s-XML: An efficient mapping scheme to bridge XML and relational database

Samini Subramaniam, Su-Cheng Haw *, Poo Kuan Hoong

Faculty of Information Technology, Multimedia University, 63100 Cyberjaya, Malaysia

Article info

Article history:
Received 27 April 2011
Received in revised form 4 November 2011
Accepted 5 November 2011
Available online 11 November 2011

Keywords:
Labeling scheme
Query processing
Mapping scheme
XML instance storage
Relational database

Abstract

XML has recently emerged as the leading medium for data storage and data transfer over the World Wide Web due to its adaptable structure and flexibility in defining the tags. Many organizations had adopted XML as the principal facet in their online business applications. On the other hand, relational database is still widely used as the back-end database in most organizations. The diversity of these models need to be taken into account to ensure transparent and seamless integration. In this paper, we propose s-XML, an effective mapping scheme to bridge XML and relational database. Experimental results indicate that (1) s-XML is robust in terms of database storage and data loading; (2) s-XML processes query efficiently for complex chain and twig queries; and (3) s-XML is able to support large and skew-structured dataset as compared to relational DTD, Attribute and Edge approaches.

1. Introduction

eXtensible Markup Language (XML) has become one of the most advanced and sophisticated technologies to convey, communicate and exchange data over the World Wide Web (WWW) [1]. In conjunction with the high implementation of XML in present days, the efficacy of XML processing needs to be improved to provide faster response to the users.

There are three ways of storing XML data [2–5]. First, storing XML data in repositories designed for semi-structured data. Second, storing the XML data in object-oriented database and finally, storing the data in relations via relational database. In this paper, we focus on storing and querying XML data based on Relational Database Management Systems (RDBMS). Storing XML data into the relational tables require the table schemas to be designed in such a way that maintains the original data in the XML document and there is no occurrence of lossless information during data retrieval from the database.

It is essential for a mapping scheme to preserve the relationship among the XML nodes so that query retrieval will be done accurately and the responses given to the user must be as precise as possible. There are two types of user queries which the mapping scheme needs to handle, i.e., the full-text query and the structural query [6]. The details of these queries will be discussed in Section 3 of the paper.

The main contribution of this paper is to propose a new mapping scheme, s-XML to bridge XML and relational database. This approach will be compared with the existing approaches which are the Edge, Attribute and relational Data Type Definition (DTD) approach in terms of storage space, time taken to map XML data into relational tables and time taken to process different types of queries with diverse combinations.

The rest of the paper is organized as follows. Section 2 illustrates some review on the existing mapping schemes. Section 3 describes the new mapping scheme, s-XML. Section 4 provides the correctness analysis of the s-XML. Section 5 explains the experimental design. Section 6 provides the experimental results and discussions. Finally, Section 7 concludes the paper.

2. Related work

Fig. 1 represents the XML document which will be used as an example throughout the paper.

Edge approach. Edge approach [3] is the most simplest and straightforward approach which stores the entire XML document into a single relational table. This approach uses the edge-labeling for the XML tree and labels the nodes using the breadth first traversal. Fig. 2 shows the edge-labeling method used in the Edge approach.

In the edge-labeling, XML nodes are labeled using integers and the edges are labeled using the element names which appear in the XML document. This is the most commonly used mapping scheme in the early era. Table 1 describes the schema defined in the Edge approach. Based on the example depicted in Fig. 1, Table 2 shows the partial of data stored in the Edge table.

Nevertheless, the Edge approach is only pertinent for simple XML document devoid of complex combinations because the table may suffer from ‘excessive table size’ error. This is due to the fact...
that it stores the entire XML document into a single Edge table. Consider query, Q1 to retrieve the title of the book where the publisher is Dynamic Enterprise in XPath [7] notation.

\[
\text{Q1: } /\text{library/book/}[\text{title}]/\text{publisher} = 'Dynamic Enterprise'
\]

In order to retrieve the answer, excessive self-joins are required as it stores all the data in a single table. The self-join could be implemented based on the following property: A node, \(X\), can be joined to another node, \(Y\), if and only if \(X\).TargetID = \(Y\).SourceID' [8].

The query access plan starts with the predicate selection: tracing the sourceID of publisher with the value equal to 'Dynamic Labeling'. In this case, sourceID with '9' will be returned. Next, the row occurrence where it consists of targetID '9' will be retrieved. Similarly, the row occurrence where targetID is '2' will be retrieved. This process continues until it reaches the root of the query, which in this case is the 'library'. As such, Q1 will be transformed to Structure Query Language (SQL) as below:

```sql
Select tte.data
From Edge lib, Edge bk, Edge pub, Edge tte
Where lib.name = 'library' and bk.name = 'book'
And pub.name = 'publisher' and tte.name = 'title'
And lib.sourceID = 0 and lib.targetID = bk.sourceID
And bk.targetID = pub.sourceID
And bk.targetID = tte.sourceID
And pub.targetID = pub.sourceID
And tte.targetID = tte.sourceID
And pub.data = 'Dynamic Enterprise'
```

that it stores the entire XML document into a single Edge table. Consider query, Q1 to retrieve the title of the book where the publisher is Dynamic Enterprise in XPath [7] notation.

\[
\text{Q1: } /\text{library/book/}[\text{title}]/\text{publisher} = 'Dynamic Enterprise'
\]

In order to retrieve the answer, excessive self-joins are required as it stores all the data in a single table. The self-join could be implemented based on the following property: A node, \(X\), can be joined to another node, \(Y\), if and only if \(X\).TargetID = \(Y\).SourceID' [8].

