Relational Universal Index Structure for Evaluating XML Twig Queries

Samir Mohammad
Patrick Martin
Wendy Powley

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

Technical report No. 2010-576

May 2010

Relational Universal Index Structure for Evaluating XML Twig Queries

Samir Mohammad, Patrick Martin, Wendy Powley

School of Computing, Queen's University, Canada

{samir,martin,wendy}@cs.queensu.ca

***Abstract*— Numerous approaches to storing XML data in relational databases have been proposed that take advantage of the maturity of relational database management systems. Index structures to support these approaches have been developed to speed-up XML query processing. Typical drawbacks of these approaches include the lack of support for twig queries and the large storage requirements for the index structures. In this report we propose a novel index structure that is compact and effectively supports processing of XML twig queries. Experimental results show that our approach achieves lower response time than similar approaches while using less space to store the XML data.**

I. INTRODUCTION

Due to its flexibility, XML is becoming the standard for exchanging data over the World Wide Web. XML data can be stored and queried by using either native XML repositories [2,7,18,20,22,31], or relational database management systems [3,6,10,19,23,24,25,26,29,30]. Native approaches for storing and querying XML data are still relatively new. On the other hand, relational database management systems are well founded, tuned, and standardized by several decades of work. In addition, huge volumes of data are already stored in relational database management systems. Motivated by these facts, researchers and vendors (such as IBM®, Oracle®, Sybase®, and Microsoft®) are working on ways to improve the capabilities of RDBMS to store and retrieve XML [1,5,6,10,19,23,24,25,26,29,30].

Elements in XML data are linked through a hierarchical structure. Any two elements are linked through their common ancestor. Therefore, indexing common ancestors can facilitate the evaluation of twig queries. For example, consider the XPath query below:

Query 1: *//student* [*/ fname = 'Sue' and lname = 'Jones'*] */ program*

This query returns the program of the student *Sue Jones*. Its pattern can be represented as a node-labeled tree as shown in Figure 1. A single line represents a parent-child relation and a double line represents an ancestor-descendent relation.

Figure 2 contains an XML document, which is represented as the hierarchical node-labeled tree in Figure 3. The node labels are shown inside the nodes of Figure 3. Query 1 can be evaluated over the data in Figure 3 as follows. We first evaluate the branch with *fname='Sue'*. This part returns the node <4.2.3> and the branching node <3.1.3>, assuming that all branching nodes for each node in the data-tree are recorded in the database. We call the branch that is evaluated first the *base branch*, and the branch(es) evaluated afterward the *secondary branch(es)*.

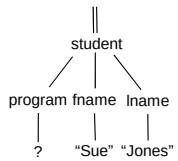


Fig. 1 Query 1 hierarchical pattern

Now to solve the second branch *lname='Jones'*, we have to search for the *lname* element that has a value "*Jones"* and whose parent node label is <3.1.3>. The only node that matches these criteria is node <4.3.3>. Note that the other two nodes that have the same last name *Jones*, namely, nodes <4.3.1> and <4.3.2>, are excluded early in the search because their parent node is not <3.1.3>. Finally, we search for branch *program* relative to its parent <3.1.3>. So the value "*CS*" is returned as the final answer.

```
<course number="251">
   <name>XML</name>
   <students>
      <student>
         <program>Math</program>
         <fname>Omar</fname>
         <lname>Jones</lname>
      </student>
      <student>
         <program>Physics</program>
         <fname>Ayah</fname>
         <lname>Jones</lname>
      </student>
      <student>
         <program>CS</program>
         <fname>Sue</fname>
         <lname>Jones</lname>
      </student>
   </students>
   <instructor>Beth</instructor>
</course>
```

Fig. 2 An XML document

From the above example, we can see that twig queries can be evaluated by using knowledge of their branching nodes. We propose an approach that utilizes this idea to evaluate twig queries efficiently by building a Universal Index Structure for XML databases (UISX). This index structure guarantees to find a *complete* and *precise* match for each node of any

arbitrary base branch by executing a single index lookup. That is, all matching tuples are retrieved without any false positives.

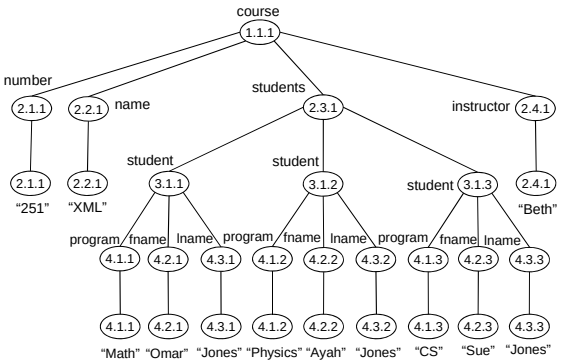


Fig. 3 The data-tree representation of the XML document in Figure 2

Finding matching elements of a twig is a core operation in XML query processing [3]. Much research has been done to match elements at different branches of twig queries [3,4,8,12,14]. Generally, these approaches suffer from either producing structures that are physically large to support twig queries (Queries with multiple paths) efficiently, or not being able to support twig queries as efficiently as they support single path queries as a consequence of reducing the size. A good study of the trade-off between index space and evaluation efficiency is given by Chen et al. [3]. They implement two index structures: ROOTPATHS and DATAPATHS. ROOTPATHS has small size, but it is not as efficient as DATAPATHS, whose size is much larger. The reason behind the DATAPATHS superior performance is the fact that it indexes all possible subpaths of root-to-leaf paths, which are used to match any two arbitrary branches.

Our proposed approach has a compact size, yet, it supports efficient evaluation of twig queries. It uses a RDBMS to store and query XML data. We use path summaries based on DataGuides [8], to facilitate query evaluation. The path summary, which is modeled as a simple table in a relational database, reduces the number of matches required to evaluate a query by preserving a path summary of the original XML data structure before shredding. Path summaries reduce the size of the stored XML databases. This reduction in size is achieved by: (1) eliminating redundant data from the database, such as the path of an element, which can be regenerated when needed from the summary; and (2) by using the summary to regenerate the internal nodes of the XML data-tree along with their subtrees. Therefore, internal nodes do not need to be shredded and stored in relational tables. In our approach, only the leaf nodes are shredded and stored in relational tables. The root-paths are recorded for all leaf nodes, where the information of the internal nodes is encoded. Zhang et al. [30] observed that RDBMSs do not support the inequality-joins efficiently, while they support the equality-joins efficiently. Our XML-relational approach evaluates XML queries by using equijoins, while most XML-relational approaches use inequality-joins [9,30].

The UISX index structure has been implemented successfully using the DB2® DBMS [11], and the experimental results show that it performs well in comparison to existing state of the art approaches in terms of size and response time. The contributions of this report are as follows:

- A novel index structure for storing and querying XML data, where all nodes of XML data-trees are indexed in relation to their branching nodes.
- A unique way of storing and using XML path summaries to facilitate query processing.
- An efficient storage method.

The rest of this report is organized as follows. Section 2 discusses related work. Section 3 introduces the UISX approach, explains the XML data and path summary models used to build UISX, introduces XPath query expressions, explains how the proposed index structure optimizes the use of the space to store XML data, and illustrates how queries are evaluated by using this index structure. Section 4 presents an experimental evaluation of the UISX approach in comparison to existing approaches. Finally, Section 5 presents the conclusions and outlines future work.

II. Related Work

RDBMSs are known for their strength in data storage and manipulation, query processing and optimization, concurrency control, recovery, and security. Consequently, many research projects have proposed mapping XML data to RDBMSs. These proposals can be divided into two groups: mappings that are based on the schemas of XML data, which are referred to as *structure-mappings*; and mappings that are not based on XML schemas, which are referred to as *model-mappings*. In structure-mapping, XML data is mapped to different relational schemas depending on the existing XML schemas. In model-mapping, the XML data is mapped to the same relational schema regardless of the structure of the mapped data, whether an XML schema exists or not. Shanmugasundaram et al. [24] and Florescu et al. [6] proposed two of the early approaches for mapping XML data. The first approach is based on structure-mapping, and the latter is based on model-mapping. Our approach is based on model-mapping.

