Xregion: A New Approach to Storing XML Data in Relational Databases

Li-Yan Yuan* and Meng Xue
University of Alberta, Canada

Abstract
In this paper, we propose a new structure-based approach, called Xregion, to storing XML data in relational databases. Our approach first partitions an XML document into several disjoint regions according to the cardinality of element nodes, and then maps these regions into separate relations. The experimental results demonstrate that the proposed approach dramatically improves the performance of queries on the XML data over the generic mapping approaches.

Key words: XML, DBMS, XQuery

1 Introduction
With increasing popularity of XML documents, it is crucial to store and query XML documents efficiently in order to exploit the full power of this new technology. A motivated application of this research can be an XML repository system that stores millions heterogeneous XML documents, which are well-formed but have no DTD or the DTDs of which are not known beforehand.

One promising approach to managing XML documents is to store and query them in a relational database. In this approach, XML data must be converted into a set of tuples and stored in relational tables, due to the difference between relational database structure and the hierarchical structure of XML documents. Queries posed on XML documents then need to be translated into SQL statements against those relational tables, and the query results need to be constructed in the desired XML format. Thus, the problem of query efficiency over XML documents shifts to the effectiveness of the database schemas in terms of query performance by SQL.

The database schema for XML storage varies among different XML-to-Relation mapping techniques. One of the mapping techniques, called generic mapping, is to design relational database schemas for XML documents without the knowledge of DTD or XML Schema information. Multiple generic mapping techniques have been studied, such as Edge Mapping [6] and Path Based Mapping [10] [14] [7].

The basic idea behinds existing XML generic mapping approaches is to model an XML document as a tree, and record the parent-child relationships among nodes in the XML tree as tuples in relational tables, with each tuple representing an edge or a node in the XML data tree. As a result of this decomposition, the hierarchical structure of an XML document is flattened to binary relationships scattered in the database tables. Although the generic mapping approach can be used to store any XML documents, with or without schema, into current relational database, querying performances of existing generic mapping approaches are still far from satisfactory, especially for large XML documents.

There are two main reasons of this inefficiency. First, XML data are scattered in relation schemas with a very high degree of fragmentation. At the query processing time, a great amount of join or $\theta$ join operations are required to restore the hierarchical structure of an XML document. Second, only parent-child binary relationships are stored in relation schemas, so it is very expensive to search ancestor information.

In this paper, we propose a new generic mapping technique, called Xregion, to store XML data in relational databases. Our solution for reducing the fragmentation is simple, but very effective. We first partition an XML document into several disjoint regions according to the cardinality of node occurrences, and then store these regions, including their parent information, into separate tables. An example for our mapping approach is given below.

The graph in the Figure 1 can be interpreted as a structure summary of a sample XML document for a university registration system. Each ellipse represents a node in the document while a double ellipse represents a node with repeated occurrences, i.e. set-valued node. For example, each course has one course number and one title, but many sections and many TAs. Therefore, both “section” and “TA” node are represented by a double ellipse.

Each set-valued node represents a distinct region in the structure summary, and each distinct node in a region is represented by a separate field in the table for the region. For example, $R_{21}$ in Figure 1 is the region represented by the “course” node, and its corresponding

*Computing Science Department
Edmonton, Canada T6G 2H1
yuan@cs.uaberta.ca
The advantages to partition and store XML documents based on cardinality of node occurrences are as follows. First, since each region contains no further nested structure, i.e., no other set-valued nodes, it can be stored a set of tuples in one table. Second, we avoid the very expensive join operations required for query evaluation by other approaches, because all children nodes, except for the set-valued child nodes, of an element are stored in the same relation as the element. Since regular relations do not support set-valued attributes, we create a separate relation for each set-valued element and all its descendants with one-to-one relationships to the element. Since regular relations do not support set-valued attributes, we create a separate relation for each set-valued element and all its descendants with one-to-one relationships to the element. So in our system, XML documents are decomposed according to the nested level of the data, rather the binary relationships between nodes, which are widely used in the existing generic mapping methods.

We have implemented Xregion and several other existing generic mapping techniques, including XParent [7], Edge Mapping[6] and Xrel[14]. We have conducted extensive experiments using above approaches to storing two XML documents (200 MB and 2GB) into an Oracle database, and then compare their performance by evaluating 7 and 5 typical queries respectively. Figure 2 describes the elapsed time for querying the database storing an XML document with size 2GB.

The figure demonstrates that the proposed approach dramatically improves the performance of queries on the XML data over the existing generic mapping technologies, by the order of one or two magnitudes. The improvement is particularly striking for large XML documents.

The rest of the paper is organized as follows. In Section 2, we review the current state of generic mapping approaches. In Section 3, we give formal definitions of the XML data models used in our approach. Section 4 formalizes the proposed approach, Xregion. Section 5 goes into the implementation details of the XML bulk loader system. The experimental setup and results are described in Section 6. Finally, in Section 7 we provide a summary and present future work.

2 Background

XML is becoming the standard for data interchange and representation on the Web and elsewhere. Its nested and self-describing nature makes its transmission, and presentation more intuitive than any other data standard. Figure 3 is an example XML document describing courses information.

An XML documents may have a DTD (Document Type Definition) or an XML Schema, which can be used to define and validate the data structure of the document. In this paper, we focus on the problem of storing XML documents into relational database without the knowledge of the DTDs or XML schemas.

The nested and self-describing nature of XML provides simple and flexible means for exchanging data among applications. However, it is not designed to facilitate efficient data storage or retrieval.

One method to store XML data is to employ current Relational Database Management Systems (RDBMS) for XML storage and retrieval. This takes advantage of the mature technologies already provided in current database systems, such as concurrency control, powerful query optimizers and indexing techniques.

When we look at mapping XML to a relational database, we are considering the difference between relational database structures and XML data structures. Conventional relational database systems do not sup-
Figure 3: An example XML document for university courses.

port the inherent hierarchical and semi structured format of XML data. Instead, the nested XML data need to be transformed into tables according to the database schemas generated by mapping approaches.

The design of database schema is crucial to the performance of query processing and result publishing, because it stipulates how XML data are stored into the underlying relational database systems. In the case that XML documents are very large or they are in huge numbers, it is imperative to convert those data to a format where they can be retrieved effectively.

To further understand XML relational storage models and their effectiveness in terms of query processing, we will now describe existing generic mapping approaches.

**Edge Based Mapping:** Florescu and Kossmann [6] proposed the Edge approach to modeling an XML document as an ordered and directed graph, where every node is assigned an identifier oid and each edge is explicitly labeled by the name of incoming element type or attribute. All edges of an XML data graph are stored in a single table called the Edge table, which has the following structure.

\[
\text{Edge}(\text{source}, \text{ordinal}, \text{target}, \text{label}, \text{flag}, \text{value})
\]

Each tuple in the Edge table represents one edge in the directed graph. An edge is defined by the Source and Target fields, which are oids of the two nodes connected by the edge. The Label field records the label of an edge. The local order of the edge among its siblings is stored in the Ordinal field. The Flag field indicates whether the target node is an internal node (“ref”) or a leave node with a value (“string” or “int”). The data graph for the example XML document is shown in Figure 4, and Table 1 is its corresponding Edge table.

Table 1: A relational storage example using Edge approach.

