Virtual Joins for XML Data

Dao Dinh Kha
Graduate School of Information Science
Nara Institute of Sci. & Tech., Japan
kha-d@is.aist-nara.ac.jp

Masatoshi Yoshikawa
Information Technology Center
Nagoya University, Japan
yosikawa@itc.nagoya-u.ac.jp

Shunsuke Uemura
Graduate School of Information Science
Nara Institute of Sci. & Tech., Japan
uemura@is.aist-nara.ac.jp

Abstract

Establishing the hierarchical order among the XML elements is an essential function of the XML query processing techniques and there are a number of proposals for the task. Although most of XML documents have associated DTD or XML schema, the prior query processing techniques have not utilized the document structure information efficiently. Each of prior techniques has an advantage in processing only a type of queries and it is difficult to incorporate them to complement each other.

In this study, we propose a novel XML query processing method that uses DTD or XML schema to improve the I/O complexity of XML query processing. We design a Structure-based Coding for XML data (SCX) that incorporates both structure and tag name information extracted from the document structure descriptions. Given the tag name and the structure code of an element, SCX allows to determine the tag name and the structure code of the parent element without I/O. This property of SCX provides a Virtual Join mechanism that greatly reduces I/O workload for processing XML queries. We present the algorithms to apply the Virtual Joins for processing the queries of both path and twig patterns, where SCX can be integrated with other structural join techniques to improve the XML query processing. Our experimental results indicate that SCX accelerates the processing of path queries significantly and the efficiency increases in correspondence with the join workload of XML queries and the size of data sets.

1. Introduction

XML [18] data has a recursive tree structure, which can be represented by a rooted label tree. The structure of XML documents in a class can be described by a grammar that is a set of rules of the hierarchical relationship among XML elements in the instance documents of this class. The hierarchical order of elements in an XML document physically depends on their location in the physical storage structure of the document. On the other hand, the elements have to obey the *schematic* order described in the grammar associated with the class containing the document.

Queries on XML data typically specify elements by selection predicates and their tree structure relationship. There are a number of proposed methods for verifying the structure relationship of XML elements. An enumeration that allows the identifier of the parent element to be computed from the identifier of a child element has been presented in [19], hence the "`parent-child`" relationship can be determined using a calculation on the identifiers. The oversize length of identifier and the lack of ability to determine the tag name of the parent element limit the efficiency of this numbering scheme in query processing. The pattern of node paths has been used in [7] to select the elements having the similar hierarchical order. In [6, 8, 10, 12, 15], the 3-tuple (`startPos`, `endPos`, `level`) and equivalent tuples have been used to present the hierarchical order of XML elements in an XML document. The recent approaches to the problem use the structural joins, which select the pairs of XML elements from the candidate sets such that a given hierarchical order holds.

Note that most of XML documents in use are associated with a DTD or XML schema. Since the descriptions integrate the document structure with data types and define the relation of schemata to XML document instances, the XML documents exchanged over the web can share their grammars efficiently. However, the prior techniques to process XML queries have not utilized the information about the schematic structure of XML documents expressed in the descriptions. For example, the enumeration in [19] has been designed for a general tree without any restriction on its structure. Similarly, the presentation (`startPos`, `endPos`, `level`) in [6,8,10,12,15] is extracted from an XML instance document with the assumption that the XML tree can be in any shape.

In structural joins, the indexing data of the candidate sets, a portion of which is only used to produce the partial answers, has to be provided before joining, normally by I/O access to the secondary memory. However, the previous papers have not sufficiently investigated the I/O workload needed to get the candidate sets. This study aims at the improvement of the I/O complexity for XML query processing. The issue is important since the I/O speed is much slower than the computation speed in the main memory. We observe that the schematic structure of XML documents can be used to verify the structural requirement of the elements in an XML query, without the knowledge of their actual position in the physical storage structure. Based on this observation, we propose a new approach to interpret XML queries. For example, to process the XML query "`a/b`", the prior joining techniques require the indexing data of the candidate sets $\{\mathtt{a}\}$ and $\{\mathtt{b}\}$ to be loaded then join these sets to find the pairs $\{\mathtt{a}_i, \mathtt{b}_j\}$ such that $\mathtt{a}_i$ is the parent element of $\mathtt{b}_j$. Actually, the elements $\{\mathtt{a}_i\}$ are not of the interest in the final answer of the query, which consists only of the elements $\mathtt{b}_j$. The indexing data of $\{\mathtt{a}\}$ is used just for checking the parent-child relationship of $\mathtt{a}_i$ and $\mathtt{b}_j$. Using the new approach, rather than joining the elements sets $\{\mathtt{a}\}$ and $\{\mathtt{b}\}$, for each $\mathtt{b}_j$ we seek a method to check if there exists $\mathtt{a}_i$ that is the parent element of $\mathtt{b}_j$ using the indexing data of $\mathtt{b}_j$ only. Therefore, we can save the I/O workload for loading the indexing data of the set $\{\mathtt{a}\}$. Figure 1 illustrates the difference of the prior approaches and our approach in processing the XQuery expression:

```
"FOR $b IN /site/people/person/[address='Japan']
RETURN $b/city/text()"
```
(S_1)

The prior approaches require the indexing data of six elements `site`, `people`, `person`, `address`, `name`, and `text`. Using the new approach, the indexing data of only two elements `address` and `text` is required.

In this paper, we propose a novel Virtual Join mechanism for XML query processing that utilizes the information about the schematic structure extracted from the DTD or XML schema of XML documents. The core of the mechanism is a Structure Coding for XML data , SCX, that compactly expresses the schematic structure of XML elements. The SCX has the following property:

> If the tag name and the structural code of an element are known, the tag name and the structural code of the parent element can be determined without any I/O.

Therefore, for a given element, the tag names of all of its ancestor elements can be determined recursively. Note that the tag names of elements are essential for evaluating XML path expressions. Our study provides evidence that the information about the schematic structure of XML documents declared in DTD or XML schema can be used effectively in the indexes for XML data. In addition, it shows that different indexing techniques can be integrated to complement each other to improve the XML query processing.

The outline of this paper is as follows. In Section 2, we present the preliminaries of our study. Our main contribution is presented the next three sections:

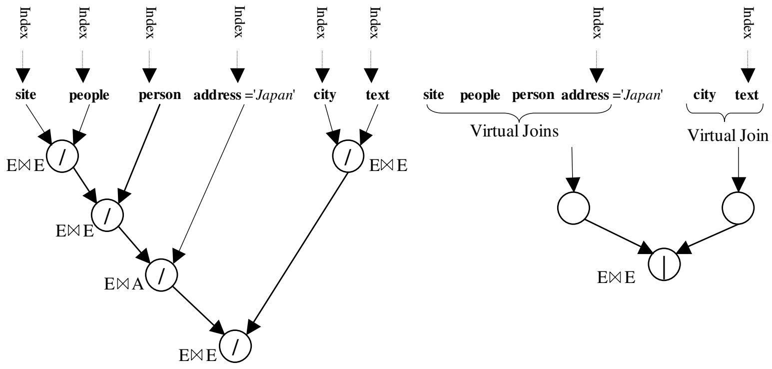


Figure 1. The prior and new approaches for processing the query S_1

- In Section 3, we give the definition and the construction algorithm for SCX that enumerates XML elements based on the schematic structure deduced from DTD or XML schema. Incorporating both structural and tag name information, SCX allows the navigation to the parent of an element, as well as testing the tag name of the parent in the time independent of the document size.
- In Section 4, we describe the Virtual Join mechanism to evaluate XML queries in both path and twig patterns. The mechanism has an optimal I/O workload: The indexing data of only the candidate sets that contain the output elements or relate with the selection predicates is needed. It does not require the indexing data to be sorted. Many intermediate joins can be avoided using the operations on SCX.
- In Section 5, we present the experimental results to show the efficiency of our method in XML query processing with various configurations. Not any special indexing structure except the B^+-tree is required.