There are three types of model-mapping approaches: edge, node, and path approaches. The edge model-mapping approach proposed by Florescu and Kossmann [6] is based on the edge-labeled data model. It maps all edges in an XML data-tree into a single relational table that has the scheme (*Source*,*Target*,*Tag*,*Flag*,*Value*). Each edge represents an element that has a *Source* and *Target* identification. An XPath query is evaluated by matching the *Target* id of one element (edge) with the *Source* id of the following element in the path of a query starting from one end and finishing at the other end. The *Flag* represents the type of the node (e.g. int, string). The

edge approach requires a minimum of *n-1* join operations to evaluate a query with *n* elements for both single path and twig path queries. In addition, it does not efficiently evaluate queries with the ancestor-descendent "//" axis.

Zhang et al. [30] proposed a model-mapping approach based on the node-labeled data model. They use intervals to label the nodes and map XML tree elements to a relational table that has the scheme (*Start*,*End*,*Tag*,*Level*,*Value*). Two elements can be joined together if the interval (*Start*,*End*) of one element contains the other element's interval. Unlike the edge approach, node model-mapping can efficiently evaluate queries with the ancestor-descendent "//" axis, but it still requires *n-1* joins to evaluate a single path or a twig path query with *n* elements.

Yoshikawa et al. [29] proposed a model-mapping approach that is based on forward-paths of elements in an XML data-tree. A forward-path is a path that starts from an element in the higher part of an XML data-tree (e.g. the root element) and ends at an element at the lower part (e.g. the mapped element). In this approach, elements are shredded into a relational table with the scheme (*Path*,*Start*,*End*,*Value*). Each element is identified by its root-path (which is a forward-path). Single path queries are evaluated with one match. Twig queries, however, are evaluated by decomposing the twig into multiple single paths. Each path is evaluated separately and then joined together to obtain the final answer. The number of joins required to evaluate a twig query is usually equal to the number of branches in the query. The forward-paths approach reduces the number of joins required to evaluate a query, however, it may produce incorrect answers when recursion exists in XML data [9]. To overcome this problem Pal et al. [19] proposed a similar approach using reversed-paths instead of forward-paths. A reversed path is a path that starts from an element at a lower part in an XML data-tree and ends at an element in a higher part. The reversed-paths approach not only eliminates the possibility of producing false results, but also improves the performance of query evaluation. The reversed-paths approach has been used by IBM® System RX, Microsoft® SQL Server 2005, and Oracle® DB [9].

Chen et al. [3] used a reversed-path approach where each node in an XML data-tree is given a global id, and then shredded into relational tuples with the scheme (*HeadId*, *SchemaPath*, *LeafValue*, *IdList*). The *HeadId* is the id of the node at which a reversed-paths ends, *SchemaPath* represents the reversed-paths of XML data nodes, *LeafValue* represents the values of the leaf nodes in the path of the mapped elements, and *IdList* contains lists of the global ids of the nodes that constitute a path from the *HeadId* to the designated mapped nodes. Two index structures were proposed. The first was the ROOTPATHS index, which indexed only the prefixes of the root-to-leaf paths. The second was the DATAPATHS index, which indexed all subpaths of root-to-leaf paths, including the root-to-leaf paths. The key idea of this approach is to create an index for all branching nodes. To process a twig query, in the case of ROOTPATHS index, all branches are evaluated and the returned *IdLists* are then merged or hash-joined to arrive at the final solution. In the case of DATAPATHS index, a twig query is processed by evaluating the base branch first to get the ids of the branching nodes which are available in the *IdList*. Then a search is carried out for the secondary paths that are rooted at the identified branching nodes and that have the exact reversed-path given in the query. The reversed-paths that are used to evaluate a twig query in DATAPATHS index start from the leaf node of the query and end at the branching nodes. The DATAPATHS index reduces access to the index to a single index lookup in order to find a match for fully specified, single path query without any recursion. Consequently, solving twig queries, which can be divided into multiple single path queries, requires a relatively small number of index lookups.

Chen's et al [3] index structure does not have a path summary table like our approach. Their approach, however, has a dictionary to encode schema paths by using special characters to designate elements and attributes instead of using the whole names. This dictionary has to be accessed at an early stage of an XML query evaluation process. Our approach, in contrast, uses the path summary table, which has approximately the same size as the dictionary table. The key idea of both approaches is to index all leaf nodes in relation to the branching nodes, and so minimize the number of index accesses required to evaluate a twig query.

III. Universal Index Structure For XML Data

Based on the observation that branching nodes are the key element in solving twig queries, we propose the UISX approach to efficiently match and join any two arbitrary nodes that share the same branching node. In this approach the base branch is evaluated first. Then, for each returned base branch node, the secondary branches are examined, and the matching nodes of each branch are located through their common ancestor node by using only one index lookup.

A. XML Data and Path Summary Models

In this subsection we describe our basic data model, and path summary. Then in the following subsections we discuss the query language, the size optimization, and the query processor of the UISX.

Definition 1. We model XML documents as trees. An *XML tree* is a directed ordered graph $G=(R,V_R,V_L,E,tagg,labelg,T)$. R is the root node. V_R is the set of internal nodes. V_L is the set of leaf nodes. $V_L=(V_E \cup V_T)$, that is, V_L consists of the empty leaf node V_E (for empty elements), and the set of value (text) leaf nodes V_T. Nodes in V_R and V_L are tagged through the *tagg* function (The extra *g* stands for *G*). V_R and V_E nodes are tagged according to the tag of the elements or attributes they represent. Nodes in V_T have the same tag as their V_R parent nodes. Internal nodes V_R have to have one or more child nodes, which could be V_R and/or V_L node(s). 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_L to form a tree. The total number of edges equal to $|E|$, where $|E|=|V_R| + |V_L|$, $|V_R|$ is the total number of internal nodes, and $|V_L|$ is the total number of leaf nodes in the tree. Each and

every node in V_R and V_L is associated with only one parent through an edge, since each node in a tree structure can have only one parent, except R, which does not have a parent[1].

Each node in V_R and V_L are assigned a unique label through the *labelg* function, which is determined by the LLS labeling scheme [16] as follows. Each node v, such that $v \in (V_R \cup V_E)$ is assigned a unique vector label $<d.p.s>$, where d and p are taken from the label of the o node in the path summary I to which v node belongs. That is, v node is an instance of an o node (instances and path summaries are defined shortly). s is the instance serial number of node o. Nodes in V_T are labeled according to the labels of their parent V_R 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 leaf nodes $|V_L|$. We define serial path r in Definition 3 below. In our model, an edge e of a node v, where $e \in E$ and $v \in V$, is equal to the serial number s of the parent node p, denoted $e(v)=s(p)$. The data-tree representation G of the data in Figure 2 is illustrated in Figure 3, which is used in the examples throughout this report, unless we specify otherwise.

Definition 2. 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. For example, the tag path of node <4.1.2> is *course.students.student.program*.

Definition 3. A *serial path* r for a node v is a sequence of serial numbers $s_1.s_2\ldots s_i$ ($i \geq 1$), of the nodes on the path from the root node to v node. For example, the serial path of node <4.1.2> in Figure 3 is (*1.1.2.2*). 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 (*1.1.2.2*) are (1,2,3,and 4), respectively.

Definition 4. A *node path* n for a node v is a sequence of alternating 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 <4.1.2> in Figure 3 is *course.1.students.1.student.2.program.2*. 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 *course.1.students.1.student.2.program.2* is *course.students.student.program*. 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 *course.1.students.1.student.2.program.2* is (*1.1.2.2*).

