LTIX: A Compact Level-based Tree to Index XML Databases

Samir Mohammad
Patrick Martin

School of Computing
Queen's University
Kingston, Ontario, Canada K7L3N6
{samir,martin}@cs.queensu.ca

Technical report No. 2010-570

May 2010

ABSTRACT

Indexing XML data is essential for XML query optimization. Most of the existing approaches that combine a labeling scheme with a path index use labeling schemes that reflect the structure of the indexed data. In addition, the labeling rules do not depend on the combined path indexes. By designing a labeling scheme that does not reflect the structure of XML data, since it is available in the accompanied path index; and by aligning the data nodes' labels with the path index nodes' labels, we can support the join process more efficiently. We propose a novel index structure called LTIX (Level-based Tree Index for XML databases). This index structure is based on Level-based Labeling Scheme (LLS) that not only minimizes the number of joins and matches required to evaluate twig queries, if it is used with path indexes, but also facilitates effective query optimization through early pruning of the space search. Experimental tests show the performance benefits of our proposed approach.

KEYWORDS

XML indexes, path indexes, query processing, labeling schemes, data models, structural indexes

Report Index

1. INTRODUCTION

XML is becoming the dominant method of exchanging data over the Internet and the rapid growth in XML databases has resulted in the need to efficiently query this XML data. One way to achieve fast retrieval of data is through indexing. XML queries can contain single or multiple paths, and typically have predicates that involve structure. Generally, XML structural indexes can be grouped into three categories: node indexes, path indexes, and sequence indexes.

Solving a twig query using node indexes requires *n-1* structural joins where *n* is the number of nodes in the query. In contrast, less joins are needed if we use path indexes. In this case, the required number of joins to solve a twig query is usually equal to the number of branches in the query, but path indexes, in general, require a large amount of memory. No joins are required to solve a twig query if we use sequence indexes. Sequence indexes, however, suffer from two anomalies, namely, false positives and false negatives. Refinement steps are added to the evaluation process of a query to overcome these problems.

Some researchers combine node indexes with path indexes to expedite query processing and reduce the number of structural joins. For example, Kaushik et al. [14], Moro et al. [20], and Haw et al. [12] integrate the (*start*,*end*) interval node index with the DataGuide path index. In these approaches element labels are assigned and then subsequently associated with their designated nodes in the path index. In this case, the path indexes, as well as the interval node indexes, hold the structural information of the data. We believe that it is sufficient for only one of them to hold the structure information in order for them to work well together. We can therefore plan a labeling scheme that is structure independent and link it with a path index to provide the structural information. We implement this concept in our approach. Our approach is therefore similar to the approaches that integrate interval node indexes with DataGuide path indexes [12,14,20] with the exception of the labeling scheme. We propose a novel index structure that is based on Level-based Labeling Scheme (LLS) [19], which is shown to work efficiently with DataGuides. The contributions in this report are as follows:

- A novel index structure, LTIX, which combines the LLS and the DataGuide path index.
- A unique compact representation of DataGuide path index.

- A light-weight query processor that uses LTIX to efficiently prune the space search and remove false positives.

The remainder of the report is structured as follows. We review related work in Section 2. We formally define our index structure and explain the LTIX system in Section 3. We present our experimental results in Section 4 and conclude in Section 5.

2. RELATED WORK

Structural indexes can be grouped into three categories: node indexes, path indexes, and sequence indexes. Node indexing approaches [16] depend on labeling schemes including interval labeling [12,16,27], and prefix labeling [21,24]. Node indexes hold values that reflect the nodes' positions within the structure of an XML tree. They can be used to find a given node's location in relation to other nodes, which are used to solve simple (single) path and twig path queries. The main shortcoming of these indexes is the number of structural joins required to solve a query, which is equal to $n-1$ where n is equal to the number of nodes in the query.

Path indexing schemes include indexes that cover either single path queries [8,9] or both single path and twig path queries [1,11,13,17]. Path indexes, in general, require a large amount of memory [11]. Path indexes consider paths as a whole, during query evaluation, instead of dealing with each node in the path separately. Consequently, the number of joins is reduced during query processing and hence query performance is improved.

Sequence indexes [22,26] interpret the whole query as a structured-encoded sequences and search for a match in the structured-encoded sequence of an XML document. They suffer, however, from false positive and false negatives [18]. Refinement steps are added to the evaluation process of a query to overcome these problems.

An example of interval node indexes is shown in Figure 2. It is based on the (*start*,*end*) labeling scheme of the XML document in Figure 1.

```
<students>
  <student address="Kingston">
      <name>
          <fname>Tim</fname>
          <lname>Wang</lname>
      </name>
      <courses>
          <course>Art</course>
          <course>History</course>
      </courses>
  </student>
  <student address="Ottawa">
      <name>
          <fname>Sarah</fname>
          <lname>Ahmad</lname
      </name>
      <courses>
          <course>Math</course>
      </courses>
      <children>
          <child>
              <name>
                  <fname>Mike</fname>
                  <lname>Salem</lname
              </name>
          </child>
      </children>
  </student>
</students>
```

Figure 1. An XML document

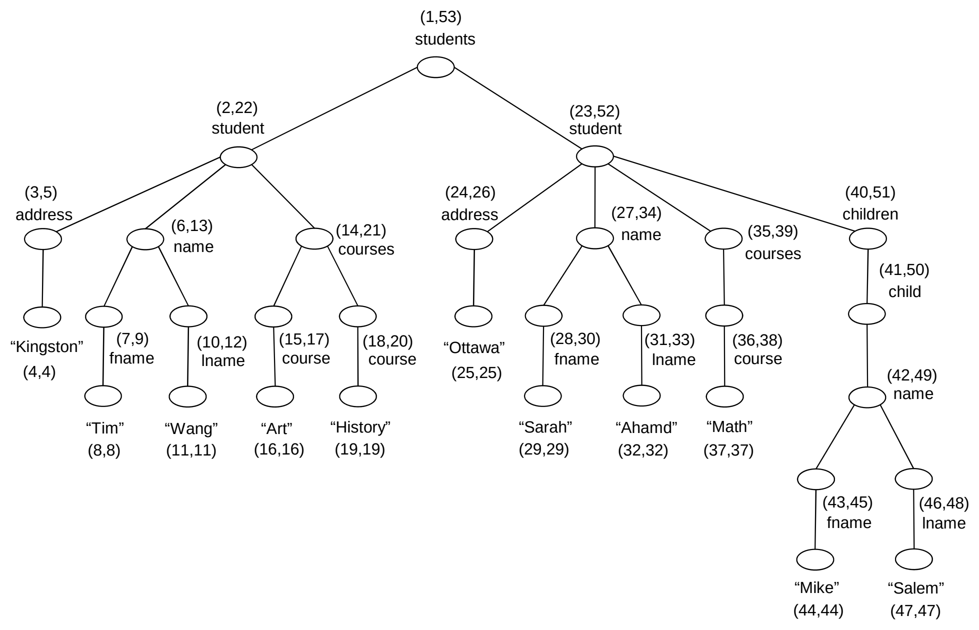


Figure 2. An interval labeled tree representation of the XML data in Figure 1

The labels are given according the sequential traversal of the document in Figure 1. In this type of node indexes, a relation between two elements is established if one element's interval contains the other element's interval.