<table>
<thead>
<tr>
<th>Src</th>
<th>Ord</th>
<th>Gt</th>
<th>Label</th>
<th>Flag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;0</td>
<td>1</td>
<td>&amp;1</td>
<td>catalogue</td>
<td>ref</td>
<td>ABC</td>
</tr>
<tr>
<td>&amp;1</td>
<td>1</td>
<td>&amp;2</td>
<td>univ</td>
<td>string</td>
<td>Dr. Lin</td>
</tr>
<tr>
<td>&amp;3</td>
<td>1</td>
<td>&amp;3</td>
<td>course</td>
<td>ref</td>
<td>291</td>
</tr>
<tr>
<td>&amp;3</td>
<td>1</td>
<td>&amp;4</td>
<td>@cno</td>
<td>string</td>
<td>Database Systems</td>
</tr>
<tr>
<td>&amp;3</td>
<td>1</td>
<td>&amp;5</td>
<td>title</td>
<td>string</td>
<td>H1</td>
</tr>
<tr>
<td>&amp;3</td>
<td>1</td>
<td>&amp;6</td>
<td>sections</td>
<td>ref</td>
<td>Dr. Dean</td>
</tr>
<tr>
<td>&amp;6</td>
<td>1</td>
<td>&amp;7</td>
<td>section</td>
<td>ref</td>
<td>539</td>
</tr>
<tr>
<td>&amp;7</td>
<td>1</td>
<td>&amp;8</td>
<td>@sno</td>
<td>string</td>
<td>Dr. Lin</td>
</tr>
<tr>
<td>&amp;7</td>
<td>1</td>
<td>&amp;9</td>
<td>instructor</td>
<td>string</td>
<td>Dr. Dean</td>
</tr>
<tr>
<td>&amp;6</td>
<td>2</td>
<td>&amp;10</td>
<td>section</td>
<td>ref</td>
<td>539</td>
</tr>
<tr>
<td>&amp;10</td>
<td>1</td>
<td>&amp;11</td>
<td>@sno</td>
<td>string</td>
<td>Dr. Lin</td>
</tr>
<tr>
<td>&amp;10</td>
<td>1</td>
<td>&amp;12</td>
<td>instructor</td>
<td>string</td>
<td>Dr. Dean</td>
</tr>
<tr>
<td>&amp;1</td>
<td>2</td>
<td>&amp;13</td>
<td>course</td>
<td>ref</td>
<td>123</td>
</tr>
<tr>
<td>&amp;13</td>
<td>1</td>
<td>&amp;14</td>
<td>@cno</td>
<td>string</td>
<td>Dr. Hanks</td>
</tr>
<tr>
<td>&amp;13</td>
<td>1</td>
<td>&amp;15</td>
<td>title</td>
<td>string</td>
<td>Program.</td>
</tr>
<tr>
<td>&amp;13</td>
<td>1</td>
<td>&amp;16</td>
<td>sections</td>
<td>ref</td>
<td>112</td>
</tr>
<tr>
<td>&amp;16</td>
<td>1</td>
<td>&amp;17</td>
<td>section</td>
<td>ref</td>
<td>112</td>
</tr>
<tr>
<td>&amp;17</td>
<td>1</td>
<td>&amp;18</td>
<td>@sno</td>
<td>string</td>
<td>Dr. Hanks</td>
</tr>
<tr>
<td>&amp;17</td>
<td>1</td>
<td>&amp;19</td>
<td>instructor</td>
<td>string</td>
<td>Dr. Hanks</td>
</tr>
<tr>
<td>&amp;13</td>
<td>1</td>
<td>&amp;20</td>
<td>TA</td>
<td>ref</td>
<td>123</td>
</tr>
<tr>
<td>&amp;20</td>
<td>1</td>
<td>&amp;21</td>
<td>@sid</td>
<td>string</td>
<td>123</td>
</tr>
<tr>
<td>&amp;20</td>
<td>1</td>
<td>&amp;22</td>
<td>lab</td>
<td>string</td>
<td>D01</td>
</tr>
<tr>
<td>&amp;13</td>
<td>2</td>
<td>&amp;223</td>
<td>TA</td>
<td>ref</td>
<td>112</td>
</tr>
<tr>
<td>&amp;23</td>
<td>1</td>
<td>&amp;24</td>
<td>@sid</td>
<td>string</td>
<td>112</td>
</tr>
<tr>
<td>&amp;23</td>
<td>1</td>
<td>&amp;25</td>
<td>lab</td>
<td>string</td>
<td>D02</td>
</tr>
</tbody>
</table>

**XParent:** XParent [7] is a four-table path-based mapping system, which uses fixed schemas to store variant XML documents, according to the XPath model. The Element table and Data table, respectively, are created for storing the element nodes and the values of attribute nodes and text nodes. Each tuple in these tables represents a node in the XML tree. The binary relationships among nodes are stored in the DataPath table. Another table, LabelPath table, stores all distinct label-paths and their depth. These four relational
tables are as follows.

- LabelPath (ID, Len, Path)
- DataPath (Parentid, Childid)
- Element (PathID, Did, Ordinal)
- Data (PathID, Did, Ordinal, Value)

Compared with Edge Mapping, the path based mapping speeds up the query processing on simple XML queries by storing path expressions explicitly in relations. However, when processing queries with multiple paths or multiple conditions on different branches they still need a number of joins or self-joins to check nodes connections.

**XRel:** XRel proposed by M. Yoshikawa and T. Amagasa et al [14] is a generic mapping approach, which keeps both the simple path expressions and element positions in relations. The position of an element is recorded by the byte-offset of its start and end positions in the XML document. For example, the positions of node 1 (document root “catalogue”) and node 19 (“instructor”) in the XML tree shown in Figure 4 are (0, 450) and (299, 332) respectively. In addition to the Element table, another two tables are created to store attributes and text contents, i.e. Attribute table and Text table. Same as Edge and XParent approaches, each tuple in these relations represents one node in the XML tree. The basic XRel schemas are as follows.

- Text (docID, pathID, start, end, value)
- Attribute (docID, pathID, start, end, value)
- Element (docID, pathID, start, end, ordinal, reverse_ord)

A contribution of XRel was that they introduced a new format for representing paths by using two characters to separate steps in a path expression, e.g. ‘#/catalogue#/course’ instead of ‘/catalogue/course’. The advantage of this transformation is that it simplifies the query translation process for simple queries on paths with wildcards, and guarantees the correctness of string matching in query processing.

In XRel schemas, the containment relationships among nodes in an XML document can be captured by comparison between start and end positions. So for XRel, sometimes, it does not need to verify all the intermediate edge connections one by one between two nodes, e.g. node a→b→c→node d, and only need to check whether one node (d) is reachable from the other node (a). Thus XRel will use less join operations for searching ancestors of a node. However, this simplification in query processing does not improve the query performance as expected, and sometimes it is even worse especially for large documents.

**Monet Model:** Monet [10] is another path based XML relational storage model proposed by Schmidt et
al. The basic idea is similar to the Edge mapping, which identifies parent-child relations from the XML data graph. At the mapping stage, they apply a different approach by creating separate relational tables for every distinct path in the graph. Thus data stored in the same table has a strong structural relation and each table is relative small compared to Edge approach. However, similar argument with the Binary approach of Edge mapping, this approach might not be viable for large collections of XML documents, due to the limit of total number of tables in database systems. For example, the Monet approach created 2587 tables for a single XML document for Webster’s Dictionary [10].