Definition 5. A node with a node path n is an *instance* of a tag path t if the sequence of the tag path of n is identical to the sequence of tag path t, that is, if $t(n)=t$. For example, the nodes <4.1.1> and <4.1.2> are instances of the tag path *course.students.student.program*.

Definition 6. *Extension* of a tag path t, denoted $ext(t)$, is a set of nodes whose node paths are instances of the tag path t, that is, $ext(t)=\{n : t(n)=t\}$. For example, the extension of tag path *course.students.student.program* includes nodes <4.1.1>, <4.1.2>, and <4.1.3>.

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

Definition 7. A *path summary* is a directed ordered tree $I=(O,M,tagi,labeli,C)$. O is the set of summary nodes. $O=(R \cup O_R \cup O_L)$, where R is the same as the data graph root element since a tree can have only one root element, O_R is the set of internal nodes, and O_L is the set of leaf nodes. M is a set of child-parent edges that connects O nodes to form a tree. $|M|=|O|-1$, where $|M|$ is the total number of edges in the summary tree and $|O|$ is the total number of nodes in the summary tree. Nodes in O are tagged through the *tagi* function. We refer to the tags of O nodes as the tag name of the element or attribute they extend. All nodes in the path summary are assigned a unique label through the *labeli* function, which is determined by the LLS labeling scheme [16] as follows. Each node's label consists 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. 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, $C=\{c_1,c_2, \ldots, c_i : i=|O|\}$. 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 $O=\{o_1,o_2, \ldots, o_i : i=|O|\}$, $t=\{t_1,t_2,\ldots,t_i : i=|O|\}$, and node o_j has tag path t_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 z, then the s values of the instances of o_j would be 1 for the first instance, 2 for the second instance, … , and z for the last instance. Figure 4 contains an example of a path summary I of the XML data-tree G in Figure 3. Note that V_T nodes in G are represented by their parent nodes.

In UISX, an XML data-tree G can be summarized by a path summary I such that the tag path t of every node path n of G has exactly one tag path t in I, and every tag path t of I is a tag path of a node path n of G. That is, every distinct path in the source data appears only once in the path summary, and all the paths in the summary 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 [8,17].

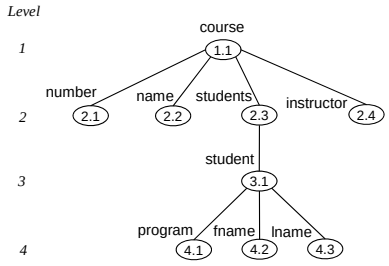


Fig. 4 The path summary of the data in Figures 2 and 3

For each node o_i in I that has 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 $s_g=\{1,2,\ldots,n\}$ where n equal to the count of instances of o_i, that is, $n=C_i$. Note that the labels of the summary nodes (e.g. nodes of Figure 4) are created first, and then used to create the labels for the data-tree nodes (e.g. nodes of Figure 3). For example, nodes <3.1.1>, <3.1.2>, and

<3.1.3> in *G* are extensions of the same node in the path summary *I*, namely, node <3.1>.

The path summary in Figure 4 is mapped to the relational table *PathSummary* as shown in Table 1. The leaf nodes data of Figure 3 is mapped to the relational table *LeafNodes* as shown in Table 2.

In Table 1, the *Tag* field contains the tags of the elements of the nodes in the summary, which is assigned through the *tagi* function of *I*. The *Level* and *PerLv* fields represent the *d* and the *p* parts of the path summary nodes labels as indicated in Figure 4, respectively. These labels are allocated through the *labeli* function of *I*. The *Parent* field holds the label of the parent nodes, which are the *p* value of the parent node. The *Level* (*d*) 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 *PathSummary* table. Note that the *Parent* value of the root element is zero since it does not have a parent. The *Type* represents the type of node (e.g. element or attribute). The *Count* value (*C*) is the number of nodes in the original XML data that belong to the same summary group. It is used mainly to reconstruct the subtrees that are rooted at the internal nodes (see Section C).

TABLE I. THE PATHSUMMARY TABLE

Tag	*Level*	*PerLv*	*Parent*	*Type*	*Count*
course	1	1	0	E	1
number	2	1	1	A	1
name	2	2	1	E	1
stduents	2	3	1	E	1
instructor	2	4	1	E	1
student	3	1	3	E	3
program	4	1	1	E	3
fname	4	2	1	E	3
lname	4	3	1	E	3

Table 2 shows the *LeafNodes* table, which is populated with the data of all leaf nodes V_L in the XML tree. In this table the *Level*, *PerLv*, and *No* values together form the label of the leaf nodes *d*, *p*, and *s*, respectively, as shown in the data-tree in Figure 3. These labels are allocated through the *labelg* function of *G*. The *Value* field contains the values of the node for V_T nodes, and *null* for V_E nodes. The *Lev1*,…, *Lev4* fields are explained below.

TABLE II. THE LEAFNODES TABLE

Level	*PerLv*	*No*	*Value*	*Lev 1*	*Lev 2*	*Lev 3*	*Lev4*
2	1	1	251	1	1	0	0
2	2	1	XML	1	1	0	0
2	4	1	Beth	1	1	0	0
4	1	1	Math	1	1	1	1
4	1	2	Physics	1	1	2	2
4	1	3	CS	1	1	3	3
4	2	1	Omar	1	1	1	1
4	2	2	Ayah	1	1	2	2
4	2	3	Sue	1	1	3	3
4	3	1	Jones	1	1	1	1
4	3	2	Jones	1	1	2	2
4	3	3	Jones	1	1	3	3

Branching Indices: In order to achieve high performance of the UISX index structure, and since *s* value uniquely identifies a node among other nodes of the same class, we split the serial path and save each part in a different field (see Table 2). Each field is titled after the level of the *s* values it contains. That is, each field is titled *Lev*(*i*), where $i \in [1,\ldots,n]$, and *n* is the number of levels in *G*. Each field is used for indexing branching nodes located at the corresponding level. We define *H* as a set of *branching indices* that we create to index s_i values, where $H=\{H_1,H_2,\ldots,H_n\}$, $i \in [1,\ldots,n]$, and *n* is the number of levels in G. Each index for each level is based on the concatenation of (s_i,*d*,*p*) values (see Table 2). All *s* values of *V* nodes in *G* are covered by the *H* set.

The index structure of UISX has mainly three components: *path summary* table, *leaf nodes* table, and *branching indices*. The tables' key fields are underlined in Tables 1 and 2. The key field of the *PathSummary* table is (*Tag*), and the key fields of *LeafNodes* table are (*Level*,*PerLv*,*Value*). The two tables are related through the (*Level*,*PerLv*) fields. The branching indices are the *H* set of indexes, which are used to facilitate the link between two arbitrary nodes in a twig query. Our index structure covers nodes that belong to the same XML document; the extension to multiple documents can be implemented by adding the document id to the labels of *I* and *G* nodes.

B. X-Path Query Expressions

In what follows we formally define X-Path query expressions as they are used in the UISX.

Definition 8. A query *Q* is *covered* by a summary *I* if and only if : (1) the nodes of *Q* exist in *I*, and (2) the *Q* nodes exist in *I* according the structure specified by *Q*.

For example, the query "/*instructor*[/*name*= '*XML'*]/*course*" is not covered by the path summary *I* in Figure 4. Although the first condition is met, but not the second. If we switch the positions of *course* and *instructor* tags "/*course*[/*name*= '*XML'*]/*instructor*," then the mapping of *Q* nodes to *I* nodes succeed and *I* covers *Q*.

Definition 9. To evaluate a twig query *Q* over a data graph *G* by using *I*, we say that the matching of an instance of one group with the instances of another group is *complete* if the returned nodes contain all the relevant nodes.