Path indexes [1,8,9,10,11,13,15,17] partition element nodes in the source XML data-tree based on their path similarity. The DataGuide summary in Figure 7 is an example path index for the data-tree in Figure 2. The numbers inside the oval shaped nodes represent the labels of the summary (or path index) nodes. Unlike node indexes, which return the answers of XML queries at the granularity of individual instances of elements, path indexes return the answers of XML queries at the granularity of the whole groups of instances of elements. Then a node index, such as the interval node index above is used to perform structural joins in a post-processing phase to arrive at the answers to a query. In the structural join operations, each element' instances in a group is compared with the other elements' instances in the other groups to find a match.

To evaluate a simple XML query a number of joins and comparisons are required if we use node indexes. To overcome this shortcoming node indexes can be integrated with path indexes. Our proposed LTIX approach integrates a special labeling scheme (LLS) [19] with a DataGuide path index as a way of reducing join and match operations. To illustrate this consider evaluating Query 1 below over the data in Figure 2. Query 1 returns the first and the last names of the students in an XML document. The node-labeled tree representation for Query 1 is given in Figure 3:

Query 1: *//student/name*[*fname*]*/lname*

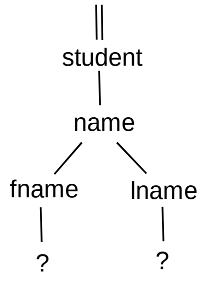


Figure 3. The node-labeled tree representation of Query 1

The instances of Query 1 elements in the XML data-tree in Figure 2 are saved in an index structure similar to the one shown in Figure 4 as suggested by Zhang et al. [27], which is based on (*start*,*end*) labeling scheme.

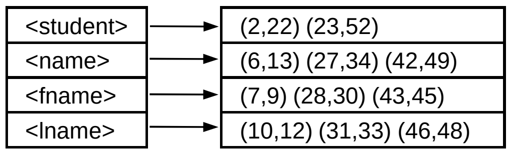


Figure 4. The (*start,end*) interval node index for instances of Query 1 elements in Figure 2

To evaluate Query 1 over the data in (*start*,*end*) interval index in Figure 4, we need to implement 3 structural joins and 18 matches, in the worst case. This worst case is reached if we used the standard merge join algorithm to arrive at the final answer that contains the following tuples:

fname	**lname**
Tim	Wang
Sarah	Ahmad

Zhang et al.[27] propose the MPMGJN algorithm to reduce the number of joins. Much subsequent research has been done in this area to reduce the number of joins and comparisons [2,3,5]. Discussion of the suggested approaches is beyond the scope of this report.

To evaluate Query 1 above by using an integrated system such as the one shown in Figure 5, which integrates the DataGuide path index (Figure 7) with the interval node index (Figure 4), we need to perform 2 join operations and 8 matches in the worst case. This worst case is reached if we used the standard merge join algorithm.

Level	*PerLv*	*Tag*	*Start*	*End*
2	11	student	2	22
2	11	student	23	52
3	21	name	6	13
3	21	name	27	34
4	11	fname	7	9
4	11	fname	28	30
4	21	lname	10	12
4	21	lname	31	33
5	11	name	42	49
6	11	fname	43	45
6	21	lname	46	48

Figure 5. Integration of the node index (Figure 4) with the interval node index (Figure 7).

From the above discussion we notice that integrating interval node indexes with path indexes dropped the number of joins, in our example, from 3 to 2 join operations, and the number of matching operations have been reduced from 18 to 8 matches. Our proposed LTIX approach, which integrates a special labeling scheme (LLS) with a DataGuide path index, requires implementing only one join operation during which two matches are performed in the worst case, to evaluate Query 1 above. We will return to Query 1 example and explain how we can achieve this in Section (3.2) after we elaborate on our approach in the next section.

LLS labeling [19] scheme preserves the best traits of both interval labeling [12,24,27] and prefix labeling schemes [21,24]. Similar to interval labels, the size of LLS labels is constant regardless of the data-tree depth, and hence requires modest storage space. In contrary, the prefix labels length grows as the depth grows, and hence more space is required to save them. Like interval labeling, integer comparison operations are used to establish a relation between two nodes with LLS, which is more efficient than the substring matching operations that are used to establish a relation with prefix labeling. Furthermore, a relation between two nodes can be identified with a single equality comparison operation with LLS, while with interval labeling, a relation is identified using two inequality comparison operations.

ORDPATH labels [21] are a variant of Dewey prefix labels [7,21]. They do not need to be updated when new nodes are inserted, but they suffer from the shortcomings of prefix indexes. In the interval node index approach proposed by Zhang et al. [27], they suggest including the level of elements as a part of node labels. In contrast, our approach not only has the level of elements as part of the node labels, but we provide a path index (absent from Zhang's approach), and the levels are also added to this path index node labels.

3. LTIX APPROACH

In this section, we first introduce the XML data model used in LTIX, the path index, and the mapping of XML data-tree into native XML path index and data repository. We then trace two examples to demonstrate how LTIX is used to solve twig queries and to improve the efficiency of query evaluation process.

3.1 XML Data and Path Index Models

We model an XML document as a directed graph $G=(R,V_R,V_T,E,tagg,labelg,T)$. R is the root node. V_R is the set of elements and attribute nodes – excluding R and V_T – and V_T is the set of text nodes that contain the values of the nodes for nodes that have values, which usually are leaf nodes. V_R and V_T nodes are tagged with the *tagg* function (the extra g stands for the graph G). V_R nodes are tagged according to the tag of the element or attribute they represent. V_T nodes, however, are tagged with the tag of its V_R parent node[1]. A node v, such that $v \in V_R$, may have zero child nodes, or one or more child nodes, which could be V_R or V_T node(s). V_T nodes are always leaf nodes. E is a set of child-parent edges, $E=\{e_1, e_2, \ldots e_i\}$, that connects all nodes of V_R and V_T to form a tree. The total number of edges is $|E|$ and the total number of nodes is $|V_R + V_T|$, where $|E|=|V_R + V_T|$ since $R \notin V_R$. Each node in V_R and V_T is associated with exactly one parent through an edge. R does not have a parent since it is the root node.

All nodes in V_R and V_T are assigned unique labels through the *labelg* function, which is determined by the LLS labeling scheme as follows. Each node v, such that $v \in V_R$, is assigned a unique vector label $<d.p.s>$ where d and p are taken from the label of o node in the path index I (Figure 7) to which v node belongs according to an earlier implemented partition. That is, v node is an instance of an o node. (instance and path index are defined later). s is the instance serial number of node o, which uniquely identifies this node among similar nodes of the same class. V_T nodes carry the same labels as their V_R parent nodes. The set of serial paths is defined by T, where $T=\{r_1, r_2, \ldots, r_n\}$ and n is the number of text nodes $|V_T|$. We define serial path r in Definition 2 below. In our model, an edge e of a node v, where $v \in V_R$ *and* $e \in E$, is equal to the serial number s of the parent node p, denoted $e(v)=s(p)$. The data-tree graph representation G for the data in Figure 1 is illustrated in Figure 6, which is used in the examples throughout this report, unless we state otherwise. Next, we give several definitions, which are used in describing the LLS labeling scheme and the LTIX system.

[1] REF/IDREF are encoded as values in XML, and can be related through their values, hence we do not consider them as edges.

