University of Alberta

XREGION: A STRUCTURE-BASED APPROACH TO STORING XML DATA IN RELATIONAL DATABASES

by

Meng Xue 

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 2004

Your file Votre référence
ISBN: 0-612-95884-1
Our file Notre référence
ISBN: 0-612-95884-1

To My Family

Acknowledgements

I would like to express my gratitude to my supervisor Prof. Li-Yan Yuan, for his unending guidance, advice, and motivation throughout the thesis research.

I would like to thank Dr. Mario A. Nascimento and Dr. Marek Reformat, for their valuable suggestions that improved the thesis.

To my dear husband Lin, thank you for your love, encouragement and support. Special thanks to my parents for their continuous love and support.

I would also like to thank Reza Sherkat, Stanley Oliveira, Gabriela Moise and all other lab mates for creating a pleasant environment in the database systems laboratory.

Special thanks to all my friends for various help that they have given me.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

As a markup language designed for describing data, XML is becoming a popular medium, and a major standard, for data representation over Internet. With the development of Web applications, increasing amount of data are now available in XML format, and the sizes of XML documents are growing very quickly. It is imperative to store and query these XML documents efficiently in order to exploit the full power of this new technology.

The motivation for our research is the development of an XML repository system that is capable of storing a large number of heterogeneous XML documents, which are well-formed but have no DTD (Document Type Definition) or for which the DTDs 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 reconstructed in the desired XML format, i.e., XML result publishing.

One of the XML-to-Relation mapping techniques, called generic mapping, involves designing relational database schemas for XML documents without the knowledge of DTD or XML Schema information. Multiple generic mapping techniques have been studied—for example, Edge mapping [12] and path-based mappings [22] [28] [14].

The basic idea behind 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 a relational database, query performances of existing generic mapping approaches are still far from satisfactory, especially for large XML documents.

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

This thesis focuses on generic mapping approaches that are able to lower the fragmentation degree of XML data in relations, and our goal is to improve query performance.

1.2 Contributions

In this thesis, we propose a new generic mapping technique, called *Xregion*, to store XML data in relational databases. Our solution for reducing the frag-

mentation is simple, but very effective. We first partition an XML document into several disjoint regions according to the cardinalities of node occurrences, and we then store these regions, including their parent information, into separate tables. An example for our mapping approach is given below.

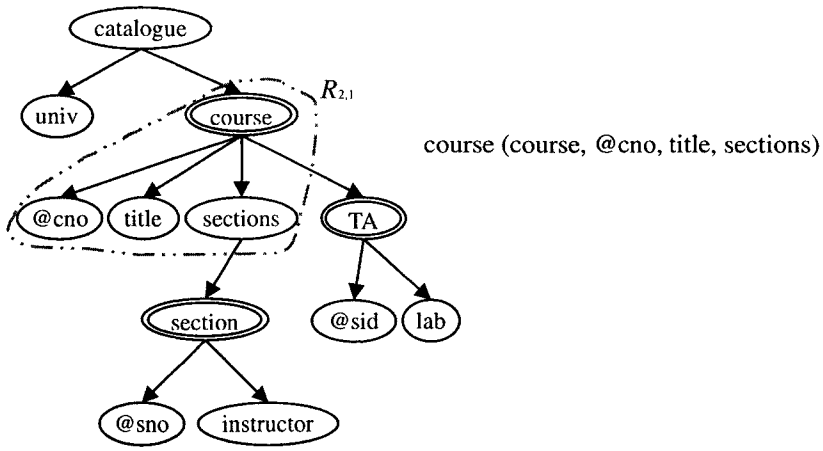


Figure 1.1: An example for Xregion.

The graph in Figure 1.1, called an XML structure tree, describes the structure of a sample XML document for a university registration system. Each ellipse represents a distinct node—identified by its path—in the document, while a double ellipse represents a node with repeated occurrences under its parent node, i.e., a 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.

The above structure tree is then partitioned into four disjoint regions by three set-valued nodes, e.g., "course", "section" and "TA". Each region is mapped into a separate table, and each distinct node in a region is represented by a separate column in the table for the region. For example, R_{21} in Figure 1.1 is the region represented by the "course" node, and its corresponding relation schema is

course_table (course, @cno, title, sections).

