head$;
    /* *CONSTRUCTION* */
7. **while** ($S.head.next$ is not *null*) **do**
8. $s := S.head.next$;
9.   **while** (s is not *null*) **do**
10. $s.v_{old} := s.v_{old}.parent$; $s.id := s.v_{old}.nid$;
11.   Create a new node, u, where $u :=_{id} s.v_{old}$;
12.   **for each** $e \in s.v_{new}.ext$ **do**
13. $u.ext := \{e' : e' \in s.v_{old}.ext$, such that $e' < e$ and if $e' < e'' < e$, $e'' \notin s.v_{old}.ext\}$;
14.   Connect u as the parent of $s.v_{new}$;
15. $s.v_{new} := u$;
16. $s' := s$; $s := s.next$;

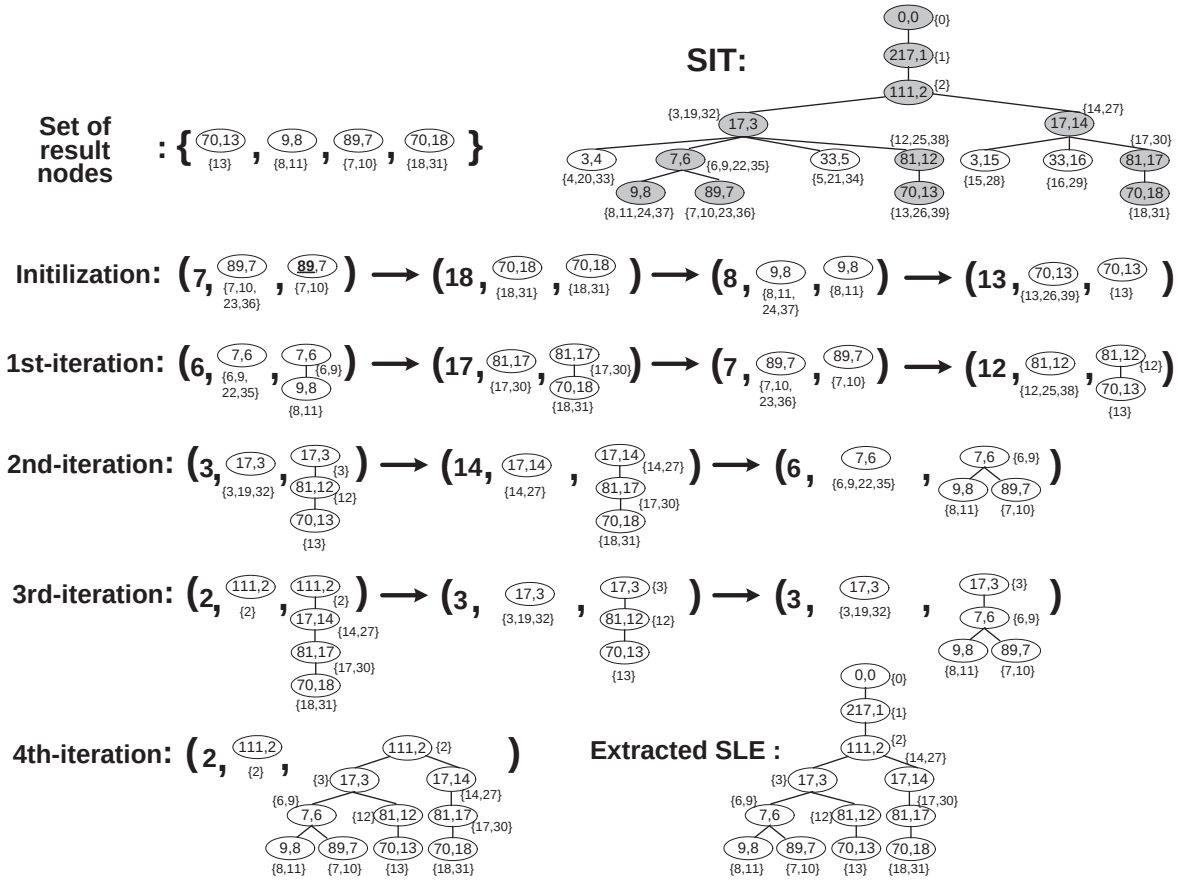


Figure 9: An SLE Extraction Example

17. **if** ($s'.id < S.head.id$) **then**
18. Swap s' with $S.head$;
19. **else if** ($s'.id = S.head.id$) **then**
20. $S.head.v_{new}.ext := S.head.v_{new}.ext \cup s'.v_{new}.ext$;
21. Connect $s'.v_{new}$'s children as the children of $S.head.v_{new}$;
22. Remove s' from S;

/* *COMPLETION* */

23. **while** ($S.head.v_{old}$ is not the `root` of L) **do**
24. $S.head.v_{old} := S.head.v_{old}.parent$;
25. Create a new node, u, where $u :=_{id} S.head.v_{old}$;
26. **for each** $e \in S.head.v_{new}.ext$ **do**
27. $u.ext := \{e'$: $e' \in S.head.v_{old}.ext$, such that $e'<e$ and if $e'<e''<e$, $e'' \notin S.head.v_{old}.ext\}$;
28. Connect u as the parent of $S.head.v_{new}$;
29. $S.head.v_{new} := u$;
30. **return** the tree rooted at $S.head.v_{new}$;

end

Example 4.1 Figure 9 illustrates an example that extracts the SLE shown in Figure 3 from its upper bound SLE, the SIT, shown in Figure 5. The specification of the SLE is evaluated using the SIT and the set of result nodes, V, obtained is shown at the top left corner of