The query access plan starts with the predicate selection: tracing the sourceID of publisher with the value equal to 'Dynamic Labeling'. In this case, sourceID with '9' will be returned. Next, the row occurrence where it consists of targetID '9' will be retrieved. Similarly, the row occurrence where targetID is '2' will be retrieved. This process continues until it reaches the root of the query, which in this case is the 'library'. As such, Q1 will be transformed to Structure Query Language (SQL) as below:

```sql
Select tte.data
From Edge lib, Edge bk, Edge pub, Edge tte
Where lib.name = 'library' and bk.name = 'book'
And pub.name = 'publisher' and tte.name = 'title'
And lib.sourceID = 0 and lib.targetID = bk.sourceID
And bk.targetID = pub.sourceID
And bk.targetID = tte.sourceID
And pub.targetID = pub.sourceID
And tte.targetID = tte.sourceID
And pub.data = 'Dynamic Enterprise'
```

<table>
<thead>
<tr>
<th>No.</th>
<th>Column name</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SourceID</td>
<td>Stores the ID of the source node</td>
</tr>
<tr>
<td>2</td>
<td>TargetID</td>
<td>Stores the ID of the destination node</td>
</tr>
<tr>
<td>3</td>
<td>Ordinal</td>
<td>Stores the relative position of a node among its siblings</td>
</tr>
<tr>
<td>4</td>
<td>Name</td>
<td>Stores the element name of a node respectively.</td>
</tr>
<tr>
<td>5</td>
<td>Flag</td>
<td>Indicates if a node is pointing to another node (denoted as REF) or is a leaf (denoted as VALUE)</td>
</tr>
<tr>
<td>6</td>
<td>Data</td>
<td>Stores value if any otherwise NULL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>No.</th>
<th>Column name</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SourceID</td>
<td>Stores the ID of the source node</td>
</tr>
<tr>
<td>2</td>
<td>TargetID</td>
<td>Stores the ID of the destination node</td>
</tr>
<tr>
<td>3</td>
<td>Ordinal</td>
<td>Stores the relative position of a node among its siblings</td>
</tr>
<tr>
<td>4</td>
<td>Name</td>
<td>Stores the element name of a node respectively.</td>
</tr>
<tr>
<td>5</td>
<td>Flag</td>
<td>Indicates if a node is pointing to another node (denoted as REF) or is a leaf (denoted as VALUE)</td>
</tr>
<tr>
<td>6</td>
<td>Data</td>
<td>Stores value if any otherwise NULL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>No.</th>
<th>SourceID</th>
<th>TargetID</th>
<th>Ordinal</th>
<th>Name</th>
<th>Flag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Library</td>
<td>REF</td>
<td>NULL</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>Book</td>
<td>REF</td>
<td>NULL</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>Book</td>
<td>REF</td>
<td>NULL</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
<td>2</td>
<td>Year</td>
<td>REF</td>
<td>NULL</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>5</td>
<td>2</td>
<td>Publisher</td>
<td>REF</td>
<td>NULL</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>Title</td>
<td>REF</td>
<td>NULL</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>2</td>
<td>2</td>
<td>Year</td>
<td>REF</td>
<td>NULL</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Title</td>
<td>VALUE</td>
<td>XML Mapping</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Year</td>
<td>VALUE</td>
<td>2010</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Publisher</td>
<td>VALUE</td>
<td>Dynamic Labeling</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Title</td>
<td>VALUE</td>
<td>XML Labeling</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Year</td>
<td>VALUE</td>
<td>2010</td>
</tr>
</tbody>
</table>
Since self-join is the most expensive exercise from RDBMS perspective, this certainly cause overhead in resources for complex query processing.

**Attribute approach.** In contrast with the Edge approach, the Attribute [3] mapping scheme stores the XML data into relational tables based on the element names that appear in the XML document. Attribute approach creates as much table as distinct element name that appear in the document. Table 3 describes the schema defined in the Attribute approach. Table 4 shows the mapping results of the sample XML document using the Attribute approach.

Compared to Edge approach, the Attribute approach shreds XML document in many tables. This is one of the drawbacks of the Attribute approach where the number of tables depends on distinct element names in the XML document. Imagine a complex XML document with various element names. This will lead to excessive creations of tables which consumes resources and machine memory. Consider the same query defined earlier, Q1. Based on the Attribute approach, the SQL is as below:

```
Select tte.data.
From publisher pub, title tte, book bk, library lib.
Where lib.targetID = bk.sourceID.
And bk.targetID = tte.sourceID.
And bk.targetID = pub.sourceID.
And pub.data = 'Dynamic Enterprise'
```

Although from the RDBMS perspective, Attribute approach produces lesser joins as compared to Edge approach, the processing of complex queries with assorted combinations will delay query retrieval time because Attribute approach produces joins within multiple tables.