Other Mapping Approaches: In addition to generic mapping approach, several other XML-to-Relation mapping techniques have also been investigated recently. Some of them use XML DTDs or XML schemas information for generate relational schemas [11] [8][9]. Others construct the database schema according to analysis and statistics on frequent structure or query work load, such as STORED system [4] and a cost based system LegoDB [2]. But it is difficult for them to deal with XML data that has irregular structures. In addition extra operations, such as gathering statistics and analyzing query work load, are also required.

Cooper, Sample and Franklin et al pursue a different direction to improve the performance of querying XML data in database. To facilitate the navigation and selection of nodes on the XML trees, they build a special index, Index Fabric [3], on top of RDBMSs for storing path information. The experimental results show that the fast index improves performance, but mainly for refined paths, which are specialized paths for tuning frequently occurring queries.

Xing, Guo and Wang use node grouping to mapping XML documents to databases which reduces the index space and thus significantly improve the query performance [13].

3 Xregion Data Model

In this section, we first formally define the XML data tree, XML structure tree, region and nested level, which forms the basis of the data model used in our mapping approach.

Given an XML document, we use \( El \) to denote the set of element names, \( Attr \) the set of attribute names, \( Vert \) the set of node identifiers, \( Str \) the set of possible string value of elements or attributes. (Note that the symbol ‘@’ is added as the prefix of all attribute names.)

3.1 XML Data Tree

Following previous work on XML data, we model an XML document as a node-labeled tree, XML data tree, which is defined below. The XML data tree used in our approach is slight different from the XPath tree models [5] discussed in Section 2. We model the text of an element as the value of that element node, rather than a separate text node on the tree. For this reason, we adapt the formal definition of XML data tree from XNF [1] by modifying the total function “ele” and adding a new function “val”.

**Definition 1** [1] An XML tree \( T \) is defined as a tree \((V, lab, ele, val, attr, root)\):

- \( V \subseteq Vert \) is a finite nonempty set of vertices.
- \( lab: V \rightarrow El \).
- \( ele: V \rightarrow V^* \).
- \( val: V \rightarrow Str \cup \text{null} \).
- \( attr \) is a partial function \( V \times Attr \rightarrow Str \). For each \( v \in V \), 
  - \( \{ al \in Attr \mid attr(v, al) \text{ is defined} \} \) is a finite set.
- \( root \in V \) is the root of XML data tree \( T \).

Every element in the document is modeled as a node, characterized by a unique node identifier. All attributes of an XML document are modeled as children of their associated element nodes. Given an XML data tree, a path of a node is a sequence of ancestor labels starting from the root to the node. Figure 5 is a graphic depiction of the data tree for the sample XML document in Figure 3.

3.2 XML Structure Tree

An XML document contains both meta data and data itself, and its meta data, including all path and other information, can be described by the structure tree, which is a summary of the XML data tree.

**Definition 2** Let \( Pt \) be the set of paths in an XML data tree. An XML structure tree \( S \) is defined as a tree \((Vs, Pt, multi, r)\):

- \( Vs: Pt \rightarrow El \cup Attr \).
- \( multi: Pt \rightarrow n. \)
- \( r \in Vs(Pt) \) is the root of XML structure tree \( S \).

\( multi(p_{1}) \) shows the maximum cardinality of a node identified by a path \( p_{1} \in Pt \). The parent-child relationship is captured by paths. Figure 6 shows an XML structure tree for the example XML document. All multi-valued nodes that occur repeatedly under their
Figure 5: An XML data tree for the example XML document

Figure 6: An XML Structure tree

has one course number and one title, but many sections and many TAs. Therefore, both course number “@cno” and “title” are represented by a single ellipse while “section” and “TA” are represented by a double ellipse.

Features of the XML structure tree are:

1. It represents the complete structure of a given XML document, i.e., it contains the structure of every element type in the XML document.

2. It is exactly as deep as the corresponding XML data tree.

3. Generally, the XML structure tree will be much smaller than the XML data graph.

From this graph we can see that the structured information for an XML document includes not just path information, but also cardinality of node occurrences. Because of the above features, we will use the structure tree as the basis for partitioning any given XML documents.

3.3 Region

The key idea of the proposed approach is to partition the input XML documents into disjoint regions according to the cardinality of node occurrences. The definition of the region is given below.

Definition 3 A region in a structure tree is a subtree $R$ of the structure tree such that

1. the root of $R$ is either a set-valued node (i.e., a double ellipse) or the root of the structure tree, and

2. all the subtrees of $R$ rooted with set-valued node (i.e., a double ellipse) are removed.

Obviously, a structure tree with $N$ double ellipses contains $N + 1$ disjoint regions. Each region consists of all and only those descendants that have one-to-one relationships with the set-valued element (region root), so we can map all nodes in a region into one relation with every node in this region represented by a separate field in the relation. The other feature of a region is that all the nodes in a region share the same ancestors and cousins outside of the region. Consequently, storing all the regions in separate tables may significantly
reduce the number of joins needed for query evaluation. Figure 7 shows all the regions for the sample document, and the corresponding relations for storing regions.

For example, the relation for the region “course” is:
Relation course (course, @cno, title, sections)

3.4 Region Tree and Nested Level
In order to specify the table schema according to the regions, we define the region tree of a given document as the tree obtained from the structure tree by replacing each region with one node. Since each node in a region tree will be stored as one relation, a region tree is also called a relation tree. Figure 7 (b) describes the relation tree for the sample document.

Given a region tree $T_R$, the nested level of a region is then defined as the height of its corresponding region node in the region tree.

All nodes in a given region belong to the same nested level, although they are on different depths of the XML structure tree. For example, in Figure 7, while the node “section”, “instructor” and “TA” are on different depths in the structure tree, they are at the same nested level – the nested level 3. Thus we transform the complicated hierarchical structure of an XML document to a relatively simple nested structure among regions (relations).

Based on its topological position in the region tree, each region can then be labeled as $R_{i,j}$, where $i$ denotes its nested level, and $j$ the order of regions in their respective level, with the left most as 1. Consider Figure 7 (b) again. Relation “root” is labeled as $R_{1,1}$, Relation “course” is labeled as $R_{2,1}$, Relation “section” is labeled as $R_{3,1}$, and Relation “TA” is labeled as $R_{3,2}$.

3.5 Region Instance
Once all regions are identified, the XML data tree will be traversed to map all the instances of each region into the corresponding tables specified by the region.

**Definition 4** Let $Pr$, $r_{\text{edge}}$ be the set of paths and edges in a region $R$ of an XML structure tree respectively, and $root$ be the path of the region root. An instance $I$ of the region $R$ is defined as a tree $(Pi, edge, r, val)$, which is a subtree of the XML data tree.

1. $Pi \subseteq Pr$ is a finite set of paths.
2. $val: Pi \rightarrow Str \cup \text{null}$.
3. $edge \subseteq r_{\text{edge}}$ is a finite set of edges.
4. $r = root$ is the root of sub tree $I$ that represents the instance.

We call each occurrence of a region structure as an instance of the region, because it instantiates the abstract structure of a region with data in the XML document. When loading XML document into the database, each instance of a region is stored as one tuple in the table of this region.

For example, the $I_1$ and $I_2$, marked on the XML data tree shown in Figure 8, are two instances of the region “course”.

4 Xregion Storage Schema
In this section, we specify the basic database schema for our proposed approach, Xregion. The basic idea of Xregion is to store all children nodes, including attributes, of an element in the same relation with the element, except its set-valued child nodes, in order to reduce the fragmentation degree of XML data in relations.