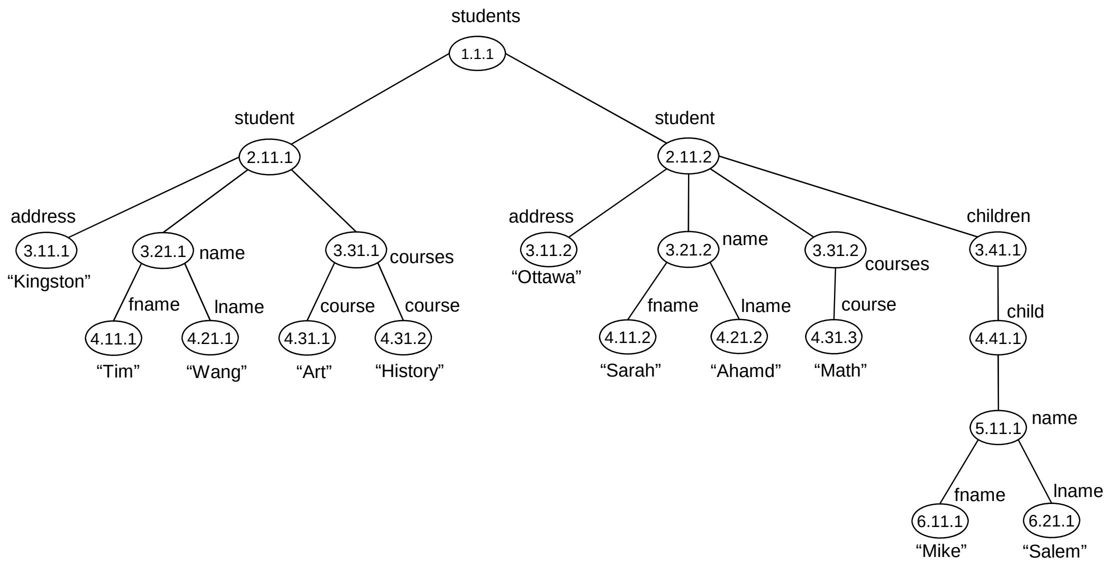


Figure 6. LTIX data model of the data in Figure 1

Definition 1. A *tag path t* for a node *v* is a sequence of tags, $l_1.l_2. \ldots . l_i$ ($i \geq 1$), of the nodes on the path from the root node to *v* node, separated by dots. For example, the tag path of node <4.31.3> is *students.student.courses.course*.

Definition 2. A *serial path r* for a node *v* is a sequence of dot separated serial numbers, $s_1.s_2. \ldots .s_i$ ($i \geq 1$), of the nodes on the path from the root node to *v*. For example, the serial path of node <4.31.3> is (*1.2.2.3*) , which contains the third part of the labels of the nodes in the path from the root node to this node. Note that the *d* values (the levels) of the components of a serial path *r* of a node *v*, where $r = (s_1.s_2. \ldots .s_i)$, is $d = (1,2, \ldots , i)$, respectively, where *i* is the level of *v*. For example, the levels of the component of the serial path (*1.2.2.3*) are (1,2,3, and 4), respectively.

Definition 3. A *node path n* for a node *v* is a dot-separated alternating sequence of tags and serial numbers $l_1. s_1.l_2.s_2. \ldots . l_i.s_i$ ($i \geq 1$), of the nodes on the path from the root node to *v* node. For example, the node path of node <4.31.3> is *students.1.student.2.courses.2.course.3*. The tag path *t* of a node path *n*, denoted *t(n)*, is the sequence of tags that exist in *n*. For example, *t(n)* of *students.1.student.2.courses.2.course.3* is *students.student.courses.course*. Similarly, the serial path *r* of node path *n*, denoted *r(n)*, is the sequence of serial numbers that exist in *n*. For example, *r(n)* of *students.1.student.2.courses.2.course.3* is *1.2.2.3*.

Definition 4. A node with a node path *n* is an *instance* of a tag path *t* if the sequence of the tag path in *n* is identical to the sequence of the tag path *t*, *t(n)=t*. For example, the node paths of nodes <4.31.2> and <4.31.3> are instances of the tag path *students.student.courses.course*.

Definition 5. *Extension* of a tag path *t*, denoted *ext(t)*, is a set of all node path instances of a tag path *t*, that is, $ext(t)=\{n : t(n)=t \}$. For example, the extensions of the tag path *students.student.courses.course* are nodes <4.31.1>,<4.31.2>, and <4.31.3>.

In LTIX, an XML data-tree *G* structure can be summarized by a path index *I* such that all node paths of *G* that share the same tag path *t* are represented by exactly one tag path *t* in *I*, and every tag path *t* of *I* is a tag path of at least one node path *n* of *G*. That is, every distinct path in the source data to appear only once in the path index, and all the paths in the path index have at least one matching path in the original source data. Basically, *G* nodes are partitioned into equivalence classes in *I* where the nodes of a class have the same root path.

We define *path index* as a directed graph *I=(R,O,M,tagi,labeli,C)*. *R* is the same as the data graph *G* root element, since XML document can have only one root element. *O* is the set of index nodes excluding *R*. *M* is the set of child-parent edges that connects *O* nodes to form a tree. $|M|=|O|$, where $|M|$ is the total number of edges in the index tree and $|O|$ is the total number of nodes in the index tree. Nodes in *O* are tagged through the *tagi* function. We tag *O* nodes with the tag name of the element or attribute they extend. All nodes in the path index are assigned a unique label through the *labeli* function, which is determined by LLS labeling scheme as follows. Each node's label consist of two parts vector <*d.p*>, where *d* is the level (depth) of the node, and *p* is the number of this node across *d* level (denoted *PerLv*). An edge *m* of a node *o*, where $m \in M$ and $o \in O$, is equal to the *p* value of the parent node *x*, denoted *m(o)=p(x)*. *C* is the set of counts of instances for each node in *O*, that is, $C=\{c_1,c_2, \dots , c_i : i =|O|\}$. For each node o_j, and count c_j, where $o_j \in O$ and $c_j \in C$, c_j is the count of instances of the tag path t_j of node o_j, where $O=\{o_1,o_2, \dots , o_i : i =|O|\}$, $t=\{l_1,l_2, \dots , l_i : i=|O|\}$, and node o_j has tag path l_j. If we assume that in *O* there is a node o_j whose count of instances is c_j, and c_j value is *x*, then the *s* values of the instances of o_j would be 1 for the first instance, 2 for the second instance, … , and *x* for the last instance. Figure 7 contains an example of a path index *I* of the XML data-tree *G* in Figure 6.