The advantages to partitioning and storing 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 as 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. So in our system, XML documents are decomposed according to the nested level of the data, rather than 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[14] and Edge[12]. We have conducted extensive experiments using the above approaches to store three XML data sets into an Oracle database, and then compare their performance by evaluating eighteen typical queries. Experimental results demonstrate that the proposed approach dramatically improves the performance of query evaluation on the XML data over the existing generic mapping technologies, by one or two orders of magnitude. The improvement is particularly striking for large XML documents.

In summary, the contributions of the thesis are as follows. We propose a new generic XML-to-Relation mapping approach, called Xregion. Experiments show that Xregion outperforms existing generic mapping techniques (such as Edge mapping and XParent) significantly—especially for complex queries involving multiple paths and conditions. This approach enjoys high performance even with large XML documents.

We have also implemented an XML importing system based on the proposed approach. This system can create database schema for any well-formed

XML document and load its data into the database automatically without user interventions. No DTD information is required for this mapping.

1.3 Overview

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

Chapter 2

Background and Related Work

2.1 An Overview of XML Documents

Extensible Markup Language (XML)[1], 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 2.1 is an example XML document describing course information. Each pair of matching tags in an XML document, together with the enclosed XML data, forms an XML **element**. The **element type** of an element is identified by its corresponding tag name. Any XML element can contain other elements as sub elements. An element can also contain **attributes**, which are additional information included as part of the start tag of the element, and include attribute values in quotes, for example, <course cno="291">. Different from some other markup languages, XML tags are not predefined; users need to define their own tags, which makes XML more flexible and adaptable for information identification.

Despite the flexibility in content, an XML document must be syntactically correct in format, i.e., **well-formed**. There are several rules that determine whether an XML document is well-formed. First, in an XML document, all tags must have a matching end tag or be themselves self-

```
<catalogue>
  <univ>ABC</univ>
  <course cno="291">
    <title>Database Systems</title>
    <sections>
      <section sno="H1" >
        <instructor>Dr. Lin</instructor>
      </section>
      <section sno="H2" >
        <instructor>Dr. Dean</instructor>
      </section>
    </sections>
  </course>
  <course cno="539">
    <title>Programming</title>
    <sections>
      <section sno="H1" >
       <instructor>Dr. Hanks</instructor>
      </section>
    </sections>
    <TA sid="123"> <lab>D01</lab> </TA>
    <TA sid="112"> <lab>D02</lab> </TA>
  </course>
</catalogue>
```

Figure 2.1: An example XML document for university courses

ending. Overlapping tags are not permitted, therefore all sub elements must be properly nested within their parent element. Furthermore, all XML documents must contain a single tag pair to define the root element, e.g. the <catalogue>....< /catalogue> in the example XML document.

An XML document 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 thesis, we focus on the problem for storing XML documents in relational databases without the knowledge of the DTDs or XML schemas.