Figure 9. The shaded nodes in the SIT are the nodes to be visited during the SLE extraction. The extraction procedure is divided into three major phases: initialization, construction and completion. We show the contents of the list, S, after the initialization and for each iteration of the construction. The initialization phase constructs S and determines the head of S. The main idea of the construction phase is as follows: for each iteration (*Lines 9-22*), we trace up the SIT to find a common ancestor for all nodes in V. Since there may be more than one intermediate common ancestor for some nodes in V and the path length of these nodes to their common ancestors may be different, we compare the *id* of the elements in S with the *id* of $S.head$ for each move of a node up the SIT. Intuitively, this means that the node that moves highest in the SIT will always "wait" for the other nodes. For each move of a node up the SIT, we construct a part of the new SLE. As a result, after the construction phase, i.e., the 4th-iteration in this example, we build a tree whose root is the common ancestor of all nodes in V, i.e., the node (111,2). We then add the path from this node to the root in the completion phase.

The complexity of the SLE extraction algorithm is linear in the size of the upper bound SLE. As illustrated in the example, for each step in each iteration during the construction phase, we traverse one step up the old SLE, create a new node for the new SLE and compare the *id* values. These operations can be performed in constant time. Since we can at most build an SLE as large as the original one, it follows the linear time complexity.

5 Experimental Evaluation

In this section, we report on experiments that consider the following three issues: (1) the effectiveness of using the SLEs to control both the structure size and the extent size of the index and the query performance of using SLEs; (2) a comparative analysis of the SLEs and Kaushik et al.'s index definition scheme [17]; and (3) the efficiency of using SLEs to aid querying large XML datasets in memory-limited devices.

We ran the first two sets of experiments on a Windows XP machine with a Pentium 4, 2.53 GHz processor and 512 MB of RAM. In the last experiment, we used a Toshiba Pocket-

PC with a 400 MHz Intel PXA250 processor and 64 MB of SDRAM; we loaded the SLEs in the Pocket-PC's main memory and retrieved the data contents of the result nodes from the PC via a wireless LAN with a transfer rate of 11 Mbps.

We use three benchmark XML datasets: XMark [30], which is an XML benchmark project modelling a deeply nested auction database; SwissProt [23, 32], which describes DNA sequences; and DBLP [19], which is a popular bibliography database. Table 1 shows some brief descriptions of the three XML datasets such as the size, the number of distinct tags/attributes, and the maximum depth of each dataset. $|V_T|$ is the number of nodes in the structure tree, which is the extent size of the SIT, and $|V_I|$ is the number of nodes in the SIT, which is the structure size of the SIT. The ratio of $|V_I|$ to $|V_T|$ shown in the last column of Table 1 indicates the degree of its redundancy (a higher ratio indicates less redundancy) and regularity (a lower ratio indicates greater regularity) of the dataset. Thus, the $|V_I|/|V_T|$ ratios show that DBLP is relatively regular and SwissProt has the lowest level of redundancy.

Datasets	Size	Tags/Attrs	Depth	$\lvert V_T \rvert$	$\lvert V_I \rvert$	$\lvert V_I \rvert / \lvert V_T \rvert$
XMark	111 MB	86	11	1837608	30071	1.64%
SwissProt	109 MB	100	5	5166890	1466332	28.38%
DBLP	127 MB	38	5	3733320	1874	0.05%

Table 1: Dataset Descriptions

5.1 Effectiveness of Using SLEs

We study the effectiveness of using SLEs to control both the structure size and the extent size of the index and the gain in query performance of using SLEs.

Queries and SIT-lattice Elements. We evaluate the SLEs using a set of practical queries for each dataset and then specify seven SLEs for each set of queries based on the heuristic rules discussed in Section 3.2. While we include the union and the intersection operations in the seven SLEs for each dataset, we evaluate the subtraction operation separately in Section 5.1.4 due to the complementary nature of the operation. We list the queries (Q_1 to Q_5) and the SLEs (L_1 to L_7) in Appendix, while we depict an overview of the relationships between the SLEs and the queries for each dataset in Figure 10. In the figure, a (dotted) path from

an SLE, L_i, to a query, Q_j, means that L_i covers Q_j, while a (solid) path from an SLE, L_i, to another SLE, L_j, indicates that $L_j \leq L_i$. For simplicity, we use $L_{i,\ldots,j}$ to denote $L_i, \ldots, L_j$ in subsequent discussions.