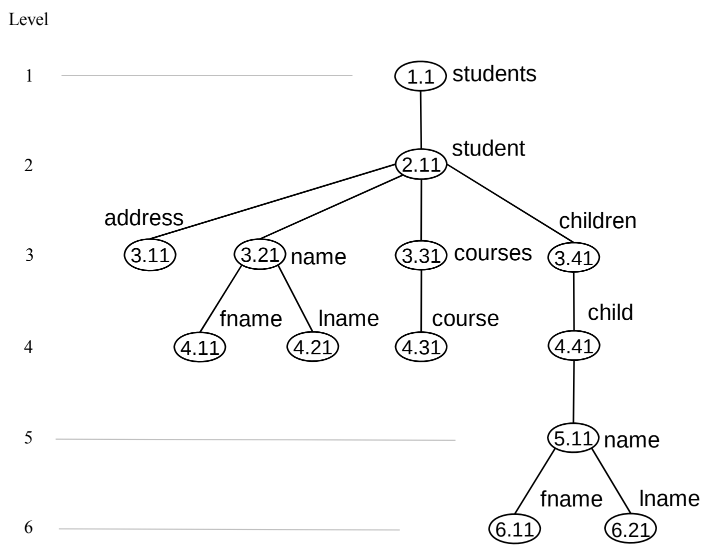


Figure 7. The path index *I* of the XML data-tree *G* in Figure 6

For each node o_i in *I* that has a label $<d_i.p_i>$, there are instances in *G* that have labels in the form $<d_g.p_g.s_g>$, such that $d_i=d_g$, $p_i=p_g$, and $s_g=\{1,2, \dots ,n\}$ where *n* equal to the count of instances of o_i, that is, $n=c_i$. For example, the numbers inside the oval shaped nodes in Figures 6 and 7 represent the labels of the nodes according to *labelg* and *labeli* functions, respectively.

Note that the labels of the path index nodes in Figure 7 are created first, and then used to create the labels for the data-tree nodes in Figure 6. The gaps between the *PerLv* numbers in Figure 7 allow for expansion while maintaining the order of the elements. The path index *I* information of Figure 7 is mapped into table representation as shown by the *Path Index,* and *Elements and Attributes Dictionary* tables in Figure 8 (B and A), and the data-tree *G* information of Figure 6 is mapped into table representation as shown by the *Value Index*, and *Elements and Attribute Index* tables in Figure 8 (C and D). We implement the *path Index* as a binary file; and the *Elements and Attributes Dictionary, Value Index,* and *Elements and Attributes Index* as B+trees in our LTIX system. The key of each index is underlined in Figure 8.

Tag	*Level*	*PerLv*	*Type*
address	3	11	A
child	4	41	E
children	3	41	E
course	4	31	E
courses	3	31	E
fname	4	11	E
fname	6	11	E
lname	4	21	E
lname	6	21	E
name	3	21	E
name	5	11	E
student	2	11	E
students	1	1	E

(A) Elements and Attributes Dictionary

Level	*PerLv*	*Parent*
1	1	0
2	11	1
3	11	11
3	21	11
3	31	11
3	41	11
4	11	21
4	21	21
4	31	31
4	41	41
5	11	41
6	11	11
6	21	11

(B) Path Index

Level	***PerLv***	*No*	***Value***	*SerialPath*
3	11	1	Kingston	1,1,1
3	11	2	Ottawa	1,2,2
4	11	1	Tim	1,1,1,1
4	11	2	Sarah	1,2,2,2
4	21	1	Wang	1,1,1,1
4	21	2	Ahmad	1,2,2,2
4	31	1	Art	1,1,1,1
4	31	2	History	1,1,1,2
4	31	3	Math	1,2,2,3
6	11	1	Mike	1,2,1,1,1,1
6	21	1	Salem	1,2,1,1,1,1

(C) Value Index

Level	***PerLv***	***No***	*Parent*
1	1	1	0
2	11	1	1
2	11	2	1
3	11	1	1
3	11	2	2
3	21	1	1
3	21	2	2
3	31	1	1
3	31	2	2
3	41	1	1
4	11	1	1
⋮	⋮	⋮	⋮
4	41	1	1
5	11	1	1
6	11	1	1
6	21	1	1

(D) Elements and Attributes Index

Figure 8. Data dictionary and indexes

Figure 8 (A) contains a list of all elements and attributes in the path index and is referred to as the *Elements and Attributes Dictionary*. The *Tag* field is the tag of the elements or attributes of the nodes in the path index, which is assigned through the *tagi* function of *I*. The *Level* and *PerLv* columns represent the *d* and the *p* parts of the path index nodes labels, respectively, as indicated in Figure 7. These labels are allocated through the *labeli* function of *I*. The *Type* represents the type of node (e.g. element or attribute).

The *Parent* field in Figure 8 (B) holds the *PerLv* labels of the parent nodes, which is the *p* values of the parent nodes. The *Level* value of the parent node is equal to the current node *Level* value minus one, so we do not need to list the parent node level in the *Path Index*. We assigned a zero value for the parent of the root node since it does not have any parent. Tables A and B in Figure 8 could be combined, but we prefer to keep them separate because they are used in different stages of queries evaluation process.

Figure 8 (C) shows the *Value Index* table, which is populated with the values of attributes and elements of the XML data-tree in Figure 6. The *Level*, *PerLv*, and *No* values together form the labels of the leaf nodes <*d.p.s*>, as shown in the data-tree in Figure 6. These labels are allocated through the *labelg* function of *G*. The *Value* field contains the values of all leaf nodes, and the values of internal nodes in the case where complex elements exist. Note that the value labels (which consist of *Level*, *PerLv*, and *No*) are the same as the element or attribute labels to which they belong. Finally, the *SerialPath* field contains the serial paths *r* of each node in the tree. It represents a vector of the *No* values of the nodes that constitute a path from the root node to the designated node. It is used in structural joins to solve twig queries, as we shall see in the forthcoming examples in Section (3.2).

All nodes in the XML tree are represented by the *Elements and Attributes Index* as shown in Figure 8(D). The *Elements and Attributes Index* can be extended to have the serial paths of all attributes and elements similar to the serial paths of values, but it is not necessary in our approach. Note that the *Parent* values in table (D) are different than the *Parent* values in table (B). In table (D) they stand for the *No* values of the parent nodes.

Definition 6. In order to achieve high performance of the LTIX index structure, and since an *s* value uniquely identifies a node among other nodes of the same class, we define the *Serial Path Index* that is based on the concatenation of (*Level*, *PerLv, SerialPath*) of values. The *Serial Path Index* is used to facilitate the link between two arbitrary nodes in two different twigs of a twig query.

Our index structure covers nodes that belong to the same XML document; the extension to multiple documents is straightforward. Our index structure of LTIX system consists of five components: The *Elements and Attributes Dictionary*, *Path index*, *Value Index*, *Elements and Attributes Index*, and *Serial Path Index*. By using the LTIX index structure, we can retrieve any leaf node, and reconstruct any internal node *v* along with the subtree that is rooted at *v*.

3.2 Two Simple Examples

In this section we trace two examples. The first example shows how LTIX is used to minimize the number of join and comparison operations. The second example illustrates the power of LTIX approach in pruning false positive early during the evaluation process.

Example 1: We evaluate Query 1 below, which was introduced in Section2, over the mapped data in Figure 8. This query returns the first name and the last name of all students.

