XTRON: An XML Data Management System using Relational Databases

Jun-Ki Min [*] , Chun-Hee Lee [1] , Chin-Wan Chung [2]

[*] *School of Internet-Media Engineering*
Korea University of Technology and Education
Byeongcheon-myeon, Cheonan, Chungnam, Republic of Korea, 330-708

[1,2] *Division of Computer Science*
Department of Electrical Engineering & Computer Science
Korea Advanced Institute of Science and Technology (KAIST), Daejon,Republic of Korea

Abstract

Recently, there has been plenty of interest in XML. Since the amount of data in XML format has rapidly increased, the need for effective storage and retrieval of XML data has arisen. Many database researchers and vendors have proposed various techniques and tools for XML data storage and retrieval in recent years. In this paper, we present an XML data management system using a relational database as a repository. our XML management system stores XML data in a schema independent manner, and translates a comprehensive subset of XQuery expressions into a single SQL statement. Also, our system does not modify the relational engine. In this paper, we also present the experimental results in order to demonstrate the efficiency and scalability of our system compared with well-known XML processing systems.

1 Introduction

As data are collected over diverse application areas, the requirement of interoperability for sharing and integrating them has increased. Therefore, W3C has proposed the eXtensible Markup Language(XML) [4]. Due to its flexibility and self-describing nature, XML is considered as the *de facto* standard for data representation and exchange in the Internet.

* Corresponding Author, Email: jkmin@kut.ac.kr
[1] Email: leechun@islab.kaist.ac.kr
[2] Email: chungcw@islab.kaist.ac.kr

To retrieve XML data, several query languages have been proposed. Among them, XQuery [2] is considered as the standard query language for XML data since it is broadly applicable across all types of XML data.

Since the amount of data in XML format has rapidly increased, the need for effective storage and retrieval of XML data has arisen. Many database researchers and vendors have proposed various techniques and tools for XML storage and retrieval [3,6,11,10,13,21,28,35,39].

Text file systems, native XML management systems (special-purpose systems), and traditional databases can be used as repositories for XML data. Using a text file system as an XML repository is the most convenient and prevalent approach. But, it is inefficient in retrieval since XML data is always parsed into an intermediate format such as a DOM tree.

An alternative for an XML repository is to use native XML database systems such as LORE [25] and Strudel [14]. LORE is designed for managing semi-structured data. Its data model is Object Exchange Model (OEM) which is a simple and nested object model. Strudel is a web-site management system and its data model is a labeled directed graph similar to OEM. Obviously, these works suggest valuable techniques and insights for the management of irregularly structured data. However, it is uncertain whether these approaches will be widely accepted in the real world since they are not mature enough to process queries on a large amount of data and in multi-user environments [17].

In addition, native XML management systems have two potential drawbacks as pointed in [33]. First, they do not use the existing mature storage and query capability. Second, they have difficulty in integrating the existing data, most of which is relational data. And, major DBMS vendors (SQL Server, Oracle, and DB2) have developed an XML management system using an RDBMS. Therefore, we focus an XML management system using an RDBMS.

1.1 The goals of XTRON

In this paper, we present an XML data management system using an RDBMS called XTRON. We have chosen to use an RDBMS due to its ability to behave as a stable repository as well as an efficient query optimizer and executor.

The design goals of XTRON are as follows:

- **No modification of the relational engine**: The modification of a relational engine may incur unintended side effects such as the consistency problem. Thus, the main goal of our design is to use the relational engine

without modification.

- **schema independence**: Some works [23,36] ignore the schema information of XML data. In [39], the relational schema using DTD is different from that without DTD. Thus, a design goal of XTRON is to utilize schema information if it is available and to store XML data over identical relational tables whether DTD exists or not.
- **Efficient evaluation of path expressions**: To support an efficient evaluation of XML queries, some work uses path indexes which incur the modification of the engine. In contrast to the previous work, we represent a label path as an interval in [0.0, 1.0). Using the containment relationships between intervals of label paths and an interval of the path expression, the path expression can be efficiently evaluated without the modification of a relational engine.

In addition, to demonstrate the efficiency and scalability of XTRON, we implemented XTRON and comparison systems: edge approach, region approach, and region with path table approach. Also, we show the effectiveness of XTRON compared with well-known XML processing systems: Galax and Berkeley DB XML.

1.2 Organization

The remainder of the paper is organized as follows. In Section 2, we present various methods for storing and retrieving XML data. After we show the architecture of XTRON in Section 3, we describe the details of storing XML data in XTRON in Section 4. In Section 5, we present mechanisms for XML data retrieval. Section 6 and Section 7 show GUI of XTRON and the results of our experiments. Finally, in Section 8, we summarize our work and suggest some future studies.

2 Related work

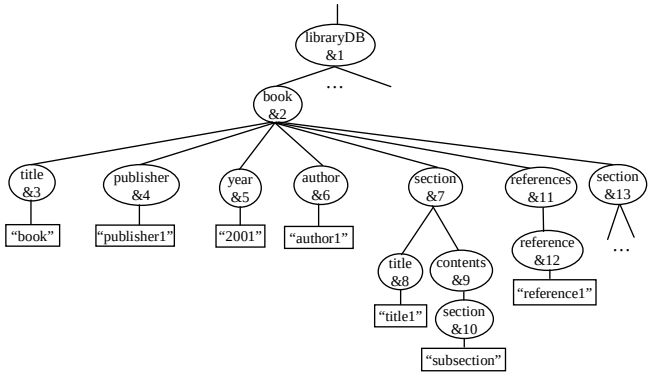
Recently, in order to store XML documents using relational database systems, many XML storage systems and techniques using relational tables have been proposed.

With respect to mapping of the graph model to relational tables, these mapping schemes are basically classified into two groups: One is the *edge approach* [17] and the other is the *region approach* [20,23,36,37,41].

The edge approach stores the edges in the XML graph into the relational ta-

```
<libraryDB>
    <book>
        <title> book1 </title>
        <publisher> publisher1 </publisher>
        <year> 2001 </year>
        <author> author1 </author>
        <section>
            <title> title1 </title>
            <contents>
                <section> subsection </section>
            </contents>
        </section>
        <references>
            <reference>reference1</reference>
        </references>
        <section> … </section>
    </book>
    …
</libraryDB>
```

(a) LibraryDB.xml



(b) The corresponding XML graph of (a)

Fig. 1. An XML data

bles. In this approach, a unique node identifier (*nid*) is assigned to each node of the XML graph. An edge of the XML graph is represented as $<nid_1, label, nid_2>$ where nid_1 is a node identifier for the source of the edge, nid_2 is a node identifier for the target of the edge, and *label* is the label of the target node. Generally, the edge approach is efficient in computing parent-child relationships. However, the edge approach is inefficient in computing ancestor-descendant relationships since ancestor-descendant relationships are computed by the massive joins for parent-child relationships.

The edge approach has many alternatives according to the mapping rule from a set of edges to relational tables. Florescu and Kossman [17] provided three alternatives. The first one is to store all edges in a single table, called an edge table. The second one is to partition all edges with respect to the label. Then, each sub-group of edges is stored in distinct tables, called a binary table. The last one, a universal table approach, is to store all sequences of edges to leaf nodes of the XML graph in a single table which is equal to the result of a full outer join of binary tables.

Based on the edge approach, [35] suggested the inlining technique which utilizes the structural information contained in a DTD (Document Type Definition). The intuitive behavior of the inlining approach is that if one-to-many or

many-to-many relationship between element nodes are defined in DTD, then sets of edges between the element nodes are mapped to different tables, otherwise (i.e., one-to-one relationship), sets of edges between the element nodes are mapped to the same table using the inlining technique. Thus, the inlining approach reduces the join overhead for evaluating queries. However, this inlining technique may lose the order information of child elements.

The region approach originated from the information retrieval (IR) field [7,29]. In this approach, XML data is considered as a tree structured data. As shown in Figure 2 which is an example of the region approach corresponding to Figure 1, this approach assigns a region to an element in XML data. Generally, a region is represented by a pair (start,end) consisting of the position of the start tag and the position of the end tag of an element.

region_table

start	end	label
1	2000	library
2	1000	book
3	5	title
6	8	publisher
…	…	…
19	500	contents

XQuery X1: //book//title

SQL S2:
```
SELECT  t.*
FROM    region_table b, region_table t
WHERE   b.label = "book" AND t.label = "title"
    AND  b.start < t.start AND t.end < b.end
```

Fig. 2. An example of the region approach

The root node &1 in Figure 1-(b) is represented by (1,2000) when XML data in Figure 1-(a) has 2000 words. In this case, a region satisfies the following property.

Property 1 *An element* e =(start, end) *in an XML document is an ancestor of* e' = (start', end') *if* $start \leq start' \leq end' \leq end$.

Using the above property, ancestor-descendant relationships can be efficiently evaluated. For example, as described in Figure 2, the SQL S2 corresponding to XQuery X1 uses only one self join.

To improve the performance of path expressions, Shimura et al. [36,40] devised the path table approach which is combined with the region approach. In the path table, all distinct simple paths and their identifiers (i.e., *path_id*) are recorded. Then, instead of the *label* column of region_table, the *path_id* column is used. Thus, by the join of the path table and the region_table, a path expression can be evaluated efficiently.

Besides the edge and region approaches, the prefix approach [8,22,27,39] is proposed to support updates and preserve the document order efficiently. With prefix encoding, an XML element is represented by a vector which is the concatenation of the parent element's label and the local order among siblings. Generally, the prefix approach is used on native XML systems. The disadvantages of the prefix approach on the relational engine are the waste of disk space to keep the prefix value (i.e., a vector of integers) and the requirement

for user defined operators since database systems do not support the operator for the vector type generally.

3 The Architecture of XTRON

In order to achieve the goal of our system, we design the architecture of XTRON. The architecture of XTRON is shown in Figure 3. As shown in Figure 3, XTRON consists of two parts: the store part and the retrieval part.

