An Efficient Index Lattice for XML Query Evaluation

Wilfred Ng and James Cheng

Department of Computer Science and Engineering
The Hong Kong University of Science and Technology, Hong Kong
{csjames, wilfred}@cse.ust.hk

Abstract. We have defined an XML structural index called the *Structure Index Tree* (*SIT*), which eliminates duplicate structures arising from the equivalent subtrees in an XML document by merging them into a concise structure. In this paper, we impose a lattice structure on the SIT and call the structure a *SIT-lattice* in order to enhance the applicability of the index. A *SIT-Lattice Element* (*SLE*) is an index of an arbitrary subset of paths in the document. Since paths represent the structure of the 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 demonstrate that SLEs are able to support effective querying over very large XML documents in memory-limited hand-held devices.

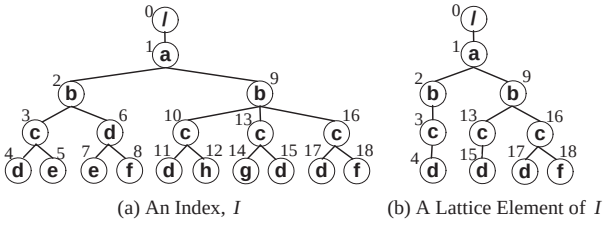
1 Introduction

It is well recognized that establishing an efficient index to aid in processing queries on XML data is important, for example, Dataguides [4], 1-index [11], A(k)-indexes [7], D(k)-indexes [2], M(k)-indexes [5], and F&B-index [6]. However, 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 defined on an XML structural index, called the *Structure Index Tree* (or the *SIT* in short) [3]. The SIT has been introduced in our preliminary work [3] to aid in efficient evaluation of XPath queries on compressed XML data. The SIT is constructed based on the partitioning of paths in an XML document, while an element in the lattice is the 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.

How do we address the structure size problem? We consider 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 [6] definition scheme and other indexing techniques [7], 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.



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

Fig. 1. A Full XML Index and a Lattice Element

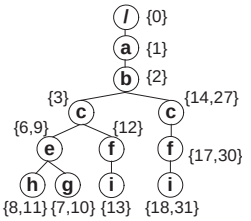


Fig. 2. A Sample SLE

Example 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, "`//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 "$\texttt{c}_{10}$" (node `c` with identity = 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.

How do we address the huge extent size problem? Consider an XML document that 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 there are 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 2. Consider an XML document tree in Figure 3, 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 = ''A'']//h`", we need only

to access the shaded nodes in Figure 3. As mentioned before, we select a (any) combination of paths in an XML document and the resultant SLE is a very small index for the selected path. The SLE selected for our example is shown in Figure 2, which is an index of the shaded nodes in Figure 3. The SLE also pre-computes the common predicate "`[@id = ''A'']`" of the query workload.

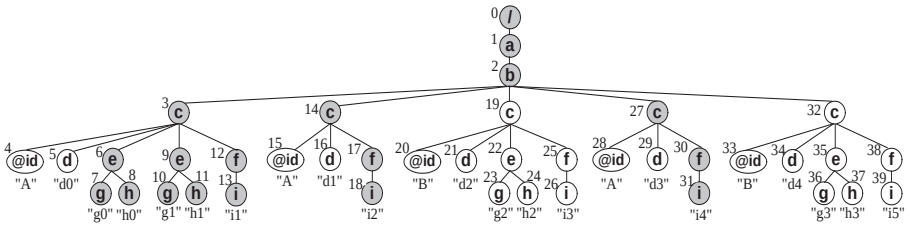


Fig. 3. An XML Document Tree

In Figure 2, we can further combine the two equivalent paths, $\langle$`c`, `f`, `i`$\rangle$, into one; however, the collapsed index does not cover branching path expressions. For example, consider the query "`//c[e]/f`". The "`f`" elements are not distinguishable with the two paths combined, but are distinguishable with the SLE in Figure 2. In fact, we find that the main factor that accelerates query evaluation is the reduction in the extent size, rather than further reduction in the structure size obtained by the coalescence of the two paths.