We describe the related work in Section 6 and conclude this paper with a suggestion for the future work in Section 7.

2. Preliminaries

2.1. Structure information in DTD and XML schema

An XML document may have a reference to a DTD or an XML Schema, which contain the description of the hierarchical relationship of XML elements. A DTD that defines the elements of the XML document in Figure 2 may have the following element type declarations:

```
<!ELEMENT personnel (company, business, person+)>
<!ELEMENT person (name, email?, person*)>
<!ELEMENT name (family, given)>
```

The first element type declaration indicates that the `personnel` element has one element `company`, one element `business`, and one or many elements `person`.

The hierarchical order of elements can be found also in the complex element descriptions in the XML schema of the same XML document. For example, the following statement:

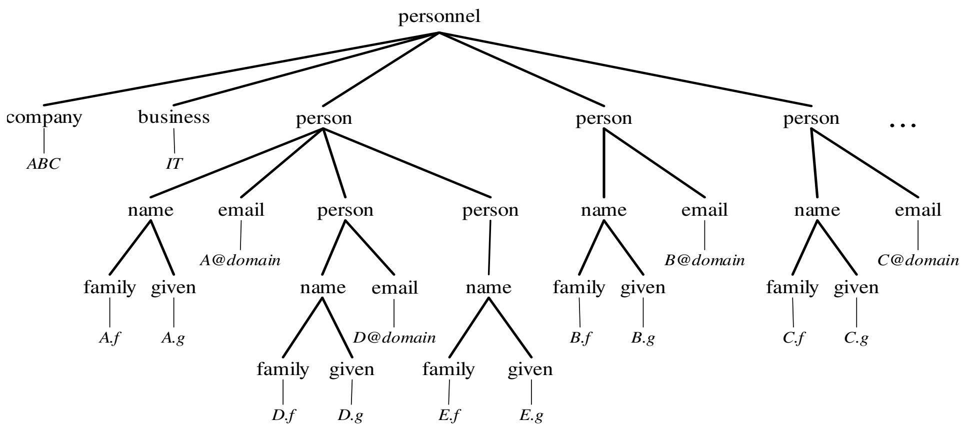


Figure 2. An XML example

```
<xs:element name="name">
  <xs:complexType>
    <xs:sequence>
    <xs:element name="family" type="xs:string"/>
    <xs:element name="given" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

indicates that the complex element `name` has one element `family` and one element `given`. The element hierarchy extracted from the DTD and XML schema for the XML document in Figure 2 is depicted in Figure 3.

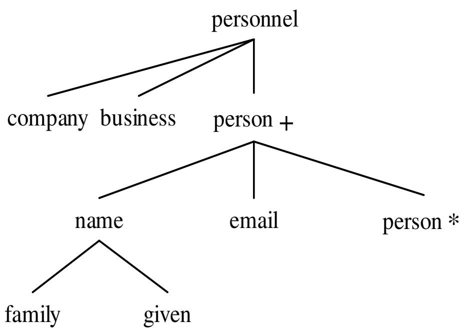


Figure 3. Element hierarchy

In practice, the size of a DTD or XML schema is much smaller than the size of the XML documents associating with them. Let us call the number of parent-child pairs in the DTD or XML schema referenced by an XML document the *structural cardinality* of the document. The specifications of the DTDs of the XMark[1] and Shakespeare[2] data sets are shown in Table 1.

[1] http://monetdb.cwi.nl/xml/downloads.html
[2] http://sunsite.unc.edu/pub/sun-info/xml/eg/shakespeare.1.10.xml.zip

Data set	DTD	Size	Stru. Card.
XMark	“auction.dtd”	4304 bytes	117
Shakespeare	“play.dtd”	1184 bytes	44

Table 1. The specification of some DTDs

2.2. Notation

In this paper, we present an XML document as a labeled rooted tree, which is abstracted out from the DOM data model [17]. Each node of the tree corresponds to an element, an attribute, or a text. A node is the parent of another node if the element corresponding to the former node contains the element, or attribute, or text corresponding to the latter node.

Hereafter, let T denote an XML tree rooted at $\boldsymbol{r}$, $\boldsymbol{n}$ be a node of T, the parent node of $\boldsymbol{n}$ be determined by $parent(\boldsymbol{n})$, the tag name of $\boldsymbol{n}$ be determined by $\boldsymbol{n}.tag$, and the length of the path from $\boldsymbol{r}$ to $\boldsymbol{n}$ be the level of $\boldsymbol{n}$.

3. SCX: a structure coding

In this section, we describe the design of our novel structure coding SCX and the construction algorithms. For simplicity, we will use the notation of DTD. The main goals of the SCX design is the efficiency in XML query processing and the robustness in structural update.

The main component of SCX is the *structural identifier* that presents the schematic order of XML elements. The schematic order of an XML element is determined by the DTD element type declarations, in which the element participates. For example, if a tag name `b` appears in the element type declarations of the tag names `a` and `f` in a DTD then in any instance of the XML document conforming to the DTD, the tag name of the parent element of an element having the tag name `b` must be either `a` or `f`.

The structural identifier of a node uniquely determines both structural identifier as well as the tag name of the parent node using two *index functions*: the function parentSID returns the structural identifier of the parent node and the function nameID returns an integer value used together with the tag name of the child node to find the tag name of the parent node.

The function nameID is a novel and essential part of our method. Intuitively, each pair of the tag names of parent and child nodes in the DTD is mapped into an integer called the *child order*. The child order together with the tag name of the child node uniquely determines the tag name of the parent node. In other words, the following dependencies hold:

$$\texttt{parent\_tag}, \texttt{child\_tag} \longrightarrow \textit{child order} \tag{1}$$

$$\texttt{parent tag} \longleftarrow \texttt{child tag}, \textit{child order} \tag{2}$$

The benefit of the introduction of these functions is the ability to determine the tag names of all ancestor nodes of a node without the necessity to access to the secondary memory. Therefore, intermediate structural joins can be avoided and the indexing data of only the leaf nodes in the query tree structure is necessary for evaluating XML queries.

3.1. The description of SCX

SCX represents the schematic and actual orders of an XML node by the pair [sid, ord], where sid is the structural identifier and ord is a presentation of the node in an XML instance. There are several proposed presentations for the position of XML elements in an XML document, each of them uses the `docID` and `nodeLevel` together with one of the pairs (`preorder`, `postorder`), (`preorder`, `range`), or (`startPos`, `endPos`). The methods to determine the hierarchical relationship among XML nodes using these presentations are similar and can be converted from one to other with a minor modification. In this paper, we adopt the 3-tuple (`docID`, `startPos`, `endPos`) to present XML nodes, i.e. the component ord of SCX. The `nodeLevel` parameter used in the prior presentations is omitted. The element (`docID`, $\texttt{startPos}_1$, $\texttt{endPos}_1$) is an ancestor of the element (`docID`, $\texttt{startPos}_2$, $\texttt{endPos}_2$) iff $\texttt{startPos}_1 < \texttt{startPos}_2$ and $\texttt{endPos}_2 < \texttt{endPos}_1$. Since the encoding is very useful in defining the preceding and following orders, the inclusion of the ord in SCX guarantees that SCX can help the evaluation of queries related to these orders. As will be shown in Section 4, sid and ord complement each other in query processing. Since the design of SCX helps to reduce the number of elements and attributes, the indexing data of which needed to be loaded for processing a query, the size of the combination makes a little impact on the performance of the technique.