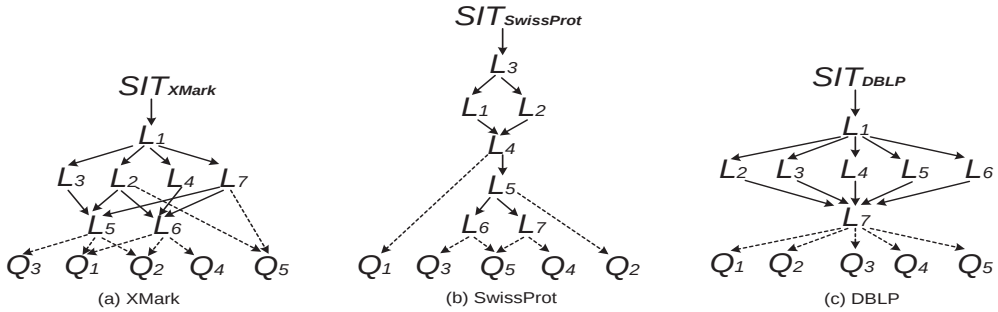


Figure 10: SIT-Lattice Elements and Queries

5.1.1 Performance on SLE Construction

We investigate (1) the effectiveness of the SLEs in controlling the structure size and the extent size of the index and (2) the efficiency in constructing the SLEs. In Table 2, we show the Structure Ratio and Extent Ratio of the SLEs of the three XML datasets, L_1 to L_7, which represent the ratio of the structure size and the extent size of the respective SLEs to those of their corresponding SIT, respectively.

The results show that the structure size and the extent size of the SLEs can essentially vary from as small as 0% to as large as 100% of the SIT, and many points in between. This implies that we have great flexibility in choosing an SLE to aid in query evaluation.

SIT-Lattice Elements		L_1	L_2	L_3	L_4	L_5	L_6	L_7
XMark	Structure Ratio (%)	11.98	0.84	4.95	7.91	0.31	0.40	0.58
	Extent Ratio (%)	34.18	0.59	6.04	16.42	0.41	0.43	0.69
	Build Time (sec)	0.233	1.231	1.032	1.520	0.001	0.001	0.011
SwissProt	Structure Ratio (%)	81.11	57.80	88.43	42.95	35.67	22.56	31.81
	Extent Ratio (%)	79.48	59.33	90.60	45.28	37.20	23.79	33.12
	Build Time (sec)	5.123	7.020	0.078	0.021	0.230	0.167	0.188
DBLP	Structure Ratio (%)	22.96	10.34	11.72	9.16	7.25	8.58	0.64
	Extent Ratio (%)	54.23	2.32	13.39	2.54	1.13	0.16	0.001
	Build Time (sec)	0.560	1.709	1.121	1.530	1.002	1.402	0.044

Table 2: SLE Construction Results

We also record the time (Build Time) taken to construct the SLEs in Table 2. The Build

Time includes the time taken to load the SLE into the main memory, though the loading time is usually negligible compared to the construction time. When the SLEs (such as $L_{1,2,3,4}$ of XMark, $L_{1,2,5,6,7}$ of SwissProt and $L_{1,2,3,4,5,6}$ of DBLP) are extracted from their upper bounds, it is usually more costly if value-based predicates are imposed, since we need to access the disk to retrieve the data contents of the nodes for the evaluation of the predicates. However, when the SLEs (such as $L_{5,6,7}$ of XMark, $L_{3,4}$ of SwissProt and L_7 of DBLP) are constructed as the union or the intersection of some existing SLEs, the construction time is only on average tens of milliseconds.

5.1.2 Query Evaluation Speedup

We study the query evaluation speedup obtained by using the SLEs instead of the SITs. Our goal is to investigate the effect of a reduction in the structure size and/or the extent size on the query performance.

We measure the response time of each query that is evaluated using the SLEs and the SIT. Then, we compute the speedup as the ratio of the response time of a query evaluated using an SLE to that using the SIT. The speedup ratio is expressed as "*milliseconds per second*". We show the speedup ratio in Table 3. For example, for XMark, the speedup ratio of L_4 against Q_1 is 80, which means that it takes 80 milliseconds to evaluate Q_1 using L_4, while it takes 1 second to evaluate Q_1 using the SIT. Thus, a lower speedup ratio indicates a higher speedup. In Table 3, a slash "/" indicates that the SLE does not cover the query.

We record impressive speedup for all the three datasets. We now discuss some insights of the performance speedup on each dataset observed from the experimental results.

XMark: Tremendous speedup is achieved by using all SLEs except L_1. From Table 2, both the structure size and the extent size of the SLEs are considerably reduced, except for L_1, whose extent size is not reduced as much as its structure size. The specification of L_1 (c.f. Appendix) is only structural, indicating that the extent size of no index nodes in L_1 is reduced. This implies that the main factor contributing to the speedup is the reduction in the extent size of the SLEs.