**Table 3**

<table>
<thead>
<tr>
<th>No.</th>
<th>Column</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Aname</td>
<td>Aname indicates the element name that appears in XML document</td>
</tr>
<tr>
<td>2</td>
<td>SourceID</td>
<td>Stores the ID of the source node</td>
</tr>
<tr>
<td>3</td>
<td>TargetID</td>
<td>Stores the ID of the destination node</td>
</tr>
<tr>
<td>4</td>
<td>Ordinal</td>
<td>Stores the relative position of a node among its siblings</td>
</tr>
<tr>
<td>5</td>
<td>Flag</td>
<td>Indicates if a node is pointing to another node (denoted as REF) or leaf node (denoted by VALUE)</td>
</tr>
<tr>
<td>6</td>
<td>Data</td>
<td>Stores value if any otherwise NULL</td>
</tr>
</tbody>
</table>

**Table 4**

<table>
<thead>
<tr>
<th>SourceID</th>
<th>TargetID</th>
<th>Ordinal</th>
<th>Flag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>library</td>
<td>1</td>
<td>1</td>
<td>REF</td>
<td>NULL</td>
</tr>
<tr>
<td>book</td>
<td>2</td>
<td>1</td>
<td>REF</td>
<td>NULL</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2</td>
<td>REF</td>
<td>NULL</td>
</tr>
<tr>
<td>publisher</td>
<td>2</td>
<td>9</td>
<td>REF</td>
<td>Dynamic Labeling</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>0</td>
<td>VALUE</td>
<td>NULL</td>
</tr>
<tr>
<td>title</td>
<td>2</td>
<td>5</td>
<td>REF</td>
<td>NULL</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>10</td>
<td>REF</td>
<td>NULL</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>0</td>
<td>VALUE</td>
<td>XML Mapping</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>0</td>
<td>VALUE</td>
<td>XML Labeling</td>
</tr>
<tr>
<td>year</td>
<td>2</td>
<td>6</td>
<td>REF</td>
<td>NULL</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>11</td>
<td>REF</td>
<td>NULL</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0</td>
<td>VALUE</td>
<td>2010</td>
</tr>
</tbody>
</table>

**Relational DTD.** This mapping scheme requires the DTD [9] to generate table schema and shred the XML document into relational database based on the following properties:

a. Elements with in-degree=0 will be stored in separate relation.
b. Elements below '*' and '+' will be stored in separate relation to cater for set-valued child.
c. Nodes with in-degree=1 will be inlined.
d. Any elements with recursive edges will be stored in separate relation.
e. Elements with in-degree >1 will be stored in separate relation.

The relational DTD approach maps XML data based on the frequency of the element occurrence in an XML document. The elimination of less important elements and grouping of elements based on incidence allows lesser space consumed, straightforward table schema and efficient mapping to tables. However, this approach can only be used if the DTD of an XML document is available. The following example shows XPath query to retrieve title of all books where publisher name is ‘Dynamic Enterprise’ and the corresponding SQL query.

**Q2:/**library/**//title/**publisher = 'Dynamic Enterprise'

Select lib.title from library lib,book bk,publisher publ where publ.publisher = 'Dynamic Enterprise' and lib.ID=bk.parentID and bk.ID=pub.parentID

**SUCCENT.** SUCCENT [10] is an efficient path-based approach to store and query XML documents which does not require the knowledge of the DTDs of the XML documents being stored. The schemes are shown below.

a. **Document** (DocID, Name) – stores the name of XML document in database.
b. **Path** (PathID, PathExp) – PathExp refers to the path from the current leaf node to the root node which uses the ‘.’ separator to differentiate the element names. These paths are uniquely identified using PathID.
c. **PathValue** (DocID, PathID, LeafOrder, SiblingOrder, LeftSibIxLevel, LeafValue) – LeafValue refers the position of a leaf node in the XML document, SiblingOrder is use to store sibling information, LeftSibIxLevel known as Left Sibling Intersection Level which refers to the level of the highest common ancestor of a node and finally LeafValue stores the value of a leaf node.
d. **AncestorInfo** (DocID, SiblingOrder, AncestorOrder, AncestorLevel) – stores ancestor information of a leaf node.
e. **TextContent** (DocId, LinkId, Text) – stores the value of a leaf node

Since SUCCENT only stores the path to the leaf nodes, query retrieval will be faster especially for full-text query. However, reconstruction of an XML tree from the data stored in the relational tables could be inefficient. This is because the internal nodes are stored within the path expression of a leaf node. Thus, the path of each leaf node needs to be parsed to acquire the information of the internal nodes to decide on the position of the nodes in the XML document. Therefore, this process is time consuming and certainly causes inefficient XML tree construction.

**XRel.** This approach decomposes an XML document and map the information into the relational database based on the types of nodes [11]. XRel maintains (1) simple path expression and (2)
region information of a node in an XML tree. Simple path expression refers to the path of a node from the root node. Region information is used for the management of containment relationships among the nodes. Region is a pair of start and end position of a node in an XML tree which is determined using depth-first search. Node $k_r$ is reachable from node $k_b$ if node $k_b$ is within the region of node $k_r$, i.e., the Start position of node $k_r$ < start position of node $k_b$ and the End position of node $k_r$ < end position of node $k_b$. Containment relationship maintains the ancestor–descendant relationship among the nodes. This is an added advantage of this approach because the relationships among nodes can be easily determined using the $\theta$-joins. Unfortunately, $\theta$-joins are more costly than the equijoins which degrades the performance of this approach.