Definition 1 *A c-group is the maximal group of the consecutive sibling nodes having the same tag name.*

The notion of c-group in the Definition 1 can be extended to encompass the sequence of the same subgroup of elements that appear multiple times. In a DTD element type declaration, a c-group corresponds to a single child element or a subgroup of elements, the cardinality of which is multiple, such as '`b`', "`b*`", or "`(c, d)*`". We will discuss the decomposition of DTD in section 3.2.1 to deal with the extension. For simplicity, the current form of the Definition 1 is used.

Definition 2 *An enumeration of the nodes in an XML tree T is called "forehand" if the identifier of a node is smaller than the identifier of the siblings from the other c-groups to the right of the node in the same parent node.*

The forehand property guarantees that the identifiers of XML nodes reflect their order as described in DTD declarations.

Definition 3 *A function childOrd: $(a,b) \rightarrow Integer$, that maps a pair of tag names a and b, where b appears in the element content of a in a DTD element type declaration, to an integer, is called "parent-name-determinable" if $\neg\exists$ a_1 and a_2 such that $a_1 \neq a_2$ but childOrd(a_1, b) = childOrd(a_2, b).*

In other words, the child order and the child tag name uniquely determine the parent tag name. The construction of the "parent-name-determinable" function childOrd from the declarations of a DTD will be described in section 3.2.1.

The structural identifiers of the nodes in an XML tree are generated in a preorder traversal by a forehand enumeration. All the nodes in a c-group are assigned the same integer equal to the sum of a basic value computed from the structural identifier of the parent node and the child order of the c-group computed by the parent-name-determinable function childOrd. The root node, itself is a c-group, is enumerated by 1.

3.2. Generating SCX

Before describing the algorithm to generate SCX in section 3.2.2, we present the process to construct the parent-name-determinable function childOrd from DTD.

3.2.1. The construction of the childOrd function

The function childOrd returns an auxiliary integer used to distinguish the same child in different parents in a mapping from the child tag to the parent tag. The function is represented as a table constructed based on the DTD listing all possible arguments and values. In general, a DTD can be complex because of the complex specification of the type of an element. For example, an element `a` can be defined by a DTD declaration such as `<!ELEMENT a ((b, c, d)?, (e, f)*)>`. A direct transformation of such a DTD to the function childOrd is not the best solution. Therefore, we can decompose it using the *intermediate* elements recursively to reduce the complexity, where the intermediate elements are assigned unique tag names. The above example declaration can be presented by an equivalent group of DTD declarations <!ELEMENT g (b, c, d)>$, <!ELEMENT h (e, f)>, and <!ELEMENT a (g?, h*)>. The primary decomposition rules of DTD declarations are:

1. (b|c)* is decomposed into d* and d = (b|c)
2. (b, c)* is decomposed into d* and d = (b, c)
3. b*|c* is decomposed into d|e and d = b*, e = c*

Note that the commonly found in practice DTDs do not contain such complex DTD declarations. For simplicity, we consider only the $meaningful$ DTDs, i.e. any child element in actual data conforming the DTDs can be mapped uniquely to a child element in the DTD declaration of its parent element.

The core form of DTD declaration. In a DTD declaration, an element may have a number of subelements having the same tag name. The cardinality is omitted in construction of childOrd.

Definition 4 *The core form of a DTD declaration is received from the DTD declaration by replacing the cardinalities of the subelements by one.*

For example, the core form of the DTD declaration <!ELEMENT a (b, c?, d*)> is <!ELEMENT a (b, c, d)>.

Core child-orders. Each element is assigned a core child-order in its parent element equal to the index of the corresponding c-group.

Definition 5 *The index of a c-group is the order of the corresponding tag name in the core form of the DTD declaration of the parent element. The core child order of the node n in its parent node p, denoted by $p \dashv n$, is equal to the index of the c-group containing n in p.*

Note that the core child order is different from the actual order of child nodes and independent from the data size.

Example 1 *If an element* a *that conforms the DTD declaration* <!ELEMENT a (b, c, d*, e)> *has five child nodes* d *then the actual orders of the child nodes of* a *are:* b←*1,* c←*2,* d[*1*]←*3,* d[*2*]←*4,..,* e←*8. The core child orders are:* a ⊣ b←*1,* a ⊣ c←*2,* a ⊣ d[*1*]←*3,* a ⊣ d[*2*]←*3,..,* a ⊣ e←*4.*

Extended child-order. We extend the core child-order notion to guarantee the dependency (2). If ∃ a_1 and a_2 such that $a_1 \neq a_2$ but childOrd(a_1, b) = childOrd(a_2, b) then an integer value is added to the core child order of all the child nodes of a_2 such that childOrd(a_1, b) ≠ childOrd(a_2, b). Since the cardinalities of XML documents are finite, the extended child-orders always exists.

The extend child-orders are stored in the table StruDTD that has three columns PAR, CHI, and cOrder containing the tag name of the parent elements, the tag name of the child elements, and the extended child-orders of the child elements, respectively. A row (a, b, o) of the table StruDTD means that any element having the tag name b can appear only in the o^{th} c-group of a parent element having the tag name a.

In Algorithm BuildStruDTD, a $segment$ is a sequence of consecutive rows having the same value in the PAR column. The step 1 lists the pairs of parent-child tag names. For example, the DTD declaration <!ELEMENT a (b*, c|d)> corresponds to three rows in StruDTD, the columns PAR and CHI of which are a and b, a and c, a and d, respectively. Steps 2-7 compute the initial values of cOrder. Steps 8-12 generate the extended child orders. Steps 13 returns the table StruDTD and the fanout f.

Algorithm: BuildStruDTD

Input: A DTD or XML schema
Output: Table StruDTD and fanout f