A practical problem arising from 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 partitions the full index into a set of SLEs, each of which can fit into the main memory. This approach is feasible in practice, since we usually access only a portion of the full index at any time. We make two main advancements on the SIT [3] in this paper.

First, we propose a novel lattice structure on the SIT. The lattice elements can effectively filter out irrelevant elements to accelerate query evaluation. Our method is efficient to tackle the problem of both the structure size and the extent size of an index on XML data. Second, 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, we can efficiently query large XML datasets in a pocket-PC. Compared with Kaushik et al.'s index definition scheme [6], the SLEs are much easier and less costly to construct and more effective in controlling both the structure size and the extent size of an XML index.

In the rest of the section, we discuss the related work. We define the SIT-lattice and its related operations in Section 2. We evaluate the performance of the SLEs in Section 3. Finally, we give our concluding remarks in Section 4.

1.1 Related Work

A considerable amount of research has been conducted on indexing XML or semi-structured data [4, 11, 7, 6, 2, 5]. 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 [7], D(k)-indexes [2], M(k)-indexes [5], and the index definition scheme [6] to reduce the structure size of an index in Section 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.

Marian et al. [8] constructs a projected document from a set of paths extracted from a given XQuery to reduce memory requirement for query processing. Their method works on the original XML document instead of an index. As the projected document in [8] is constructed from simple XPath expressions without predicates, the irrelevant nodes of value-based conditions are not filtered out. Buneman et al. [1] 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.

2 An Index 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, called the *structure tree*, $T = (V_T, E_T, \texttt{root}_T)$, where V_T and E_T are the sets of 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 nids of the set of equivalent nodes that are coalesced into v. We set $v.ext = \{v.nid\}$ (i.e. $v.ext$ is a singleton), which is later to be combined with the exts 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 lids 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 3.

Each text node in an XML document is attached to a unique path, p, in the structure tree, which is given by $p = v_0 v_1 \cdots v_n$, where v_n is a leaf node. We take into account the numerical order of lid and nid and define a path ordering as follows.

Definition 1. (Path Ordering) Given two paths, $p_1 = u_0 \ldots u_m$ and $p_2 = v_0 \ldots v_n$, $p_1 \preceq p_2$ if one of the following two conditions holds:

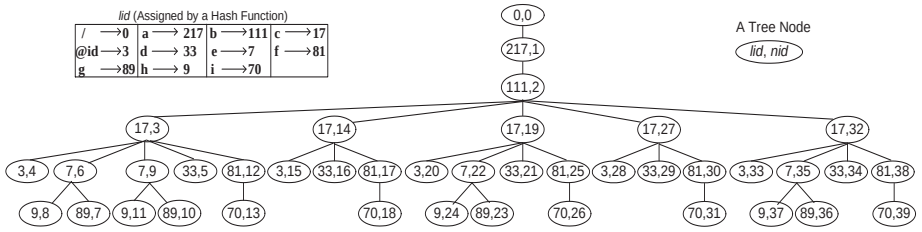


Fig. 4. The Structure Tree of the XML document presented in Figure 3

1. $p_1 \prec p_2$: 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. $p_1 = p_2$: $u_i.nid = v_i.nid$, for $0 \leq i \leq m$ and $m = n$.