Definition 10. To evaluate a twig query *Q* over a data graph *G* by using *I*, we say that the matching of an instance of one group with the instances of another group is *precise* if the returned nodes do not contain any irrelevant node.

The pattern of single path query expressions can be represented as $t_1.rel.t_2 \ldots rel.t_x$, where ($t_1$,$t_2$,…,$t_x$) are tags of the query and *rel* represent the relationship between the adjacent tags. This may be a parent-child relation "/" or ancestor-descendent relation "//." We refer to single path query expressions that have only the "/" axis as single simple path queries, and to single path query expressions that have one or more "//" axes as single complex path queries. Both types are evaluated by finding the extension of t_x, that is, $ext(t_x)$. In the relational tables in UISX, the mapped data are sorted by <*d*.*p*> keys, and hence one index look up is sufficient to evaluate these types of queries by probing the index for tuples that match <*y*.*x*>, where <*y*.*x*> is the label of t_x, and *d*=*y* and *p*=*x*. Twig queries patterns can be represented as:

$t_1.rel.t_2...rel.t_b[rel.t_1.rel.t_2...tf_1][rel.t_1.rel.t_2...tf_2]...rel.t_1.rel.t_2...tf_i$

This twig pattern expression consists of multiple single path expressions. The expressions inside the square brackets and the expression that follows at the end are the branches of the twig. The branching element tags are denoted by tb. (tf_1, $tf_2,\ldots,tf_i$) are the leaf elements' tags of the first branch, second branch, and i^{th} branch, respectively, where i is the number of branches in the twig. Given an XML data-tree G with a path summary I, in general, with UISX we evaluate a twig query Q against G in two steps. First, we map nodes of Q to nodes of I. If the mapping succeeds (i.e. Q is covered by I), we move to the next step in the evaluation process. In the second step we use only the extension of tags t_b and ($tf_1,tf_2,\ldots,tf_i$) to evaluate the query. Since our index structure is based on tree data and uses a path summary, it always returns complete and precise query results [9,14,17]. Before we present an example, we need to introduce the following theorem.

Theorem 1. In UISX, one index lookup into a branching index H is sufficient to join a pre-defined node of one group of the leaf nodes with all matching nodes in another group of leaf nodes of a twig query.

Proof. First consider the following twig query with two branches:

Q : $t_1.rel.t_2...rel.t_b[rel.t_1.rel.t_2...tf_1][rel.t_1.rel.t_2...tf_2]$

In this query, we assume that the level of the branching node t_b is L_b, and Q has two leafs: tf_1 and tf_2. The extension of tf_1 is a set of nodes Vf_1, that is, $ext(tf_1)=Vf_1=\{vf_{11},vf_{12},\ldots,vf_{1n}\}$, and similarly $ext(tf_2)=Vf_2=\{vf_{21},vf_{22},\ldots,vf_{2n}\}$, where n is the number of instances in each set. According to query Q, we want to prove that one index lookup into HL_b is sufficient to join a single node in Vf_1 node-set with all matching nodes in Vf_2 node-set.

From definition 1, the labels of Vf_j sets, where $j \in [1,2]$, consist of the three parts $<d.p.s>$. The first two parts (d and p) are the same for all nodes in each set. The third part s is the part that uniquely distinguishes each node among all nodes of the same class or group. Each node in Vf_j sets has a serial path r (definition 3), which consist of the s part of the labels of the nodes in the path from the root node to the designated node. Since s is unique for each instance of a class, then r can be used to uniquely identify the labels of all nodes in the serial path of a node. Assume that the value of s of the branching node t_b that is located at level L_b is s_x. The two branches' nodes that share t_b node in their serial paths are matched if the value of each serial path r at t_b node is equal to s_x value. This way, the matching process will return either an empty set if there is no match, or it will return the exact and precise matches since all nodes that share this common ancestor t_b node have their r values at t_b set to s_x. Consequently, there is no chance for any false positives to be retrieved. Since all s values of V nodes in G are covered by the H set of indexes (see branching indices), and HL_b index is based on s values of $Lev(L_b)$ field, then by using an index structure that contain HL_b, it would require only one index lookup to find a match for any arbitrary node in one branch with one or more nodes in any other branch of a twig provided that they share a joining node. This matching process can be extended to solve multiple branches queries with n branches by evaluating two branches at a time until all branches are evaluated as illustrated in Algorithm 2 (to be discussed shortly) □

Example 1. Consider the following twig Query 2 over the data-tree G shown in Figure 3, which asks for the list of students' first name and the programs in which they are enrolled:

Query 2: */course//student [/ program]/fname*

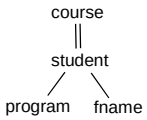


Fig 5. The node-labelled tree representation of Query 2.

This query node-labelled tree representation is shown in Figure 5. It is easy to see that I covers Q because the mapping of this Q query over I path summary of G data-tree can be carried out successfully. In this case *student* node is the branching node t_b, *program* node is the first leaf node tf_1, and *fname* node is the second leaf node tf_2. These three Q nodes map to I nodes <3.1>,<4.1>,and <4.2>, respectively. Note that L_b=3. We next retrieve $ext(tf_1)$, the extension of tf_1, which returns the tuples:

Level	*PerLv*	*No*	*Value*	*Lev 1*	*Lev 2*	*Lev 3*	*Lev4*
4	1	1	Math	1	1	1	1
4	1	2	Physics	1	1	2	2
4	1	3	CS	1	1	3	3

To find a match for the first tuple above, and since L_b=3, we use the index structure to probe the H_3 branch index, which is based on columns (*Lev3*,*Level*,*PerLv*), to retrieve all nodes that match (1,4,2). Similarly, the second and the third tuples are matched by probing the same H_3 branch index for nodes that match (2,4,2) and (3,4,2), respectively, and hence the following tuples below are returned. If there are multiple nodes that match a search criterion, we retrieve them by invoking only one index lookup.

Program	*Fname*
Math	Omar
Physics	Ayah
CS	Sue

C. Size Optimization

The UISX only maps leaf nodes because internal nodes can be regenerated using the *PathSummary* and the *LeafNodes* tables.

Claim 1. Suppose I is a path summary for an XML data-tree G, V_L is the set of leaf nodes of G, and T is the set of serial paths of V_L. Then we can use I and T to reconstruct the subtree that is rooted at any internal node v, where $v \in V_R$.

Next, we present an algorithm (Algorithm 1) that we developed to reconstruct a subtree that is rooted at an internal node v where $v \in V_R$. We design this algorithm as a proof for claim 1 above, which establishes that a subtree rooted at any internal node can be reconstructed.

Algorithm 1 : Publish an internal node

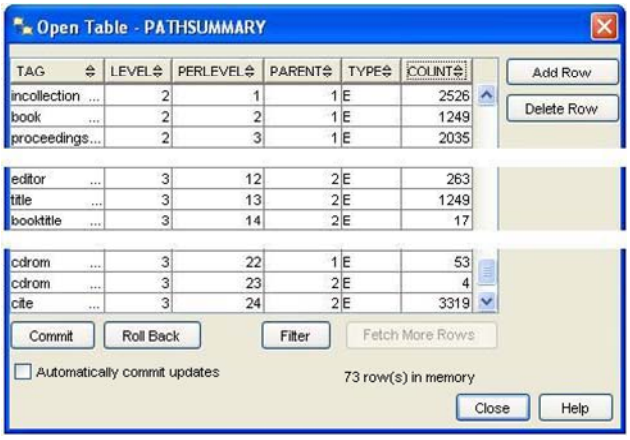
Input : An internal node v .
Output : Subtree rooted at v .

```
1 Identify by using I:
    a- The branching node v according to a given path,
    b- structure of the subtree S that is rooted at v , and
    c- leaf nodes L of S, and sort them by Level and PerLev .
    CheckedNodes = empty set { }
2 For each node l in L
   Begin
    For i = 1 to lc // lc is the count of node l
     Begin
      While(CurrentNode.Level ≤ v.Level and CurrentNode Not in CheckedNodes)
            Add CurrentNode (ChildNode,ParentNode ) to CheckedNodes
            CurrentNode=CurrentNode.Parent
     End
   End
3 Sort nodes in CheckedNodes based on <d,p,s>.
   For each root node v in CheckedNodes
   Begin
      Subtree = empty tree { }
      IdentifyChildren(v)
       Begin
          Add v to Subtree
          ChildrenSet ={v.child}
          For each child y in ChildrenSet
            Begin
              if y Is Not in LeafNode
                  IdentifyChildren(y)
              else
                  Add v to Subtree
            End
       End
   End
   Return the subtree rooted at v node .
```