Given an XML document, we first build an XML document structure tree for it, partition the document structure tree into several regions, and then map all nodes of each region into a separate relational table. After processing all the nodes on the XML document structure tree of a given XML document, the mappings from XML structure to database schema are recorded in a database table as meta data. Because of the limit on the size of region trees, a limited number of tables are needed to store any number of XML documents.

The XML data tree is then traversed to load the XML data into their corresponding tables in the database system. Every query to the XML data will be translated into a query to the underlying database, and then the XML query result will be constructed from the answer returned from the underlying database.

4.1 Basic Database Schema for Xregion
The underlying database for Xregion consists of one meta table, used to store all the meta information, such as paths, in the document structure trees and their corresponding mappings, and a limited number of data tables, named table $I_i,j$ for region $R_{i,j}$, where the upper limit of both $i$ and $j$ depends on the size of regions trees of stored XML documents.

Therefore, for an XML document with $N$ regions, we need only $N + 1$ tables. With the increase of $N$, new data tables can be created accordingly.

For simplicity, we assume that the number of nodes in any region is limited to $n$, say $n = 20$. Should the number of nodes exceeds this limit, we can either increase the number of columns of the table for the region or to create a new table to store information for the extra nodes.
4.1.1 Data Tables

Each data table is used to store all the instances of the corresponding region, as well as the data needed to identify the parent information of any node in the region.

Since a region does not contain any set-valued node, each instance of a region can be represented by a tuple of n columns, one for the value of each node in the region.

To uniquely identify each tuple in the region, we will create one unique id, named tuple id.

In Xregion, the parent of any node in a tuple of a given region table, is either in the same table, or is stored in the region table of upper level (parent of the region root). Therefore, we also create a column, called parent id, or p id for short, to store the tuple id of the parent instance of the given tuple in a region.

In order to preserve the order among all sibling nodes, we will introduce a column to record the ordinal position of the set-valued nodes, which are root nodes of each region. Finally, since a data table will be used to store multiple XML documents, we do need a column to store the document name.

In summary, each table $table_{i,j}$ is defined as a table with $n+4$ columns, that is,

$\text{table}_{i,j}(\text{doc name}, \text{tuple id}, \text{p id}, \text{ordinal}, \text{col}_1, ..., \text{col}_{n-1}, \text{col}_n)$.

Where doc name is used to store the document name of the XML document, tuple id is used to store the unique id of a tuple in the region, p id is used to store the tuple id of the parent node, ordinal is used to store the ordinal position of the tuple, and col$_i$, for $1 \leq i \leq n$, are used to store the value of the corresponding node in the region.

For convenience, we simplify the table for root region $R_{1,1}$ by removing the p id, ordinal, since the root region does not have any parent and sibling. So the structure of the root table is:

$\text{table}_{1,1}(\text{doc name}, \text{tuple id}, \text{col}_1, ..., \text{col}_{n-1}, \text{col}_n)$.

4.1.2 Meta Table

The meta table, named meta table, is designed to store all the meta information from the document structure trees, one tuple for each path in the structure tree.

Since each path represents one node in the structure tree and each node is stored in the corresponding column of its region relation, the tuple shall contain the name of the region table, the column name in the region table, and the parent table name of the region table.

Of course, we shall also store, for each tuple, the name of the XML document.

In summary, the meta table consists of five columns, that is,

$\text{meta table}(\text{doc name}, \text{path}, \text{table name}, \text{col name}, \text{p table})$.

It is easy to see that the aforementioned $N+1$ tables store all the information about an XML document.

4.2 Example

Given an XML document to be imported into a RDBMS system, the database schemas generation process works as follows. First, the XML document is parsed to get
the path summary of the document. Second, an XML document structure tree is built and all set-valued nodes are identified. Once the document structure tree is created, it is partitioned into regions represented by set-valued nodes. Then separate database schema is created for each region, and the relational table assignments are recorded in the meta_table.

After creating or assigning tables for each region on the XML document structure tree, the XML data tree is traversed in depth-first-order to load the XML data, instances of each region, into their corresponding tables in the database system.

All the data tables and meta_table for the sample document described in Figure 3 are shown in Tables 2 - 6.

As shown in the Table 4, the two instances, $I_1$ and $I_2$ (Figure 8), of the “course” region are stored as two tuples in table_2_1.

In this example, the values of “tuple_id” of all the region instances are assigned based on the document order. Figure 9 describes the tuple orders for the example XML document. The “tuple_id” together with the “doc_name” field serves as the primary key of each data table. The “p_id” and “doc_name” field forms a foreign key refer to the parent table in the upper level.

### 4.3 Discussion

#### 4.3.1 Storing Different XML Documents

It is not feasible to create different schemas for every individual XML document in a system that store a large number of XML documents, because some of the existing database systems still enforce a limit on the number of tables that the database system can hold. So we propose to let different XML documents share a set of relational tables in the system, and only create a new table...
when there is no corresponding table existed in the system. To distinguish data of different XML documents stored in the same table, we add a document identifier attribute to all the tables in the system.

Although different XML documents may have heterogeneous structures, the concept of nested level is the same. In Xregion, every data table conforms to the same general structure, which make it feasible for different XML documents to share and reuse the existing tables. So we can store XML data of different XML documents into the same table as long as they are at the same nested level of their own relation trees. Table 7 is a snapshot of table table_2_1, in which the data of “course.xml” and “test.xml” are stored.

4.3.2 Query Evaluation and Result Publishing

The meta_table of Xregion provides a standard interface for query processing and XML query result publishing. The mapping information of all component of an XML document are recorded as meta-data identified by paths and document name in the meta_table. At query processing time, the system looks up the meta_table and translates XML queries to SQL statements against the relational tables in the database system.

In Xregion, every instance of a region is stored as a tuple in its corresponding region table, any non-set-valued node in a instance is stored in the same tuple with its parent, and the parent of a set-valued node is stored in a record of its parent region table and is identified by the parent_id (p_id) of the set-valued node. So compared with other existing generic mapping approaches, Xregion is much more efficient in evaluating queries.

First, queries on nodes within a region is simplified to one or a limit number of selections on one region table, without the need of join operations, which are otherwise required by other existing mapping approaches. For example, SQL 4, 5 and 6 are translated SQL statements of Xregion, Edge and Xparent, respectively, for the XML query given below.

Example 1 Given the example XML document in Figure 3, find the course title of the course with a course number “291”.

**Q3** /

catalogue/course[@cno=“291”]/title

**SQL 4** A translated SQL query statement for Q3 using Xregion.

```
SELECT col3
FROM table_2_1
WHERE col2=’291’
```

**SQL 5** A translated SQL query statement for Q3 using Edge.

```
SELECT title.value
FROM edge root,edge crs,edge cno,edge title
WHERE root.label=’catalogue’
AND crs.label=’course’
AND title.label=’title’
AND cno.label=’@cno’
AND root.tgt=crs.src
AND crs.tgt=title.src
AND crs.tgt=cno.src
AND cno=’291’;
```

**SQL 6** A translated SQL query statement for Q3 using XParent.

```
SELECT title.value
FROM data cno, data title,
labelpath lp_cno, labelpath lp_title,
datapath dp_cno, datapath dp_title
WHERE lp_title.path=’/catalogue/course/title’
AND lp_ta.path=’/catalogue/course/@cno’
AND cno.pathid=lp_cno.id
AND title.pathid=lp_title.id
AND cno.value=’291’
AND title.did=dp_title.childid
AND cno.did=dp_cno.childid
AND dp_title.parentid=dp_cno.parentid;
```

As we can see, both Edge and XParent approaches require join operations to ensure that the “@cno” nodes and “title” nodes are belong to the same “course” elements. For Xregion, the “@cno” and its corresponding “title” are stored in the same tuple with “course”, so it only need a single selection on “@cno” and a projection on “title”.