SIT-Lattice Elements		L_1	L_2	L_3	L_4	L_5	L_6	L_7
XMark	Q_1	933	103	147	80	10	7	21
	Q_2	912	138	212	96	17	9	27
	Q_3	986	33	46	/	9	/	24
	Q_4	877	35	/	41	/	18	25
	Q_5	987	93	/	/	/	/	19
SwissProt	Q_1	356	171	836	41	/	/	/
	Q_2	334	194	719	71	33	/	/
	Q_3	455	310	987	112	133	87	/
	Q_4	519	441	1031	106	118	/	81
	Q_5	414	426	761	209	126	108	106
DBLP	Q_1	904	92	537	123	45	19	1
	Q_2	810	64	424	60	26	10	1
	Q_3	940	159	577	219	83	52	1
	Q_4	1034	88	243	107	37	35	1
	Q_5	911	146	751	128	69	27	1

Table 3: Query Evaluation Speedup Ratio (msec/sec)

SwissProt: According to Table 1, the SwissProt data has a very low level of redundancy, which implies that the structure size is large while the extent size of each index node is small. This accounts for the relatively lower speedup (higher ratio) measured for this dataset.

DBLP: We observe that the speedup obtained is roughly proportional to the reduction in the extent size and the structure size of other SLEs is not greatly reduced compared to that of L_1. This again shows that a reduction in the extent size is essential for better query performance. We also note that L_7 is tailored towards the queries, Q_1 to Q_5, and thus the evaluation of all these queries is optimized.

Based on the experimental results, we derive a guideline to achieve better query performance using SLEs: more emphasis should be put on reducing the extent size (by imposing value-based predicates) than on reducing the structure size (by imposing structural predicates). However, we note that for less regular data sources, such as SwissProt, reducing the structure size and reducing the extent size is equally important, because it is likely that every index node is associated with only a few extent elements. For such datasets, it is more effective to reduce the structure size, since a reduction in the structure size also effectively brings down the extent size of the index, as shown by SwissProt.

Finally, we remark that in this experiment, we evaluated all the predicates imposed on

the queries, even though part of them is already pre-computed by the SLEs. The reason for the re-computation is to give an accurate account of the effects of the reduction in the structure size and the extent size on query performance. However, we mention that when we made use of predicate pre-computation, significantly greater speedup was measured for almost all of the SLEs. In real-world database applications, a user can take advantage of this feature of the SLE to obtain efficient query performance gain.

5.1.3 Query Performance Gain

In this subsection, we measure the gain in query performance obtained by using the SLEs instead of the SIT and then illustrate the applicability of the SLEs by an example. We measure the performance gain as $(1 - (\textit{SLE Construction Cost} + \textit{Query Evaluation Cost using the SLE})/\textit{Query Evaluation Cost using the SIT})$, i.e., $G = \{1-(c_l+\sum_{i=1}^{n} c'_i)/\sum_{i=1}^{n} c_i\} \times 100\%$, where c_i and c'_i is the cost of evaluating the i-th query of the query workload using the SIT and the SLE, respectively, and c_l is the cost of building the SLE. We present in Table 4 the percentage gains for two scenarios: $G+$ reports the gain of using an SLE assuming that the SLE was constructed from some existing SLEs other than the SIT, while $G-$ reports the gain of an SLE that was constructed (all the way) from the SIT. For example, the construction cost of L_7 of XMark is 0.011 second, as reported in Table 2, for the $G+$ scenario. However, the cost is 4.029 seconds, which is the sum of the construction time of all the seven SLEs, for the $G-$ scenario, since all other SLEs must be constructed before L_7 can be constructed.

SIT-Lattice Elements		L_1	L_2	L_3	L_4	L_5	L_6	L_7
XMark	$G+$ (%)	4.06	86.43	76.89	79.99	98.74	98.94	97.76
	$G-$ (%)	4.06	85.53	74.95	78.08	77.93	74.46	82.10
SwissProt	$G+$ (%)	55.43	63.70	12.66	87.73	89.06	90.02	90.43
	$G-$ (%)	55.43	63.70	8.26	83.34	84.07	80.87	81.63
DBLP	$G+$ (%)	7.60	89.33	53.31	88.11	95.00	96.73	99.98
	$G-$ (%)	7.60	89.04	53.03	87.82	94.71	96.45	96.26

Table 4: Query Performance Gain

On average, using the SLEs instead of the SIT achieves significant improvement in query evaluation in both scenarios. The percentage gain is over 70% for most of SLEs, in both

$G+$ and $G-$ scenarios. The small difference between $G+$ and $G-$ also implies the great efficiency in constructing the SLEs.

Those less obvious performance gains shown in Table 4 can be explained by the small query evaluation speedup measured for these SLEs. This is also because we only used five queries for each SLE in this experiment. In practice, more queries are generally posed at a given time and hence the performance gain can be further increased.

5.1.4 Evaluation of Subtraction

We evaluate the subtraction operation by a practical scenario of using SLEs. Consider the following eight queries of the DBLP dataset.

Q_1: `//inproceedings[ee]/*`
Q_2: `//inproceedings[crossref]/*`
Q_3: `//inproceedings[ee` *or* `crossref]/*`
Q_4: `//inproceedings[ee` *and* `crossref]/*`
Q_5: `//inproceedings[`*not* `ee]/*`
Q_6: `//inproceedings[`*not* `crossref]/*`
Q_7: `//inproceedings[`*not* `[ee` *and* `crossref]]/*`
Q_8: `//inproceedings[`*not* `[ee` *or* `crossref]]/*`

Suppose that initially we have the SLE, L_1 = "`//inproceedings`". To answer the first three queries, we extract from L_1 a finer SLE, L_2 = "`//inproceedings[ee` *or* `crossref]`". From L_2 we extract L_3 = "`//inproceedings[ee` *and* `crossref]`". Then, we construct $L_4 = (L_1 - L_3)$ to answer Q_5, Q_6, Q_7 and Q_8. Note that L_2, L_3 and L_4 give instant answers to Q_3, Q_4 and Q_7, respectively. We can also construct $(L_4 - L_2)$ to give instant solution to Q_8.