Step 1 of algorithm 1 identifies the internal node v that needs to be published, the structure of the subtree S rooted at v, and the leaf nodes L of S. This step also initializes an empty set of checked nodes. Step 2 identifies all instances of all nodes that exist in S and adds them to the temporary storage repository *CheckedNodes*. For each node, it adds the labels of the child (the current node) and the parent nodes, which are connected through an edge. Step 3 contains a recursive function that takes all nodes in the *CheckedNodes* repository and builds the subtrees that consist of these nodes according to parent-child relations using the labels obtained at the previous step. Note that step 2 follows a bottom-up tree traversal direction, while step 3 follows a top-down tree traversal direction. This algorithm is designed to reconstruct a subtree rooted at single I node that satisfies a query path. Adjustment to adapt to multiple I nodes that stratify a given query path can be implemented by adding an outer loop to the algorithm to cover all satisfying nodes. All nodes N of S are scanned and retrieved only once in which they are added to a temporary repository that are used at a subsequent step to rebuild the original subtrees, and hence the cost of the algorithm is $O(N)$ database accesses in the worst case. Since nodes are clustered by their $<d.p>$ values, the actual database accesses are less than that predicted by the worst case analysis. Next, we trace a simple example that shows how an internal node is published to demonstrate our claim.

Example 2. To illustrate how the reconstruction of an internal node is carried out, we use parts of the DBLP XML database that we use in our experimental evaluation. Table 3 represents a portion of the *PathSummary* table of the DBLP database. Figure 6 illustrates a portion of the DBLP summary tree. The numbers below the elements' tags represent the count C of the extent nodes in the source XML database for the designated elements in the summary, which are taken from the *COUNT* field in the *PathSummary* table (Table 3). For simplicity, we assume in this example that the *book* element has only three child elements (*title*, *cdrom*, and *cite*).

TABLE III
PART OF THE PATHSUMMARY OF THE DBLP XML DATABASE



To evaluate the query *"//book"* we have to reconstruct the internal node *book* as per the structure shown in Figure 6. We use the *PathSummary* and the *LeafNodes* tables to implement the reconstruction as follows.

- From the *PathSummary* table we can see that the C value of *book* element is 1249, in other words, there are 1249 instances of the *book* element, and these instances are associated by child relations with: 1249 instances of the *title* element, 4 instances of the *cdrom* element, and 3319 instances of the *cite* element. For repetitive referencing, we refer to the *book* element here as the parent element, and the *title*, *cdrom*, and *cite* elements as the child elements.
- At this stage we want to determine which child instances are associated with each parent instance. In order to show how to do that, we use the *LeafNodes* table. We take only the instances of the *cdrom* element in the *LeafNodes* table, which are shown in Table 4, as an example.
- Note that the parent element (the root element of the subtree) is located at level 2 (L_b=2) and the child elements are located at level 3 as shown in Figure 6. Also, from Table 4, we can see that the first instance (the first tuple in Table 4), whose *SerNo=1*, of the *cdrom* element is associated with instance number 4 (s value at *Lev2*) of the *book* parent element. Similarly, the second instance of *cdrom* element is associated with instance number 22 of the *book* parent element, and so on.

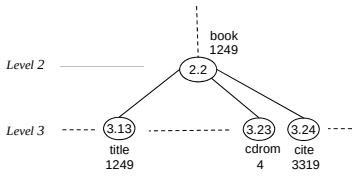


Fig. 6 A portion of the DBLP XML path summary tree

In this way we can reconstruct and publish the internal nodes. In our example in Figure 6, each instance of the *book* element has only one *title* child element. Just 4 instances of the *book* element have 4 instances of *cdrom* child element, in one-to-one relation. Finally, some instances of the *book* parent element have multiple instances of the *cite* child element.

TABLE IV
THE TUPLES OF CDROM ELEMENT IN THE LEAFNODES TABLE

Level	*PerLevel*	*SerNo*	*Value*	*Lev1*	*Lev2*	*Lev3*	*Lev4*
3	23	1	AHV/Toc.pdf	1	4	1	0
3	23	2	BERNSTEIN/Contents.pdf	1	22	2	0
3	23	3	MAIER/CONTENTS.pdf	1	151	3	0
3	23	4	Wiederhold/toc.html	1	443	4	0

D. UISX Query Processor

This section discusses the components of the UISX query processor and the algorithm used in evaluating twig queries. We evaluate twig queries using a light-weight native XML engine on top of an SQL engine as illustrated in Figure 7. Hence, we refer to this method as a *hybrid* query processor. The job of the native XML engine is to explore potential query optimization processes that are related to the structure of XML data, which can not be exploited by SQL engines. The SQL engine handles the XML-Relational data after shredding.

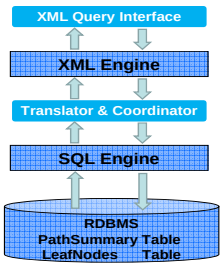


Fig. 7 The UISX hybrid query processor

We developed the algorithm that is outlined below (Algorithm 2) to evaluate twig queries with one branching node. To evaluate a query with multiple branching nodes, the query is divided into several subtrees that are rooted at the branching nodes. The most nested subtree is evaluated first, and then the result is used to solve the subtree that is rooted at the next higher branching node, and so on.

The algorithm consists of 5 parts, which are indicated on the left-hand side of the algorithm. Please note that curly brackets stand for a set that can contain one or more node(s) or element(s); (d,p) represents the node in the path summary whose *Level* is specified by d and *PerLv* is specified by p; and $(d,p,V,Lev(x))$ stands for the tuple in the *LeafNodes* table whose *Level* is d, *PerLv* is p, the *Value* of the tuple is V, and the $Lev(x)$ is the value in the *LeafNodes* table where x is equal to the level of the branching node, namely L_b.

Step 1 of Algorithm 2 identifies the labels d and p (*Level*, and *PerLv*) of the leaf nodes for all branches in a twig query, in addition to the level of the branching node L_b. The second step identifies: the branch with the minimal cardinality if no predicates are given in the query, where the cardinality can be identified from the *Count* field in the *PathSummary* table; or the branch with the minimal selectivity if predicates are used in the query. We need this step to minimize the number of nodes examined to identify a match, and hence reduce the evaluation cost. Step 3 in the algorithm identifies the set of secondary branches, which contains all the branches identified in Step 1 minus the base branch identified in Step 2. Step 4 evaluates the base branch by identifying the set of tuples that satisfy the predicates obtained in the preceding steps. These predicates include the values of d, p, and x . Note that the first two predicates are obtained from Step 2, and the third predicate is obtained from Step 1 where x (to be used in $Lev(x)$) is equal to L_b. We use these predicates to identify V and $Lev(x)$ values of the leaf nodes of the base branch. Finally, in Step 5, the information obtained in the previous steps - specifically d, p, and $Lev(x)$ values - is used to evaluate the secondary branches and obtain the final answers, which are returned to the user as the answer to the given query .