Furthermore, Xregion makes the searching for ancestor information more efficient than existing mapping approaches. Xregion transforms the hierarchical structure of an XML document to a relatively simple nested structure among regions (relations), which are normally much shallower than the structure tree. So the process for searching ancestor of nodes is converted to search ancestor of regions, which reduces the number of join operations for complicated queries. It is this feature that makes efficient query evaluations possible.

As an example, SQL 7, 8 is a translated SQL statement of Xregion and Xparent, respectively, for the example XML query discussed in Section 2.

**Example 2** Find the TAs who work with Dr. Hanks.

**Q2** /

catalogue/course[sections/section/instructor

```
SELECT title.value
FROM edge root,edge crs,edge cno,edge title
WHERE root.label=’catalogue’
AND crs.label=’course’
AND title.label=’title’
AND cno.label=’@cno’
AND root.tgt=crs.src
AND crs.tgt=title.src
AND crs.tgt=cno.src
AND cno=’291’;
```

**SQL 7** A translated SQL query statement for Q2 using Xregion.
SELECT ta.col1, ta.col2, ta.col3
FROM table_3_1 inst, table_3_2 ta
WHERE inst.col3='Dr. Hanks' AND
      ta.p_id=inst.p_id

SQL 8: A translated SQL query statement for Q2 using XParent.

SELECT ta.did
FROM data ta, data inst,
     labelpath lp_ta, labelpath lp_inst,
     datapath dp_ta, datapath dp_inst,
     datapath dp_section, datapath dp_sections
WHERE lp_inst.path='/catalogue/course/
        sections/section/instructor'
   AND lp_ta.path = '/catalogue/course/TA'
   AND ta.pathid = lp_ta.id
   AND inst.pathid = lp_inst.id
   AND inst.value = 'Dr. Hanks'
   AND inst.did = dp_inst.childid
   AND dp_inst.parentid=dp_section.childid
   AND dp_section.parentid=
        dp_sections.childid
   AND ta.did = dp_ta.childid
   AND dp_sections.parentid=dp_ta.parentid

Using Xregion, we only need to check whether the
"instructor" and "TA" are connected by same instances
of their parent region course, whereas Xparent requires
a number of joins to check the connection
instructor←section←sections←course←TA, in order
to make sure that the pairs of nodes, "TA" and "In-
structor", are connected by the same "course" nodes.
Figure 10 is a graph depiction for tracing the nearest
common ancestor "course" under Xregion and XParent
schemas, which shows that XParent needs to check two
more steps than Xregion for the example query.

XML query result publishing is another important
aspect for evaluating an XML-to-Relation mapping ap-
proach. Most XML query languages return the query
result in XML format, which consists of the value of
the satisfied element nodes together with all their de-
cendants.

Xregion also speeds up the XML query result pub-
lishing process. Because all children nodes, except for
the set-valued child nodes, of an element are stored in
the same relation as the element. So under Xregion schema, transforming the answer from relational
database to XML format, does not involve expensive
operations. For example, shown in the SQL 7, all the
content of "TA" elements can be retrieved from a sin-
gle table. However, for other existing approaches, they
still need a number of join operations to construct the
query result in XML format, due to the high fragmen-
tation of the XML data stored in the database. For
example, four more joins on Data tables and DataPath
tables are involved for answering the above query by
XParent approach.

5 Implementation

We have implemented a prototype of an XML import-
ing system, called XML Loader, as per the mapping ap-
proach described in previous section, for storing XML
documents into relational databases. Figure 11 de-

![XML importing system diagram]

Figure 11: The prototype of the XML importing sys-
tem.

scribes the architecture of our XML importing system.
There are three main modules: XML Parser, Schema
Generator and XML Data Loader. The XML Parser
reads the input XML document(s), extracts XML tag
name and value of each node in the document, ma-
terializes the path for each node and detects all the
set-valued nodes. The Schema Generator identifies re-

gions for each set-valued node and creates the corre-
sponding mapping schemas for the XML document.
The Data Loader module takes the schema informa-
tion generated in the Schema Generator module, com-
poses tuples according to the relation assignment and
loads those tuples into their corresponding tables in the
underlying database. The whole programs have been
completely written by us using the JAVA program-
ning language (Java™ 2 Platform, Standard Edition,
v 1.4.1) and JDBC (Java Database Connectivity 2.0).

XML Parser Module: our XML parser module
is implemented based on the SAX interface. The ma-
or output of the XML Parser module is the completed
paths set and set-valued nodes set, i.e. the document
structure tree, of the input XML document. Since our
parser does one sequential scan over the input XML
document, we cannot navigate back and forth to re-
trieve the ancestor information of a given element node.
So in implementation, we introduce a path stack to
trace the local hierarchical structure and node occur-
rence of an XML document. At any given time the peak of the stack is the parent information of the incoming node.

**Schema Generator Module** The Schema Generator module is the core component of our XML Loader system. Figure 12 shows a simplified view of the schema generation procedure. First, the XML document structure tree obtained from XML Parser module is partitioned into regions by set-valued nodes. Then relational tables are assigned to all regions according to their nested levels on the region tree. In the case that there is no corresponding table for a given region existed in the underlying database, a new table representing the region will also be created in the schema generating process. At the end of the schema generation procedure, the meta data table will be updated for this new XML document.

**Data Loader Module** The Data Loader module reads the schema meta data generated by Schema Generator Module, identifies region instances and composes tuples with tuple id and parent information, and loads the tuples into their corresponding tables.

In our system, an XML document is loaded into database incrementally by one-pass sequential scan. So it can process very large XML documents, as long as the size of the document supported by the underlying operating system.

The important feature of identifying region instance is that the procedure of constructing one region tuple may interleave the process of composing tuples of other regions, due to the nested structure of XML documents and sequential scan. For example, shown in Figure 14, the construction of a course tuple, is interrupted by that of two section tuples.

---

**Figure 10:** Ancestor tracing route for XParent VS XRegion

**Figure 12:** Schema Generator
Figure 13: Generating a relation for the deepest set-valued node *section* and update the structure tree by cutting the *section* sub tree off.

<catalogue>
    ......
    <course cno="291">
        <sections>
            <section sno="H1">
                <instructor>Dr. Lin</instructor>
            </section>
            <section sno="H2">
                <instructor>Dr. Dean</instructor>
            </section>
        </sections>
        <title>DBMSs</title>
    </course>
    ......
</catalogue>

Figure 14: An instance of *course* region split by two *section* region instances

6 Experiments

To evaluate the effectiveness and scalability of the proposed XML generic mapping approach Xregion, extensive experiments have been conducted. In this section, we compare the performance of the Xregion with the Edge mapping, and XParent approach. The capability of storing and querying a large number of different XML documents in a single database has also been tested, and corresponding results will be discussed in this section.