**XPARENT** XPARENT [12] is an edge-based mapping scheme which contains of five relations as follows. The LabelPath to maintain path information, the DataPath to maintain parent–child information, the Element to store element information, the Data to store values in an XML document and the Ancestor to store ancestor–descendant information. The schema used in XPARENT is similar to the one used in XRel. Nevertheless, the region encoding used in XRel is replaced by single attribute, Data-path-id (Did) which improves query processing time and uses equijoins instead of $\theta$-joins for query retrieval. Query with a combination of multiple criterions can be processed efficiently due to the management of XML data in separate relations. However, the Ancestor table may consume high disk space to store the data paths and may introduce overhead due to the storage of redundant information in the table.

**ShreX** Du et al. [13] proposed ShreX (Shredding XML), a system for shredding, loading and querying XML documents in relational database. The XML-to-relation mapping is indicated through an XML schema which is efficient, simple and allows for validation to take place. The mapping is described by adding annotations to the XML schema which depicts how the nodes should be mapped to relational database. This annotation allows the combinations of other mapping strategies which can be specified in the corresponding XML schema such as Edge approach and interval encoding.

**Other mapping approaches.** Agora [14,15], Mars [16] and SilkRoute [17] are examples of mapping schemes which requires the knowledge of XML schema or DTD for XML processing. The mapping knowledge of Agora and Mars are pre-defined at initialization time where the XML documents are represented in relational way using intermediate relational meta schema which is referred as generic relational representation of XML (GreX). On the other hand, SilkRoute converts relational schema to canonical XML view that denotes relational tables and attributes in XML format. Thus, the mapping knowledge is pre-defined within the system to produce the canonical view. Amer-Yahia and Srivastava [18] and Amer-Yahia et al. [19] proposes a mapping scheme and interface to query these mappings, which they named it as My XML Mapper (MXML). The mapping scheme is parsed and kept in a repository (i.e. relational database) which will be captured by a relational schema. Then, Interface-MXML (IMXML) will be used to generate relational schema and a set of loading programs that parses an XML document and populates tables [20].

Apart from this, Clio [21] is a schema-mapping tool which generates a set of logical mappings that captures the relationships between the instances. Clio preserves the semantics of data transformation and integration efficiently. The schema mappings in Clio can be denoted as a set of tuple-generating dependencies (tgd$s$) which is a constraint between the source and the target schemas. Clio consists of three crucial phases which are tuple extraction which can be obtained from all possible instantiations of the variables, generation of XML fragments that is the transformation of flat tuple into XML fragment and merging the XML fragments. This is to remove the duplicate XML fragments and allow for sharing of the same key among fragments. On the other hand, archival information system (ARCHIS) [22] is a system that preserves the history of relational database. This system maintains $H$-document which is a XML-based hierarchical view that stores the history of a table such as creation or deletion time of a table and the child elements in the document.

In recent years, many researchers have adopted XML as the knowledge representation in ontology and software development. Thomopoulos et al. proposed utilizing XML to represent knowledge to predict “prospective queries” on ontology [23]. They demonstrated that their proposed solution works on food quality prediction domain. On the other hand, Foetsch and Pulvermueller proposed a solution to achieve higher-level transformation languages to ease the software development by applying the XML operator hierarchy concept to the transformation language such as XSLT [24].

### 3. s-XML: our proposed approach

Fig. 3 depicts the four main types of relationships that need to be maintained to aid user’s. These information need to be maintained by the mapping technique to retrieve users queries with any combinations.

User’s queries can be divided into two forms namely full-text query and structural query. Consider a full-text query which only involves keywords such as select all the hotel with the name ‘Renaissance’. Users will be probably be furnished with hundreds over answers by the search engine because there is no other criteria been stipulated as a condition. This is the most commonly used query which is simple and easy but deficient and imperfect because the responses are inexact.

In order to narrow down the responses from the search engine and to retrieve accurate responses, structural query will be the prominent choice. Consider a query which involves multiple conditions such as to retrieve information about hotels which is located in ‘Klang Valley’, more than ‘four stars’, has ‘laundry service’ and the name of the hotel starts with ‘Renaissance’. This query will definitely returns accurate and limited answers because it is more specific. In order to cater for structural queries, the relationships among nodes need to be preserved by the mapping. Nevertheless, the dilemma that has been enduring for sometime is the inability of many existing mapping techniques to support structural queries since the association among the nodes are discarded and not given any priority. As such, s-XML approach focuses on structural queries retrieval. Yet, s-XML is able to support full-text queries retrieval efficiently.

s-XML adopts the Persistent Labeling [25] scheme to annotate each node. Each node will be labeled as $(l, [n_{p}, d_{p}], [n, d])$, where:

- $l$ is the level of the node in the tree
- $n, d$ is the local label of the node where $n$ denotes the position of the node among its sibling and $d$ is assigned to 1 in static labeling. Pair of $(n, d)$ represents the n/d rationals.
- $n_{p}, d_{p}$ is the local label of the parent node. Parent label of a node is the self label of the parent node

![Fig. 3. The relationship among nodes in an XML tree: (i) parent–child; (ii) ancestor–descendant; (iii) sibling; (iv) level information.](image-url)
d. Root element will be labeled as (0,[1,1]) where 0 represents the level and (1,1) represents the local label of the node. This element does not have a parent label since the node is the origin of the XML tree.

Fig. 4 explains [n,d] and [n_p,d_p].