With the path ordering, we can specify a structure tree (or a structure subtree), T, as the set of all its paths ordered as follows: $T = p_0 \preceq \cdots \preceq p_n$ (or simply $T = p_0 \prec \cdots \prec p_n$ as the paths are distinct in T). 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. (SIT-equivalence) Two paths, $p_1 = u_0 \dots u_m$ and $p_2 = v_0 \dots v_n$, are *SIT-equivalent*, if $u_i.lid = v_i.lid$ for $0 \leq i \leq m$ and $m = n$. Two subtrees, $T_1 = p_{10} \preceq \cdots \preceq p_{1m'}$ and $T_2 = p_{20} \preceq \cdots \preceq p_{2n'}$, 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 3. Given p_1 = "$(0,0)\dots(3,4)$", p_2 = "$(0,0)\dots(9,8)$" and p_3 = "$(0,0)\dots(3,15)$" in Figure 4, and p_4 = "$(0,0)\dots(3,15)$" in Figure 5, we have $p_1 \prec p_2 \prec p_3$ and $p_3 = p_4$. 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 (cf. [3]) to eliminate the redundant tree structures by merging the SIT-equivalent subtrees, T_1 and T_2. We skip the details of the algorithm but only give an example of the operation here: 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.

2.2 The SIT-Lattice

Given a set of paths, $P = \{p_0, \dots, p_k\}$, in the SIT, we define the path-join operation, *join*, as shown in Procedure 1, which joins the paths in P one by one to obtain a partial tree.

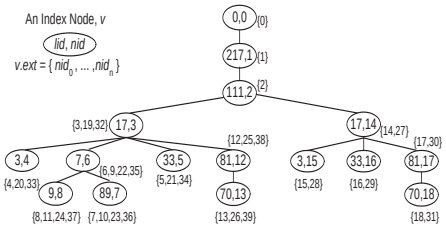


Fig. 5. The SIT of the XML Document in Figure 3

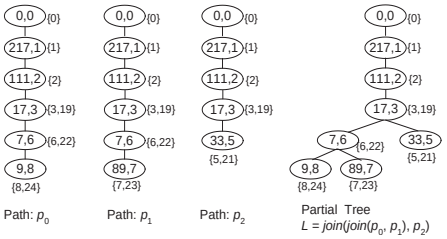


Fig. 6. A Partial SIT Constructed by Joining Three Paths, p_0, p_1 and p_2

Procedure 1 *join*(L, p)
/* L=$p_0 \preceq \cdots \preceq p_{k-1}$ and $p_{k-1} \preceq p_k$, 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 set of selected paths to obtain a tree, which we call a *partial SIT*, as defined in Definition 3.

Definition 3. (Partial SIT) Let $P = \{p_0, p_1, \ldots, p_k\}$ be a set of paths in the SIT. Without loss of generality, we assume that $p_0 \preceq p_1 \preceq \cdots \preceq p_k$. A *Partial SIT*, L, over P, is a tree constructed as follows: $L = join(\cdots join(L', p_1),\ldots, p_k)$, where L' is the initial tree that consists of only one path, p_0.

Example 4. If we apply the *join* operation to the three paths, p_0, p_1 and p_2, in Figure 6, we obtain the partial SIT, $L = join(join(p_0, p_1), p_2)$. Note that the paths are joined together by the SIT-equivalent portions of the paths.

Each path 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 an *index path* that concisely represents any subset of P_T.

Definition 4. (Index Path) Let P_T be the set of all paths in a structure tree represented by a path in its SIT and $p \in P_T$, $p = u_0 \ldots u_n$. An *index path*, p_I = $v_0 \ldots v_n$, is a path in a partial SIT such that $v_i.nid = u_i.nid$, $v_i.lid = u_i.lid$, and $v_i.ext = \bigcup_{\exists p \in P_T}\{u_i.nid\}$, for $0 \leq i \leq n$.

Example 5. The three paths in Figure 6 are index paths of some partial SIT. For example, p_0 represents the two paths, "$(0,0)\ldots(9,8)$" and "$(0,0)\ldots(9,24)$", in Figure 4 and its corresponding index path in the SIT is the path "$(0,0)\ldots(9,8)$" shown in Figure 5.

Theorem 1. The set of all partial SITs defined over a SIT is a lattice. □

We call this lattice defined over the SIT the *SIT-lattice* and an element in the SIT-lattice, i.e., a partial 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 $L_x \leq L_y$.