Algorithm 2: Evaluating twig queries

Algorithm to evaluate twig queries by using the hybrid method.
Input : Multiple paths XML query Q .
Output : Answer to all leaf nodes of the query .
1 //Use the *PathSummary* table to identify the *Level* & *PerLev* sets $\{(d_i, p_i)\}$
  //for all leaf nodes of all branches, plus the Level of the branching node L_b.
  $M(Q) \rightarrow \{(d_i, p_i)\}$ // mapping of Q in I, identify leaf nodes
  $M(Q) \rightarrow (L_b)$ // mapping of Q in I, identify the branching node level
2 //define the base branch (d_{min}, p_{min}).
  $(d_{min}, p_{min})=(d_1, p_1)$
  For $k=2$ to i
    if Count(d_k, p_k) < Count(d_{min}, p_{min})
    then $(d_{min}, p_{min})=(d_k, p_k)$
3 //define the secondary branches (d_r, p_r).
  $\{(d_r, p_r)\}=\{\{(d_i, p_i)\} - (d_{min}, p_{min})\}$
4 //Evaluate the base branch first.
  use *LeafNodes* table to find set of tuples $\{(d_j, p_j, V_m, Lev(x))\}$
    where $d_j=d_{min}$ and
      $p_j=p_{min}$ and
      $x = L_b$
5//Evaluate the secondary branches by using the base branch information .
  For each tuple in $\{(d_j, p_j, V_m, Lev(x))\}$ returned by step 4
  Begin
    For each branch in $\{(d_r, p_r)\}$
      Find $\{(d_r, p_r, V_n, Lev(y))\}$
        where $y = L_b$ and
          $Lev(x)= Lev(y)$
    Return $(V_m, \{\{V_n\}\})$
  End

In Algorithm 2 there are two factors that affect the number of accesses to indexes: the number of branches and the selectivity of the branches. Generally in our algorithm, the number of accesses to indexes is affected exponentially by the number of branches and linearly by the selectivity of the leaf nodes. The numbers of returned tuples and indexes accesses are inversely proportional to the selectivity of the leaf nodes of the branches in twig queries. False positive answers for a query are not possible with this algorithm since the set of retrieved tuples (candidate tuples) forms the exact answer to the query.

IV. EXPERIMENTAL EVALUATION

We performed all experiments on a 3 GHz Intel® Pentium 4 PC running Windows® XP, with 1.5 GB of RAM. We used IBM's DB2® V9.5 RDBMS [11] to store and retrieve XML shredded data. The goal of the experiments is to evaluate the performance in terms of elapsed time to execute a query and to evaluate the sizes of the databases, and the supporting indexes that are used by UISX system. We compare our approach with the approaches proposed by Chen et al. [3] for two reasons. First, they adopt a similar approach by creating branching nodes indexes that facilitate and guarantee one index lookup to find matches for each node returned by the base branch evaluation. Second, they compared their approach with five existing indexing schemes including: Edge table [14] and simulated DataGuide [8], which are based on edge model-mapping; simulated Index Fabric [4] and Access Support Relation (ASR) [13], which are based on forward-paths model-mapping; and Join Indices (JI) [28]. Chen et al. [3] proved experimentally that, in general, their approaches outperform these schemes.

A. Testing Data and Queries

Our experiments were carried out using the XMark [21], and the DBLP [27] datasets. We used the test queries proposed by Chen et al. [3] because they are broad and cover different criteria such as cardinality, selectivity, recursion, and depth of the branch node. For ease of reference, the queries are listed in Table 5. Table 6 contains a summary of the characteristics of the test query sets in Table 5. The first set covers single path queries. The second set covers twig (multiple paths) queries with different selectivity and high branch points. The third set covers twig queries with low branch points. The fourth set covers recursive queries. The "X" and the "D" in the "Qry No" column in Table 5 stand for the XMark and the DBLP databases, respectively.

We executed each query ten times against its respective dataset and used the average of the 10 readings in our analysis. The time to translate the XPath queries to SQL queries is not included and only the execution times of the queries are recorded, which reflect the impact of the index structures.

TABLE V. FOUR SETS OF QUERIES USED IN TESTING

Set No	Qry No	Testing Query	Result per Branch
1	Q1X	/site/regions/namerica/item/quantity [. = 5]	1
	Q1D	/dblp/inproceedings/year [. = 1968]	1
	Q2X	/site/regions/namerica/item/quantity [. = 2]	709
	Q2D	/dblp/inproceedings/year [. = 1988]	1746
	Q3X	/site/regions/namerica/item/quantity [. = 1]	9228
	Q3D	/dblp/inproceedings/year [. = 2004]	10660
2	Q4X	/site [/people/person/profile/@income = 46814.17]	1
		/open_auctions/open_auction/bidder[/increase = 75.00]	55
	Q5X	/site [/people/person/profile/@income = 46814.17]	1
		[/people/person/name = 'Hagen Artosi']	1
		/open_auctions/open_auction/bidder[/increase = 75.00]	55
	Q6X	/site [/people/person/profile/@income = 9876.00]	2038
		/open_auctions/open_auction/bidder[/increase = 75.00]	55
	Q7X	/site [/people/person/profile/@income = 9876.00]	2038
		[/regions/namerica/item/location = 'United States']	7519
		/open_auctions/open_auction/bidder[/increase = 75.00]	55
	Q8X	/site [/people/person/profile/@income = 9876.00]	2038
		/open_auctions/open_auction/bidder[/ increase = 3.00]	5172
	Q9X	/site [/people/person/profile/@income = 9876.00]	2038
		[/regions/namerica/item/location = 'United States']	7519
		/open_auctions/open_auction /bidder[/increase = 3.00]	5172
3	Q10X	/site/open_auctions/open_auction	
		[/annotation/author/@person = 'person22082']	2
		/bidder/time	59486
	Q11X	/site/open_auctions/open_auction	
		[/annotation/author/@person = 'person22082']	2
		[/bidder/increase = 3.00]	5172
		/bidder/time	59486
4	Q12X	/site//item[/incategory/category = 'category440']	41
		/mailbox/mail/date	20946
	Q13X	/site//item[/incategory/category = 'category440']	41
		/mailbox/mail/date	20946
		/mailbox/mail/to	20946
	Q14X	/site//item[/quantity = 2]	1543
		[/location = 'United States']	16294
	Q15X	/site//item[/quantity = 2]	1543
		[/location = 'United States']	16294
		/mailbox/mail/to	20946

TABLE VI
CHARACTERISTICS OF THE TESTING QUERY SETS IN TABLE 5

Query Set	Branches	Result Per Branch	Branch Depth	Recursion
1	1	1 – 10660	N/A	0
2	2-3	1 – 7519	High	0
3	2-3	2 – 59486	Low	0
4	2-3	41 – 20946	Low	1

B. Experimental Results

Since we can reconstruct internal nodes from the *LeafNodes* and the *PathSummary* tables, we do not therefore need to map them, and hence the size of the mapped database can be reduced significantly. For example, the actual size savings in our experiments are 115 MB and 88 MB for the XMark and the DBLP databases, respectively. Should we decided to store the internal nodes and use them as in the approaches proposed by Chen et al. [3] and Cooper et al. [4], we would need to use that much of extra space, which cause 42% and 51% increase in the size of the current XMark and DBLP tables, respectively (see Table 7). Another source of space saving in our approach is the fact that the paths of the nodes (elements and attributes) are not recorded in the database, as the case

in other approaches [3,4], because we can regenerate them from the *PathSummary* table by using the nodes' labels.

TABLE VII.
SIZES OF XMARK AND DBLP DATA-SETS WITH DIFFERENT IMPLEMENTATIONS

	Original size	UISX with Internal nodes mapping	UISX without internal nodes mapping	Saved space	Percentage of saved space
XMARK	100 MB	250	158	115	42%
DBLP	50 MB	155	85	88	51%

Table 8 summarizes the characteristics of the ROOTPATHS, DATAPATHS, and UISX index structures for the XMark and the DBLP databases. The tables and indexes sizes are in Megabytes. Note that the original sizes of XMark and DBLP datasets are 100 MB and 50 MB, respectively.