We recorded that the total querying time of evaluating the eight queries using L_1 is 9.09 seconds, while by applying the scenario of using L_2 to answer $\{Q_1, Q_2, Q_3\}$, L_3 to $\{Q_4\}$ and L_4 to $\{Q_5, Q_6, Q_7, Q_8\}\}$, the total querying time is significantly reduced to 4.20 seconds, which also includes the 1.44 and 0.50 seconds used to extract L_2 and L_3 respectively and 0.05 seconds to obtain L_4 by the subtraction operation.

5.2 A Comparative Analysis

The index definition scheme proposed by Kaushik et al. [17] is primarily designed to accelerate the evaluation of structural queries and hence cannot be applied to define an efficient

index to cover queries that contain value-based predicates. However, value-based predicates are crucial in querying XML data. In this subsection, we compare our method of using SLEs with Kaushik et al.'s index definition scheme in the evaluation of structural queries. We use three structural queries for each dataset, as listed below.

XMark:
Q_1: *count*(//person[profile/education])
Q_2: *count*(//person[homepage])
Q_3: //item[@id *in* //open_auction[bidder]/itemref/@item]
SwissProt:
Q_1: //Entry[*count*(Ref) = 1]
Q_2: //Entry[*count*(Keyword) >= 5]
Q_3: //Entry[*count*(Org) >= 5]
DBLP:
Q_1: //inproceedings[*not* [ee *or* crossref]]/@key
Q_2: //inproceedings[*not* ee]/booktitle
Q_3: //inproceedings[*not* crossref]/url

The three queries of XMark are also used in the experimental evaluation in [17] and we also generate the same *F+B-index* (c.f. Definition 5 of Table 1 in [17]) that ignores text tags and *idref* edges in the forward direction, and *idref*s pointing to "person" and "open_auction" tags in the back direction. The three queries of SwissProt and DBLP contain aggregation-based predicates and negation, respectively. For these two sets of queries, we build a covering index with only those tags and attributes present in the queries and with the parameters, k_{fwd}, k_{back} and td, set to ∞, 2 and 1, respectively. The index is thus obtained by computing the ∞-bisimilarity partition, i.e., a global similarity, in the forward direction and a 2-bisimilarity partition, i.e., a local similarity for paths of length at most 2, in the backward direction on the XML dataset for 1 iteration. We construct an SLE for each dataset, which is simply specified as a union of the set of paths used for the evaluation of the three queries. We show the XPath specification of the SLEs below.

XMark: (//person[profile/education *or* homepage] | //item |
//open_auction[bidder]/itemref/@item)
SwissProt: //Entry[[*count*(Ref) = 1] *or* [*count*(Keyword) >= 5] *or* [*count*(Org) >= 5]]
DBLP: //inproceedings[*not* [ee *and* crossref]]/(@key | booktitle | url)

Since we do not need to access the text data for processing structural queries, we evaluate all the queries in memory, except that for evaluating Q_3 of XMark using the SLE, for which we access the disk to retrieve the "`ids`" to perform the join.

Datasets	Q_1	Q_2	Q_3
XMark	2.73	2.27	0.38
SwissProt	2.43	2.89	3.02
DBLP	4.14	3.11	3.20

Table 5: Ratio of Response Time of using SLE to using Covering Index

Table 5 shows the ratio of the response time of each query using the SLE to that using the covering index. On average, using the SLE is a factor of 1.76, 2.78 and 3.48 faster than using the covering index to evaluate the queries of XMark, SwissProt and DBLP, respectively. The significant improvement by the SLE is because the SLE is more effective in selecting a more tailor-made set of paths for a given query workload. The only exceptional case is that in the evaluation of Q_3 of XMark, which contains an *idref*, using the SLE is 3.45 times slower than using the covering index. This is because the SIT does not index the *idref*s and therefore we have to perform a join to evaluate the query, while there is an *idref* edge pointing the corresponding "`item`" node in the covering index. However, we note that in practice few XML datasets contain *idref*s.

In addition to the greater speedup in query evaluation, the construction time of the SLE for each dataset is about 180 times less than that of the covering index. The reason for the tremendous difference is that the time for constructing an SLE is the same as evaluating an XPath query, while a covering index has to be built from the original dataset. Moreover, it is also obvious from this experiment that specifying an SLE, i.e. writing an XPath query, is far easier than specifying a covering index.

The results of this experiment thus show that comparing with Kaushik et al.'s index definition scheme, the SLEs are simpler to specify, less costly to construct, and more efficient and effective in building a more efficient index.

5.3 Use of SLEs in Memory-Limited Devices

The goal of this experiment is to show that the SLEs allow efficient querying of large XML data in memory-limited devices.