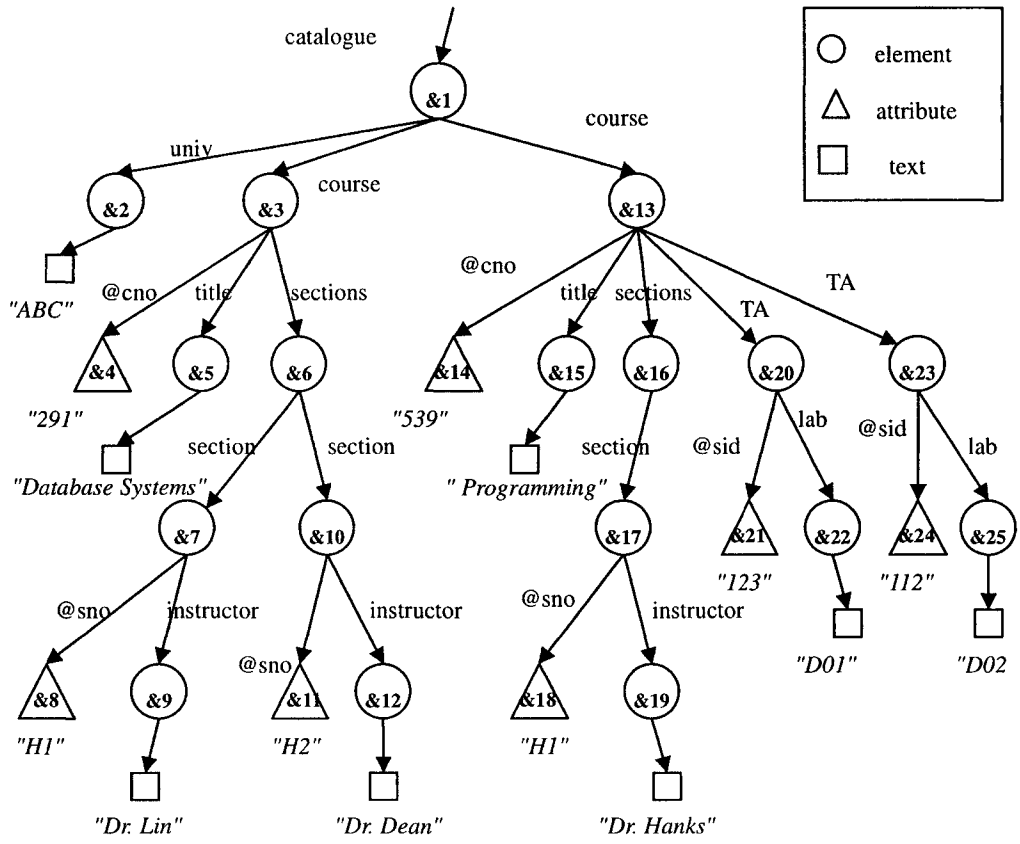


Figure 2.2: An XML tree (XPath model) for the example XML document

Because of the nested structure of XML, a well-formed XML document can be visualized as an ordered tree structure. Figure 2.2 is a graphic depiction

of an XML tree, which conforms to the definition of XPath[11] model of the World Wide Web Consortium (W3C). *Element* node, *text* node and *attribute* node are the three major node types of the XPath model, which models the element, attribute and text content of an element, respectively.

XPath (XML Path Language) is a query language defined by W3C; its syntax and semantics are used in many proposed XML query languages. In this thesis, for simplicity, all example XML queries are written in XPath syntax.

XPath describes the navigation on an XML tree by *path*s. The path associated with each node of an XML document is a sequence of tags starting from the root element to the node itself, e.g., the path of "TA" node in the above example XML document is */catalogue/course/TA* (Figure 2.2). A *predicate*, which may contain arithmetic, logical expressions or comparisons, can be added to any step of a given path as a query condition. The following are two simple XML queries using XPath expressions, and their expected results.

Example 1 Find the course number of the course with a title "Database Systems".

Q1: /catalogue/course[title="Database Systems"]/@cno

The expected query result is a string value:

```
<results>291</results>
```

Example 2 List the information of courses instructed by Dr. Dean.

Q2: //course[sections/section/instructor/name="Dr. Dean"]

The expected query result is a "course" element:

```
<results>
  <course cno="291">
    <title>Database Systems</title>
    <sections>
      <section sno="H1" >
        <instructor>Dr. Lin</instructor>
      </section>
      <section sno="H2" >
        <instructor>Dr. Dean</instructor>
      </section>
```

```
    </sections>
  </course>
</results>
```

2.2 Related Work

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. With the trend towards increasing the amount and size of XML documents, it is imperative to store and query them efficiently in order to exploit the full power of this new technology.

One of the methods used for XML storage and retrieval is the employment of Relational Database Management Systems (RDBMS). Constructing an XML storage system using existing RDBMSs takes advantage of the mature technologies already provided in database systems, such as concurrency control, query optimization and indexing techniques. Furthermore, the relational database systems are in wide-spread use, and a majority of web applications are already built on RDBMSs. Storing XML data in a RDBMS makes it possible to query seamlessly across XML and existing relational data. Given these advantages, we believe integrating XML and RDBMSs will be a promising alternative to other approaches for storing XML documents.

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 support 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 XML storage schema is crucial to the performance of query processing and result publishing, because it stipulates how XML data are

stored in the underlying relational database systems.

2.3 Generic Mapping Revisited

To further understand XML relational storage models and their effectiveness in terms of query processing, we will now look at existing generic mapping approaches which have been described in the literature. Edge mapping [12] will be discussed first, followed by three other path-based mapping approaches— XParent [14], XRel [28] and Monet [22].