Query 1: *//student /name* [*fname*] */lname*

Note that the branching occurs at the *name* node, which is the parent of the two leaf nodes, namely, *fname* and *lname*. We can see from the *Attributes and Elements Dictionary*, and the *Path Index* that the *fname* and *lname* elements in Query 1 map to nodes <4.11> and <4.21> in *I*, respectively. First, we evaluate one side by probing the key fields of the *Value Index* for values whose labels start with ("4.11") and the two returned tuples are:

Level	*PerLv*	*No*	*Value*	*SerialPath*
4	11	1	Tim	1.1.1.1
4	11	2	Sarah	1.2.2.2

These tuples are joined with the *Value Index* to arrive at the final answer of the query. In order to do that, the information of these two tuples is used by the index structure as follows. We know that the *Serial Path Index* is based on the concatenations of (*Level*,*PerLv*,*SerialPath*) columns. So the index structure probes the indexed columns in the *Serial Path* index for tuples that match (4*,21,LIKE 1.1.1%*), which is retrieved in one match. The *LIKE 1.1.1%* part retrieves all *SerialPaths* values that start with *1.1.1*. The search for a match to the second tuple is carried out in the same way by the search criterion (*4,21, LIKE 1.2.2%*), which retrieves the answer to this part in one match. The first three segments of the *serialpath* ("*1.1.1*" and "*1.2.2*") are used in the search criteria because the branching node is located at the third level. This means that the first three segments of the *SerialPath* of the two branches of the query are common and shared by the two branches.

Our approach, in contrast to the two approaches discussed previously in Section 2 – the (*start,end*) interval node index approach, and the approach that integrates the (*start,end*) interval node index with the DataGuide path index – performs only one join during which two matches are performed to evaluate the query. In our approach, the leaf nodes of the two branches are matched directly with each other without using the branching node as a mediator to join them, as opposed to the previous approaches. Further, the information of tuples obtained from evaluating the first branch leaf node is used to retrieve the exact match in one comparison for each match by using equality operator. The previous approaches, in contrast, requires multiple comparisons to find a match, since there are two parameters involved in the searching criteria (*start*, and *end*), and both are involved in an inequality comparison (*less than "<"*, or *greater than">"*).

Note that the first two fields of the *Value Index* key and the *SerialPath Index* key are the same (*Level*, *PerLv*). This fact increases the chances of a successful memory hit when the search criteria run against the *SerialPath Index* are met by multiple tuples, and thus decreases the number of disk accesses. This clustering helps to explain the shortest response time achieved by LTIX system in comparison to the previous approaches as shown in our experiments in Section 4.

Example 2: The level of XML elements in path indexes can be used to identify the element's position within an XML tree structure, and can facilitate effective query optimization through early pruning of the space search. To demonstrate that, we evaluate Query 2 below over the mapped data in Figure 8. This query returns the values of *fname* element that has a *child* element ancestor:

Query 2: *//child//fname*

Based on whether the used path index carries the level information of the indexed elements or it does not, we have two scenarios to evaluate Query 2. First, if we assume that we do not have the level information in the path index *I*, or we have it, but we do not access it in an efficient way at an early stage of the evaluation process of a query, then we solve Query 2 as follows. We would have to access the path index and search for all *child* and *fname* elements. In this process node <4.41> of *child* element, plus nodes <4.11> and <6.11> of *fname* elements are retrieved and investigated. Then node <4.11> would be excluded as the structure index would indicate that it is not a valid choice. The second scenario takes place if we assume that the levels of the path index nodes is given efficiently and at an early stage of the evaluation process, then we would need to match the extent of node <4.41> with only the extent of node <6.11>. The extent of node <4.11> would be excluded at an early stage since its level is equal to the level of <4.41> node, which contradicts with the query structure specification. Our index structure includes the level at which a node is located as part of the node label in the path index. Based on this fact, the evaluation algorithms of our LTIX approach detect invalid choices at an early step of the evaluation process and exclude them, thus improve the performance of query evaluation. To illustrate this, the

Elements and Attributes Dictionary table in Figure 8(A) can be used in our approach to evaluate Query 2 as follows. If we follow a top-down evaluation plan, then we would use the *Elements and Attributes Dictionary* table to find the *child* element information first. The search will return the following tuple:

Tag	***Level***	***PerLv***	***Type***
child	4	41	E

This information is used as a predicate to search for *fname* elements that are located at a level greater than 4. This way, the node <4.11> in the path index of *fname* element is excluded instantly without retrieving it. In contrast, other approaches will eventually exclude it, but after retrieving and testing it.

3.3 LTIX Path Index Construction

Path indexes, in general, require a large amount of memory [11]. Motivated by this fact, versions of path indexes, called approximate indexes [10,13,15] have been proposed to reduce the memory requirements, but at the expense of accuracy. In this section we describe efficient implementations for the path indexes in LTIX.

The size of path indexes for regular data is relatively small. For example, in our experimental evaluation, the path indexes for the DBLP and XMark databases contain 71 and 251 tuples, respectively. For irregular data, on the other hand, in which the same pattern is not repeated very often in the data, the size of the path indexes may be close to the size of the original indexed data, depending on the regularity of data. Our path index design covers all possibilities. Path indexes, furthermore, are used heavily in XML queries evaluation, especially for paths that have recursion. Finally, since the labeling scheme of our approach is based on the level of XML elements, our algorithms use path indexes more often than other approaches to evaluate XML queries. Based on these facts, we devoted this section to discuss different alternatives to minimize the size of the path indexes without compromising their accuracy..

We propose two types of implementations for path indexes. The first is called *Matrix Index* and the second is called *Flat Index*. Figures 10 and 11 are examples of the implementations of the first and the second types for the path index in Figure 9, respectively. Figure 9 is a portion of the path index in Figure 7. To simplify our examples we narrow the expansion gaps. The expansion gaps are reserved for inserting new nodes between the existing nodes while maintaining the order of the nodes.

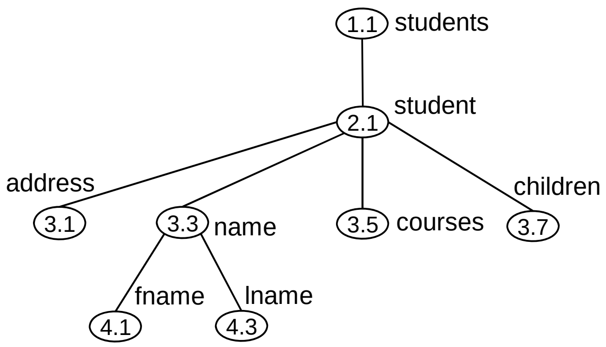
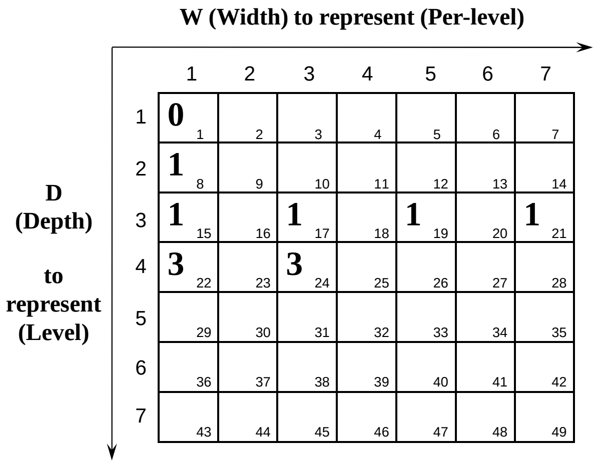


Figure 9. Fraction of the path index in Figure 7

In the matrix index (Figure 10), sequential memory slots (numbered in the bottom- right corner) are dealt with as if it is a matrix that has two coordinates. The *X* and *Y* axes coordinates represent the width (*PerLv*) and the depth (*Level*) of the path index. The first seven slots are reserved for the first level elements, the second seven for the second level elements, and so on. Each slot contains the *Parent* value for its corresponding node in the path index. The empty slots can be used for expansion. This index-probe operation is illustrated in Algorithm 1, which finds the parent node label for a given node. The matrix structure index does not have to have equal width and depth as in our example. The depth and the width of the matrix index may vary depending in the depth and the width of the path index tree, and the formed structure would still maintain the uniformity of access since the depth and the width are set ahead of time.