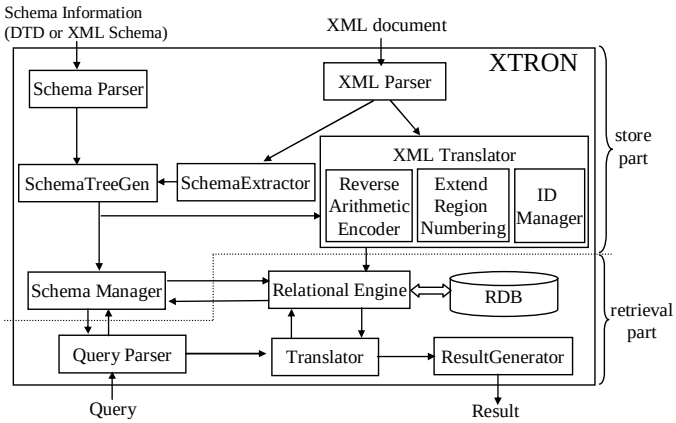


Fig. 3. The architecture of XTRON

One of design goals of XTRON is the schema independence. Also, we want to maintain XML data over identical relational tables whether the structural information exists or not. Thus, we equipped the *SchemaTreeGen* module and the *Schema Manager* module which transform textual structural data into a tree structure and maintain the structural information with a relational database. For the case that schema is not available, we designed the *SchemaExtractor* module which extracts the structural information for XML data (see details in Section 4.2).

XML data itself is transformed into structural data (i.e., relational data) by the *XML Translator* module. In the XML Translator, the path information of each element is encoded into an interval by *Reverse Arithmetic Encoder*. Among the various mapping schemes from XML to relational data, we adapt the region numbering scheme in which the element is represented by a region using *Extended Region numbering*. In addition, ID and IDREF relationship is maintained by *ID Manager*.

SQL statements for an XQuery are different according to the mapping scheme. Thus, in retrieval part, the XQuery query is translated into an SQL statement following the our storage scheme.

Details of the store part and the retrieval part are presented in Section 4 and Section 5, respectively.

4 Mapping XML to Relations

In this section, we present our approach to store XML data in relational tables.

XTRON does not require the modification of the relational engine. In order to improve the query performance, diverse XML path indexes such as DataGuide [18] have been proposed. To support these indexes in an RDBMS, the relational engine should be modified. Some work uses the path table which keeps all simple label paths in XML data. In this approach, the join between the path table and the element table is required. However, we do not use any additional data structure to handle a path but we speed up the query performance by transforming a path into an interval.

4.1 Relational Schema

At first, we describe our model for XML data. As mentioned earlier, XML data can be represented as an XML graph. In Definition 1, we do not distinguish attributes and elements since attributes are considered as specific elements. Also, we use the term element and node interchangeably in this paper.

Definition 1 *XML data is represented by the directed labeled node graph* $G_{XML} = (V, root, \Sigma, \lambda, E)$. $V = V_e \cup V_t$ *is the set of nodes where* V_e *is the set of nodes for elements or attributes in XML data and* V_t *is the set of nodes for data values.* $root \in V$ *is the root node of* G_{XML}. Σ *is the universe of labels for elements and attributes and* λ *is a function mapping* V_e *to* Σ. $E = E_e \cup E_t \cup E_r$ *is the set of edges where* $E_e \subseteq V_e \times V_e$ *is the set of edges between elements and/or attributes,* $E_t \subseteq V_e \times V_t$ *is the set of edges whose target nodes are for data values, and* $E_r \subseteq V_e \times V_e$ *is the edge set for referential relationships (i.e., ID-IDREF).* □

Also, as shown in Figure 4, the referential relationship is represented by an edge [3] from a node for an IDREF typed attribute to a node for the element which has the corresponding ID typed attribute as a child node. Since ID and IDREF typed attributes are specified by the structural information such as DTD [4] and XML Schema [12], the edge set E_r for referential relationships is specified when XML data is accompanied with DTD or XML Schema.

[3] It is represented by the dotted line in Figure 4

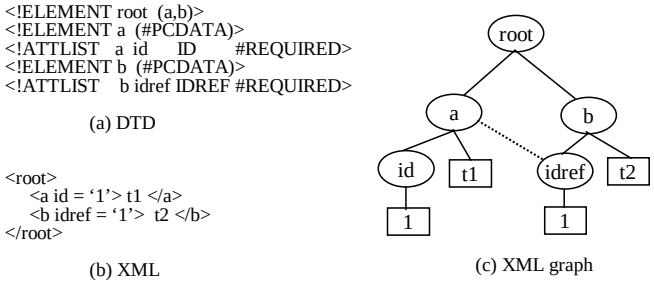


Fig. 4. An example of referential relationships

With respect to Definition 1, we create the relational schema (Table 1) which does not lose any information of the XML graph.

Table	Column	Description
`DocTable`	doc_name	XML document name
	doc_id	Corresponding identifier
	schema_id	Corresponding schema identifier
`Vertex`	did	Document id
	path	Path information for a node in V_e
	start	Start number of a region for a node in V_e
	end	End number of a region for a node in V_e
	level	Level of a node in V_e
	order	Order among same tagged siblings
	parent	Start number of the parent node
`Text`	did	Document id
	start	Start number of a region for a node in V_t
	end	End number of a region for a node in V_t
	parent	Start number of the parent node
	value	Data value of a node in V_t
`ID`	did	Document id
	owner	Start number of a node, the owner of the ID typed attribute
	id_value	Data value of ID typed attribute
`SchemaTable`	schema_name	Name of *SchemaTree*
	schema_id	Corresponding identifier
`SchemaTree` (see details in Section 4.2)	schema_id	*SchemaTree* id
	label	Label of a node in *SchemaTree*
	p_start	Start number for Interval_{label} (see Section 4.3)
	p_end	End number for Interval_{label}
	start	Start number of a region for a node in *SchemaTree*
	end	End number of a region for a node in *SchemaTree*
	level	Level of a node in *SchemaTree*
	parent	Start number of the parent node
	type	Cardinality and type information of a node in *SchemaTree*

Table 1
Relational Schema

Relations `DocTable, Vertex, Text` and `ID` are for XML data. `SchemaTable` and `SchemaTree` are for the structural information of XML data.

The relation `DocTable` records the XML document name, the corresponding document identifier and structural information identifier. The relations `Vertex` and `Text` correspond to V_e and V_t of Definition 1, respectively.

In the relations `Vertex` and `Text`, a node in G_{XML} is represented as an interval (`start`, `end`), which is generated by the Extented Region Numbering module in Figure 3. In addition, the subsets E_e and E_t of E are represented by `parent` and `start`, where `parent` is equal to the value of the parent node's `start`.

The column `path` in `Vertex`, storing a real number generated by the *Reverse Arithmetic Encoder*, is for λ of Definition 1 (see details in Section 4.3). The columns `level` and `order` of `Vertex` are extra information used to improve the query performance. The column `level` keeps the number of nodes in the path [4] from the root to itself. The column `order` keeps the ordering value among the same tagged siblings. For example, in Figure 1, there are two sections that are children of book. In this case, the order value for section &7 is 1 and that for section &13 is 2.

The relation `ID` is for E_r in Definition 1. The constraints for ID are that data values of ID typed attributes should be unique in an XML document and an element should have only one ID typed attribute. The *ID Manager* in Figure 3 gathers the elements having ID typed attributes and records them in the relation `ID` with data values of ID typed attributes. All attributes including IDREF typed attributes are recorded in `Vertex`. The data values of IDREF are stored in `Text`. Then, by 4-way equi-join of `Vertex, Text` and `ID`, the referential relationships can be computed. Thus, we explain our storage scheme based on the tree structured data. That is, we do not further discuss E_r in this section.

The structural information of each XML document is identified by `schema_id` which is stored in `SchemaTree`(see details in Section 4.2). The name of the structural information and the corresponding `schema_id` are maintained in `SchemaTable`. As mentioned above, the structural information such as DTD and XML Schema is optional. For XML documents without DTD or XML Schema, we extract the structural information using the SchemaExtractor module in Figure 3 and obtain `schema_name` by concatenating of the XML document name and the extension ".sch". Since the structural information is represented as a tree in XTRON, the attributes of SchemaTree are similar to `Vertex`.

[4] This path consists of the edges in E_e only.

4.2 Management of the Structural Information

In this section, we describe how to extract, maintain, and store the structural information of an XML document.

The structural information of an XML document can be described using DTD. Since a general DTD can be too complex to use for XML data storage, we adapt the simplifying DTD technique in [35]. The simplifying DTD technique gets rid of some information such as relative orders among subelements but preserves the cardinality information of subelements such as zero or one (?) and zero or more (*).

As a result of simplifying DTD, a set of trees, called *Element_Tree*s, is generated. An *Element_Tree* describes the structure of each element. The root of the Element_Tree denotes an element. Each leaf node (i.e., subelement) keeps a name of a subelement and a flag for cardinality information. Some *Element_Tree*s for Figure 1 are shown in Figure 5.

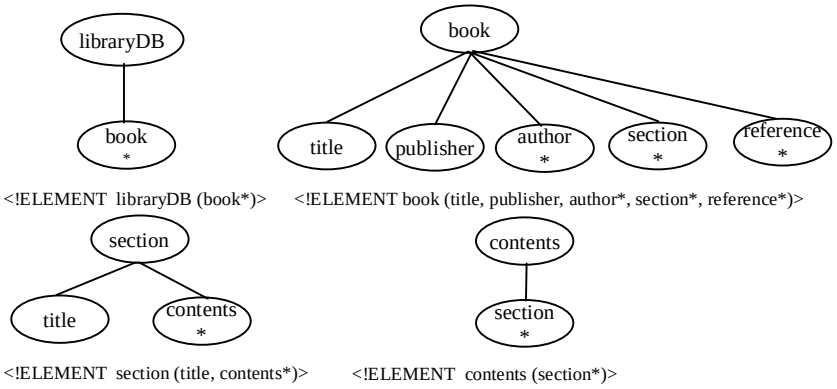


Fig. 5. Examples of *Element_Tree*s

Unfortunately, the structural information such as DTD is not mandatory for XML data. In this case, we extract *Element_Tree*s from the XML document.