```
1. save pairs a->b in PAR-CHI
/*initiating the possible value of the cOrder */
2. for each pair a->b
3.     if element content type is 'choice'
4.        cOrder ← 1;
       else /*sequence*/
5.        if (a->b is the start of a segment)
6.           cOrder ← 1;
7.        else cOrder←previous cOrder + 1; endif
       endif
   endfor
/* eliminating the duplication*/
8. loop
9.     for each pair a->b
10.       if ∃ a preceding pair a'->b &&
          cOrders are equal then
11.          ⇑ cOrder of a->* by 1
          endif
       endfor
12.    if all segments are fixed then break;
   endloop
13.return PAR, CHI, cOrder, f←max(cOrder)
```

Note that this algorithm runs once with DTD or XML schemas, the size of which are independent from the size of the data sets associated with them. The table StruDTD is loaded in to main memory when the queries are processed. The intermediate elements resulted from decomposition of complex DTD declarations, if exist, also are included in the `StruDTD` table.

Example 2 *The table* `StruDTD` *of the element hierarchy in Figure 3 is shown in Table 2. The fanout f is equal to four. Both of the nodes* `personnel` *and* `person` *may have a node* `person` *as a child node, hence* `personnel` $\dashv$ `person` *must be different from* `person` $\dashv$ `person`.

PAR	**CHI**	cOrder
personnel	company	1
personnel	business	2
personnel	person	3
person	name	2
person	email	3
person	person	4
name	family	1
name	given	2

Table 2. A `StruDTD` table

Function childOrd. The function childOrd takes the values in the columns PAR and CHI in each line of the table `StruDTD` and returns the integer in the column cOrder. For example, in Table 2, childOrd(`person`, `name`) = 2. Since the dependency (2) holds, let parentTAG denote the function from (cOrder, CHI) to PAR, and we have

$$\mathsf{parentTAG}(\mathsf{childOrd}(\texttt{a}, \texttt{b}), \texttt{b}) = \texttt{a} \tag{3}$$

3.2.2. Algorithm for SCX construction

The Algorithm ConstructSCX generates SCX of the nodes of an XML document in a preorder traversal. Set the fanout f equal to the maximal value of the column $cOrder$ in StruDTD that is independent from the size of XML documents. If $\boldsymbol{n}$ is a child node of $\boldsymbol{p}$ then the structural identifier of $\boldsymbol{n}$ is the sum of the base value equal to ($\boldsymbol{p}$.sid -1) $\times$ f + 1 and childOrd($\boldsymbol{n}.tag$, $\boldsymbol{p}.tag$).

Algorithm: ConstructSCX

Input: T rooted at $\boldsymbol{r}$, a fanout f >1
Output: SCX of nodes in T

```
1. travel T in the preorder
2.   if n is the root
3.     n.sid ← 1;
4.   else p ← parent(n);
5.     corder ← childOrd(p.tag, n.tag);
6.     n.sid ← f * (p.sid - 1) + 1 + corder;
     endif
7.   n.ord.startPos ← start position;
8.   n.ord.endPos ← end position;
   endtravel
```

The steps 5 and 6 of the algorithm ConstructSCX incorporate the child-orders computed by the function childOrd into the structural identifiers. The sid of the nodes of the XML document in Figure 2 are shown in Figure 4, where the fanout f is equal to four.

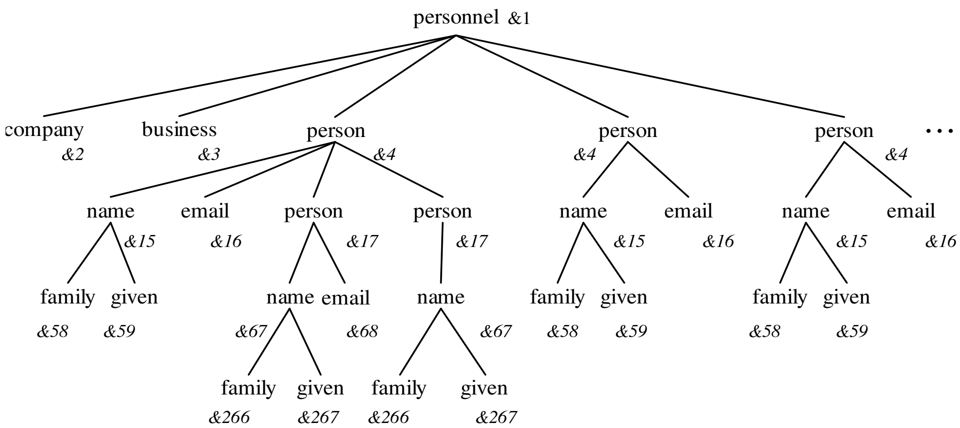


Figure 4. sid **of the document in Fig. 2**

Example 3 *In Figure 4, suppose we have to generate the* sid *of the node* `name` *of the first node* `person`. *Since the code is generated in a preorder traversal, the* sid *of the parent node* `person` *is already known to be equal to four. The function nameID(*`person`, `name`*) = 2. Therefore, the* sid *of the node* `name` *is equal to* $4 \times (4 - 1) + 1 + 2$, *or 15.*

3.3. Index functions

The index functions are used to navigate between a node and its parent node based on the structural identifier of SCX. Given the fanout f and a node $\boldsymbol{n}$, the function $parentSID$ is defined as the following:

$$\mathsf{parentSID}(\mathbf{n}.sid) = \lfloor (\mathbf{n}.sid - 2)/f \rfloor + 1 \tag{4}$$

The purpose of the function parentSID is to compute the structural identifier of the parent node. This property of SCX is similar to the enumeration introduced in [19]. However, our structural identifier system is more robust and realistic. In addition, the introduction of the nameID function makes SCX more efficient than the method in [19]. The function nameID is defined as the following:

$$\mathsf{nameID}(\mathbf{n}.sid) = \mathbf{n}.sid - f \times \lfloor (\mathbf{n}.sid - 2)/f \rfloor - 1 \tag{5}$$

This function returns the child order of $\boldsymbol{n}$ in its parent node. In other words, it computes the $distance$ from the start of the interval of possible structural identifiers for the child nodes to the structural identifier of $\boldsymbol{n}$.

Lemma 1 *If the structural identifiers of SCX are generated by Algorithm ConstructSCX then given a node, the structural identifier and the tag name of the parent node of the node can be determined.*

Proof: Let $\boldsymbol{n}$ and $\boldsymbol{p}$ denote a node and its parent node. We shall show

$$\mathbf{p}.sid = \mathbf{n}.sid - f \times \lfloor (\mathbf{n}.sid - 2)/f \rfloor - 1 \tag{6}$$

From the steps 5 and 6 of the algorithm ConstructSCX, we have

$$\mathbf{n}.sid = f \times (\mathbf{p}.sid - 1) + 1 + \mathsf{childOrd}(\mathbf{p}.tag, \mathbf{n}.tag) \tag{7}$$

Since $1 \leq$ childOrd($\boldsymbol{p}$.tag, $\boldsymbol{n}$.tag) $\leq f$,

$$f \times (\mathbf{p}.sid - 1) + 2 \leq \mathbf{n}.sid \leq f \times (\mathbf{p}.sid - 1) + 1 + f \tag{8}$$

or

$$f \times (\mathbf{p}.sid - 1) \leq \mathbf{n}.sid - 2 < f \times (\mathbf{p}.sid) \tag{9}$$

Divide to $f > 0$, we have

$$\mathbf{p}.sid - 1 \leq (\mathbf{n}.sid - 2)/f < \mathbf{p}.sid \tag{10}$$

Therefore,

$$\mathbf{p}.sid = \lfloor (\mathbf{n}.sid - 2)/f \rfloor + 1 \tag{11}$$

Thus, (6) holds and the structural identifier of $\boldsymbol{p}$ can be computed by Formula 4. Furthermore, to determine the tag name of $\boldsymbol{p}$, from (11) and $f > 0$, we have

$$f \times \lfloor (\mathbf{n}.sid - 2)/f \rfloor = f \times (\mathbf{p}.sid - 1) \tag{12}$$

or

$$\mathbf{n}.sid - f \times \lfloor \frac{\mathbf{n}.sid - 2}{f} \rfloor - 1 = \mathbf{n}.sid - f \times (\mathbf{p}.sid - 1) - 1 \tag{13}$$

From (5), (7), and (13), we have

$$nameID(\mathbf{n}.sid) = \mathsf{childOrd}(\mathbf{p}.tag, \mathbf{n}.tag) \tag{14}$$

From (3) and (14),

$$\mathbf{p}.tag = parentTAG(nameID(\mathbf{n}.sid), \mathbf{n}.tag) \tag{15}$$

The lemma holds. □

Note that the determination of the sid and the tag name of the parent node is independent from the ord of the child node.

Example 4 *In Figure 4, for the node* `given`, *the sid of which is equal 267, the sid of the parent node is equal to* $\lfloor$*(267 - 2) / 4* $\rfloor$ *+ 1, which is equal to 67. The functions nameID returns* $267 - 4 \times \lfloor (267 - 2)/4 \rfloor - 1$, *which is equal to 2. The function parentTAG(*`given`, *2) returns the value* `name`. *Similarly, the tag names of the ancestor nodes are found to be* `person`, `person`, *and* `personnel`, *respectively. Therefore, the full node path for the node* `given` *is* "`personnel/person/person/name/given`".

3.4. Other features of SCX

3.4.1. Coding complexity

We generated SCX for an XML document using the SAX parser [2]. A stack, the size of which is the maximal height of the XML tree, keeps the current node path. Two other buffers keep the tag names and the sids of the previous nodes. A node is visited after the sid of its parent node is known. The cost of the function childOrd is log_2(size(StruDTD)). Therefore, the cost for generating SCX is $O(log_2$(size(StruDTD)) $\times$ (data size)). In our experiments, it took 90 seconds for generating in the main memory the SCX of a data set of 4103211 elements and attributes.

3.4.2. Coding size

The interesting feature of the SCX is that the fanout used to compute sid depends on the number of c-groups rather than the degree of the nodes in the XML instances. For example, according to the DTD declaration <!ELEMENT a (b?, c*, d)>, the degree of a increases in parallel to the number of c. However, the number of c-groups is 3, a fixed value. The small size of the fanout keeps the size of SCX small. In Table 3, we provide the sizes of the data sets used in our experiments and of the SCX indexing data (in the column **StrInxSize**) generated from the data sets as an illustration.

3.4.3. Robustness of SCX

The structural identifier of SCX is robust for the structural update. Taking into account the DTD or XML schema, SCX $anticipates$ the position of updated nodes in the associated XML documents. The structural updates that significantly affect the robustness of the numbering schemes are:

1. **The increase of the number of nodes having the same tag** in a parent node. For example, according to the DTD declaration <!ELEMENT a (b?, c*, d)>, a node having the tag name a may have a number of nodes having the tag name c. All these nodes c have the same sid. A newly inserted node c must belong to the c-group of existing c, hence it has the same sid.
2. **The uncertain occurrence** of an element in its parent element: Let consider the same DTD declaration <!ELEMENT a (b?, c?, d)>. A node having the tag name a may or may not have a child node having the tag b. According to the construction of SCX, a place for such a b is reserved.

Therefore, in both cases, the insertion of a new node does not cause the change of the sid of the other nodes.

3.4.4. Coping with changing DTD

A radical change of DTD, which leads to the entire change of document content and structure, requires rebuilding the index from scratch. The insertion of an element in the content specification of the DTD declaration of an existing element may change the SCX of related elements in the actual data. If insertions are predicted then a $sparse$ mode of SCX construction can reserve the $location$ for the elements to be inserted. If an inserted element is the last child element of its parent element and does not increase the maximal value of $cOrder$ then SCX of existing data is not changed. In practice, the changes in a DTD are rarer than the changes in the content and structure of the XML documents conforming to the DTD.

4. Virtual Joins with SCX

SCX provides the Virtual Joining mechanism that can avoid many the intermediate structural joins in XML query processing. To perform Virtual Join mechanism, we need several the basic functions.

Determining the tag name of the parent node. For a given node, the function findParentSidAndTag calls the parentSID and nameID functions to compute the sid of the parent node and the child order, which is used by the function parentTAG to find the tag name of the parent node.

Function: findParentSidAndTag