Fig. 5 shows an example of labeling scheme based on Persistent Labeling [25]. Algorithm 1 shows our labeling algorithm (created based on our understanding on Persistent Labeling [25]).

Algorithm 1: Labeling using persistent labeling scheme

1. **Input**: An XML tree, \( \tau \)
2. **Output**: Labeled XML nodes, \( \eta \) in \( \tau \) using persistent labeling scheme \( \{ \{ n_p,d_p \}, \{ n,d \} \} \)
3. int level = 0
4. 5. function printNode(document)
6. //level 0
7. if \( \eta_r \) = root node in \( \tau \)
8. then label for \( \eta_r(level, NULL, \{ sn = 1, d = 1 \}) \)
9. level++
10. if level is 1 and \( i \) is 1
11. for each child node of \( \eta_r, \eta_{child}(i) \) do { 
12. label for node \( \eta_{child}(i)(level, \{ n_p = sn, d_p = 1 \}, \{ n = 1, d = 1 \}) \)
13. getData(\( \eta_{child}(i), n, \) level)
14. n++
15. i++
16. }
17. endfunction
18. 19. function getChildNodes(\( \eta_{child}(i), n_pchilde, levelchild \) )
20. levelchild++
21. for each child node of \( \eta_{child}(i), \eta_{nextchild}(i) \) do {
22. label for node \( \eta_{nextchild}(i)(levelchild, \{ n_p = n_pchilde, d_p = 1 \}, \{ n = 1, d = 1 \}) \)
23. getData(\( \eta_{nextchild}(i), n, levelchild \) )
24. n++
25. i++
26. }
27. endfunction
28. 29. function getData(\( \eta_{nextchild}(i), pName, levelleaf \) )
30. levelleaf++
31. for each child node of \( \eta_{nextchild}(i), \eta_{leaf}(i) \) do {
32. label for node \( \eta_{leaf}(i)(levelleaf, \{ n_p = n_pchilde, d_p = 1 \}, \{ n = 1, d = 1 \}) \)
33. endfor
34. endfunction
35. 36. The printNode() function will be used to label the root node, \( \eta_r \) in the XML document. Since root node does not have a parent node, the parent label, \( \{ n_p,d_p \} \) will be NULL as in line 8 of the algorithm. Next, the child nodes of the root node, \( \eta_{child}(i) \) will be labeled using getChildren() function as in line 19–29. Each child node will be sent to getChildren() function to read its subsequent child nodes while increasing the level of each XML node. The leaf nodes will be labeled by calling the getData() function as shown in line 31–36 of the algorithm.

These information will be mapped into two tables which are ParentTable and ChildTable that has the following scheme respectively. ParentTable will be storing the non-leaf (internal) nodes where else ChildTable will be used to store the leaf (external) nodes.

**ParentTable(\( IdNode, pName, cName, Level, LParent, SelfLabel \) ) where:**

(a) \( IdNode \) – uniquely identify the nodes stored in the ParentTable.
(b) \( pName \) – stores parent node name.
(c) \( cName \) – maintains child name.
(d) \( Level \) – maintains level information.
(e) \( LParent \) – maintains the parent label of the node which stores the reference of the parent label (IdNode).
(f) \( SelfLabel \) – maintains the self-label or local label of the node which is \( [n,d] \) in Persistent Labeling.

**ChildTable(\( IdNode, Level, pName, SelfLabel, LParent, Value \) ) where:**

(a) \( IdNode \) – uniquely identifies the nodes stored in the ChildTable.
(b) \( Level \) – stores the level information of the node in the XML document.
(c) \( pName \) – stores the element name of the parent node.
(d) \( SelfLabel \) – maintains the self-label or local label of the node which is \( [n,d] \) in Persistent Labeling.
(e) \( LParent \) – maintains the parent label of the node which stores the reference of the parent label (IdNode) from the ParentTable.
(f) \( Value \) – stores the value of the node.

Algorithm 2 explains the mapping processes to ParentTable and ChildTable. The XML document will be traversed as similar way as shown in Algorithm 1. The labels derived from Algorithm 1 which are level \( l \), self label \( ([n,d]) \), and parent label \( ([n_p,d_p]) \) will be mapped into level, selfLabel and LParent columns respectively. Algorithm 2 also shows that LParent column will be stored with the idNode of the parent node which is shown in line 10, 15 and 22.
Table 5 and 6 are examples of s-XML mapping into the ParentTable and ChildTable respectively.

Algorithm 3 describes the query retrieval process. If a condition of a query consists of a keyword and combinations of other relationships, then the keyword has to be searched in the ChildTable.

Tables 5 and 6 are examples of s-XML mapping into the ParentTable and ChildTable respectively.