We partition XMark by Vertical Partition (c.f. Section 3.3) and construct an SLE for each child of the root of its SIT. We horizontally partition SwissProt into 12 SLEs of roughly the same size by specifying each SLE as "`//Entry[@seqlen[.` $<=$ *range_lower and* `.` $>=$ *range_upper*`]`". For DBLP, we first apply Vertical Partition by constructing an SLE for each child of the root of the SIT of DBLP and then horizontally partition the over-sized child "`inproceedings`" as "`//incproceedings[@key` *starts-with* ``` ``conf/somevalue/'' ```]". Using the partition strategies, the indexes of all the three datasets are able to be loaded into the main memory of the pocket-PC. Note that the SLEs are constructed from their corresponding SITs in the PC machine, since the SITs are too large to be loaded into the main memory of the pocket-PC.

To assess the query performance, we construct, in the pocket-PC, $L_{2,3,4,5,6,7}$ (c.f. Appendix) from L_1 for XMark and DBLP. However, L_1 of DBLP is too large to be loaded into the main memory of the pocket-PC. We thus horizontally partition L_1 of DBLP into four SLEs: L_{11}, L_{12}, L_{13} and L_{14}. Then, we extract $L_{2j,3j,4j,5j,6j}$ from L_{1j} and construct L_{7j} as the intersection of $L_{2j,3j,4j,5j,6j}$, where j is $1, 2, 3$ and 4, respectively. Finally, L_i of DBLP is constructed as the union of $L_{i1,i2,i3,i4}$ for $2 \leq i \leq 7$ and then loaded into the pocket-PC. Then, we evaluate the same set of queries (c.f. Appendix) by using the SLEs. We measure the speedup ratio as the ratio of the response time of evaluating a query using an SLE to that using L_1. The query performance gains that we obtain for each of the SLEs are on average slightly better than but roughly of the same pattern as those obtained on the PC machine as shown in Sections 5.1.2 and 5.1.3 (detailed experimental results thus omitted).

In real-world applications, the partition strategies can also be used by database administrators to set access permissions to different users on XML data, which is analogous to the case of granting access privileges by relational views [11]. Administrators can build appropriate SLEs for each user and grant access to more or less data by performing the union,

intersection, subtraction or extraction of existing SLEs.

6 Related Work

A considerable amount of research has been conducted on indexing XML or semi-structured data [12, 24, 20, 18, 9, 17, 27, 28, 7, 29, 16]. However, none of the work has attempted to speed up the evaluation of value-based query conditions, which is crucial in querying XML data. We have discussed the A(k)-indexes [18], D(k)-indexes [7], M(k)-indexes [16], and the index definition scheme [17] to reduce the structure size of an index in Section 1.1. However, a new index of smaller structure size must be constructed from the base data, while the SLEs can be very efficiently constructed from existing SLEs by a set of lattice operations.

The same technique to define the lattice structure on the SIT can be also applied to define a lattice structure on other structural indexes, such as 1-index [24] and F&B-index [1, 17], by ordering the paths in the index in a specific way, for example, according to the document order of the paths. However, the study on the lattices defined on these indexes, in particular, on the approximate indexes [18, 7, 16], is our future work.

Marian et al. [21] constructs a projected document from a set of paths extracted from a given XQuery [4] to reduce memory requirement for query processing. Their method works on the original XML document instead of an index. As the projected document in [21] is constructed from simple XPath expressions without predicates, the irrelevant nodes of value-based conditions are not filtered out. Buneman et al. [6] also proposes a lattice structure, which is defined on a class of equivalent *tree instances* based on bisimulation. However, they do not focus on constructing a lattice element of smaller size from existing lattice elements to accelerate query evaluation.

7 Discussion and Conclusions

We have presented the SIT-lattice defined on the SIT. With the SIT-lattice, we are able to select any subset of relevant paths from an XML document. A SIT-lattice element (SLE) is specified as an XPath expression and we have proposed a set of heuristic rules and three

partition strategies to aid in the SLE-specification. Moreover, we have presented an efficient set of SIT-lattice operations, which consist of union, intersection, subtraction and extraction, in order to generate appropriate SLEs. We have also justified, throughout the paper, the practical value of our SIT-lattice to real-world database applications.

We carried out a wide range of empirical studies of SLEs as follows. First, we showed with experimental evidence that the SLEs can be constructed very efficiently and that using the SLEs, instead of the full index, can tremendously improve query performance. Second, we showed that the SLEs are more effective in defining a desirable refined index to accelerate query evaluation than is Kaushik et al.'s index definition scheme. Third, we demonstrated in a Pocket-PC that the SLEs can be used to query large XML data with impressive query performance.

We remark that, in general, it is difficult to check whether an SLE fully covers a given query workload, as studied in the containment problem of XPath fragments in [33, 24, 2, 22, 25]. However, in a distributed environment, especially with the rapidly growing popularity in the use of hand-held devices in peer-to-peer networks, it is more important for users to obtain a fast response of query results, despite the fact that the results may not be complete. In such environments, the SLEs can not only be used as efficient query accelerators, but can also be used to partition the indexes to allow them to fit into the main memory of the memory-limited devices. Therefore, our development of the SIT-lattice provides a solid foundation for querying XML data in peer-to-peer networks of hand-held devices.