Figure 6 describes the procedure SchemaExtractor which extracts *Element_Tree*s from XML data without DTDs. To extract *Element_Tree*s, we traverse G_{XML} in a child-depth-first fashion [1] starting from *root*. While extracting the structural information for XML data, we consider whether each of the subelements is optional(?) or multiple(*). The extracted structural information does not contain the relative order information of subelements, but the order information among elements still is still kept in the XML data. Thus.

When the procedure visits a node x, the procedure checks whether the corresponding *Element_Tree* T_{tag} exists or not (Line (4)) using the label of x (denoted by $\lambda[x]$). If the corresponding *Element_Tree* does not exist, then the procedure creates a new *Element_Tree* for the node (Line (5)). Let the set of

```
Procedure SchemaExtractor(G_XML)
begin
1.  r := root ∈ G_XML
2.  Extract_Element_Tree(r)
end

Procedure Extract_Element_Tree(x)
begin
3.  Element_Tree T_tag := find(λ[x])
4.  if (T_tag = NULL) {
5.      T_tag := new Element_Tree(λ[x])
6.  }
7.  p := set of child nodes of x
8.  for each unique λ[y ∈ p] do {
9.      t_sub := a leaf node in T_tag whose name is λ[y ∈ p]
10.     if (t_sub = NULL) {
11.         t_sub := new node(T_tag, λ[y ∈ p])
12.         t_sub.flag := '?'
13.     }
14.     sub_λ[y∈p] := subset of p whose elements' tags are λ[y ∈ p]
15.     if (number of elements in sub_λ[y∈p] > 1) t_sub.flag := '*'
16. }
17. for each child node y of x do {
18.     Extract_Element_Tree(y);
19. }
end
```

Fig. 6. An algorithm for Extraction of Element_Tree

x's children be p (Line (8)). For each unique label (denoted by $\lambda[y \in p]$) of x's children, the procedure checks whether the corresponding node t_{sub} exists or not in T_{tag} (Line (9),(10)).

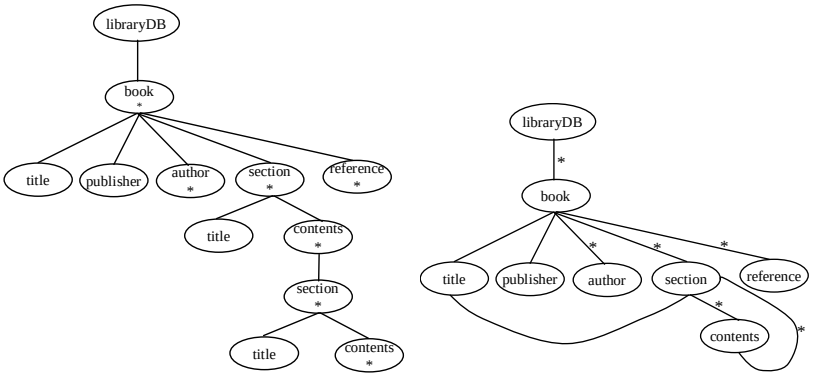
If the t_{sub} does not exist, we make the node in T_{tag} (Line (11)(12)). In this case, we set the flag as '?' (Line (12)). Suppose that the element B always appeared as a subelement for element A in an XML data. However, we cannot guarantee that B is a mandatory subelement of A since we do not know the XML data creator's intention. Thus, we use '?' as the default flag value. If the number of x's children whose labels are $\lambda[y \in p]$ is greater than 1, we set the flag of t_{sub} to '*' (Line (14)(15)). Finally, for each node of x, we extract and update the *Element_Tree* by recursion (Line (17)-(19)).

The *Element_Tree* shows only the structure of an element. Thus, as shown in Figure 7-(a), the overall structure information of XML data is required. To obtain the overall structural information of XML data, we consolidate the set of *Element_Tree*s into a tree called *SchemaTree*. The *SchemaTree* is used for XML data storage in XTRON.

We store and retrieve the *SchemaTree* by reusing the method to store and retrieve XML data. Our storage method for XML data originated from region numbering which is based on the tree structured data model. In addition, the cardinality information (e.g., ?, *) or type information (e.g., ID and IDREF) of each node is recorded in the attribute `type` in the relation `SchemaTree`.

[35] introduces a notion of a DTD graph to represent the structure of DTD.

However, as shown in Figure 7-(b), the DTD graph may contain a cycle. To store and retrieve a DTD graph in the rigid structure (i.e., relational tables), specific methods should be devised.



(a) SchemaTree (b) DTD Graph

Fig. 7. SchemaTree vs. DTD Graph

To make a *SchemaTree* in a tree form, we consider two complications: *sharing* and *recursion*. As shown in 7-(b), title is shared by book and section. This sharing violates the tree form. Thus, we set shared elements apart. The recursion is represented as a cycle in DTD graph. Section and contents show the recursive relationship in Figure 7-(b). To represent the recursion in tree form, we unfold the cycle. In this case, an infinite chain may appear. We restrict the level of unfolding such that a name of a certain node appears at most twice in the path from the root to the node. Therefore, only one section node and one contents node have a section node and a content node as descendants, respectively, as shown in Figure 7-(a). The schemaTree is utilized by the *Extend Region Numbering* module.

4.3 Reverse Arithmetic Encoding

In this section, we present the behavior of the Reverse Arithmetic Encoder which generates a real number to represents the path information of an element. The generated real number is recorded on the `path` column in the `Vertex` relation.

We first define some notations on a simple XML snippet to explain the reverse arithmetic encoding method.

Definition 2 *A* simple path *of a node $v_n \in V_e$ in G_{XML} is a sequence of one or more dot-separated labels $t_1.t_2 \ldots t_n$, such that there is a path of $n-1$ edges (e_1,...,e_{n-1}) starting from root to v_n where $e_i =< v_i, v_{i+1} > \in E_e$, $v_1 = root$, and $\lambda[v_i]$ is t_i.* □

For example, in the XML data shown in Figure 1, the simple path of a *title* element is *libraryDB.book.section.title*.

Definition 3 *When the simple path of a node $v \in V_e$ is $a_1.a_2 \ldots a_n$, a dot-separated tag sequence $b_k.b_{k+1} \ldots b_n$ is a* label path *of v if we have $b_k = a_k$, $b_{k+1} = a_{k+1}$, ..., $b_n = a_n$, where $1 \leq k$ and $k \leq n$. For two label paths, P= $p_i \ldots p_n$ and Q=$p_j \ldots p_n$ of v, if $i \geq j$, then we call P a* suffix *of Q.* □

Again in Figure 1, *section.title* is a label path of the *title* element, and *title* is a suffix of *section.title*.

The reverse arithmetic encoding originates from XPRESS which is a queriable XML data compressor [26]. Basically, the reverse arithmetic encoding represents the simple path of an element (or attribute) by an interval of real numbers between 0.0 and 1.0.

To generate the interval, the statistics (i.e., the frequencies of labels) are required. In XPRESS, the frequencies of labels can be obtained by a scan of XML data. But, in XTRON, the statistics are obtained by traversing *SchemaTree* instead of an XML data scan since *SchemaTree* contains whole tags in XML data and is much smaller than XML data.

Then, according to the frequencies, each tag T has its own interval Interval_T in [0.0, 1.0). The size of Interval_T is proportional to the frequency (normalized by the total frequency) of tag T. Each Interval_T is recorded as `p_start` and `p_end` attributes in the `SchemaTree` relation.

Function reverse_arithmetic_encoding(v_n, [$min_{v_{n-1}}$, $max_{v_{n-1}}$))
begin
1. $p_n := \lambda[v_n]$
2. $[min_{v_n}, max_{v_n}) := \text{Interval}_{p_n}$
3. **if**($[min_{v_{n-1}}, max_{v_{n-1}}) = \text{NULL}$) **return** $[min_{v_n}, max_{v_n})$
4. length := max_{v_n} - min_{v_n}
5. $min_{v_n} := min_{v_n}$ + length* $min_{v_{n-1}}$, $max_{v_n} := min_{v_n}$ + length* $max_{v_{n-1}}$
6. **return** $[min_{v_n}, max_{v_n})$

end

Fig. 8. An algorithm of reverse arithmetic encoding

The reverse arithmetic encoding encodes the simple path $P= p_1.\ldots.p_n$ of an element v_n into an interval $[min_{v_n}, max_{v_n})$ using the algorithm in Figure 8.

An input parameter $[min_{v_{n-1}}, max_{v_{n-1}})$ of the function reverse_arithmetic_encoding is the interval for the simple path ($= p_1.\ldots.p_{n-1}$) of v_n's parent node v_{n-1}. Basically, we encode the simple path of a node in G_{XML} starting from the root element in the depth first tree traversal. Therefore, $[min_{v_{n-1}}, max_{v_{n-1}})$ has already been computed in the time of encoding the simple path of v_{n-1}.

Intuitively, the reverse arithmetic encoding reduces Interval_{p_n} in proportion to the interval $[min_{v_{n-1}}, max_{v_{n-1}})$ (Line (4)-(6)). In the case of the root node,

since it does not have a parent node, $[min_{v_{n-1}}, max_{v_{n-1}})$ is null. Thus, the interval for the root node is the Interval_{p_n} itself (Line (3)).

The following example shows the intervals for nodes in Figure 1.

Example 1 *Suppose that the Interval$_T$s of labels = {libraryDB, book, section, title, year, ..., reference} are {[0.0,0.3), [0.3,0.5), [0.5,0.6), [0.6,0.7), [0.7,0.75) ..., [0.9,1.0)}, respectively. Then, the interval [0.653, 0.6536) for a simple path libraryDB.book.section.title in Figure 1 is obtained by the following process:*

element	simple path	Interval$_T$	subinterval
libraryDB	*libraryDB*	[0.0, 0.3)	[0.0, 0.3)
book	*libraryDB.book*	[0.3, 0.5)	[0.3, 0.36)
section	*libraryDB.book.section*	[0.5, 0.6)	[0.53, 0.536)
title	*libraryDB.book.section.title*	[0.6, 0.7)	[0.653, 0.6536)

The intervals generated by the reverse arithmetic encoding express the relationship among label paths as follows:

Property 2 *Suppose that a simple path P is represented as the interval I, then all intervals for suffixes of P contain I.*

For instance, the interval [0.6, 0.7) for a label path *title* and the interval [0.65, 0.66) for a label path *section.title* contain the interval [0.653, 0.6536) for a simple path *libraryDB.book.section.title*. As a result, path expressions are effectively evaluated, thanks to Property 2 without the join of the path table.