Example 6. 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 $(L_x \vee L_y)$.

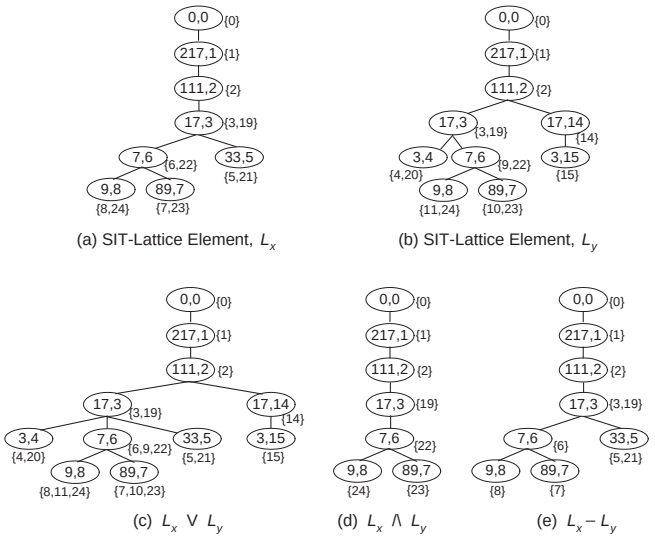


Fig. 7. SIT-lattice Elements and Operations

2.3 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 [13]), that consist 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 [11]. However, the containment problem for the XPath fragment that consists of all three constructs is shown to be co-NP complete [9], while adding the union expression "|" to the fragment makes the containment problem to be in EXPTIME [12].

The SLE containment problem is even harder, since we allow a richer set of XPath features such as aggregation-based and value-based predicates. Therefore, we employ a set of heuristic rules to aid the specification of an efficient SLE. 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`". We skip the details of our rules due to space limitation.

The indexes of real XML datasets [10] 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. For example, "`//c[.//d >= 10]/(e | f)`" partitions the SIT in Figure 5 into two SLEs, as shown in Figure 8.

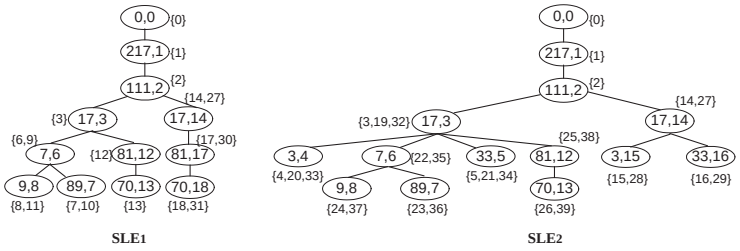


Fig. 8. Horizontal Partition of the SIT in Figure 5

3 Experimental Evaluation

We carried out two sets of experiments. The first is on a Windows XP machine with a P4, 2.53 GHz processor and 512 MB of RAM. The second is to use 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 used the following three datasets [10] XMark, SwissProt and DBLP. We list the queries (Q_1 to Q_5) and the SLEs (L_1 to L_7) in Appendix [14], while we depict an overview of the relationships between the SLEs and the queries for each dataset in Figure 9. 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.

3.1 Effectiveness of Using SLEs

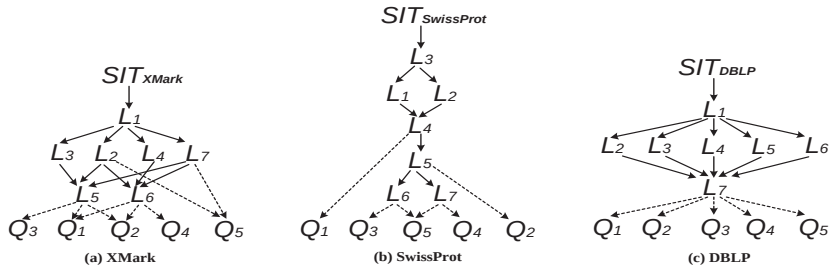


Fig. 9. SIT-Lattice Elements and Queries

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 1, 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.