Algorithm 2: Mapping XML nodes to relational database using s-XML mapping scheme

1. function createTable()
2. long timeBefore = System.currentTimeMillis();
3. set connection to database
4. // end function
5. // insert root node and node at the first level
6. insert into parentTable
7. function
8. insert into parentTable(pName, cName, level, selfLabel, LParent) values(childNode.getNodeName(), getNodeName(1), level, [n, d], 'NULL')
9. // insert node at the first level
10. insert into parentTable(pName, cName, level, selfLabel, LParent) values(childNode.getNodeName(), getNodeName(1), childNode.getNodeName(), level, [n, d], idNode)
11. // end function
12. // insert other non-leaf nodes
13. function Child(\(p_{\text{nextchild}}\))
14. insert into parentTable(pName, cName, level, selfLabel, LParent) values(childNode.getNodeName(), getNodeName(1), cN.getNodeName(1), cN.getNodeName(1), level, [n, d], IdNode)
15. // call function to insert leaf nodes
16. LeafNode(\(p_{\text{nextchild}}\))
17. // end function
18. function LeafNode(\(p_{\text{nextchild}}\))
19. insert into childTable(level, selfLabel, LParent, value) values
20. \((\text{level, [n, d], IdNode, cData.getNodeValue()})\)
21. // end function
22. long timeAfter = System.currentTimeMillis();
23. Time taken to map = timeAfter – timeBefore
24. // end function

Fig. 6 illustrates the retrieval process (based on Algorithm 3) on the four main relationships identified earlier.

This approach allows the relationships among the nodes to be determined easily. For instance, consider the Q1 described earlier. In order to retrieve the answer, firstly, the ChildTable needs to be traced to retrieve the LParent of the intended node (title) which is stored as idNode in the ParentTable. Next, LParent of the cName (title) in ParentTable will be retrieved and compared to the LParent of the given node (publisher) in the ParentTable. Finally, the LParent of both the nodes (title and publisher) must be the same in the ParentTable to prove that these nodes are sibling nodes. As such, the corresponding SQL is as below:

```
Select value from childTable where LParent = (select idNode from parentTable where cName like 'title' and LParent = (select distinct LParent where idNode = (select LParent from childTable where value like 'Dynamic Enterprise')));
```

Fig. 6. Relationship supported by s-XML.
4. Correctness analysis of s-XML

**Definition 1.** Let \( \eta_r \) be the root of the XML document which has child nodes, \( \eta_{\text{child}(i)} \) under it. The mapping process starts from the root of the document and traversed using the breadth-first traversal.

**Definition 2.** Once the first level of the node is traversed, the second layer of the XML document which consists of the child nodes, \( \eta_{\text{nextchild}(i)} \) of node \( \eta_{\text{child}(i)} \) and the descendants of \( \eta_r \) will be read and mapped to the tables.

**Definition 3.** All the subsequent layers will be traversed to store the non-leaf nodes before storing the leaf nodes, \( \eta_{\text{leaf}(i)} \).

**Lemma 1.** If the node traversed is not a leaf node or the last node in the XML document, the node will be stored in the ParentTable as the internal nodes of the XML document.

**Proof.** The intended XML document will be parsed so that the elements in the document can be read and accessed as shown. Function \( \text{printNode}() \) will be called to read the root node of the document, \( \eta_r \) as in line 5 of Algorithm 1 and mapped to the ParentTable as in line 8 of Algorithm 2 using rootC() function. The root node will be retrieved using the \( \text{getDocumentElement}() \) which is an object of the Document class in DOM. Subsequently, the first level of the XML document, which is the child nodes, \( \eta_{\text{child}(i)} \) will be read as shown in line 11–12 of Algorithm 1 and the information will be mapped to the ParentTable in the function rootC() as line 10 of Algorithm 2. After that, child nodes of node \( \eta_{\text{child}(i)}, \eta_{\text{nextchild}(i)} \) will be traversed until the leaf nodes is achieved as in line 22–24 of Algorithm 1 using the function getChildNodes(). Then, the nodes will be mapped to the ParentTable as in Algorithm 2 in line 15–16. Therefore, all the non-leaf nodes will be traversed and mapped into the ParentTable as in line 14–16 where the function Child() will be looped until the leaf node is achieved.

**Lemma 2.** The leaf nodes in the XML document will be mapped to the ChildTable once all the internal nodes have been traversed and mapped into the ParentTable.

**Proof.** The \( \text{getData()} \) function will be invoked as in line 31 of Algorithm 1 where the leaf nodes will be traversed and labeled using persistent labeling from line 31–36 of Algorithm 1. The information will be mapped to the ChildTable and the information will be stored in their respective columns as shown in line 21–23 of Algorithm 2.

**Theorem 1.** Given an XML document with multiple layers, the nodes will be traversed level by level and mapped to their respective tables. While traversing the document, each node will be labeled using the persistent labeling scheme before they are mapped.

**Proof.** The root node will be labeled and mapped to the ParentTable and subsequently followed by the other non-leaf nodes as in Lemma 1. In Lemma 2, all the leaf nodes will be labeled using Algorithm 1 and mapped to the ChildTable. Thus, the XML document traversing and mapping is divided into two broad classifications which are the internal node mapping to the ParentTable and leaf nodes mapping to the ChildTable.

5. Experimental design

Sequence of executions was performed to evaluate the efficiency of the proposed method as compared to the existing approaches, which are Edge, Attribute, and relational DTD approach. The evaluation was done to measure the time taken to load XML data into relational database, storage space after shredding process and time taken to execute queries from the relational tables.
5.1. Description of the experimental datasets

Two datasets namely the DBLP and Protein were used to facilitate the execution processes which were gathered from Washington UW Repository [26]. The snapshot of the XML document of the DBLP and Protein datasets are shown in Figs. 7 and 8 respectively.

(a) DBLP dataset
DBLP dataset has a structured XML document where the XML document is balanced. This reduces the complexity of the XML document.