```
Input:  n.sid, n.name
Global:  the fanout f of T
1. sid←parentSID(n.sid);
2. tid←nameID(n.sid);
3. name←parentTAG(n.name, tid);
4. return sid, name;
```

Establishing node path. The function generateNodePath establishes the full node path for a given node by recursively performing the function findParentSidAndTag.

Checking the existence of an ancestor having a specific tag name. For a given node with its isd and tag name, the function findAncByName look for the lowest ancestor that has a given name.

Function: findAncByName

```
Input:  n.sid, n.name, ancname
Global:  the fanout f of T
1. name←n.name, sid←n.sid;
2. loop(sid >1)
3.   sid, o ←findParentSidAndTag(sid, name);
4.   name←parentTAG(name, o);
5.   if (name is empty) return null;
6.      else if (name = ancname) return sid, name;
   endloop
```

The function findAncByName can be modified by replacing the step 6 by the command:

`if(name = ancname &&((l >0 && i=l)|| l=0))`

to incorporate the level of the ancestor to be looked for. It checks if the l-level ancestor has a given tag name. If l is equal to 0, the level requirement is omitted.

4.1. Basic *path-predicate* queries

In this section, we describe how to apply the Virtual Join mechanism to process the queries represented by a path expression ended with a predicate.

Definition 6 *A path query is called basic path-predicate query if it is expressed in the form:* $\mathtt{a}_1\ell_1\mathtt{a}_2\ell_2\cdots\ell_{k-1}\mathtt{a}_k$ *or* $\mathtt{a}_1\ell_1\mathtt{a}_2\ell_2\cdots\ell_{k-1}$*[*$\mathcal{P}$ *of* $\mathtt{a}_k$*], where* $k \geq 1$*,* ℓ_i *(i = 1 to k-1) is either the parent axis '/' or the ancestor axis '//', and* $\mathcal{P}$ *is a predicate of* $\mathtt{a}_k$*.*

For example, "person/name/[given = 'Smith']" is a basic path-predicate query. In the basic path-predicate queries, all the nodes having the tag names $\mathtt{a}_i$, $i \neq k$, and the nodes having the tag name $\mathtt{a}_k$

filtered by the predicate $\mathcal{P}$ participate in the structural joins. The predicate $\mathcal{P}$ is optional and can be void.

A basic path-predicate query is presented by a table called *query pattern* with four columns. The column TAG contains the tag names in the query path expression in reverse order, i.e. $\mathtt{a}_i$, (i = k-1 to 1). The column AXIS contains the ℓ_i (i = k-1 to 1). The column ANS indicate the lines having the axis '//' in the column AXIS. In the column FROM, only the values of the lines having the axis '//' in the AXIS column are used and initially equal to 0.

The function matchPattern checks if a node having the tag name $\mathtt{a}_k$ matches a query pattern. The function step(i) looks for the element having the tag name TAG[i] in the axis AXIS[i] of the current node by calling the function findAncByName. If the AXIS[i] is '//' and FROM[i] > 0 then sid of the node to be found must be less than FROM[i]. The function step returns $null$ if there is no such a node, otherwise returns the found sid and tag name. The step 11 seeks the next possible root for a sub-path starting by '//'. The function matchPattern returns $true$ if all the steps in the query pattern are satisfied.

Function: matchPattern