W (Width) to represent (Per-level)
D (Depth) to represent (Level)

	1	2	3	4	5	6	7
1	**0** 1	2	3	4	5	6	7
2	**1** 8	9	10	11	12	13	14
3	**1** 15	16	**1** 17	18	**1** 19	20	**1** 21
4	**3** 22	23	**3** 24	25	26	27	28
5	29	30	31	32	33	34	35
6	36	37	38	39	40	41	42
7	43	44	45	46	47	48	49

Figure 10. The *Matrix Index* structure that hold the *Parent* value of the path index in Figure 9

Algorithm 1: Find the parent node of a given node using the Matrix Index

```
// F(di,Pi)  is a function to find the parent node label for a given node.
// Input  :  (di,Pi) is the label of a node where di and Pi are the Level and
             the PerLevel of the input node, respectively.
// Output :  (do,Po) is the label of a node where do and Po are the Level
             and the PerLevel of the output(parent) node, respectively.
1  $d_o = d_i - 1$;                          // return the Level of the output node
2  $P_o = V(l) = V(((d_i-1) * W) + P_i)$     // return the PerLevel of the output node
```

Due to the increase in the number of nodes in the path index as levels increase, and due to the fact that this index structure width has to be the same for all levels, this index structure has limited control over the unused slots. Motivated by this fact, we propose the flat index (Figure 11) that divides the path index into three parts. In the first part, we save the number of levels of the path index (assume it is equal to *n*). The second part contains *n* storage units. These storage units are used to specify how many nodes there are in each level. For example, level 3 (in storage unit 4) has room for seven nodes. Finally, the third part contains the *PerLevel* of the parent node of all nodes in all levels, if they exist. Otherwise, *null* value is presented. This index-probe operation is illustrated in Algorithm 2.

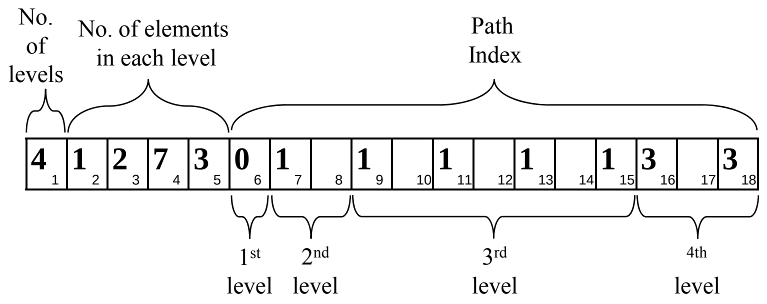


Figure 11. More efficient dynamic *Flat Index* structure for the path index in Figure 9

Algorithm 2 : Find the parent node of a given node using the Flat Index

```
// F(di,Pi)   is a function to find the parent node label for a given node.
// Input  :  (di,Pi) is the label of a node where di and Pi are the Level and the
             PerLevel of the input node, respectively.
// Output :  (do,Po) is the label of a node where do and Po are the Level and
             the PerLevel of the output (parent) node, respectively.
1  $Y <= V(1)$ ;            // Assign the value in storage unit 1 to variable Y
2  Target = $T = 0$ ;       // initialize the value of target level
3  For $k=2$ to $d_i$      // This loop is to find the address
4      { $T=V(k)+T$ }      // of the level specified by di
5  $d_o = d_i - 1$;        // returns the value of do
6  $P_o = V(1+Y+T+P_i)$    // returns the value of Po
```

In order to solve a query *Q*, for example, */students//fname* against the mapped data of *G* in Figure 8, we first verify that the two elements of *Q* exist in *I* (the path index of *G*). If so, we get their labels, which consist of two sets of labels, {<1,1>} and {<4,11>,<6,11>} for *students* and *fname* elements, respectively. We then use Algorithm 3 to verify if a relationship exists between the instances of these two sets of elements before going any further in the query evaluation. The function $R((d_1,p_1),(d_2,p_2))$ in Algorithm 3 is used to verify a child-parent or descendant-ancestor relationship between any two arbitrary nodes.

Algorithm 3: Confirm a relationship between two given nodes

```
// R((d1,p1),(d2,p2))  is a function to find if a relationship
                         exists between two arbitrary nodes.
// Input   :  (d1,p1) is the node in higher level and
              (d2,p2) is the node in lower level.
// Output  :  Boolean value: true if the relationship exists,
              or false otherwise.
 1  n = d2-d1;
 2  di=d2;
 3  Pi=P2 ;
 4  for t = 1 to n
 5     {  (do,po) = F(di,pi); // The function of Algorithm 1 or Algorithm 2 is used
 6        di=do;
 7        Pi=Po  }
 8  if (do==d1  and  po==p1)
 9      then return  true;
10      else  return  false;
```

The size of the matrix and the flat indexes are dependent on the number of spare space available for insertion. The more space we have, the more robust the path index will be, but at the expense of size. There is a trade-off between the path index size and its ability to adapt to insertion. Flat index structure, however, has more control over the index size.

We believe that these types of index structure representations are useful for XML databases. They transform the irregularity of XML databases into regular data that can be accessed uniformly. Moreover, the address of a node itself is used as part of information to reconstruct the index tree, that is, we used the address as a representation for *Level* and *PerLevel* information instead of saving them inside the file, and hence save memory. The size of the flat index is equal to $O(n + v)$ were *n* is the number of nodes in the path index tree and *v* is equal to the number of levels in the path index plus one. In real life situations, where the nodes in the path index can reach hundreds or thousands of nodes, v becomes negligible compared to n, and hence the size of the path index is approximately $O(n)$ of nodes that a path index can hold. Since our path index is based on the DataGuide path index, the size is relatively small for a regular data-tree, and grows linearly for irregular data-trees, but does not exceed the size of the source data in the worst case [9,11]. More on the cost of updating the index structure can be found in Mohammad and Martin [19].

Another alternative for building the path index is the B+trees, which handle growth gracefully. But the B+trees structure may require more number of accesses to retrieve specific information, which depends on the size of the tree that dictates the depth of the tree. In addition, B+trees require huge space. In contrast, our path index structures require one access to retrieve specific indexed information, and they require modest space. Our path index structure is similar to a dynamic hash index to some extent.

4. EXPERIMENTAL EVALUATION

All experiments were performed on a 3 GHz Pentium 4 PC running Windows XP operating system, with 1.49 GB of RAM. The goals of the experiments are to evaluate the performance of our LTIX method that uses the LLS labeling scheme. We therefore compare three different indexing methods. First, we implement a basic interval node index with the multiple predicate merge join (MPMRJN) algorithm proposed by Zhang et al. [27]. Second, we modify the MPMRJN algorithm to use the path index we described above. This allows us to observe the impact of our path index on performance. Third, we implement our LTIX method, which consists of the LLS labeling scheme and our path index. We evaluate the LLS labeling scheme's effect in the LTIX system by comparing it with the extended version of Zhang's interval labeling scheme. We have two different labeling schemes integrated with the same path index so performance differences should be due to the labeling schemes.