Also, since Property 2 is valid even though the interval is replaced by the minimum value min_{v_n} of the interval, we record only min_{v_n} as the `path` attribute in the `Vertex` relation instead of the interval.

4.4 Numbering on XTRON

Now, we explain our numbering scheme for elements in XML data.

As mentioned earlier, the edge approach is efficient in obtaining the parent-child relation among elements and the region approach is efficient in computing the ancestor-descendant relation among elements. Thus, we combine the edge approach and the region approach into a tuple of a relation.

In XTRON, an element x is represented by a region $(start_x, end_x)$ based on the region approach. The representation of each XML element has the following property.

Property 3 *If and only if an XML element x is an ancestor of an XML element y, $start_x < start_y < end_y < end_x$. And, if and only if XML element x is a parent of an XML element y, $start_x = parent_y$.*

A weakness of the original region approach is concerned with updates. To prevent the renumbering of region numbers at each insertion, we preserve extra space between elements. In this view, our approach is similar to the numbering scheme (i.e., *extended preorder*) of XISS [20]. However, *extended preorder* uses ad-hoc numbering to assign the extra space. In contrast to *extended preorder*, we utilize the structural information to assign extra space in each region.

For example, as shown in Figure 5, an element book can have title, publisher, author, section, and reference elements as children. Among them, title and publisher elements are mandatory children. Thus, only three kind elements (i.e., author, section, and reference elements) can be inserted several times in future. Our numbering scheme preserves the extra space for three kind elements between each child element of book element.

Although the efficient update of XML data is important, it is not the focus of our paper. Therefore, we omit any further discussions on efficient updates. The update issue will remain as our future work since the efficient supporting to update is a major research issue.

5 Query Processing on XTRON

In this section, we present a method for translating from XQuery to SQL over the relational representation presented in Section 4. The syntax and semantics of XQuery are too wide to discuss since XQuery is designed to support a broad class of applications. Thus, we present our translation mechanism from XQuery to SQL based on two core expressions of XQuery: path expressions and FLWR expressions.

5.1 Translation of Path Expressions

Path expressions are used to select (address) parts of an XML document. A path expression traverses its input document using location steps. A step consists of 3 parts.

step::= *axis*"::" *NodeTest* ("[" *predicate*"]")*

For each step, an *axis* describes the direction in which the nodes should be traversed. In XQuery, thirteen axes are available such as `child, descendant, attribute, self, descendant-or-self` and `parent`. XQuery includes XPath's shorthand notation for some axes. Among the thirteen axes, `child` can be omitted since `child` is the default axis. Also, `descendant` is replaced by // [5]. *NodeTest* specifies the node type and the name of nodes selected by the axis. *predicate* uses arbitrary expressions to further refine the set of nodes selected by the step.

As mentioned in Section 4.3, the simple path of each element is encoded as a real number by the reverse arithmetic encoder and recorded in the `path` attribute of the relation `Vertex`. The structural relationships among elements such as parent-child and ancestor-descendant relationships can be computed by the attributes `start, end` and `parent` in the relation `Vertex`.

In general, to process multi-step (k > 1) XPath path expressions, the system generates an SQL query that is nested to depth k, and then this scheme is improved to generate an SQL query involving a k-way self-join [19]. XTRON reduces the number of self-joins using the reverse arithmetic encoding.

The algorithm for translating a path expression into an SQL query is shown in Figure 9. An object *sql* for an SQL statement has attributes :*select, from, where, orderby,* and *with* for SELECT, FROM, WHERE, ORDERBY and WITH clauses. Also, to keep the relation name for the temporary result of an XQuery expression, *prev_rel* is used.

For brevity, we only describe the behavior for `child` and `descendant` among the various *axes* (Line (6)-(15)) and label among diverse *NodeTest*s (Line (16)-(21)) in Figure 9. The portion in Line(22)-(27) is for predicates of a step.

Based on Property 2, a path expression formed in $//l_1/l_2/\ldots/l_n$ or $/l_1/l_2/\ldots/l_n$ is transformed into an interval $[min_{path}, max_{path})$ with $level\_gap(= n)$ and *desc* (Line (2)-(4)). At Line (3), *pathexp[i]* denotes the i'th step in the path expression.

The variable *desc* is used to specify that the first axis of the current path expression (or subpath expression) is `descendant` or not. Then, we can generate the corresponding SQL query using $[min_{path}, max_{path})$, *level* and *desc* in the function gen_SQL().

For understanding of the algorithms, we will use examples to explain the parts of the algorithms.

Example 2 *Suppose that the tags and the corresponding Interval$_T$s are the*

[5] Strictly speaking, // is the abbreviation form of `descendant-or-self::node()/`

```
function TranslatePath(sql, pathexp)
begin
1.  desc := false, level_gap := 0, min_path := 0.0, max_path := 1.0
2.  for i := 1 to length(pathexp) do {
3.     sql := tran_step(sql, pathexp[i], desc, level_gap, min_path, max_path)
4.  }
5.  return sql
end
function tran_step(sql, step, desc, level_gap, min_path, max_path)
begin
6.  switch(step.axis) {
7.     case "//":          /* descendant */
8.        if([min_path,max_path) != [0.0,1.0)) {
9.           sql := gen_SQL(sql, desc, level_gap, min_path, max_path)
             /*genete SQL fragment for the subpath expression */
10.          level_gap := 0, min_path := 0.0, max_path := 1.0
11.       }
12.       desc := true, level_gap := level_gap+1
13.    case "/":           /* child */
14.       level_gap := level_gap+1
15. }
16. switch(step.NodeTest) {
17.    case label: [min_label, max_label) := get Interval_label
18.       length := max_label - min_label
19.       min_path := min_label+length·min_path
20.       max_path := min_label+length·max_path
21. }
22. if(step.predicates != NULL) {
23.    sql := gen_SQL(sql, desc, level_gap, min_path, max_path)
24.    desc := false, level_gap := 0, min_path := 0.0, max_path := 1.0
25.    for i = 1 to length(predicates) do
26.       sql := tran_predicate(sql, i, step)
27. } else if(step is the last step of pathexp) sql := gen_SQL(sql, desc, level_gap, min_path, max_path)
28. return sql
end
```

Fig. 9. An algorithm of translation from Path expression to SQL

same as those in Example 1. A path expression P = //book/section/title is translated into the following SQL query:
SELECT V_1*.**
FROM Vertex V_1
WHERE 0.653 <= V_1*.path AND* V_1*.path < 0.655 AND* V_1*.level >= 3*

Example 2 illustrates the translation from a path expression such as $//l_1/l_2/\ldots/l_n$ to an SQL query. Since the axis of the first step is `descendant`, *desc* is set as true (Line (12)). Also, since the axes of the remaining steps are `child`, *level* is increased to 3 (Line (14)). The *NodeTest* of each step is *label*. Thus, we obtain a single interval (Line (17)-(20)) based on the reverse arithmetic encoding.

As mentioned in Section 4.3, the labels and the corresponding Interval_Ts of an XML document are stored in the relation `SchemaTree`. Thus, an Inteveral_T for a label is obtained efficiently (Line (17)). Finally, at Line (27), the corresponding SQL query of P is generated by calling the function gen_SQL().

XTRON is similar to BLAS [6] in the aspect of the utilization of interval representation. But, BLAS uses an ad-hoc numbering to assign a region, whereas XTRON utilizes the structural information of XML data to assign a region.

In addition, only path expressions are considered in BLAS. However, we support a more powerful query facilities based on XQuery features such as the dereference operator (i.e., =>), FLWR expressions and Element Constructors.

Unlike the path expression P in Example 2, some path expressions which contain '//' (i.e., descendant) in the middle of the path expression cannot be translated into a single interval. Conceptually, these kinds of path expressions are divided into subpath expressions and consolidated using the difference of the levels.

For instance, a path expression $Q = //l_1/\ldots/l_i//l_{i+1}/.../l_j$ is divided into $Q_1 = //l_1/\ldots/l_i$ and $Q_2 = //l_{i+1}/.../l_j$.

During the process from the first step to i'th step ($= /l_i$), $level$ ($= i$), $desc$ (= true) and $[min_{path}, max_{path})$ are computed like the path expression P in Example 2. When visiting the $i+1$'th step ($= //l_{i+1}$), $[min_{path}, max_{path})$ is not [0.0, 1.0). So, the SQL statement for Q_1 is generated (Line (8)-(11)). And, $desc$, $level$, and $[min_{path}$,$max_{path})$ are reinitialized, respectively.

Then, we can obtain $level$ ($= j - i$), $desc$ (= true) and $[min'_{path}$,$max'_{path})$ for Q_2. In this case, the level of the result elements of Q_2 is greater than or equal to $(j - i)$ + *level of the elements accessed by* Q_1 . In addition, an element accessed by Q_2 is a descendant of an element accessed by Q_1. Suppose that the result relation for Q_1 is V_1 and the result relation for Q_2 is V_2. Then, the SQL query is such that :

SELECT V_2.*
FROM Vertex V_1, Vertex V_2
WHERE min_{path} <= V_1.path AND V_1.path < max_{path} /*path condition for Q_1*/
AND V_1.level >= i /*level condition for Q_1*/
AND min'_{path} <= V_2.path AND V_2.path < max'_{path} /*path condition for Q_2*/
AND V_2.level >= V_1.level+$(j - i)$ /*level condition for Q_2*/
AND V_1.start < V_2.start AND V_2.end < V_1.end
/*structural condition between Q_1 and Q_2*/

Next, let us consider the path expressions which contain predicates (Line (22)-(27) in Figure 9).