TABLE VIII
CHARACTERISTICS OF TESTING DATABASES AS IMPLEMENTED BY THE INDICATED APPROACHES

	ROOTPATHS	DATAPATHS	UISX
XMARK Tables Size	267	1,285	158
DBLP Tables Size	151	381	85
XMARK Indexes Size	509	2,535	325
DBLP Indexes Size	282	402	183
XMARK No. of Tuples	2,995,272	15,734,707	1,158,492
DBLP No. of Tuples	2,709,327	8,022,673	1,296,328

The ROOTPATHS and DATAPATHS indexes in Table 8 were not subjected to the compression methods listed in Chen et al.'s paper. To save the shredded data of the XMark database, UISX used a space equal to 59% of the space used by ROOTPATHS, and 12% of the space used by DATAPATHS. Similarly, to save the shredded data of the DBLP database, UISX used a space equal to 56% of the space used by ROOTPATHS, and 22% of the space used by DATAPATHS. With regard to the indexes size for XMark database, UISX used 64% of the space used by ROOTPATHS, and 13% of the space used by DATAPATHS. With regard to the indexes size of the DBLP database, UISX used 65% of the space used by ROOTPATHS, and 46% of the space used by DATAPATHS. Finally, the number of tuples that is required by UISX to shred the XMark XML database is equal to 39% of the tuples required by ROOTPATHS, and 7% of the tuples required by DATAPATHS; and for DBLP, UISX requires 48% of the number of tuples required by ROOTPATHS and 16% of the tuples required by DATAPATHS.

The results of the performance tests of UISX compared to ROOTPATHS and DATAPATHS with regards to the sets of test queries in Table 5 are illustrated in Figures 8-10.

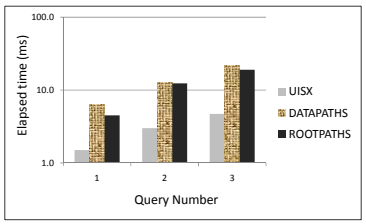


Fig. 8 Performance comparison of UISX with ROOTPATHS and DATAPATHS using the DBLP database

We tested the DBLP data-tree with just the single branch queries since its depth is shallow. The results of the 3 test queries, given in Figure 8, show that UISX performs 67% - 76% better than the ROOTPATHS, and 76% - 79% better than DATAPATHS.

Figure 9 presents the elapsed execution time of the test queries with UISX compared with ROOTPATHS using the XMark database. We notice that the gain in performance is fairly steady (53% – 64%) for the first type (single branch queries 1-3). On the other hand, the percentage gain in performance for the second type of queries (queries 4 – 9) decreases from the 81% to 31% as the selectivity decreases, since the number of pages that contain the returned tuples of the elements with high selectivity are smaller than those with low selectivity. The gain in performance for the third type of test queries is extremely high (99%) because with ROOTPATHS, each tuple returned by the base branch evaluation result has to be hash-joined or merged with the tuples returned by the secondary branches in order to find the matching tuples. While in UISX, the matching tuples of each secondary branch are retrieved with one comparison for each tuple returned by the base branch. The gain in performance for the fourth type of queries is high (79% - 89%) for the selective queries (queries 12 and 13), and relatively low (19% - 37%) for the queries with low selectivity (queries 14 and 15).

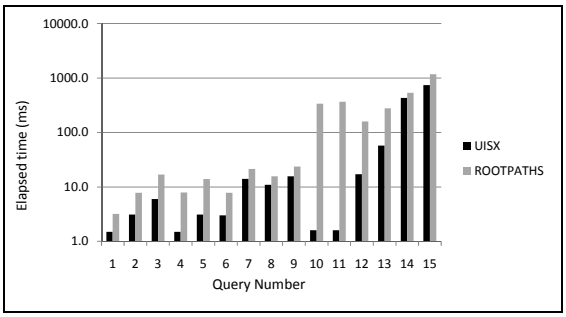


Fig. 9 The performance comparison of UISX with ROOTPATHS using XMark database

Figure 10 shows the comparison of UISX with DATAPATHS using the XMark database. We notice that the gain percentage for the first type of queries is steady and it is in the fifties. Similar to the performance tests against the ROOTPATHS, the gain percentage in the performance of the second type of queries decreases as the selectivity decreases and ranges between (9% - 53%). The gain percentage in performance for the third type is ranging between (48% - 50%). Also, the gain percentage in performance for the first two queries of the forth type of queries (queries with high selectivity) ranges between (36% - 44%), which is higher than that of the last two queries (queries with low selectivity), which ranges between (9% -24%).

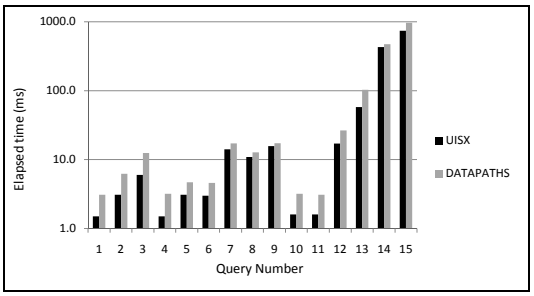


Fig. 10 The performance comparison of UISX with DATAPATHS using XMark database

Our approach performs well in comparison to ROOTPATHS and DATAPATHS. The UISX performance gains over ROOTPATHS are mainly because UISX does not produce any false positive answers, while ROOTPATHS does. DATAPATHS does not produce any false positives, and it is an efficient index structure, but its size is large. UISX performance gains over DATAPATHS are due to the relatively small size of the UISX index structure. Larger indexes require a deeper B+tree, and hence require more search. An efficient way to evaluate a query using DATAPATHS index structure is by evaluating the base branch first. Then a mechanism must be implemented in order to extract the ids of the branching nodes from the returned *IdLists* (e.g. scan the *IdList* string by implementing a string matching operations). In UISX, in contrast, when the base branch is evaluated, the branching nodes ids (labels) are returned in the fields (*Lev2*,*Lev3*,*Lev4*,..*etc*), and are ready to be used in matching operations directly without the need for extra techniques to extract the branching nodes ids.

To evaluate recursive queries (queries 12-15), the reversed-path approaches use the Optional String Pattern Matching (OSPM) function ("LIKE") to evaluate a path with ancestor-descendent axis [9]. For example, if we assume that *S* and *A* stands for *student* and *address* elements, respectively, then the query *"//student/address"* would be evaluated by using an SQL query that would contain the statement "*SchemaPath LIKE AS%*" along with other statements. In contrast, the UISX approach uses only the exact string pattern matching ("="). For example to find the nodes that match the path in the above query, we would run the following SQL query:

```
select  s1.level, s1.perlv
from    PathSummary as s1, PathSummary as s2
where   s1.tag='address'        and
        s2.tag='student'        and
        s1.parent=s2.perlv
```

It is known that SQL supports exact string pattern matching efficiently by using the B+tree indexes, while B+tree indexes do not support ("LIKE") efficiently [9].

V. CONCLUSIONS

We have proposed an index structure that indexes all nodes in an XML data-tree in relation to their branching nodes. This index structure can be used to establish a relation between any two arbitrary nodes that have a common ancestor in a data-tree with a minimal number of matches. Consequently, this index structure facilitates efficient evaluation of twig queries, while maintaining competitive performance for single path queries. The index structure has a path summary of the XML data-tree. Relational tables are used to store shredded XML data and path summaries. Our approach uses less space compared to other state-of-the-art approaches while having similar or better query performance.