6.1 Experimental Setups

For the purpose of performance comparison, we also implemented the Edge mapping and XParent mapping approaches using Java programming language and SAX parser API. All experiments were conducted on a
TABLE 8: Data sets information

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>#paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHAKS</td>
<td>7.65MB</td>
<td>57</td>
</tr>
<tr>
<td>DBLP</td>
<td>200MB</td>
<td>156</td>
</tr>
<tr>
<td>SYN2G</td>
<td>2GB</td>
<td>156</td>
</tr>
</tbody>
</table>

PIII/1GHz PC with 1G RAM running Red Hat Linux release 7.1. The relational database system used in the study was Oracle 9i Database Standard Edition release 2. We selected three different XML data collections, with size of 7.65M, 200MB and 2GB respectively, as our data sets, which contain XML documents with various depths and sizes. In order to cover different aspects of XML queries over XML data, queries were selected carefully for the corresponding data collections. Table 8 summaries the features of three XML collections in our experiments.

6.1.1 Data Sets

- **SHAKS**
  SHAKS consists of 37 Shakespeare plays in XML document format with the average size of 277kB and 30,000 words. The maximum depth of nested XML tags is 5 (Play/Act/Scene/Speech/Line). The whole collection, created by Jon Bosak, is available at http://metalab.unc.edu/bosak/xml/eg/shaks200.zip.

- **DBLP (200MB)**
  The DBLP data set is a large XML document downloaded from the DBLP server (http://dblp.uni-trier.de/xml/). The DBLP data set used in our experiment contained the data till Jan, 2004, which listed more than 470,000 articles. The size of this DBLP XML document is 200M.

- **SYN2G (2GB)**
  To test the scalability of these three XML relational mapping approaches, Xregion, Edge and XParent, we generated a synthetic XML document with the size of 2GB from the DBLP data set.

The SYN2G XML document used in this experiment is constructed by concatenating 9 modified DBLP with the original DBLP. The whole process is one sequential scan of the DBLP file using SAX for Java.

The rule used in modification is to keep numeric values unchanged, and only modify letters. In order to simulate the real distribution of authors and their publications, we also keep the values of “author” elements unchanged. A randomly generated 5-letter dictionary is used to translate each occurrence of any letter specified in the dictionary to its corresponding letter, at every time creating a new version of DBLP. For example, a title “An Object-Based Approach to the B Formal Method” is translated into “An Objezt-Based Ajjroazr to tre B Formal Metrod” based on the following 5-letter dictionary.

<table>
<thead>
<tr>
<th>letter</th>
<th>new value</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>h</td>
</tr>
<tr>
<td>h</td>
<td>r</td>
</tr>
<tr>
<td>k</td>
<td>o</td>
</tr>
<tr>
<td>p</td>
<td>j</td>
</tr>
<tr>
<td>v</td>
<td>f</td>
</tr>
</tbody>
</table>

6.1.2 Query Set

For experiment on the DBLP and SYN2G XML documents, we used the queries from XParent and those presented by F. Tian et al [12] as query templates. The queries experimented in Xrel [14] was selected for testing the SHAKS data set. These queries test a variety aspect of query performance. In all, we test eighteen different queries in our benchmark.

All benchmark XML queries in the experiment were translated by us into a set of SQL statements, one for each XML query, and executed in ORACLE with session setting SQLTRACE enabled.

6.1.3 Performance Measurement

We compare the Xregion with Edge and XParent approaches with respect to query performance, such as query elapsed times and I/O blocks, as well as the size of resulting database for these mapping schemas.

In all our experiments, the size of the database buffer is set to 32 MB, which is much less than the size of DBLP and SYN2G XML documents, while four times larger than the size of SHAKS. Indexes are built properly on relational tables for all three mapping approaches. For Edge and XParent, we created indexes as proposed in [6] [7]. For Xregion, major indexes are composite indexes on table_i_j(doc_name, tuple_id), table_i_j(doc_name, p_id) and meta_table(doc_name, path).

All benchmark data, such as total elapsed time and I/O blocks of the translated SQL queries, are obtained from Oracle database trace file and formatted with the TKPROF utility provided by Oracle RDBMS. The sizes of database tables and indexes are calculated from the statistics generated by the Oracle database.
6.2 Experiment Results

In this paper we present experimental results of XML queries on three different XML collections with size of 7.65M, 200MB and 2GB respectively. In order to get reproducible experimental results, all the benchmark queries are executed for 5 times before timing. All the results reported are obtained after warming up the database buffer, and are based on the average elapsed times of 10 to 40 executions. The experiment results and discussion are given below for each data set.

6.2.1 Experiment on SHAKS

The SHAKS data set was experimented by many XML Relational mapping approaches in literature. We also experimented on SHAKS using the same queries used in Xrel[14] and XParent [7], in order to compare the query performance of our proposed approach Xregion with the other three mapping approaches, Edge, XParent and Xrel, on small XML documents, as well as check the correctness of our system by comparing the query results published in other papers[14].

The queries for the SHAKS XML collection are as follows.

- SQ1 /PLAY/ACT
- SQ2 /PLAY/ACT/SCENE/SPEECH/LINE/STAGEDIR
- SQ3 //SCENE/TITLE
- SQ4 //ACT//TITLE
- SQ5 /PLAY/ACT/SCENE/SPEECH [SPEAKER=“CURIO”]
- SQ6 /PLAY/ACT/SCENE[//SPEAKER=“Steward”]/TITLE

The elapsed times of all queries for SHAKS are shown in the Figure 15 in logarithmic scale. All queries were executed 40 times to get a better precision. Because the size of SHAKS (7.65MB) is far less than the database buffer size (32MB), all mapping approaches do not require physical reads for evaluating all benchmark queries. Table 9 shows the number of logical I/Os, which are the database buffer cache reads, involved in each query for all three mapping approaches.

For all queries, Xregion outperforms other mapping approaches, and the elapsed time for each query is less than 0.01 sec. XParent performs similarly for SQ1, SQ2, SQ3 and SQ4, which queries only contain one path expression. However, for SQ5 and SQ6, which contain more than one simple path expressions, Xrel is much slower compared with other mapping approaches, because non-equijoins on element start and end positions are involved for evaluating these two queries. Edge approach consumes much time for SQ2 and SQ4, which either query on a long path expression or contains a “//” in the middle of a path expression, since it need a lot of joins to check the connection of all possible steps on the path expressions.

Table 11 shows the size of the resulting relational database tables for each mapping schema.

6.2.2 Experiment on DBLP

For testing the DBLP, we adopted the queries from XParent and those presented by F. Tian et al [12] as query templates, which test a variety aspect of query performance. The followings are 6 query templates for the DBLP XML document.

- Q1 Select title of inproceedings by year and a keyword, such as “XML”.
- Q2 Select articles written by author A.
- Q3 Select papers written by author A or author B.
- Q4 Select titles of papers published between year a and year b, with titles starting with a keyword, e.g. “Database”.
- Q5 Select journal papers by a label of a cite entry.
- Q6 Select journal papers by author A quoted by papers published in a given year.
- Q7 Select journal papers by author M that are quoted by author N.

Query Retrieval
The elapsed times for all queries are shown in the Figure 16 in logarithmic scale and the number of I/O blocks involved together with elapsed times are shown in Table 12. These results show that Xregion dramatically improve the performance of query evaluation.

For simple queries that only contain simple path expressions and key search, such as Q2, Q3 and Q5, Edge and XParent performs comparably with Xregion, in that all re in the same magnitude.