(b) Protein dataset
Protein dataset is a skewed structured document which grows deeper and has massive child nodes which is appended as a child node to the parent node that increases the level of the XML document. Certainly, skewed structured document is more complex than the structured XML document and requires more time to read and map the XML nodes.

Table 7 shows the summary of the datasets used in this experiment.

5.2. Description of the queries

Five queries ranging from simple to complex queries were identified to be evaluated. These benchmark queries were selected in accordance to the popular benchmark tool, ‘The Michigan Benchmark’ [27]. Simple query identifies the tuples that satisfy a given predicate over its attributes or elements. Join query involves connecting nodes that need to be structurally related. Query on an aggregation hierarchy is unique and differs from general query on association relationships. Complex chain query defines query on one single element at a time while complex twig query defines query on two or more elements. In other words, chain query consists of only one leaf node while twig query has two or more leaf nodes. Thus, they are also known as simple path expression and branching path expression respectively. Fig. 9 illustrates the XML query, XPath notation and the corresponding SQL query for each group of queries.

From Fig. 9, as the query become more complex, the number of joins required increases based on the complexity. As
mentioned in Section 3 earlier, support for complex (ad-hoc) queries has been very crucial especially over the web to provide quick results.

5.3. Experimental setup

We used Acer Intel Pentium dual-core processor T2390 with 160 GB HDD and 1 GB DDR2. The execution was performed using IntelliJ IDEA Community Edition 9.0.1 using JDK 1.5.0. MySQL was employed as the database server and Java was used as the programming language. The time recorded was based on the average of five consecutive executions.

Table 8
Query description for DBLP dataset.

<table>
<thead>
<tr>
<th>Query no.</th>
<th>Query description</th>
<th>XPath notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>QT1</td>
<td>Retrieve all the information for the books with the title start with 'Advanced topic'</td>
<td>/dblp/book[title='Advanced Topic']</td>
</tr>
<tr>
<td>QT2</td>
<td>List all the authors who have produced some materials in the year '1992'</td>
<td>/dblp/j[year = 1992]/author</td>
</tr>
<tr>
<td>QT3</td>
<td>Retrieve the number of books or proceedings written by different authors</td>
<td>count (/book/author)</td>
</tr>
<tr>
<td>QT5</td>
<td>List number of pages, title and the year of proceedings for all the 'inproceedings' publications</td>
<td>/dblp/inproceedings/pages</td>
</tr>
</tbody>
</table>

Table 9
Query retrieval time on DBLP dataset for each approach.

<table>
<thead>
<tr>
<th>Query</th>
<th>Edge (ms)</th>
<th>Attribute (ms)</th>
<th>DTD (ms)</th>
<th>s-XML (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>QT1</td>
<td>1731</td>
<td>832</td>
<td>751</td>
<td>932</td>
</tr>
<tr>
<td>QT2</td>
<td>3951</td>
<td>1735</td>
<td>1621</td>
<td>1752</td>
</tr>
<tr>
<td>QT3</td>
<td>2040</td>
<td>1000</td>
<td>961</td>
<td>1752</td>
</tr>
<tr>
<td>QT4</td>
<td>7642392</td>
<td>23857</td>
<td>5284</td>
<td>5204</td>
</tr>
<tr>
<td>QT5</td>
<td>6129391</td>
<td>22572</td>
<td>7323</td>
<td>6021.8</td>
</tr>
</tbody>
</table>

Fig. 11. Storage space consumption on DBLP and Protein dataset.

Fig. 12. Query on DBLP dataset based on the Edge, attribute, relational DTD and s-XML approaches.
6. Performance evaluations and discussions

6.1. XML data mapping into relational database

The first experiment was to measure the time taken to map XML data into relational database and the result is shown in Fig. 10. There are three steps involved in the mapping process which are database creation, table creation and loading of the XML data. The relational DTD and Attribute approaches took more time to load XML data compared to s-XML because the table creation and data loading imply multiple tables which depends on the rate of recurrence of the elements and the discrete element names in the XML document respectively. s-XML performed the best for the reason that it only involves two tables to maintain the XML data efficiently.

6.2. RDBMS storage size

The second experiment was conducted to evaluate the storage space consumed by Edge, Attribute, relational DTD and s-XML approaches after XML data were mapped into their respective tables. Fig. 11 shows the results of the evaluation.

Edge approach utilized lesser space in view of the fact that the entire document is stored in a single table. s-XML consumes adequate space because only two tables were created to stored the XML document. Attribute and relational DTD approaches create tables for distinct element names that appear in XML document and based on the frequency of the elements that appear on the XML document respectively.

6.3. Query processing

6.3.1. Query on DBLP dataset

The first experiment was conducted on DBLP dataset with 130 MB (see Table 8 for query description). Fig. 12 shows the SQL commands to retrieve each query based on the mapping strategies. Table 9 records the experimental results.