A step can include a sequence of predicates. As mentioned above, predicates are used to prune the set of nodes selected by the current step. Thus, the SQL query for the relation which is the result of the step is generated (Line (23) in Figure 9). Let the resulting relation for the step be kept in $sql.prev\_rel$.

To translate the predicates of a step, more dedicated handling is required. The algorithm which translates a predicate into an SQL query is shown in Figure 10.

```
function tran_predicate(sql, i, step)
begin
1.  predicate := step.predicates[i]
2.  switch(predicate) {
3.    case [pathexp]: /*path expression
4.      temp_rel : = sql.prev_rel
5.      sql := TranslatePath(sql, pathexp);
6.      sql.prev_rel := temp_rel
7.    case [n]: /*order based predicate
8.      if(i = 1) {
9.        sql.where := sql.where+ AND + sql.prev_rel.order = n
10.     }
11.     else { // in form of label[path][2]
12.       x := sql.prev_rel
13.       sql.from :=
14.         ( SELECT DISTINCT x.did, x.path, x.start, x.end, x.level,
15.             ROWNUMBER() OVER (ORDER BY x.start) AS order, x.parent
16.           FROM sql.from WHERE sql.where) AS X_t
17.       sql.where := X_t.order = n
18.       sql.prev_rel := X_t
19.     }
20.   case [pathexp op value]: //conditional expression
21.     temp_rel := sql.prev_rel
22.     sql := TranslatePath(sql, pathexp)
23.     sql.from = sql.from+ TEXT T_t
24.     sql.where = sql.where+ AND sql.prev_rel.start = T_t.parent
25.     sql.where = sql.where+ AND T_t.value op value
26.     sql.pre_rel := temp_rel
27.   case => name_test: //derefence
28.     if(IS_IDREF(step) = false) return ERROR
29.     sql.from := sql.from + Text T_t, ID id_t, Vertex V_t
30.     sql.where := sql.where+ AND sql.prev_rel.start = T_t.parent //get data value of current node
31.     sql.where := sql.where+ AND T_t.value = id_t.id_value //get the corresponding ID attribute
32.     sql.where := sql.where+ AND V_t.start = id_t.owner
33.     if(name_test is label) {
34.       [min_label, max_label) := get Interval_label
35.       sql.where := sql.where+ AND min_label <= V_t.path AND V_t.path < max_label //for label
36.     }
37.     sql.prev_rel := V_t
38.     t := t+1
39. }
40. return sql
end
```

Fig. 10. An algorithm for translation from a predicate to SQL

A path expression can be applied as a predicate. Such a predicate selects all nodes whose paths starting from themselves satisfy the path expression. For instance, //book[title] selects all book elements which have title as a child.

The path expression as a predicate is handled in Line(3)-(6). Basically, the SQL query for a predicate is added in the SQL query for a step by calling TranslatePath() which in turn calls gen_SQL(). In this case, a θ-join of the relation for the predicate and the relation for step is involved. According to the definition of a θ-join, the tuples satisfying the θ condition remain as results. Thus, by a θ-join, the elements which do not satisfy the predicate are pruned. After calling TranslatePath(), *sql.prev_rel* is changed. Thus, the recovery of *sql.prev_rel* is required (Line(6)).

Another type of the predicate is the order-based predicate formed in [position() = n]. Such a predicate selects a node whose position is n among the

currently visited nodes. The order-based predicate simplifies like $[n]$. For instance, //section[2] selects all descendant section elements that are the second section children of their parents.

To compute each order-based predicate, Tatarinov et. al [39] uses the function RANK(), originally proposed as an OLAP extension. However, they do not consider the more complicated semantics of the order-based predicate.

The meaning of the order-based predicate varies according to the owner step of the predicate and the location of the predicate among predicates. For instance, //section[title][2] means "among all section elements which have a title child, select the second section element." But //section[2][title] means "selects all descendant sections that are the second section children of their parent and have title as children." The approach of [39] does not distinguish between these cases.

As shown in Table 1, the order of same tagged siblings is recorded in the `order` attribute in the `Vertex` relation in XTRON. Thus, a position-based predicate located at the first position such as $label[n]$ can be easily translated into SQL without the function RANK() (Line (8)-(10)).

The position-based predicate which is not located at the first position is handled using the function ROWNUMBER() which assigns the order number of each row specified by "ORDER BY" to each row starting from 1 (Line(11)-(19)).

Also, a conditional expression can be applied as a predicate (Line (20)-(26)). To support the conditional expression, the data values of the nodes are obtained by the join of the result relation and `Text` (Line (24)). Then, the condition between the data values and *value* is inserted (Line (25)). In this case, according to the type of *value*, the type casting of data values may occur using the function CAST(), such as CAST(V_t.value AS INT).

A difference between XQuery path expressions and XPath is the support of the dereference operator (i.e., =>). In XTRON, the start value of the region for an element, which has an ID typed attribute, is stored in the owner column of the `ID` relation with the value of the ID typed attribute. Thus, XTRON supports the dereference operator (Line(27)-(38)) [6] . The left side of the dereference operator must be an IDREF typed node. Thus, type checking of a step is performed using `type` of `SchemaTree` (Line (28)). The node set of the deference operator is identified (Line (30)-(31)) by a 4-way join of the relation for currently visited nodes (i.e., *sql.prev_rel*), `Text` T_t, `ID` id_t, and V_t. The V_t is the result relation of the dereference operator. By the join of *sql.prev_rel* and

[6] Actually, the dereference operator is not categorized into predicates. But, for brevity, we explain the dereference operator in the algorithm for predicates

`Text` T_t, we can obtain the data values of IDREF attributes. Then, by the join of `Text` T_t and `ID` id_t, we can obtain the information for the corresponding ID type attributes. As mentioned in Section 4, the start number of an element which is the owner of an ID typed attribute is kept in the owner column of the ID table. Thus, by the join of `ID` id_t and `Vertex` V_t, we can select result elements of the dereference operator (Line (32)).

In addition, a deference operator is followed by *name_test* that specifies the name of the target element. If *name_test* is *label*, we refine the nodes accessed by the deference operator using the Interval_{label} for *label*.

5.2 Translation of FLWR Expressions

In this section, we briefly present the translation mechanism for FLWR expression since FLWR expression itself is complicated. Conceptually, a FLWR expression consists of FOR, LET, WHERE, and RETURN clauses.

FLWRExpr ::= (*FORClause*|*LETClause*)+ *WHEREClause*? *RETURNClause*
FORClause ::= "FOR" *variable* "in" *Expr* ("," *variable* "in" *Expr*)*
LETClause ::= "LET" *variable* ":=" Expr ("," *variable* ":=" *Expr*)*
WHEREClause ::= "WHERE" *Expr*
RETURNClause :: = "RETURN" *Expr*

FOR and LET clauses are used to bind one or more variables to one or more expressions. The FOR clause is used whenever the iteration is needed. In the FOR clause, the variables are bound to individual values returned by their corresponding expressions, whereas the LET clause simply binds each variable to the value of its respective expression without iteration. The tuples bound by FOR or LET clause can be pruned by an optional WHERE clause. The RETURN clause generates the output of the FLWR expression.

Figure 11 shows a simple FLWR expression which generates the titles of the books published in 2000 as millennium sales.

```
FOR $x IN document(librarayDB.xml)/libraryDB/book
LET $y := 2000
WHERE $x/year = $y
RETURN <millennium_sales> $x/title </millennium_sales>
```

Fig. 11. A FLWR expression

The algorithm for translating from a FLWR expression into a single SQL query is shown in Figure 12.

With respect to the syntax of XQuery, FOR and LET clauses which bind one or more variables to one or more expressions can appear in an arbitrary order.

```
1.symtable := new Hash() //symbol table for variable
function TranslateFLWR(sql, flwrexp)
begin
2.  sql_list := new SQL[length(flwrexp.bindings)]
3.  for i := 1 to length(flwrexp.bindings) do {
4.     sql_list[i] = TranslateBinding(sql_list[i], flwrexp.bindings[i])
5.     symtable.insert(flwrexp.bindings[i].variable, sql_list[i].prev_rel)
6.  }
7.  sql_where := new SQL()
8.  if(flwrexp.where != null) {
9.     sql_where := TranslateWhere(sql_where, flwrexp.where)
10. } else sql_where := null
11. sql := TranslateReturn(sql, flwrexp.return, sql_where, flwrexp.bindings, sql_list)
12. return sql
end
function TranslateBinding(sql, binding)
begin
13. switch(type(binding.exp)) {
14.    case pathexp:
15.       sql := TranslatePath(sql, binding.exp)
16.    case FLWRexp:
17.       sql := TranslateFLWR(sql, binding.exp)
18. }
19. return sql
end
```

Fig. 12. An algorithm of translation from FLWR to SQL

```
1.  ...
2.  case variable:
3.     sql.prev_rel := symtable.hash(variable)
4.  ...
```

Fig. 13. Partial extension of Figure 9 for variables

The variables and the corresponding expressions of FOR and LET clauses in a FLWR expression $flwrexp$ are kept in $flwrexp.bindings$. Each expression for a variable is translated by calling TranslateBinding() (Line (4) in Figure 12). Since the binding variables may be used in other expressions, the result of the expression for the variable is kept in a hash table symtable (Line (5)). Also, to support some path expressions in the FLWR expression starting with variables (e.g., $x/title), the portion in Figure 13 is required at Line (7) of the function TranslatePath() in Figure 9.

The function TranslateBinding() translates the expression into a single SQL query according to the type of the expression (Line (13)-(18)). In this case, path expression (Line (14)-(15)) or another FLWR expression (i.e., nested FLWR expression) (Line (16)-(17)) can be involved.

Next, the WHERE clause is translated (Line(7)-(10)). The WHERE clause which is an optional clause refines the nodes selected by FOR or LET clauses. As presented in Section 5.1, predicates of the path expression's step are similar to the WHERE clause. Thus, the translation mechanism of the WHERE clause is similar to that of predicates. We omit the details of the translation of the WHERE clause since we present the translation mechanism for predicates in Section 5.1.