We use the Berkeley B+tree to store the data for the three schemes, and we use a binary file to store the path index. We evaluate each method against two test sets (see Section 4.1). We measure the performance using two implementation-independent criteria, namely the number of comparisons performed to establish relations between two elements and the number of cases pruned by the method, as well as the average runtime of a query with each method. The size of the tables is not measured since they depend on the B+tree implementation.

4.1 The Datasets and Queries

We execute our experiments using two datasets: the DBLP Computer Science Bibliography [25] dataset and the XMark [23] dataset with scale factor (0.1). Statistics for the two datasets are summarized in Table 1.

Table 1. Details of DBLP and XMark datasets

Testing Dataset	*Size*	*No of Elements in PathIndex*	*No of Levels*	*Total Number of Elements*	*Max Cardinality*	*Avg. Cardinality*
DBLP	20 MB	71	5	582,033	109,595	8,197
XMark	15 MB	251	11	185,225	6,183	737

XPath (XML Path Language) [6] is a flexible query language that has been proposed to access XML data. An XML query may consist of either a single path or multiple paths (twig path). Both single path and twig path queries can be recursive (i.e. support ancestor-descendent "//" relationships) or non-recursive. Based on these criteria, we can divide XML queries into the following four types.

Type 1 (T1): Single path non-recursive queries.
Type 2 (T2): Single path recursive queries.
Type 3 (T3): Twig path non-recursive queries.
Type 4 (T4): Twig path recursive queries.

Since most XML queries fall into these four types of queries, we use them in our experimental evaluation, and we run them against the DBLP and XMark datasets. For each type of query, we used 4 queries as shown in Figures 12(A) and 12(B). These queries are chosen to cover different combination of query path lengths, cardinality of elements, and the number of returned tuples, which is affected by the selectivity of elements. Figures 12(A) and 12(B) contain lists of the 4 types of queries, as specified by (T1,T2,T3, and T4).

T1-Q1 : */dblp/inproceedings/cdrom*
T1-Q2 : */dblp/inproceedings/cite/label*
T1-Q3 : */dblp/inproceedings/booktitle*
T1-Q4 : */dblp/book/series/href*
T2-Q1 : /dblp//author
T2-Q2 : *//series/href*
T2-Q3 : *//book//label*
T2-Q4 : *//href*
T3-Q1 : */dblp/incollection[/year='2000']/booktitle*
T3-Q2 : */dblp/proceedings[/booktitle='ACCV']/isbn*
T3-Q3 : */dblp/inproceedings[/author='Adele E. Howe']/title*
T3-Q4 : */dblp/proceedings[/isbn='0-7695-1991-1']/title*
T4-Q1 : *//inproceedings[/mdate='2002-08-04']/title*
T4-Q2 : *//proceedings[/booktitle='ACNS']/isbn*
T4-Q3 : *//incollection[/booktitle='Temporal Databases']/year*
T4-Q4 : *//incollection[/author='Jurgen Annevelink']/title*

(A). For DBLP database

T1-Q1 : */site/regions/africa/item/id*
T1-Q2 : */site/open_auctions/open_auction/bidder/personref/person*
T1-Q3 : */site/open_auctions/open_auction/seller/person*
T1-Q4 : */site/catgraph/edge/from*
T2-Q1 : *//id*
T2-Q2 : *//africa//category*
T2-Q3 : *//regions//item//text*
T2-Q4 : *//open_auctions//text*
T3-Q1 : */site/regions/africa/item[/location='United States']/payment*
T3-Q2 : */site/regions/africa/item[/id='item0'] /location*
T3-Q3 : */site/catgraph/edge[/from='category0']/to*
T3-Q4 : */site/people/person[/name='Kaj Carey']/phone*
T4-Q1 : *//africa/item[/quantity='1']/name*
T4-Q2 : *//open_auction[/reserve='3199.90']/initial*
T4-Q3 : *//closed_auction[/type='Regular']/price*
T4-Q4 : *//regions//item[/quantity='2']/name*

(B). For XMark database

Figure 12. Representative queries for 4 types of queries

4.2 Performance Evaluation

We execute each query ten times against its respective dataset and take the average of the 10 readings. The average of each type of the 4 types of queries is used in our analysis. Tables 2(A) and 2(B) show the results of the testing of DBLP and XMark test cases, respectively. The results include the number of pruned cases, the average number of comparison operations, and the average runtime of the test cases. The pruning is due to the use of the path indexes and the information about the elements' levels. We notice that the number of pruned cases in DBLP dataset is less than those of XMark datasets. This is due to two factors. First, the number of levels is higher in the XMark dataset. Second, the number of repetitive element names (elements with the same name) is also higher in the XMark dataset. Since more elements are tested in the twig queries, we notice that the number of pruned cases for twig queries is more than those of single path queries for both datasets.

In both test cases, the number of row pairs compared drops to zero for both types of single path queries (T1 and T2) when the path index is incorporated, and hence the performance of our approach is similar to that of the extended approach. The basic interval node indexes require significantly more comparisons than the extended interval node indexes and LTIXs because, with the latter two indexes, the correct set of answers is identified by the path index. During this process, the labels of the nodes (which consist of *Level* and *PerLevel* parts) that match the exact answer criteria are identified by using the path index, then used to retrieve the answer from the data in the B+tree index. The data is clustered in the B+tree by these labels so the retrieval times are much smaller than those of the basic interval node indexes.

We see similar performance improvements for both types of twig queries (T3 and T4) in both test cases. The extended interval node indexes compare less row pairs than the basic interval node indexes (79%-90% less comparisons), and LTIXs compares 97.9%-99.9% less pairs than the extended interval node indexes. Similarly, extended interval node indexes perform 46%-78% faster than the basic interval node indexes, and our approach outperforms the extended interval node indexes by 89%-99.6%.

Tables 2. Average pruned cases, comparisons, and runtime for 4 types of queries.

(A). Against DBLP dataset.

Query Type	*Average Pruned Cases*	*Avg. No of Comparisons*			*Avg. Runtime (msec)*		
		Basic Interval	*Extended Interval*	*LTIX*	*Basic Interval*	*Extended Interval*	*LTIX*
T1	5	105,684	0	0	22,108	6	6
T2	2	32,376	0	0	15,545	12	12
T3	13	322,491	67,966	17	39,503	8,543	32
T4	14	316,409	67,384	61	35,016	18,991	96

(B). Against XMark dataset.

Query Type	*Average Pruned Cases*	*Avg. No of Comparisons*			*Avg. Runtime (msec)*		
		Basic Interval	*Extended Interval*	*LTIX*	*Basic Interval*	*Extended Interval*	*LTIX*
T1	29	25,915	0	0	3,250	7	7
T2	69	7,003	0	0	1,426	48	48
T3	205	32,805	3,272	12	2,304	738	22
T4	140	81,022	7,902	168	5,241	1,732	191

The experimental results of the twig queries (T3 and T4) show that in the case of the DBLP dataset, our approach performs 99.5% - 99.6% faster than the extended approach, while in the case of XMark dataset our approach performs 89% - 97% faster than the extended approach. This is because the XMark dataset is text oriented where the size of data is very large and it exceeds 7,000 characters for many elements; while the DBLP dataset is record-oriented and the size of the data items is often short (e.g. name, title, date).