```
Input:  a node n, a query pattern

1. cursid ←a_k.sid; curtag ←a_k.tag; curstep ←1;
2. while (true)
3.   b ← step(curstep)
4.   if (b!=null)
5.     cursid = b.sid; curtag = b.tag;
6.     if (AXIS[curstep]='//')
7.        FROM[curstep] = b.sid;
       endif
8.     if curstep = sizeOf(querypattern)
9.        return true;
10.    else curstep++;
     else
11.    Seek max j < curstep:  AXIS[j]='//'&& FROM[j] > 1
12.    if ∃:  curstep ← j; ∀ i>j FROM[i] ← 0;
13.    else break;
     endif
   endwhile;
14.return false;

Function step(i)
1. s ← cursid; t ← curtag;
2. n←findAncByName(s, t, TAG[i], AXIS[i]);
3. return n;
```

In function matchpatter, the function generateNodePath can be applied to avoid the repetition of the function step.

Lemma 2 *The matchPattern function correctly verifies if the element $\mathtt{a}_k$ satisfies the path expression* $\mathtt{a}_1\ell_1\mathtt{a}_2\ell_2\cdots\ell_{k-1}\mathtt{a}_k$. □

Performing Virtual Joins. The VirtualJoin procedure takes a set of the nodes $\mathtt{a}_k$, which satisfy the predicate $\mathcal{P}$, and a query pattern as the input. By a single scan over the set, for each index i the function matchPattern checks if the node a_k^i matches the query pattern. Note that the function matchPattern will terminate early for the disqualified nodes and the checking process run totally in main memory. The function virtualJoin can process the *ancestor-level* joins, where the hierarchy level is required, such as "a/b", "a/*/b", as well as the "a//b" join.

```
Function:  VirtualJoin
/*for basic path-predicate query*/

Input:  dlist[], queryPattern

for (i from 0 to the size of dlist - 1)
  if matchPattern(dlist[i], pattern) = true
     output dlist[i];
  endif;
endfor;
```

The Virtual Join mechanism does not require the candidate nodes to be sorted and evaluates basic path-predicate queries without I/O except the indexing data of the output candidate set. For queries with long location paths, as shown by the experiment results, the Virtual Join mechanism has a clear advantage since the indexing data of only the last elements in the location paths is needed to be loaded.

4.2. Complex *path-predicate* queries

A path query may be associated with several selection predicates.

Definition 7 *A path query is called complex path-predicate query if it is expressed in the form of a finite sequence of basic path-predicate queries separated by the parent axis '/' or the ancestor axis '//'.*

A complex path-predicate query $\mathtt{B}_1\ell_1\mathtt{B}_2\ell_2\cdots\ell_{k-1}\mathtt{B}_k$, $k \geq 1$, is evaluated by integrating the result of the basic path-predicate queries $\mathtt{B}_i$, i = 1 to k, which are evaluated separately using the Virtual Joins. Let $\{\mathtt{r}_i\}$ denote the list of result nodes of $\mathtt{B}_i$, $\{s_i\}$ denote the list of sid of the nodes in the highest hierarchical structure of $\mathtt{B}_i$ corresponding to $\{\mathtt{r}_i\}$. From the description of matchPattern, $\{s_i\}$ is generated together with $\{\mathtt{r}_i\}$. The lists $\{\mathtt{r}_i\}$ are joined using the conventional structural join technique to produce the final result. Two elements $\mathtt{r}_i^j$ and $\mathtt{a}_{i+1}^k$, $1 \leq i < k-1$, are matched in the structural joins of $\{\mathtt{r}_i\}$ and $\{\mathtt{r}_{i+1}\}$ if:

1. $\mathtt{r}_i^j.startPos < \mathtt{r}_{i+1}^k.startPos$ && $\mathtt{r}_{i+1}^j.endPos < \mathtt{r}_i^k.endPos$, and
2. (ℓ_i is '/' && $\mathtt{r}_i^j.sid$ = parentSID(s_i^j))||(ℓ_i is '//' && $\mathtt{r}_i^j.sid < s_i^j$)

For example, the complex *path-predicate* query a/b/c/[d: $\mathcal{P}_1$]/e/f/[g: $\mathcal{P}_2$] is decomposed into two subqueries a/b/c/[d: $\mathcal{P}_1$] and e/f/[g: $\mathcal{P}_2$]. For the first subquery, the d satisfying the predicate $\mathcal{P}_1$ are loaded and virtually joined with c, b, a. For the second subquery, the g satisfied the predicate $\mathcal{P}_2$ are loaded and virtually joined with f and e. The sid of the nodes e corresponding to the intermediate nodes g are also available. These d and g are joined by the condition:"d is an ancestor of g and d.sid = parentSID(e.sid), where the e corresponds to the g".

4.3. Processing twig queries

Queries on XML data typically specify elements by selection predicates and their tree structure relationship that can be represented as a node label twig pattern with elements with or without predicates in the leaf nodes.

A twig query is decomposed into three complex path-predicate subqueries that are processed separately using the Virtual Joins. The result elements of these queries then are joined using the component *ord* of SCX by conventional structural join techniques that base on the (`startPos`, `endPos`) presentation of the position of elements, e.g. [12], [8] etc. The compatibility with the prior researches is an interesting feature of SCX.

Figure 5 illustrates a twig query, the result elements of which have the tag name `b`. The twig query is decomposed into three subqueries represented by the paths from `a` to `b`, from the node bellow `b` in the left branch to `p`, and from the node bellow `b` in the right branch to `q`. The join of the outputs of these subqueries to produce the final answer is similar to the join of intermediate results in a complex path-predicate query.

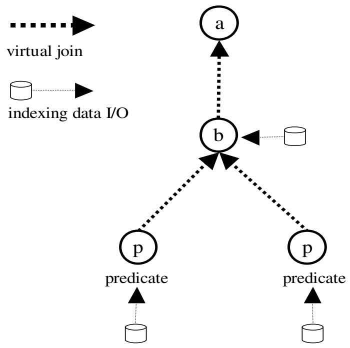


Figure 5. Processing a twig query

Example 5 *For the SCX technique, the XQuery statement* "`FOR $b IN /site/people/person WHERE $b/address/city = 'Nara'] RETURN $b/name/text`" *is decomposed into the basic path-predicate queries* "`/site/people/person`", "`address/city = 'Nara']`", *and* "`name/text`".

Note that, in general, the I/O complexity is *optimal* since only the indexing data of elements that have to be verified by predicates is loaded or belong to the candidate set of the output. For example, in Example 5, the indexing data of elements `person`, `city` and `text` are needed to perform the joins.

Data sets	Size	#element	#attribute	StrInxSize	Q1	Q2	Q3	Q4	Q5	Q6
Data set 1	23.4MB	336224	76867	23MB	1952	7350	2400	3528	3173	50
Data set 2	57.6MB	832911	191162	55MB	4875	10875	6000	8361	7471	134
Data set 3	87.2MB	1253793	286576	87MB	7312	16312	9000	12759	11341	203
Data set 4	115.7MB	1666315	381878	110MB	9750	21750	12000	16640	14954	271
Data set 5	145.1MB	2088879	478374	148MB	12187	27187	15000	21123	19030	318
Data set 6	174.0MB	2502484	573122	158MB	14625	32625	18000	25499	22917	412
Data set 7	203.4MB	2921324	669773	188MB	17062	38062	21000	29706	26551	448
Data set 8	232.2MB	3337649	765562	219MB	19500	43500	24000	33954	30377	521

Table 3. Specifications of the data sets

5. Experiment

We have implemented SCX and the Virtual Join mechanism in a system called Virtual Joins Engine for XML(VJEX). The current version of VJEX has the module for evaluating the basic path-predicate queries. We maintain the main file structure for storing the structure coding as the following:

```
<scx, element_name, add_infor>
```

where `scx` is the SCX, including both sid and ord, `element_name` is the tag name, and `add_infor` consists of an indicator whether the item is an element or attribute and a pointer to data. The primary functions on the structures are:

F$_1$: For a given `scx`, retrieve the elements or attributes having that identifier.

F$_2$: For a given name, retrieve all the elements or attributes having that name.

The XML content and the coding data are indexed using B$^+$-tree. We create two B$^+$-trees on `scx` and `element_name`. Since it is not necessary to sort the indexing data in Virtual Joins, the data is sorted by the `scx.ord.startPos` as required by most of the structural join techniques . For the purpose of this paper, the physical data presentation of the XML content is not material and its details is not relevant to the results of this paper.

VJEX keeps the table StruDTD in the main memory during the query evaluation. The input query of VJEX is transformed into a basic path-predicate, a complex path-predicate, or a twig forms. The basic path-predicate subqueries are evaluated separately using the Virtual Join mechanism. The partial results are joined using non-virtual join algorithms (not shown here) to produce the final result.

5.1. Experiment setup

We measure the *full* processing time that includes the elapsed times for loading the indexing data and for performing structural joins in XML queries.

We compare our method with the method Stack-Tree-Decs in [12] and PathStack in [8]. Both of the Algorithms use the tuple (`docID`, `startPos`, `endPos`, `nodeLevel`) to present an XML element. Stack-Tree-Decs is the highest performance method among four methods described in [12], where stacks are used to reduce the number of the match tests for the pairs of elements from the joined candidate sets in structural joins. PathStack also uses stacks to compactly present the partial and total answers to avoid the large intermediate answers.

5.2. Experimental platform and Data sets

Our experiments were conducted in a workstation running on Windows XP Professional with a 2GHz CPU. The SAX parser available from the Xerces project [2] were used to parse XML data. Other programs for extracting structure information from DTD, generating SCX, and processing the Virtual Joins were written in Java. The maximal Java heap size was set to be equal 300MB.

We used the XML data generator `xmlgen` provided by XMark [1] to generate synthetic XML documents conforming the DTD "auction.dtd". This DTD has a fairly complex structure to make the experiments objective. The specifications of these data sets are shown in Table 3. As shown in the column **StrInxSize** of Table 3, the size of the SCX indexing data is approximately equal to the size of the XMark data sets from which it was generated.

5.3. Experimental results

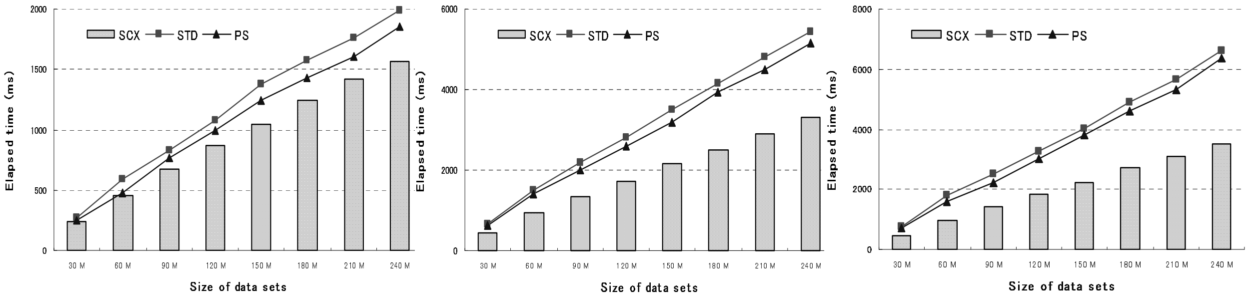
To evaluate SCX, we use the queries that features the various complexities of structural joins, some of them were borrowed from prior researches in the same topic. The queries contain both '/' and '//' axes and include short, medium, and long location paths.

Q$_1$: `//closed_auction/item`

Q$_2$: `//items/name`

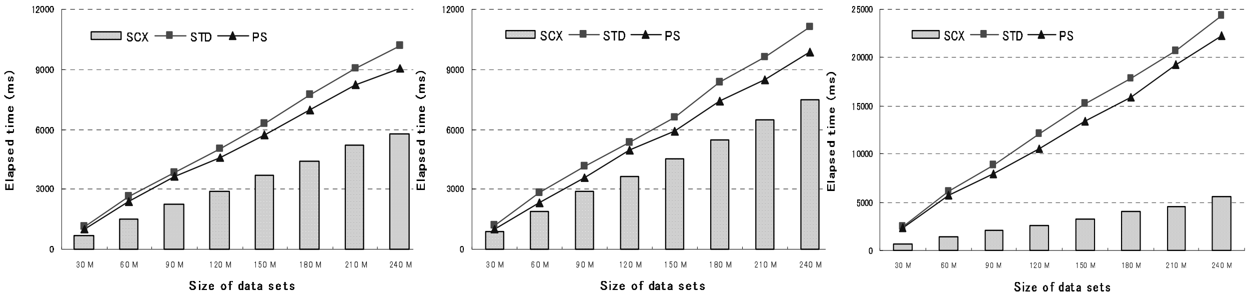
Q$_3$: `//open_auction//description`

Q$_4$: `//open_auction//description//listitem`



(a) Elapsed query time for Q_1 (b) Elapsed query time for Q_2 (c)Elapsed query time for Q_3

Figure 6. The elapsed times for processing the short queries Q_1-Q_3



(d)Elapsed query time for Q_4 (e) Elapsed query time for Q_5 (f)Elapsed query time for Q_6

Figure 7. The elapsed times for processing the medium and complex queries Q_4-Q_6

Q_5: `//open_auction//description//keyword`

Q_6: `//closed_auctions/closed_auction/annotation/description/parlist/listitem/text/emph/keyword/`

The experiment results are shown in Figures 6 and 7, where the methods SCX, Stack-Tree-Decs, and PathStack are abbreviated by SCX, STD and PS, respectively. The number of matches of the queries are shown by the columns **Q1**, **Q2**, **Q3**, **Q4**, **Q5**, and **Q6** in Table 3 according to the data sets.

5.3.1. Simple joins

The queries with a single structural join have been discussed in detail in the [10] such as parent-child joins "`E1/E2`" or ancestor-descendant joins "`E1//E2`". We omit the discussion about the element and attribute join, considering it as a special case of `E1/E2` join. Both of the queries Q_1 and Q_2 involve a single parent-child join of two element sets. The query Q_3 has an ancestor-descendant join. The cardinalities of the XML element sets participating in the structural joins are different. From the Figure 6(a), we can see the SCX is slightly better than Stack-Tree-Decs in the smallest data set and significantly better in the bigger data sets. The elapsed times for processing the short queries on different data sets are shown in Figure 6(a)-(c).

5.3.2. Medium complex queries

The queries contain several structural joins that may be ancestor-descendant or parent-child joins. The queries Q_4 and Q_5 are borrowed from [15]. In the queries Q_4 and Q_5, there are only the ancestor-descendant joins. The elapsed times for processing the queries on different data sets are shown in Figure 7(d)-(e). As expected, the axis '//' can be processed efficiently using Virtual Joins. For the location paths of Q_4 and Q_5, the indexing data of only two elements `listitem` and `keyword` was loaded and the remaining part of the evaluation process was done in main memory.

5.3.3. Complex queries

A number of queries introduced in XMark [1] have complex structure. The evaluation of such queries requires many structural joins. An example of such queries is Q_6. Performing the structural joins in **Q6** is the main workload of the evaluating the query: "*Print the keywords in emphasis in annotations of closed auctions*". The elapsed time for processing the query on different data sets is shown in Figure 7(f). For the complex query, the advantage of the SCX over STD and PS is greatly significant. It can be explained by that the amount of indexing data saved by SCX from loading from secondary memory, that is required by other indexing methods, is larger for such queries. For the location path of Q_6, the indexing data of the only element `keyword` was loaded and the remaining part of the evaluation process was done in main memory despite of the number of joins in the location path.