For query Q1, which requires text matching, e.g. ‘%XML%’, the performance of Edge and XParent are very inefficient, because they need to search the entire Data or Edge table for this matching operation. For some other complicated query, such as Q6 and Q7, Xregion outperforms the Edge and XParent significantly.

For Q6, XParent performs even worse than Edge. Although XParent or other path based mapping approaches can locate a node in the XML tree directly with the aid of path information stored in the relational schemas, they still need a lot of joins tracing nearest
common ancestors for processing queries with multiple paths and predicates specified on different branches. For example, query Q6 that contains four paths and three conditions. The following is Q6 using XQuery syntax.

Q6 Select journal papers by author Jim Gray quoted by papers published in 1995.

<result>
{
    LET $cite:= document(dblp.xml)/dblp/article
    [year=''1995'']/cite
    FOR $journal IN document(dblp.xml)/dblp/article
    WHERE $journal/author=''Jim Gray'' and
    $journal/@key=$cite
    RETURN
    $journal
}
</result>

XParen uses 4 path selections and 10 joins to locate and check the connections among nodes involved in the query. Edge approach requires 7 selection on edge labels and 6 self-joins for checking edge connections to evaluate Q6. Because Xregion stores XML documents by regions, which groups nodes with one to one relationships to each other in one relation, e.g. the “year” and “cite” are stored in the same table with “article”, so Xregion use only 2 joins for connecting “author” relation with “article”.

### 6.2.3 Database Size

The resulting database sizes of mapping schemas are also a critical issue, when storing large XML documents into RDBMSs. Table 13 shows the size of the resulting relational database tables and indexes for three mapping schemas.

<table>
<thead>
<tr>
<th>Approach</th>
<th>DB size</th>
<th>#rows</th>
<th>#tables</th>
<th>index</th>
</tr>
</thead>
<tbody>
<tr>
<td>XParent</td>
<td>323MB</td>
<td>16472978</td>
<td>4</td>
<td>416MB</td>
</tr>
<tr>
<td>Edge</td>
<td>237MB</td>
<td>5643462</td>
<td>1</td>
<td>296MB</td>
</tr>
<tr>
<td>Xregion</td>
<td>176MB</td>
<td>2777916</td>
<td>47</td>
<td>115MB</td>
</tr>
</tbody>
</table>

Table 13: Sizes of resulting database tables and indexes for Xregion, XParent and Edge schema of DBLP XML document.

The sizes of database for Xparent schemas is more than 40% larger than the DBLP file, because it stores element nodes and their text values separately, and both tuples of an element nodes are bundled with position information. In addition, XParent also use another table DataPath table to record the parent-child relationships between element nodes, so the database size of XParent is the largest among all three mapping approaches.

Xregion also use less space for indexes, while XParent consumes more than 2 times of space for indexes.

### 6.2.4 Experiment on SYN2G

In order to inspect the scalability of our mapping approach, we generate a synthetic XML document SYN2G, by enlarging the size of original DBLP XML document to 2GB. In generating the new test XML document, we take care to keep the ratio of different elements and attributes in the original DBLP document.

We used the same query templates Q1 to Q5 of DBLP data set, and the same set of author names, year and paper types for experiment on the SYN2G XML document for Xregion, Xparent and Edge approaches.

The sizes of the resulting databases of all these three mapping approaches scaled about 10 times the sizes of their corresponding database for DBLP data set.

Table 14 shows the query elapsed times ratio for Xregion on DBLP and SYN2G XML documents. The ratios for all queries except Q5 are around 10, which is the ratio of the size of DBLP and SYN2G.

Figure 18 shows the ratio of query elapsed time for DBLP and SYN2G using Xregion, XParent and Edge. The performances of XParent and Edge degrade dra-
6.2.5 Schema Generation Time

Different from other existing approaches, which use fixed schemas for storing XML documents, Xregion creates or assigns database schema for an XML document at the parsing time according to the structure of the underlying document. In our experiment, we also measured the schema generation times for Xregion, which are shown in Table 16. The results show that the schema creation for XML documents using Xregion schema can be done within several minutes, even for large documents.

7 Conclusions and Future Work

In this paper, we have presented a new generic mapping approach, called Xregion, to storing XML data in relational database systems, based on reducing the fragmentation level of XML data in relations. Performance study showed promising results that Xregion outperforms existing generic mapping techniques, such as Edge mapping and Xparent, especially for large XML documents. For example, every query on SYN2G in Xregion schema consumes less than 1.2 seconds, while it takes XParent or Edge several minutes to evaluate a single query. The new approach keeps the nested structure of XML documents and stores all non-set-valued nodes in the same tuples with their parents, which in turn reduces the number of join operations for complicated queries in query processing.

The proposed mapping method is a meta-data driven approach and no relational schema assignments are hard-coded. In addition, our method can store different XML documents in a limited number of tables. The mapping information of all component of an XML document is recorded as a meta-data identified by paths and document name. Since the schema is not hard-coded, the maintainability and flexibility are enhanced. Furthermore, this new mapping approach provides a standard interface for query processing and XML publishing. All changes of the relational schema assignments for an XML document are recorded in a single table, which can be easily updated and maintained.

Table 16: The elapsed time for schema generation

<table>
<thead>
<tr>
<th>Data set</th>
<th>Size</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHAKS</td>
<td>7.65MB</td>
<td>4.965 sec.</td>
</tr>
<tr>
<td>DBLP</td>
<td>200MB</td>
<td>66.868 sec.</td>
</tr>
<tr>
<td>SYN2G</td>
<td>2GB</td>
<td>412.569 sec.</td>
</tr>
</tbody>
</table>

Table 15: I/O blocks for querying SYN2G (size 2GB) using Xregion, XParent and Edge

<table>
<thead>
<tr>
<th>Query</th>
<th>Xregion</th>
<th>XParent</th>
<th>Edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>4464</td>
<td>1439115</td>
<td>2486350</td>
</tr>
<tr>
<td>Q2</td>
<td>3560</td>
<td>113920</td>
<td>16345182</td>
</tr>
<tr>
<td>Q3</td>
<td>5209</td>
<td>226412</td>
<td>17040573</td>
</tr>
<tr>
<td>Q4</td>
<td>20567</td>
<td>9338007</td>
<td>18676595</td>
</tr>
<tr>
<td>Q5</td>
<td>26</td>
<td>109664</td>
<td>7728207</td>
</tr>
</tbody>
</table>
XML document are transparent to the query processing module, since the interface of query processing is the meta-data stored in the meta_table.

We have implemented an XML importing system based on proposed approach. This system can create relational schema for any well-formed XML document, with or without DTD information, and load its data into the database automatically. The system can be easily built on top of off-the-shelf relational database management systems.