Table 10

<table>
<thead>
<tr>
<th>Query no.</th>
<th>Query description</th>
<th>XPath notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>QT1</td>
<td>List different feature-types that appear in the dataset</td>
<td><code>/ProteinDatabase/ProteinEntry/feature/feature-type</code></td>
</tr>
<tr>
<td>QT2</td>
<td>List all the genome in genetics where intron-status is ‘incomplete’</td>
<td><code>/ProteinDatabase/ProteinEntry/genetics [intron-status= incomplete]/genome</code></td>
</tr>
<tr>
<td>QT3</td>
<td>Count the number of information contained in the database which contains the text ‘binding-site’</td>
<td><code>count (text () = ‘binding-site’)</code></td>
</tr>
<tr>
<td>QT4</td>
<td>List all the map-position,mobile-element,genetic-origin and genetic-code for ‘mitochondrion’ genome type</td>
<td>`/ProteinDatabase/ProteinEntry/map-position</td>
</tr>
<tr>
<td>QT5</td>
<td>List the status,sequence specification and note in account info where molecular type is ‘DNA’</td>
<td>`ProteinDatabase/ProteinEntry/reference/accinfo/status</td>
</tr>
</tbody>
</table>

Table 11

<table>
<thead>
<tr>
<th>Query</th>
<th>Edge (ms)</th>
<th>Attribute (ms)</th>
<th>DTD (ms)</th>
<th>s-XML (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>QT1</td>
<td>1048421</td>
<td>834196</td>
<td>2913</td>
<td>2552</td>
</tr>
<tr>
<td>QT2</td>
<td>5832004</td>
<td>472352</td>
<td>104273</td>
<td>100111</td>
</tr>
<tr>
<td>QT3</td>
<td>3040</td>
<td>3621</td>
<td>2210</td>
<td>2305</td>
</tr>
<tr>
<td>QT4</td>
<td>8473221</td>
<td>1925882</td>
<td>103842</td>
<td>9837</td>
</tr>
<tr>
<td>QT5</td>
<td>8823103</td>
<td>1947310</td>
<td>9194.5</td>
<td>8532</td>
</tr>
</tbody>
</table>

Fig. 13. Performance Comparison in terms of scalability.
From the results obtained (Table 9 and Fig. 12), we noticed the following:

- **Edge** approach took the longest time to process all types of queries due to the excessive multiple self-joins required.
- **Relational DTD** approach performed the best followed by the **Attribute** approach for simple queries (QT1–QT3).
- **s-XML** performance is the best for complex queries (QT4 and QT5).
- **Edge** and **Attribute** approaches suffer greatly for complex queries (QT4) by about 1468 times and 4 times respectively slower compared to s-XML.
- The pattern of the dataset has impact on the performance. The relational DTD approach partitioned a DTD tree into subtrees based on the elements below the '/' and '+' (both used to represent the many relationships between elements) were shredded into separate relations. As the DTD resembles a regular structured tree with approximately 8 fan-outs, the number of relations created is in controllable size. On the other hand, s-XML requires look-up on both tables to retrieve the query results. As such, relational DTD performance is better than s-XML for simple queries.

Nevertheless, for complex queries, relational DTD requires multiple joins (6 joins for QT4 and 5 joins for QT5) while in s-XML, it replaces the joins with selections.

### 6.3.2. Query on protein dataset

Second experiment was conducted on a larger dataset, i.e., Protein with 0.67 GB dataset (see Table 10 for query description) and the summarized results of the experiment are shown in Table 11. The SQL statement (see Table 12) on each approach is placed in the summarized results of the experiment are shown in Table 11.

From the results obtained on Protein dataset, we observed the following:

- The Edge and Attribute approaches took very long (the longest time is 8473221 ms; equivalent to 2.35 h) to complete the results. As such, these approaches are not scalable to support larger datasets.
- **Relational DTD** performance degrades for a larger dataset and whenever queries involving complex/assorted combinations (QT4 and QT5).
- **s-XML** performed the best in all types of queries (except QT3).
Protein dataset is a skew-structured which causes higher response time by relational DTD approach compared to s-XML. Since relational DTD solely depending on the occurrences of elements in the dataset, it performed slower for complex queries due to multiple joins required.

Unlike relational DTD, the number of tables generated in s-XML approach is fixed regardless of the frequency occurrence of the element.

6.4. Scalability performance comparisons

In this section, we evaluated the performance of the mapping schemes using the Protein dataset at different range of sizes. This experiment was conducted to measure the performance of each mapping schemes in processing complex twig query (QT5) of Protein dataset. The result of the experiment is shown in Fig. 13.

From the results, we observed that Edge and Attribute approaches took the longest time to process complex twig query at various file size. Even though relational DTD scheme was a better approach in executing simple queries in structured XML tree, its performance degrades while processing complex twig query using skewed structured document compared to s-XML mapping scheme. This is due to the fact that relational DTD mapping scheme created scores of tables which caused massive joins among the relations. s-XML mapping scheme able to process complex twig query better than the other mapping scheme regardless of the file size while maintaining its simple structure and the number of tables.

7. Conclusion

XML document requires robust and seamless mapping approach which allows for efficient and accurate data shredding into relational database. A good mapping scheme should preserve the four main hierarchical relationships, i.e., parent–child, ancestor–descendant, siblings and level of XML document into the relational database. In this paper, we proposed a new mapping scheme named s-XML to support structural queries retrieval efficiently.

The experimental evaluations revealed that (1) s-XML is robust in terms of database storage and data loading; (2) s-XML processed query efficiently for complex chain and twig queries; (3) s-XML is able to support skew-structured dataset as compared to relational DTD, Attribute and Edge approaches; and (4) s-XML is scalable to support query retrieval on large dataset effectively.

Appendix A See Table 12.

References