We believe that building a native XML query engine on top of the SQL engine is a good way to process XML queries on shredded XML data. The native engine verifies the correctness of the hierarchical structure of the query. Meanwhile, it identifies the essential parts (elements) of an XML query that must be used to execute the query, and pass them to the SQL engine in a special order for execution, hence eliminating joins that could be performed otherwise. Further, our experiments show that mappings that are based on path summaries perform well in comparison to mappings that are based on reversed-paths.

USIX uses only two tables to store XML databases. One table holds the path summary, and the other holds data about the leaf nodes. The summary table is relatively small. These summaries save us from recording the paths of the tuples in the database, and are used to regenerate the paths from the nodes' labels.

Our method of mapping XML into relational tables does not require an XML schema so it can be applied as a general solution for any data-tree structure in addition to XML databases. Two popular data-sets are used in our experiments: the XMark dataset [21], which is document-centric database; and the DBLP dataset [27], which is a data-centric database. Our approach performs well with both types of data-sets.

Our XML-relational approach evaluates XML queries by using equijoins, while most other XML-relational approaches use inequality-joins [9,10,30], which are less efficient. Our approach, furthermore, uses exact string pattern matching, while other approaches use the "LIKE" operator [9]. Finally, we are planning to improve on the native XML query engine

that works on top of the SQL engine. We think that coordinating the query optimization tasks between these two engines can improve XML query processing. Despite the fact that the size of the shredded data is minimized, since the leaf nodes contain details about both themselves and the internal nodes, we noticed that some of these details are redundant among multiple leaf nodes. For example, leaf nodes that share the same branching node have similar information about the path from the root node to the branching node. It worth investigating if there is a way to eliminate these redundancies and improve performance at the same time.

ACKNOWLEDGMENT

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

REFERENCES

[1] S. Chaudhuri and K. Shim, "Storage and Retrieval of XML Data using Relational Database," in *Proc. ICDE 2003*, Bangalore, India, p. 802.

[2] D. Che, K. Aberer, and M. Ozsu, "Query Optimization in XML Structured-document Databases. *The VLDB Journal,* Vol. 15, No. 3 , pp. 263-289, 2006.

[3] Z. Chen, J. Gehrke, F. Korn, N. Koudas, J. Shanmugasundaram, D. Srivastava, "Index Structures for Matching XML Twigs using Relational Query Processors," *Data & Knowledge Engineering*, Vol. 60, No. 2 , pp. 283-302, 2007.

[4] B. Cooper, N. Sample, M. Franklin, G. Hjaltason, and M. Shadmon, "A Fast Index for Semistructured Data," in *Proc.VLDB*, 2001, Rome, Italy, pp. 341-350.

[5] I. Dweib, A. Awadi, S.E. Elrhman, and J. Lu, "Schemaless Approach of Mapping XML Document into Relational Database, in *Proc. of CIT*, 2008, Sydney, Australia, pp. 167-172.

[6] D. Florescu, and D. Kossmann, "Storing and Querying XML Data using an RDMBS," *Bulletin of the Technical Committee on Data Engineering,* Vol. 22, No. 3, pp. 27-34, 1999.

[7] R. Goldman, J. McHugh, and J. Widom, "From Semistructured Data to XML: Migrating the Lore Data Model and Query Language," in *Proceedings of the 2nd International Workshop on the Web and Databases, ACM SIGMOD Workshop*, 1999, Philadelphia, Pennsylvania, USA, pp. 25-30.

[8] R. Goldman and J. Widom, "DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases," in *Proc. VLDB*, 1997, Athens, Greece, pp. 436-445.

[9] G. Gou, and R. Chirkova, "Efficiently Querying Large XML Data Repositories: A Survey," *Transactions on Knowledge and Data Engineering,* Vol. 19, No. 10, pp. 1381-1403, 2007.

[10] P. Harding, Q. Li, and B. Moon, "XISS/R: XML Indexing and Storage System Using RDBMS," in *Proc. VLDB*, 2003, Berlin, Germany, pp. 1073-1076.

[11] (2007) IBM homepage on DB2. [Online]. Available: http://www01.ibm.com/software/data/db2/linux-unix-windows/

[12] R. Kaushik, P. Bohannon, J. Naughton, and H. Korth, " Covering Indexes for Branching Path Queries," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2002, pp. 133-144.

[13] A. Kemper and G. Moerkotte, "Access Support in Object Bases", in *Proc.* SIGMOD, 1990, pp. 364-374.

[14] J. McHugh and J. Widom, "Query Optimization for XML," in *Proc.VLDB*, 1999, Edinburgh, Scotland, UK, pp. 315-326.

[15] T. Milo and D. Suciu, "Index Structures for Path Expressions," in *Proc. ICDT*, 1999, Jerusalem, Israel, pp. 277-295.

[16] S. Mohammad and P. Martin, "LLS: A Level-based Labelling Scheme for XML Databases," CASCON, 2010, Toronto, Canada. pp. 115-127

[17] 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, 2010, pp. 98-124.

[18] J. Naughton, D. DeWitt, D. Maier, A. Aboulnaga, J. Chen, L. Galanis, et al, " The Niagara Internet Query System," *Bulletin of the Technical Committee on Data Engineering,* Vol. 24, No. 2, pp. 27-33, 2001.

[19] S. Pal, I. Cseri, O. Seeliger, G. Schaller, L. Giakoumakis, and V. Zolotov, "Indexing XML Data Stored in a Relational Database, " in P*roc. VLDB*, 2004, Toronto, Canada , pp. 1146-1157.

[20] S. Paparizos, H. Jagadis, J. Patel, S. Al-Khalifa, L. Ladshmanan, D. Srivastava, et al., "TIMBER: A Native System for Querying XML," in *Proc. of the ACM SIGMOD International Conference on Management of Data*, 2003, San Diego, California, USA, pp. 672-672.

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

[22] H. Schoning, "Tamino – a DBMS Designed for XML," in *Proc. ICDE*, 2001, Heidelberg, Germany, pp. 149-154.

[23] J. Shanmugasundaram, E. Shekita, J. Kiernan, R. Krishnamurthy, E. Viglas, J. Naughton, and I. Tatarinov, "A General Technique or Querying XML Documents using a Relational Database System," *ACM SIGMOD Record*, Vol. 30, No. 3, pp. 20-26, 2001.

[24] J. Shanmugasundaram, K. Tufte, and G. He, "Relational Databases for Querying XML Documents: Limitations and Opportunities, " in *Proc. VLDB*, 1999, Edinburgh, Scotland, pp. 302-314.

[25] P. Suei, J. Wu, Y. Lu, D. Lee, S. Chou, and C. Lin, "A Novel Query Preprocessing Technique for Efficient Access to XML-Relational Databases," in *Proc. of First International Workshop on Database Technology and Applications*, 2009, Wuhan, Hubei China, pp. 565-569.

[26] 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 International Conference on Management of Data*, 2002, Madison, Wisconsin, USA, pp. 204-215.

[27] The DBLP Computer Science Bibliography XML records homepage. [Online]. 2009, Available: http://www.informatik.uni-trier.de/~ley/db/.

[28] P. Valduriez, "Join Indices," *ACM Transactions on Database Systems (TODS),* Vol. 12, No. 2, pp. 218-246, 1987.

[29] M. Yoshikawa, T. Amagasa, T. Shimura, and S. Uemura, "XRel: A Path-Based Approach to Storage and Retrieval of XML Documents using Relational Databases," *ACM Transaction on Internet Technology (TOIT)*, Vol. 1, No. 1, pp. 110-141, 2001.

[30] 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 International Conference on Management of Data*, 2001, Santa Barbara, California, USA, pp. 425-436.

[31] Q. Zou, S. Liu, and W. Chu, "Ctree: A Compact Tree for Indexing XML Data," in *Proc. WIDM*, 2004, Washington, DC, USA, pp. 39-46.

TRADEMARKS