Table 1. SLE Construction Results

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

We also record the time (Build Time) taken to construct the SLEs in Table 1. 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.

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. We show the speedup ratio (*milliseconds per second*) in Table 2. 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 2, a slash "/" indicates that the SLE does not cover the query. We record impressive speedup for all the three datasets and thus no speedup ratio is presented here.

Table 2. Query Evaluation Speedup Ratio (msec/sec)

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

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 are equally important, because it is likely that every index node is associated with only a few 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 are 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, it is interesting to see 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.

Query Performance Gain. We now 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 $(1 - (\textit{SLE Construction Cost} + \textit{Query Evaluation Cost using the SLE})/\textit{Query Eval-}$

uation 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 are the costs of evaluating the ith query in the workload using the SIT and the SLE, respectively, and c_l is the cost of building the SLE. We present in Table 3 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 1, for the $G+$ scenario. However, the cost is 4.029 secs, 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.

Table 3. Query Performance Gain

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

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 3 can be explained by the small query evaluation speedup measured for these SLEs. This is also because we only used 5 queries for each SLE in this experiment. In practice, more queries are generally posed at a given time and the performance gain can still be further increased.

3.2 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 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 [14]) 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 [14]) 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 3.1 (detailed experimental results thus omitted).

4 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 by an XPath expression.

We carried out 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 demonstrated that SLEs can be used to query large XML data with impressive query performance in Pocket-PCs.

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 [11, 9, 12]. However, in a distributed environment, such as using hand-held devices in a P2P network, it is important for users to obtain a fast response of query results, despite the fact that the results may not be complete. In such environments, 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.

References

1. P. Buneman, et al. Path Queries on Compressed XML. In *Proc. of VLDB*, 2003.
2. Q. Chen, A. Lim, and K. W. Ong. D(K)-Index: An Adaptive Structural Summary for Graph-Structured Data. In *Proceedings of SIGMOD*, 2003.
3. J. Cheng and W. Ng. XQzip: Querying Compressed XML Using Structural Indexing. In *Proceedings of EDBT*, 2004.
4. R. Goldman and J. Widom. Dataguides: Enabling Query Formulation and Opeimization in Semistructured Databases. In *Proceedings of VLDB*, 1997.
5. H. He and J. Yang. Multiresolution Indexing of XML for Frequent Queries. In *Proceedings of ICDE*, 2004.
6. R. Kaushik, P. Bohannon, J. F. Naughton and H. F. Korth. Covering Indexes for Branching Path Queries. In *Proceedings of SIGMOD*, 2002.
7. 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.
8. A. Marian and J. Simeon. Projecting XML Documents. In *Proc. of VLDB*, 2003.
9. 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.
10. G. Miklau and D. Suciu. XML Data Repository, which can be found at the URL: `http://www.cs.washington.edu/research/xmldatasets`.
11. T. Milo and D. Suciu. Index Structures for Path Expressions. In *Proceedings of ICDT*, 1999.
12. F. Neven and T. Schwentick. XPath Containment in the Presence of Disjunction, DTDs, and Variables. In *Proceedings of ICDT*, 2003.
13. T. Schwentick. XPath Query Containment. In *SIGMOD Record*, **33**(1), 2004.
14. Appendix `http://www.cse.ust.hk/~wilfred/SLE/appendix.pdf`.

APPENDIX (This appendix [14] is included for reading convenience only).

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.

We use three benchmark XML datasets: XMark, which is an XML benchmark project modelling a deeply nested auction database; SwissProt, which describes DNA sequences; and DBLP, which is a popular bibliography database. Table 4 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 4 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.

Table 4. Dataset Descriptions

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%

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[$P$]