References


Li-Yan Yuan received his BS and MS in Electric Engineering from Shanghai Jiao-Tong University and Ph.D. in Computer Science from Case Western Reserve University in 1978, 1981, and 1986 respectively. He is Professor in Department of Computing Science at University of Alberta. His research interests include Database Management Systems, Knowledge Representation, and Logic Programming. He has published extensively in ACM TODS, IEEE Transactions, AI Journal, ACM PODS, ICLP.
<table>
<thead>
<tr>
<th>doc_name</th>
<th>Path</th>
<th>table_name</th>
<th>col_name</th>
<th>p_id</th>
<th>table_name</th>
<th>col_name</th>
<th>p_id</th>
<th>table_name</th>
<th>col_name</th>
<th>p_id</th>
<th>table_name</th>
<th>col_name</th>
<th>p_id</th>
<th>table_name</th>
<th>col_name</th>
<th>p_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>course.xml</td>
<td>/catalogue</td>
<td>table_1</td>
<td>col1</td>
<td></td>
<td>table_1</td>
<td>col2</td>
<td></td>
<td>table_1</td>
<td>col3</td>
<td></td>
<td>table_1</td>
<td>col4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>course.xml</td>
<td>/catalogue/univ</td>
<td>table_1</td>
<td>col1</td>
<td></td>
<td>table_1</td>
<td>col2</td>
<td></td>
<td>table_1</td>
<td>col3</td>
<td></td>
<td>table_1</td>
<td>col4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>course.xml</td>
<td>/catalogue/course</td>
<td>table_2</td>
<td>col1</td>
<td></td>
<td>table_2</td>
<td>col2</td>
<td></td>
<td>table_1</td>
<td>col3</td>
<td></td>
<td>table_1</td>
<td>col4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>course.xml</td>
<td>/catalogue/course/@cno</td>
<td>table_2</td>
<td>col1</td>
<td></td>
<td>table_2</td>
<td>col2</td>
<td></td>
<td>table_1</td>
<td>col3</td>
<td></td>
<td>table_1</td>
<td>col4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>course.xml</td>
<td>/catalogue/course/title</td>
<td>table_2</td>
<td>col1</td>
<td></td>
<td>table_2</td>
<td>col2</td>
<td></td>
<td>table_1</td>
<td>col3</td>
<td></td>
<td>table_1</td>
<td>col4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>course.xml</td>
<td>/catalogue/course/sections</td>
<td>table_2</td>
<td>col1</td>
<td></td>
<td>table_2</td>
<td>col2</td>
<td></td>
<td>table_1</td>
<td>col3</td>
<td></td>
<td>table_1</td>
<td>col4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>course.xml</td>
<td>/catalogue/course/sections/section</td>
<td>table_2</td>
<td>col1</td>
<td></td>
<td>table_2</td>
<td>col2</td>
<td></td>
<td>table_1</td>
<td>col3</td>
<td></td>
<td>table_1</td>
<td>col4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>course.xml</td>
<td>/catalogue/course/sections/section/@sno</td>
<td>table_3</td>
<td>col1</td>
<td></td>
<td>table_2</td>
<td>col2</td>
<td></td>
<td>table_1</td>
<td>col3</td>
<td></td>
<td>table_1</td>
<td>col4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>course.xml</td>
<td>/catalogue/course/sections/section/instructor</td>
<td>table_3</td>
<td>col1</td>
<td></td>
<td>table_2</td>
<td>col2</td>
<td></td>
<td>table_1</td>
<td>col3</td>
<td></td>
<td>table_1</td>
<td>col4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>course.xml</td>
<td>/catalogue/course/TA</td>
<td>table_3</td>
<td>col1</td>
<td></td>
<td>table_2</td>
<td>col2</td>
<td></td>
<td>table_1</td>
<td>col3</td>
<td></td>
<td>table_1</td>
<td>col4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>course.xml</td>
<td>/catalogue/course/TA/@sid</td>
<td>table_3</td>
<td>col1</td>
<td></td>
<td>table_2</td>
<td>col2</td>
<td></td>
<td>table_1</td>
<td>col3</td>
<td></td>
<td>table_1</td>
<td>col4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>course.xml</td>
<td>/catalogue/course/TA/lab</td>
<td>table_3</td>
<td>col1</td>
<td></td>
<td>table_2</td>
<td>col2</td>
<td></td>
<td>table_1</td>
<td>col3</td>
<td></td>
<td>table_1</td>
<td>col4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: The meta-table.

<table>
<thead>
<tr>
<th>doc_name</th>
<th>tuple_id</th>
<th>p_id</th>
<th>ord.</th>
<th>col1</th>
<th>col2</th>
<th>col3</th>
</tr>
</thead>
<tbody>
<tr>
<td>course.xml</td>
<td>2.0</td>
<td>1</td>
<td>1</td>
<td>course1</td>
<td>291</td>
<td>Database System</td>
</tr>
<tr>
<td>course.xml</td>
<td>5.0</td>
<td>1</td>
<td>2</td>
<td>course2</td>
<td>539</td>
<td>programming</td>
</tr>
</tbody>
</table>

Table 4: The table_2_1(course).

<table>
<thead>
<tr>
<th>doc_name</th>
<th>tuple_id</th>
<th>p_id</th>
<th>ord.</th>
<th>col1</th>
<th>col2</th>
<th>col3</th>
</tr>
</thead>
<tbody>
<tr>
<td>course.xml</td>
<td>3.0</td>
<td>2.0</td>
<td>1</td>
<td>section1</td>
<td>H1</td>
<td>Dr. Lin</td>
</tr>
<tr>
<td>course.xml</td>
<td>4.0</td>
<td>2.0</td>
<td>2</td>
<td>section2</td>
<td>H2</td>
<td>Dr. Dean</td>
</tr>
<tr>
<td>course.xml</td>
<td>6.0</td>
<td>5.0</td>
<td>1</td>
<td>section3</td>
<td>H1</td>
<td>Dr. Hanks</td>
</tr>
</tbody>
</table>

Table 6: The table_3_1 (section)

<table>
<thead>
<tr>
<th>doc_name</th>
<th>o_id</th>
<th>p_o_id</th>
<th>ord</th>
<th>col</th>
<th>col2</th>
<th>col3</th>
</tr>
</thead>
<tbody>
<tr>
<td>course.xml</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>section1</td>
<td>H1</td>
<td>Dr. Lin</td>
</tr>
<tr>
<td>course.xml</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>section2</td>
<td>H2</td>
<td>Dr. Dean</td>
</tr>
<tr>
<td>course.xml</td>
<td>6</td>
<td>5</td>
<td>1</td>
<td>section3</td>
<td>H1</td>
<td>Dr. Hanks</td>
</tr>
<tr>
<td>test.xml</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>h1</td>
<td>j1</td>
<td></td>
</tr>
<tr>
<td>test.xml</td>
<td>6</td>
<td>3</td>
<td>2</td>
<td>h2</td>
<td>j2</td>
<td></td>
</tr>
</tbody>
</table>

Table 7: Two different XML documents share the table_3_1.

<table>
<thead>
<tr>
<th>Query</th>
<th>Xregion</th>
<th>Xparent</th>
<th>Edge</th>
<th>Xrel</th>
<th>#tuples returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQ1</td>
<td>20</td>
<td>202</td>
<td>598</td>
<td>18</td>
<td>185</td>
</tr>
<tr>
<td>SQ2</td>
<td>47</td>
<td>387</td>
<td>21685</td>
<td>49</td>
<td>618</td>
</tr>
<tr>
<td>SQ3</td>
<td>73</td>
<td>705</td>
<td>709</td>
<td>658</td>
<td>750</td>
</tr>
<tr>
<td>SQ4</td>
<td>96</td>
<td>930</td>
<td>29599</td>
<td>876</td>
<td>951</td>
</tr>
<tr>
<td>SQ5</td>
<td>10</td>
<td>33</td>
<td>64</td>
<td>116</td>
<td>4</td>
</tr>
<tr>
<td>SQ6</td>
<td>29</td>
<td>427</td>
<td>183</td>
<td>3952</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 9: Logical I/O blocks for querying the SHAKS using Xregion, XParent, Xrel and Edge