References

[1] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the web: from relations to semistructured data and XML*. San Francisco, Calif.: Morgan Kaufmann, c2000.

[2] S. Amer-Yahia, S. Cho, L. V. S. Lakshmanan, and D. Srivastava. Minimization of Tree Pattern Queries. In *Proceedings of SIGMOD*, 2001.

[3] A. Berglund et al. XML Path Language (XPath) 2.0, 2003. `http://www.w3.org/TR/xpath20`.

[4] S. Boag et al. XQuery 1.0: An XML Query Language. `http://www.w3.org/TR/xquery`, Nov. 2002.

[5] T. Bray et al. Extensible Markup Language (XML) 1.0 (Third Edition), 2004. `http://www.w3.org/TR/REC-xml`.

[6] P. Buneman, M. Grohe, and C. Koch. Path Queries on Compressed XML. In *Proceedings of VLDB*, 2003.

[7] Q. Chen, A. Lim, and K. W. Ong. D(K)-Index: An Adaptive Structural Summary for Graph-Structured Data. In *Proceedings of SIGMOD*, 2003.

[8] J. Cheng and W. Ng. XQzip: Querying Compressed XML Using Structural Indexing. In *Proceedings of EDBT*, 2004.

[9] C. W. Chung, J. K. Min, and K. Shim. APEX: An Adaptive Path Index for XML Data. In *Proceedings of SIGMOD*, 2002.

[10] J. Clark and S. DeRose. XML Path Language (XPath) 1.0, 1999. `http://www.w3.org/TR/xpath.`

[11] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*, Prentice Hall, 2002.

[12] R. Goldman and J. Widom. Dataguides: Enabling Query Formulation and Opeimization in Semistructured Databases. In *Proceedings of VLDB*, 1997.

[13] J. Goldstein and P. Larson. Optimizing Queries Using Materialized Views: A Practical, Scalable Solution. In *Proceedings of SIGMOD*, 2001.

[14] G. A. Gratzer. *General Lattice Theory*. Birkhuser Verlag, c1998.

[15] A.Y. Halevy. Answering Queries Using Views: A Survey. In *VLDB Journal*, Volume 10 Issue 4, 2001.

[16] H. He and J. Yang. Multiresolution Indexing of XML for Frequent Queries. In *Proceedings of ICDE*, 2004.

[17] R. Kaushik, P. Bohannon, J. F. Naughton and H. F. Korth. Covering Indexes for Branching Path Queries. In *Proceedings of SIGMOD*, 2002.

[18] R. Kaushik, P. Shenoy, P.Bohannon, and E. Gudes. Exploiting Local Similarity for Efficient Indexing of Paths in Graph Structured Data. In *Proceedings of ICDE*, 2002.

[19] M. Ley. Digital Bibliography and Library Project (DBLP). `http://dblp.uni-trier.de`.

[20] Q. Li and B. Moon. Indexing and Querying XML Data for Regular Path Expressions. In *Proceedings of VLDB*, 2001.

[21] A. Marian and J. Simeon. Projecting XML Documents. In *Proceedings of VLDB*, 2003.

[22] G. Miklau and D. Suciu. Containment and Equivalence for a Fragment of XPath. In *Journal of the ACM*, Vol. 51, No. 1, pp.2-45, January 2004.

[23] G. Miklau and D. Suciu. XML Data Repository. `http://www.cs.washington.edu/research/xmldatasets`.

[24] T. Milo and D. Suciu. Index Structures for Path Expressions. In *Proceedings of ICDT*, 1999.

[25] F. Neven and T. Schwentick. XPath Containment in the Presence of Disjunction, DTDs, and Variables. In *Proceedings of ICDT*, 2003.

[26] R. Paige and R. Tarjan. Three partition refinement algorithms. In *SIAM Journal of Computing*, 16:973988, 1987.

[27] N. Polyzotis and M. Garofalakis. Statistical Synopses for GraphStructured XML Databases. In *Proceedings of SIGMOD*, 2002.

[28] N. Polyzotis and M. Garofalakis. Structure and Value Synopses for XML Data Graphs. In *Proceedings of VLDB*, 2002.

[29] P. Ramanan. Covering Indexes for XML Queries: Bisimulation Simulation = Negation. In *Proceedings of VLDB*, 2003.

[30] A. R. Schmidt and F. Waas and M. L. Kersten and M. J. Carey and I. Manolescu and R. Busse. XMark: A Benchmark for XML Data Management. In *Proceedings of VLDB*, 2002.

[31] T. Schwentick. XPath Query Containment. In *SIGMOD Record*, Vol. 33, No. 1, March 2004.

[32] SWISS-PROT Protein Knowledgebase. `http://www.expasy.ch/sprot/`.

[33] M. Yannakakis. Algorithms for Acyclic Database Schemes. In *Proceedings of VLDB*, 1981.

APPENDIX

This appendix lists, in abbreviated XPath syntax, the queries and the specification of the SLEs used in the performance evaluation. We use fully parenthesized expressions for the predicates as to avoid ambiguity.