We believe that the performance gain of LTIX, as noted in tables 2(A) and 2(B), is achieved mainly by two factors in our index structure. First, the LTIX path index is based on the levels of XML elements, which is used to prune out false positive cases early in the evaluation process. Second, multiple inequality comparisons are performed to find a match for a node using the basic and the extended node indexes, while LTIX only requires one equality comparison to find a match for a node.

5. CONCLUSIONS

Unlike other type of indexes such as node indexes [24,27], path indexes [9,8,17,10,15,13], and sequence indexes [26,22], our index structure strengthens the importance of levels in the structural index by including it as part of the node identifications (labels). The knowledge of the level can be used by an XML query optimizer to select those nodes whose level indicates a potential answer for a given query, and to eliminate the nodes that violate the element level order given by the query. Unlike previous approaches that integrate interval node indexes with path indexes [12,14,20], our approach does not have to have the structural information available in both indexes. This allows us to provide a structure independent labeling scheme for our approach that performs better in evaluating XML queries.

Query optimization in the context of XML databases is extremely challenging because the complexity of the XML data model leads to much larger search spaces for XML query optimization [4]. Experimental evaluations of our system show that LTIX system is capable of pruning improper and inefficient plans at the early stages of query optimization process. Previous approaches that use a universal label scheme across the complete document result in large labels for large documents. In contrast, in our approach we split the labels into groups of smaller numbers that require less memory and are easier to maintain and process than large labels. The join operations are carried out by using equality operator instead of inequality operator, which is used by most XML indexing approaches [11].

Much research has been done to propose a persistent labeling scheme for dynamic XML data to avoid the relabeling cost [7,21]. Cohen et al. [7] established that any persistent labeling scheme requires $\Omega(N)$ bits per label in the absence of any clues about the data, where N is the size of the data. Such long labels, however, require high storage in addition to being more expensive to process than the shorter one. In contrast, our labeling scheme in LTIX, which is tightly coupled with the summary index, requires a constant label size to cover dynamic data. The worst case update cost of LTIX requires relabeling fewer nodes than that of the interval labeling scheme [19].

We showed experimentally that the LTIX system outperforms the basic interval node indexes as proposed by Zhang et al. [27] and the extended interval node indexes. It works well for single path queries as well as for twig queries. For a future work, we are planning to use a customized XML storage media for the LTIX system, instead of using B+tree storage media. We are also planning to extend our indexing scheme to be adapted gracefully by relational database systems in term of storage and querying by building an engine that translate XPath queries into SQL queries where the hierarchy of XML paths are reflected properly.

6. ACKNOWLEDGMENTS

This work was supported by the Natural Science and Engineering Research Council of Canada.

7. REFERENCES

[1] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML.* Morgan Kaufmann Publishers. 2002.

[2] S. Al-Khalifa, H.V. Jagadish, N. Koudas, J.M. Patel, D. Srivastava, and Y. Wu. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proceedings of the 18th ICDE*, pages 141-154, 2002.

[3] N. Bruno, N. Koudas, and D. Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proceedings of the ACM SIGMOD.* Pages 310-321, 2002.

[4] D. Che, K. Aberer, and M.T. Ozsu. Query optimization in XML structured-document databases. *The VLDB Journal*, 15(3), 263-289, 2006.

[5] S. Chien, Z. Vagena, D. Zhang, V. Tsotras, and C.Zaniolo. Efficient structural joins on indexed XML documents. In *Proceedings of 28th ICDE,* pages 263-274, 2002.

[6] http://www.w3.org/TR/xpath.

[7] E. Cohen, H. Kaplan, and T. Milo. Labeling Dynamic XML Trees. In *Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 271-281, 2002.

[8] B. Cooper, N. Sample, M. Franklin, G. Hjaltason, and M. Shadmon. M. A Fast Index for Semistructured Data. In *Proceedings of 27th VLDB Conference*, pages 341-350, 2001.

[9] R. Goldman, and J. Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *Proceedings of 23rd VLDB Conference*, pages 436-445, 1997.

[10] R. Goldman, and J. Widom, Approximate Data Guide. In *Proceedings of the Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats*, 1999.

[11] G. Gou, and R. Chirkova. Efficiently Querying Large XML Data Repositories: A Survey. *Transactions on Knowledge and Data Engineering*, 19(10): 1381-1403, 2007.

[12] S. Haw, and C. Lee. Extending path index and region encoding for efficient structural query processing in native XML databases. *The Journal of Systems and Software*. 82(2009): 1025-1035, 2009.

[13] R. Kaushik, P. Bohannon, J. Naughton, and H. Korth. Covering indexes for branching path queries. In *Proceedings of the ACM SIGMOD ICMD*, pages 133-144, 2002.

[14] R. Kaushik, R. Krishnamurthy, J. Naughton, and R. Ramakrishnan. On the integration of structure indexes and inverted lists. In *Proceedings of the ACM SIGMOD ICMD*, pages779-790, 2004.

[15] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting local similarity for indexing paths in graph-structured data. In *Proceedings of 18th ICDE*, pages 129-140, 2002.

[16] Q. Li, and B. Moon. Indexing and querying XML data for regular path expressions. In *Proceedings of 27th VLDB Conference*, pages 361-370, 2001.

[17] T. Milo, and D. Suciu. Index Structures for Path Expressions. In *Database Theory –ICDT'99, Proceedings of 7th ICDT*, pages 277-295, 1999.

[18] S. Mohammad, and P. Martin. Index structures for XML Databases. In Li, C., and Ling, T. W. (Eds.). *Advanced Applications and Structures in XML processing: Label Streams, Semantics Utilization and Data Query Technologies*. IGI Global, 2009.

[19] S. Mohammad & P. Martin. LLS: A Level-based Labeling Scheme for XML Databases. Submitted to *CASCON 2010*, Toronto, Canada, 2010.

[20] M. Moro, Z. Vagena, and V. Tsotras. Tree-Pattern Queries on a Lightweight XML Processor. In *Proceedings of the 31st VLDB Conference*, pages 205-216, 2005.

[21] P. O'Neil, E. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. ORDPATHs: Insert-Friendly XML Node Labels. In *Proceedings of the ACM SIGMOD ICMD*, pages 903-908, 2004.

[22] P. Rao, and B. Moon. PRIX: Indexing and querying XML using Prufer sequences. In *Proceedings of the 20th ICDE* 2004, pages 288-300, 2004.

[23] A. Schmidt, F. Waas, M. Kersten, D. Florescu, I. Manolescu, M. Carey, and R. Busse. *The XML Benchmark Project.* Technical Report INS-R0103, 2001.

[24] I. Tatarinov, S. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and Querying Ordered XML Using a relational Database System. In *Proceedings of the ACM SIGMOD ICMD*, pages 204-215, 2002.

[25] http://www.informatik.uni-trier.de/~ley/db/.

[26] H. Wang, S. Park, W. Fan, and P. Yu. ViST: A Dynamic Index Method for Querying XML Data by Tree Structures. In *Proceedings of the ACM SIGMOD ICMD*, pages 110-121, 2003.

[27] C. Zhang, R. Naughton, D. Dewitt, Q. Luo, and G. Lohman. On Supporting containment Queries in Relational Database Management Systems. In *Proceedings of ACM SIGMOD ICMD*, pages 425-436, 2001.