In our query set, the join workloads are increased from queries Q_1 to Q_6. In all experiments, we can see an interesting tendency that when the sizes of data sets increase, the comparison result is changed in the favor of the SCX method. The advantage of SCX steadily increases in correspondence with the size of the experimental data sets as well as the join workload. The experiment result accords with our expectation that for the large data sets and the queries with the heavy join workload, the advantage SCX in I/O complexity for XML query processing becomes more significant since the amount of data saved from I/O becomes larger.

6. Related works

Querying on the structure is an essential task of the databases of semistructure and XML data. The structural summary has been presented as a graph for semistructured data in [9, 11, 14]. A path-based approach to query the XML data by storing all available node paths in a table of RDBMS and making queries over the pattern of the node paths has been proposed [7]. An integration of XML node numbers in query statements and an algorithm for transformation from XPath to SQL have been discussed in [5]. A general approach to store XML data in tables of RDBMS, where a query is evaluated by joining the tables containing the data items related to the query, has been presented in [6].

The presentation of XML elements by (`docID, startPos, endPos, nodeLevel`) and the equivalent tuples using `pre` and `post order`, `preorder` and `range` have been used in [3, 8, 10, 12, 13] for processing structural joins. In our work, to present the actual order of elements in an XML document, we adopted the presentation. Both [12] and [8] use stacks in the structural join algorithms to reduce the number of match tests between candidate sets of a join. The algorithms in [12] can produce the result sorted either by ancestor or descendant nodes. The algorithms in [8] can process both of the path and twig queries, where the partial and total answers have been in compact stacks to avoid the large intermediate answers. The indexing structures such as B^+-tree and R-tree built-in in RDBMSs have been exploited in [15] to index the presentation values.

The current works related to our approach are [4, 16]. XML documents are embedded in a binary tree in [16], hence the depth of the binary tree is high in practice. [4] has proposed the XR-tree to manage the stab lists used to find the qualified pairs of elements in the ancestor-descendant structural join. The index permits skipping over the portions of the candidate sets that are guaranteed not to produce any match. The index implementation requires a new data structure other than the widely used B^+-tree and does not support well the parent-child relationship.

Our research investigates the whole query processing procedure, including the I/O workload for the indexing data. The function (4) of SCX was inspired by the UID method presented in [19], which enumerates the nodes of a tree by sequent integers, starting from one at the root. Besides the Virtual

Join mechanism, the design of SCX solves the issues of coding size and robustness in structural update that limit the efficiency of the original UID method in query processing.

7. Conclusion

Prior techniques for processing XML queries have not utilized efficiently the document structure described in DTD and XML schema and heavily depended on the actual order of XML elements in XML instances. In this study, we proposed the Virtual Join mechanism for structural joins using the Structure Coding for XML data (SCX) that incorporates both structure and tag name information extracted from DTD or XML schema. SCX greatly improves the I/O complexity of XML query processing. Many intermediate containment joins can be avoided by computing the result set using the operations on the SCX only. According to our experiments, SCX significantly improves the query processing efficiency in correspondence with the structural join workload and the size of data sets. In addition, SCX and prior structural joins techniques can be integrated to improve the XML query processing.

Our plan for the next development of the study is the investigation into the application of SCX to query the XML data stored in RDBMS.

References

[1] A. Schmidt. XMark: A Benchmark for XML Data Management. *Proceedings of VLD, Hongkong*, 2002.
[2] Apache Software Foundation. Apache XML Project. *http://xml.apache.org/*, 2001.
[3] C.Zhang et al. On Supporting Containment Queries in Relational Database Management Systems. *Proceedings of SIGMOD, USA*, May 2001.
[4] H.Jiang, H.Lu, W.Wang, B.C.Ooi. XR-Tree: Indexing XML data for Efficient Structural Joins. *Proceedings of the ICDE, India*, 2003.
[5] I. Tatarinov et al. Storing and Querying Ordered XML Using a Relational Database System. *Proceedings of the ACM SIGMOD, USA*, June 2002.
[6] J.Shanmugasundaram, et al. Relational Databases for Querying XML Documents: Limitations and Opportunities. *Proceedings of VLDB, Scotland*, 1999.
[7] M.Yoshikawa, T.Amagasa, T.Shimura, S.Uemura. XRel: A Path-Based Approach to Storage and Retrieval of XML Documents Using Relational Databases. *ACM Transation on Internet Technologies*, 1(1), 2001.
[8] N.Bruno, N.Koudas, D.Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. *Proceedings of SIGMOD, USA*, 2002.
[9] P.Buneman, S.Davidson, M.Fernandez, D.Suciu. Adding Structure to Unstructured Data. *Proceedings of the International Conference on Database Theory, Greece*, pages 336–350, 1997.
[10] Q. Li, B. Moon. Indexing and Querying XML Data for Regular Path Expressions. *Proceedings of VLDB, Italy*, 2001.
[11] R.Goldman and J.Widom. DataGuides: enabling query formulation and optimization in semistructured databases. *Proceedings of the International Conference on Very Large Databases*, pages 436–445, September 1997.
[12] S. Al-Khalifa et al. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. *Proceedings of ICDE, USA*, 2002.
[13] S.Chien et al. Efficient Structural Joins on Indexed XML Documents. *Proceedings of VLDB, Hongkong*, 2002.
[14] T.Milo, D.Suciu. Index Structures for Path Expression. *Proceedings of the International Conference on Database Theory*, pages 277–295, 1999.
[15] Torsten Grust. Accelerating XPath Location Steps. *Proceedings of SIGMOD, USA*, June 2002.
[16] Wang Wei et al. PBiTree Coding and Efficient Processing of Containment Join. *Proceedings of the ICDE, India*, 2003.
[17] World Wide Web Consortium. Document Object Model (DOM) Level 1 Specification. *http://www.w3.org/TR/REC-xml*, 1998.
[18] World Wide Web Consortium. Extensible Markup Language (XML) 1.0. *http://www.w3.org/TR/REC-xml*, 2000.
[19] Y.K.Lee, S-J.Yoo, K.Yoon, P.B.Berra. Index Structures for structured documents. *Proceedings of the International Conference on Digital Libraries, Maryland*, March 1996.