Finally, the RETURN clause which generates the output of the FLWR expression is translated (Line(11)). The SQL queries for FOR, LET and WHERE clauses are consolidated into a single SQL query by the function TranslateReturn(). XML data are stored across multiple tables, and an XQuery query is translated into an SQL query. Then, the tuples which are the results of the SQL query are obtained by the relational engine. Thus, the reconstruction of the XML fragments using the tuples is required. Many researchers proposed various publishing techniques for relational data in the form of XML documents [5,15,16,31,32]. One of the prominent XML publishing techniques is the sorted outer union [31,32]. Thus, like most related literatures [24,34,38], we adapt sorted outer union technique for a result generator.

6 GUI of XTRON

For providing the user interface, we implemented GUI of XTRON. Figure 14 shows GUI of XTRON. There are a menu and icons in the top part of GUI. By clicking a menu or icons, we can log in XTRON, log out XTRON, choose XML data, store XML data or run XQuery. The tree in the left window represents XML data lists which are stored in each database. Currently, shake1.xml and xmark1.xml are stored in DB2 database. In the side of the tree, there are XML document and Query tab. We can see XML data through XML document tab and perform XQuery by Query tab. The contents of xmark1.xml are shown in XML document tab of Figure 14.

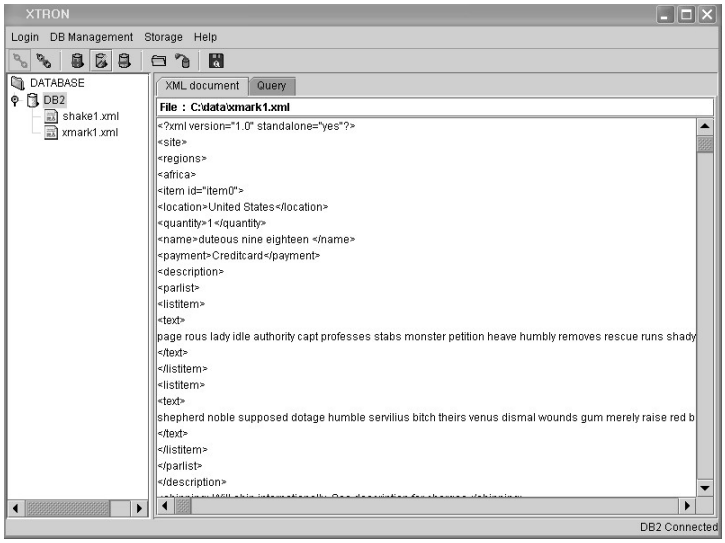


Fig. 14. GUI of XTRON

In order to execute XQuery, we must write the query in the text window of the Query tab which are located in the top as shown in Figure 15. Then, if we click the RUN button, the result for XQuery will be shown in the text window

of the Query tab which are located in the bottom. Figure 15 shows GUI after performing the following XQuery.

FOR $i in document("xmark1.xml") /site/regions/australia/item
RETURN <item> {$i/description} </item>

The result of that query is displayed in GUI. We can see the previous results or next results by clicking PREV or NEXT button in Query Tab. For removing the results, click Clear button. In the case that the size of the result is large, we can see the result by moving the scroll bar. Also, if the wrong XQuery enter, the error message window will be shown.

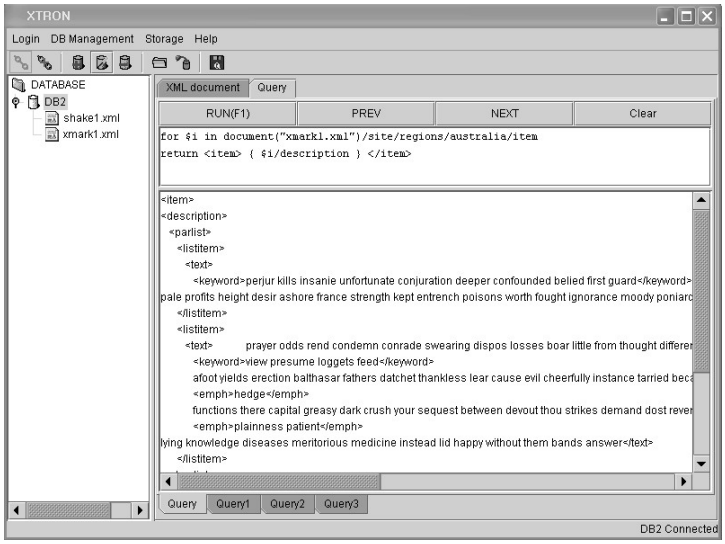


Fig. 15. Performing XQuery in GUI

7 Experiments

To show the efficiency of our system, we empirically compared it with diverse storage schemes: the edge approach, the region approach, and the path table with region approach [7] on real-life and synthetic data sets. In addition, we compared the query performance of XTRON with the well-known XML processing systems: Galax 0.3.5 [8] and Berkeley DB XML 2.0 [9].

For the experiments, we implemented XTRON and the systems for above approaches with JAVA language. Galax is a native XML query processing system which does not utilize the relational databases. Berkeley DB XML performs the XQuery processing using the relational database. To support

[7] we call it the region path approach
[8] available at http://www.galaxquery.org/
[9] available at http://www.sleepycat.com/

XML query processing, Berkeley DB XML modifies the relational engine. In contrast, as mentioned earlier, XTRON does not change the relational engine.

In this section, we first explain the experimental data sets and the query set. Then, we present the query performance.

7.1 Experimental Environments

The experiments were performed on a Pentium IV-3.6 Ghz platform with MS-Windows XP and 3.25 GBytes of main memory. We used IBM DB2 Universal Edition 8 as a repository and stored the data on a local disk. In the database, we built up the multi-column index with did (document id), path, and start columns on the Vertex table for XTRON, the multi-column index with did, source, and target columns on the edge table for the edge approach, the multi-column index with did and start columns on the region table for the region approach, and the multi-column index with did, start, path_id columns on the region_path table for the region path approach. To connect DB2, we used the JDBC driver. No other optimization such as physical storage tuning was made on DB2. We begin by describing the XML data sets and queries used in the experiments.

Data Sets We evaluated XTRON using real-life and synthetic XML data sets: Shakespeare and XMark. Each data set consists of five XML documents. The characteristics of the data sets used in our experiment are summarized in Table 2. Size denotes the disk space of XML data in MBytes and Description indicates how each XML document is constructed.

Data Set	Name	Size (MBytes)	Description
XMark	XM1	0.19	$f = 0.001$
	XM2	1.18	$f = 0.01$
	XM3	11.88	$f = 0.1$
	XM4	118.55	$f = 1$
	XM5	1185.5	$f = 10$
Shakespeare	SH1	0.28	1 Play
	SH2	0.97	4 Plays
	SH3	7.89	37 Plays
	SH4	78.95	37 Plays X 10
	SH5	789.5	37 Plays X 100

Table 2
XML Data Set

We used the XMark data [30] developed for the XML benchmark project for the synthetic data sets. The XMark data simulates the contents of an Internet

auction site. The XMark data set is generated using xmlgen [10], the XML data generator of the XMark benchmark. By adjusting the scaling factor, f, we generated five XML documents for the XMark data set.

The Shakespeare [9] data set is the real-life XML data composed of the collection of plays of Shakespeare. As shown in Table 2, the Shakespeare data set also consists of five XML documents: SH1, SH2, SH3, SH4 and SH5. SH1 is the play entitled "The Tragedy of Hamlet, Prince of Denmark." SH2 contains four tragedies (Hamlet, Macbeth, Othello, and King Lear), and all plays of Shakespeare are contained in SH3. To test the effectiveness of XTRON on large sized XML data, we scaled up SH3 by 10 times for SH4 and by 100 times for SH5.

Query Sets We evaluate XTRON using several queries. The queries used in this experiments are presented in Table 3.

The first character in the first column indicates the data set on which the query is executed: 'X' denotes XMark, 'S' denotes Shakespeare.

The number in the first column represents the type of query. The queries of type 1 are path expressions based on a simple path, the queries of type 2 are partial matching path expressions, the queries of type 3 are FLWR expressions, the queries of type 4 are FLWR expressions with an element constructor, and the queries of type 5 are nested FLWR expressions. Query Definition in Table 3 describes the corresponding XQuery queries.

7.2 Experiments for the Query Evaluation

In our experiments, some queries did not terminate within twelve hours on some approaches. We denote these cases as DNT (Did Not Terminate). The query time is the average over multiple executions.

First, we present the performance of path expressions (i.e., query types 1 and 2) compared with the edge approach since it is difficult to systematically translate a FLWR expression into a single SQL query in the edge approach. Table 4 shows the path expression performance on XM3 and SH3. Parsing denotes the parsing time of given path expressions and Translation represents time consumed to convert a parsing result to an SQL statement in milliseconds. Execution denotes the execution time of the translated SQL statement and Total is the sum of parsing time, translation time and execution time.

As shown in Table 4, the performance of the edge approach is very low.

[10] Available at http://monetdb.cwi.nl/xml/downloads.html

Name	Query Definition
X1	/site/regions/australia/item/mailbox/mail/date
X2	//description//text/keyword[1]
X3	FOR $a in /site/people//profile WHERE $a/education = "College" RETURN $a/interest
X4	FOR $i in /site/regions/australia/item RETURN <item> $i/description </item>
X5	FOR $a in /site//africa/item/mailbox RETURN <seller> FOR $b in /site//closed_auctions/closed_auction WHERE $b/date = $a/mail/date RETURN $b/seller </seller>
S1	/X/PLAY/ACT/SCENE/SPEECH/STAGEDIR
S2	//ACT//TITLE[1]
S3	FOR $a in /X/PLAY//SPEECH WHERE $a/SPEAKER = "Ghost" RETURN $a/STAGEDIR
S4	FOR $b in /X/PLAY/PERSONAE RETURN <PERSONAE> $b/PGROUP </PERSONAE>
S5	FOR $a in /X//PERSONAE/PGROUP/PERSONA RETURN <PGROUP_SPEAKER> FOR $b in /X//ACT/EPILOGUE//SPEAKER WHERE $b = $a RETURN $b </PGROUP_SPEAKER>

Table 3
XML Query Set

Data	Query	Approach	Parsing	Translation	Execution	Total(msec)
XM3	X1	XTRON	203	94	750	1047
		EDGE	328	31	31141	31578
	X2	XTRON	203	94	10875	11281
		EDGE	172	16	2856750	2857047
SH3	S1	XTRON	203	16	1531	1750
		EDGE	328	16	115828	116266
	S2	XTRON	203	16	734	953
		EDGE	1187	178	5390735	5392367

Table 4
Evaluation Cost of Path Expressions (msec)

In XTRON, the path expression is translated into intervals by using the reverse arithmetic encoder. Thus, the parsing time and the translation time of XTRON is a little worse than those of the edge approach. However, the execution time of XTRON is significantly better than that of the edge approach.