XMark:
Common predicates used in the queries and the SLE specification:
P_{x1} = [[[initial >= 100] *and* [current <= 200]] *and* [*not* [reserve]]]
P_{x2} = [interval[[start >= 01/01/2000] *and* [end < 01/01/2001]]]
P_{x3} = [[*count*(bidder) >= 10] *and* [*avg*(bidder/increase) < 5]]

Queries:
Q_1: /site/open_auctions/open_auction[P_{x1} *and* [P_{x2} *and* P_{x3}]]/@id
Q_2: /site/open_auctions/open_auction[[P_{x1} *and* [P_{x2} *and* P_{x3}]] *and* [*not* [bidder]]]/(@id | */description)
Q_3: //open_auction[[P_{x1} *and* P_{x2}] *and* [type = ''featured'']]/@id
Q_4: /site/open_auctions/open_auction[[P_{x1} *and* P_{x3}] *and* [*max*(bidder/increase) >= 10]]/annotation/description
Q_5: //open_auction[[P_{x1} *and* [P_{x2} *or* P_{x3}]] *and* [*not* [*contains*(type, ''Dutch'')]]] /(@id | bidder[increase >= 10]/date)

SIT-lattice elements:
L_1: //open_auctions
L_2: //open_auction[P_{x1}]/(@id | */description | type | bidder/(date | increase) | interval)
L_3: //open_auction[P_{x2}]
L_4: //open_auction[P_{x3}]
$L_5 = L_2 \cap L_3$: //open_auction[P_{x1} *and* P_{x2}]/(@id | */description | type | bidder/(date | increase) | interval)
$L_6 = L_2 \cap L_4$: //open_auction[P_{x1} *and* P_{x3}]/(@id | */description | type | bidder/(date | increase) | interval)
$L_7 = L_5 \cup L_6$: //open_auction[P_{x1} *and* [P_{x2} *or* P_{x3}]]/(@id | */description | type | bidder/(date | increase) | interval)

SwissProt:
Common predicates used in the queries and the SLE specification:
P_{x1} = [@seqlen[[. >= 100] *and* [. < 1000]]]
P_{x2} = [Mod[[@type = ''Created''] *and* [@date[[. >= ''01-JAN-1993''] *and* [. < ''1-JAN-2000'']]]]]
P_{x3} = [P_{x1} *and* P_{x2}]
P_{x4} = [P_{x3} *and* [*count*(Ref) = 1]]

P_{x5} = [P_{x4} *and* [*contains*(Species, ‘‘Homo’’)]]

Queries:
Q_1: //Entry[P_{x3}]/(@id | Gene)
Q_2: //Entry[P_{x4}]/(@id | Gene)
Q_3: //Entry[P_{x5} *and* [*count*(Keyword) >= 5]]/(@id | Gene)
Q_4: //Entry[P_{x5} *and* [*count*(Org) >= 5]]/(@id | Gene)
Q_5: //Entry[P_{x5} *and* [[*count*(Keyword) >= 5] *and* [*count*(Org) >= 5]]]/(@id | Gene)

SIT-lattice elements:
L_1: //Entry[P_{x1}]
L_2: //Entry[P_{x2}]
$L_3 = L_1 \cup L_2$: //Entry[P_{x1} *or* P_{x2}]
$L_4 = L_1 \cap L_2$: //Entry[P_{x3}]
L_5: //Entry[P_{x4}]
L_6: //Entry[P_{x4} *and* [*count*(Keyword) >= 5]]
L_7: //Entry[P_{x4} *and* [*count*(Org) >= 5]]

DBLP:
Common predicates used in the queries and the SLE specification:
P = [[[[[*contains*(author, ‘‘David’’)] *and* [year >= 2000]] *and*
[crossref[[*contains*(., ‘‘sigmod’’)] *or* [*contains*(., ‘‘vldb’’)]]]] *and*
[*contains*(booktitle, ‘‘SIGMOD’’)]] *and* [*contains*(title, ‘‘Data Mining’’)]]

Queries:
Q_1: //∗/@key[*ancestor-or-self*::inproceedings[P]]
Q_2: (//title[*parent*::inproceedings[P]] | //author[*parent*::inproceedings[P]])
Q_3: //*/inproceedings[P]/(booktitle | year | page | title)
Q_4: //cite[@label[. = ‘‘IBM99’’ *and* ./*ancestor*::inproceedings[P]]]
Q_5: *count*(//inproceedings[P]/author)

SIT-lattice elements:
L_1: //inproceedings
L_2: //inproceedings[*contains*(author, ‘‘David’’)]
L_3: //inproceedings[year >= 2000]
L_4: //inproceedings[crossref[[*contains*(. ‘‘sigmod’’)] *or* [*contains*(., ‘‘vldb’’)]]]
L_5: //inproceedings[*contains*(booktitle, ‘‘SIGMOD’’)]
L_6: //inproceedings[*contains*(title, ‘‘Data Mining’’)]
$L_7 = L_2 \cap L_3 \cap L_4 \cap L_5 \cap L_6$: //inproceedings[$P$]