In the edge approach, to obtain all descendant elements of the target elements which are reached by a path expression, massive joins are required. Also, to compute descendant axes (i.e., //) in type 2 queries, additional recursive clauses are required. Thus, the performance gap of type 2 queries between XTRON and the edge approach is greater than that of type 1 queries.

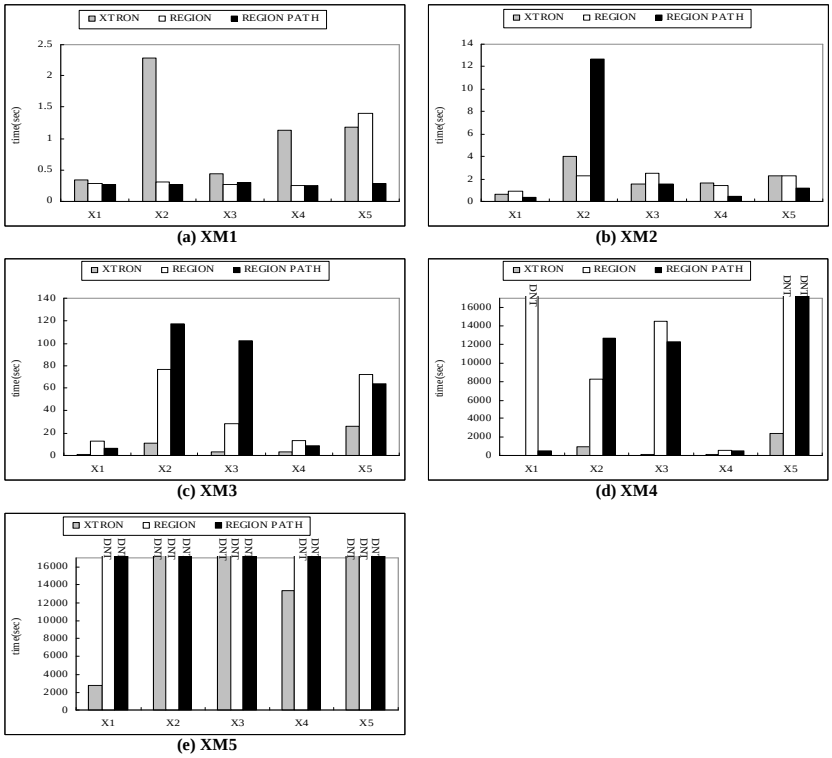


Fig. 16. Query evaluation cost for XMark Data Set

Next, we present the query performance of XTRON compared with the region approach and the region path approach over diverse query types. As shown in Table 4, the performance of the edge approach is much worse due to massive joins. However, the region based approaches (i.e., XTRON, region, and region path approach) do not require massive joins to obtain descendant elements of a certain element. Thus, in this experiment, we did not contain the edge approach.

The query evaluation times over various sized XML data are shown in Figure 16 and Figure 17. As shown in Figure 16 and Figure 17, XTRON shows good performance over most cases except when XML data is very small.

When XML data is very small (i.e., see Figure 16-(a) and Figure 17-(a)), XTRON is not superior to the other approaches. As mentioned above, XTRON translate the path expression into intervals using the reverse arithmetic encoder. Thus, the parsing time and translation time of XTRON are relatively greater than the other approaches. Therefore, total evaluation times of XTRON for small sized XML data is a little bit slower than the other approaches.

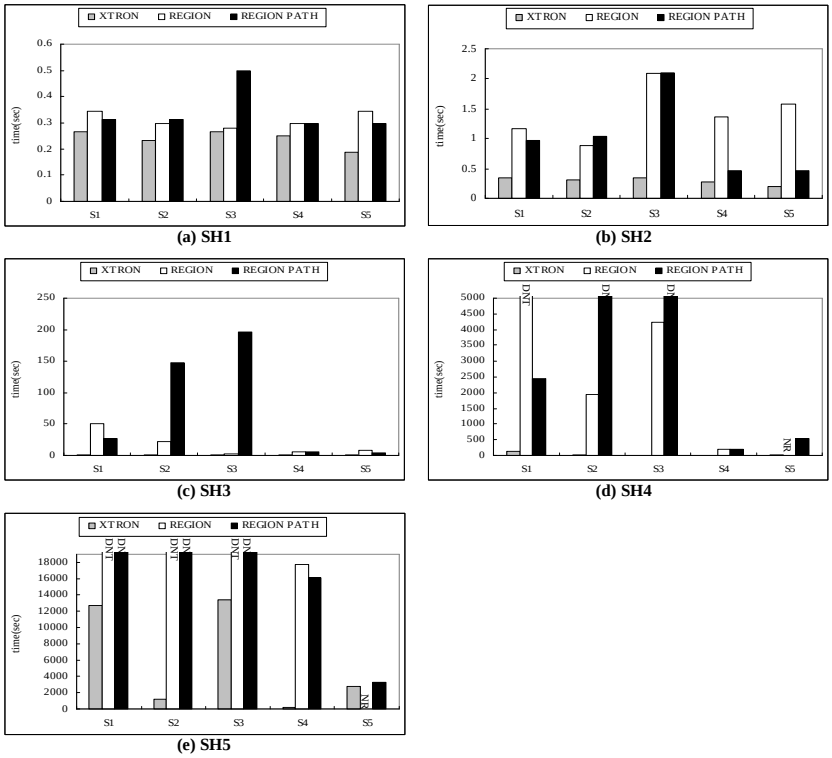


Fig. 17. Query evaluation cost for Shakespeare Data Set

For simple path expressions (i.e., type 1 queries), the region path approach is superior to the other approaches over various sized XML data. A simple path expression is mapped to a unique path identifier in the path table. Thus, selection with a single value (i.e., a unique path identifier) is performed in the region path approach.

In contrast to the evaluation results of simple path expressions, the region path approach shows the worst performance for partial matching path expressions (i.e., type 2 queries) due to joins of the path table and the element table.

Now, we compare XTRON with well-known XML processing systems: Galax and Berkeley DB XML. In our experiment, some queries do not run due to memory-full or internal exceptions. We denote these cases as NR(Not Run).

In addition, Galax and Berkeley DB XML do not handle large sized XML data (i.e., XM5 and SH5). Thus, we do not show the query performance for large sized XML data. The query performance of XTRON for large sized XML data is shown in Figure 16-(e) and Figure 17-(e). As shown in Figure 16-(e) and Figure 17-(e), XTRON can generate the query results on large sized XML data, although some queries do not terminate within five hours.

As described earlier, XTRON uses the reverse arithmetic encoding method. Thus, as shown in Figure 18 and Figure 19, when XML data is small, XTRON

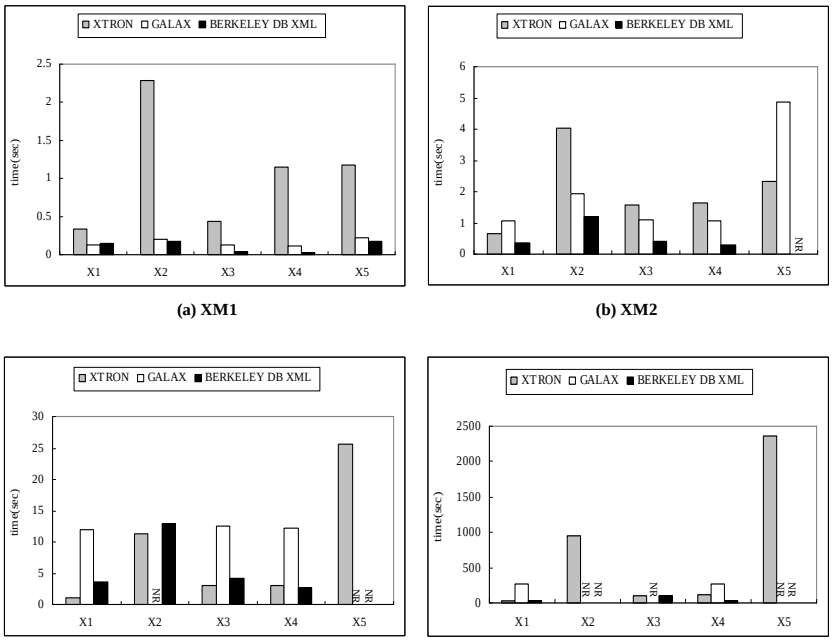


Fig. 18. Query evaluation cost with Galax and Berkeley DB XML for XMark Data Set

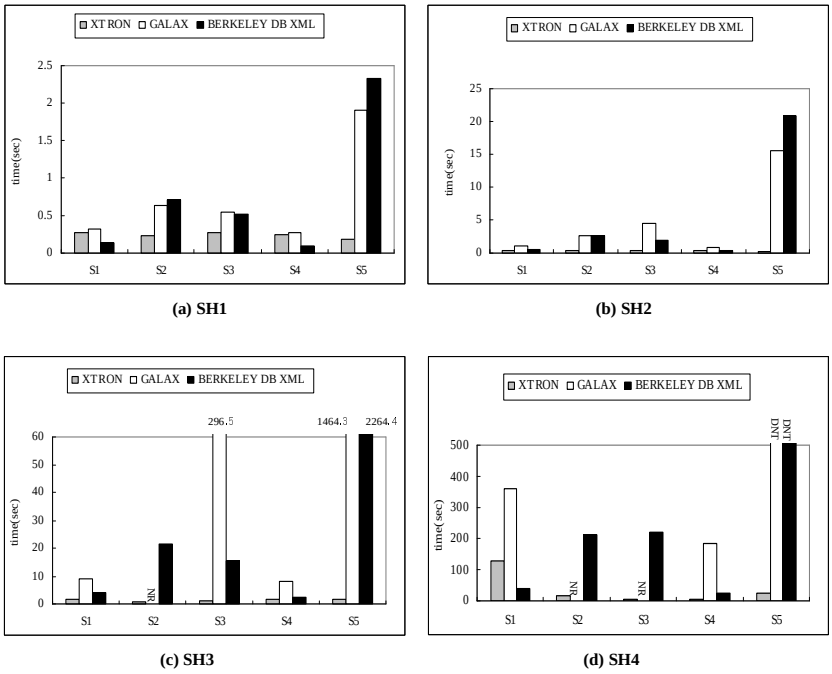


Fig. 19. Query evaluation with Galax and Berkeley DB XML for Shakespeare Data Set

does not show efficient performance compared with Galax and Berkeley XML DB due to the query translation time. In contrast, as the size of XML data increases, XTRON shows good performance. XTRON does not modify the relational engine. In contrast, Berkeley DB XML modifies the relational engine

to support XML. Nevertheless, the performance of XTRON is superior to that of the Berkeley DB XML.

Although some queries do not terminate for five hours on XTRON, our proposed method is the most efficient over almost all cases except the cases for very small sized XML data. This is because the sequence of steps connected by child axes is compacted and represented as an interval using the reverse arithmetic encoder, and ancestor-descendant relationships are easily computed by the region based numbering.

8 Conclusion

In this paper, we propose XTRON, an XML data management system using a RDBMS, which does not incur the modification of an RDBMS.

In XTRON, the transformation technique from XML to relational data is the schema independent since the schema information (e.g., DTD and XML Schema) is utilized if it is provided. If the schema information is not available, XTRON extracts structural information efficiently and utilizes it. Since XTRON utilizes the reverse arithmetic encoding method to represent the label path of each element and adopts a hybrid approach of edge and region numbering techniques, path expressions are efficiently evaluated. Also, XTRON supports a comprehensive subset of XQuery expressions including nested FLWR expressions, wildcards, order-based predicates and so on. In XTRON, an XQuery expression is translated into a single SQL statement without the need for an escape to a general-purpose programming language.

In addition, we implemented XTRON and conducted an extensive experimental study with both real-life and synthetic data sets. The experimental results show that XTRON outperforms the other approaches and well-known XML data processing systems in most cases.

Currently, we are implementing an OWL management system on XTRON since OWL is an XML data for the semantic web and we are building a transformation tool from OQL to XQuery, where OQL is a query language for OWL.

XTRON does not support update operations since the update syntax for XML data is not announced yet. Thus, as future work, we plan to construct an XML update syntax and extend XTRON to support update operations. Also, we plan to extend the query facilities of XTRON to support a wider class of XQuery (including set operators, and some XQuery functions).

References

[1] J. Banerjee, W. Kim, S.-J. Kim, and J. F. Garza. Clustering a DAG for CAD Databases. *TSE*, 14(11):1684–1699, 1988.

[2] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon. XQuery 1.0: An XML Query Language. W3C Working Draft, http://www.w3.org/TR/2002/WD-xquery-20020816, 2 May 2003.

[3] P. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, June 2006.

[4] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible Markup Language (XML) 1.0. W3C Recommendation, http://www.w3.org/TR/REC-xml, 1998.

[5] M. J. Carey, D. Florescu, Z. G. Ives, Y. Lu, J. Shanmugasundaram, E. J. Shekita, and S. N. Subramanian. Xperanto: Publishing object-relational data as xml. In *Proceedings of the Third International Workshop on the Web and Databases(WebDB)*, pages 105–110, May 2000.

[6] Y. Chen, S. B. Davidson, and Y. Zheng. BLAS: An Efficient XPath Processing System. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, June 2004.

[7] C. Clarke, G. Cormack, and F. Burkowski. An Algebra for Structured Text Search and a Framework for its Implementation. *The Computer Journal*, 38(1), 1995.

[8] E. Cohen, H. Kaplan, and T. Milo. Labeling Dynamic XML Trees. In *Proceedings of PODS 2002*, pages 271–281, 2002.

[9] R. Cover. The XML Cover Pages. http://www.oasis-open.org/cover/xml.html, 2001.

[10] D. DeHaan, D. Toman, M. P. Concens, and M. T. Ozsu. A comprehensive xquery to sql translation using dynamic interval encoding. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 623–634, June 2003.

[11] A. Deutsch, M. F. Fernandez, and D. Suciu. Storing Semistructured Data with STORED. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, pages 431–442, June 1999.

[12] D. C. Falliside. XML Schema Part 0. W3C Recommendation, http://www.w3.org/TR/xmlschema-0, 2 May 2001.

[13] W. Fany, J. X. Yuz, H. Lu, J. Luz, and R. Rastogi. Query Translation from XPath to SQL in the Presence of Recursive DTDs. In *Proceedings of 31th International Conference on Very Large Data Bases*, 2005.

[14] M. F. Fernandez, D. Florescu, J. Kang, A. Y. Levy, and D. Suciu. Overview of Strudel - A Web-Site Management System. *Networking and Information Systems*, 1(1):115–140, 1998.

[15] M. F. Fernandez, Y. Kadiyska, D. Suciu, A. Morishima, and W. C. Tan. SilkRoute: A framework for publishing relational data in XML. *ACM Tansactions on Database Systems(TODS)*, 27(4):438–493, December 2002.

[16] M. F. Fernandez, W. C. Tan, and D. Suciu. SilkRoute: trading between relations and XML. *WWW9/Computer Networks*, 33(1-6):723–745, June 2000.

[17] D. Florescu and D. Kossman. Storing and Querying XML Data using an RDMBS. *IEEE Data Engineering Bulletin*, 22(3):27–34, September 1999.

[18] R. Goldman and J. Widom. DataGuides: Enable Query Formulation and Optimization in Semistructured DataBases. In *Proceedings of 23rd International Conference on Very Large Data Bases*, pages 436–445, August 1997.

[19] T. Grust, M. van Keulen, and J. Teubner. Accelerating XPath evaluation in any RDBMS. *ACM Transactions on Database Systems.*, 29(1):91–131, March 2004b.

[20] P. J. Harding, Q. Li, and B. Moon. Xiss/r: Xml indexing and storage system using rdbms. In *Proceedings of the 29th International Conference on Very Large Data Bases*, pages 1073–1076, August 2003.

[21] R. Krishnamurthy, V. T. Chakaravarthy, R. Kaushik, and J. F. Naughton. Recursive xml schemas, recursive xml queries, and relational storage:xml-to-sql query translation. In *Proceedings of the 20th International Conference on Data Engineering*, pages 42–53, March 2004.

[22] C. Li and T. W. Ling. QED: A Novel Quaternary Encoding to Completely Avoid Re-labeling in XML Updates. In *Proceedings of ACM CIKM 2005*, pages 501–508, 2005.

[23] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *Proceedings of 27th International Conference on Very Large Data Bases*, pages 361–370, September 2001.

[24] I. Manolescu, D. Florescu, and D. Kossmann. Answering XML Queries on Heterogeneous Data Sources. In *Proceedings of 27th International Conference on Very Large Data Bases*, pages 241–250, September 2001.

[25] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A Database Management System for Semistructured Data. *SIGMOD Record*, 26(3):54–66, 1997.

[26] J.-K. Min, M.-J. Park, and C.-W. Chung. XPRESS : A Queriable Compression for XML Data. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 122–133, June 2003.

[27] P. O'Neil, E. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. ORDPATHs: Insert-Friendly XML Node Labels. In *Proceedings of ACM SIGMOD 2004*, pages 903–4908, 2004.

[28] S. Pal, I. Cseri, O. Seeliger, M. Rys, G. Schaller, W. Yu, D. Tomic, A. Baras, B. Berg, D. Churin, and E. Kogan. XQuery Implementation in a Relational Database System. In *Proceedings of 31th International Conference on Very Large Data Bases*, 2005.

[29] A. Salminen and F. Tompa. PAT expressions: an algebra for text search. *Acta Linguistica Hungarica*, 41(1-4), 1992.

[30] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 974–985, August 2002.

[31] J. Shanmugasundaram, E. J. Shekita, R. Barr, M. J. Carey, B. G. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently Publishing Relational Data as XML Documents. In *Proceedings of 26th International Conference on Very Large Data Bases*, pages 65–76, September 2000.

[32] J. Shanmugasundaram, E. J. Shekita, R. Barr, M. J. Carey, B. G. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently publishing relational data as XML documents. *VLDB Journal*, 10(2-3):133–154, September 2001.

[33] J. Shanmugasundaram, E. J. Shekita, J. Kiernan, R. Krishnamurthy, S. Viglas, J. F. Naughton, and I. Tatarinov. A General Techniques for Querying XML Documents using a Relational Database System. *SIGMOD Record*, 30(3):20–26, 2001.

[34] J. Shanmugasundaram, E. J. Shekita, J. Kiernan, R. Krishnamurthy, S. Viglas, J. F. Naughton, and I. Tatarinov. A General Techniques for Querying XML Documents using a Relational Database System. *ACM Record*, 30(3), September 2001.

[35] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *Proceedings of 25th International Conference on Very Large Data Bases*, 1999.

[36] T. Shimura, M. Yoshikawa, and S. Uemura. Storing and Retrieval of XML Documents using Object-Relational Databases. In *Proceedings of 10th International Conference, DEXA*, pages 206–217, August 1999.

[37] D. Srivastava, S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, and Y. Wu. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proceedings of 18th International Conference on Database Engineering*, Febrary 2002.

[38] I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld:. Updating XML. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, June 2001.

[39] I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and Querying Ordered XML Using a Relational Database System. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 204–215, June 2002.

[40] 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 Transactions on Internet Technology (TOIT).*, 1(1):110–141, 2001.

[41] C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman. On Supporting Containment Queries in Relational Database Management Systems. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, May 2001.