All the above mapping approaches transform an XML document into relational data through an XML tree or graph. The Edge approach was first proposed by Florescu and Kossmann in 1999 [12], and has been viewed as a representative of generic mapping approaches. It uses a single relational table to store all data of an XML document, with one record in the table representing one edge on the graph of the document. Edge mapping is simple and capable of storing any semi-structured document that can be modeled as a graph. The simplicity, however, leads to inefficiency because of the additional joins and selections required to restore the hierarchical structures of XML documents for query evaluations. For example, a very simple XPath expression $/a/b/c$ requires three selections and two joins.

Several alternative approaches, which store additional path information in relations (the Monet model [22], XRel [28] and XParent [14]), were proposed to improve query performance. Here we categorize them as path-based mapping approaches. Storing path expressions explicitly in the database schemas facilitates queries such as simple node selections, but not, however, the complicated ones.

In this section, we will give a brief description of how these systems translate the XML queries into the SQL statements, and discuss the operations involved in these statements. Detailed experimental data will be presented in

2.3.1 Edge Based Mapping

Florescu and Kossmann [12] proposed the Edge approach to model an XML document as a set of atomic structure units, which are edges on the data graph, and which store each unit as a tuple in a relational table of a RDBMS. They represented an XML document as an ordered and directed graph, in which every node is assigned an identifier *oid* and each edge is explicitly labeled by the name of incoming element type or attribute.

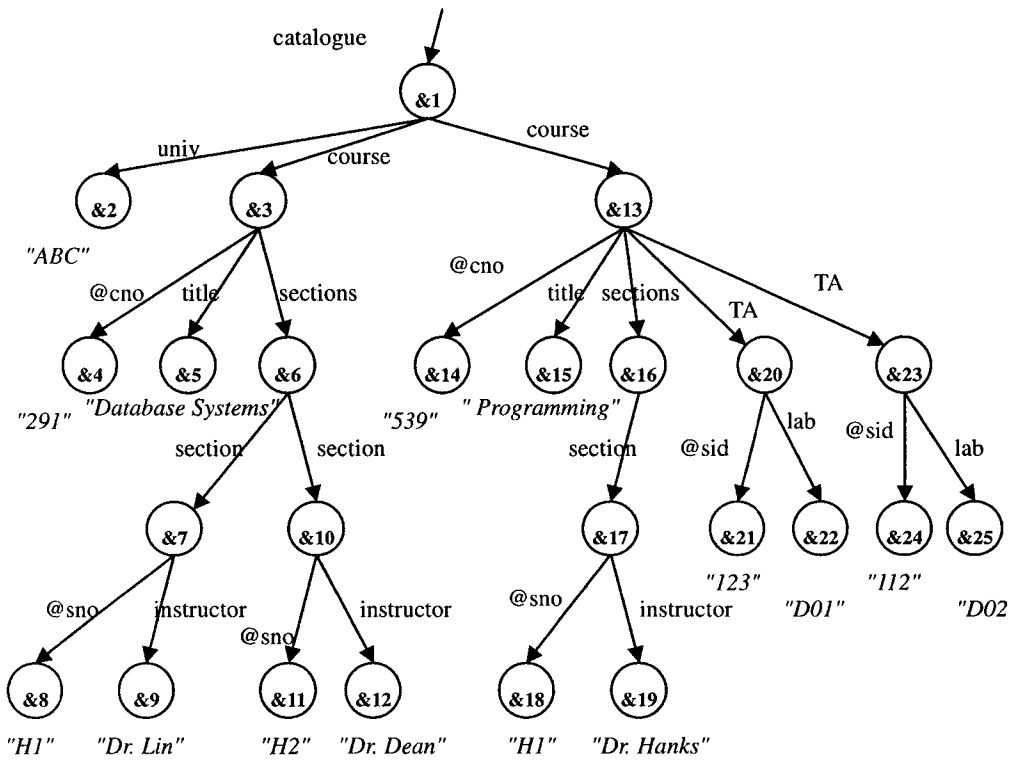


Figure 2.3: An XML data graph of Edge mapping

All edges of an XML data graph are stored in a single table called the **Edge** table, which has the following structure.

```
Edge (source, ordinal, target, label, flag, 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 *oid*s 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 leaf node with a value ("string" or "int"). The data graph for the example XML document is shown in Figure 2.3; Table 2.1 is its corresponding Edge table.

Source	Ordinal	Target	Label	Flag	Value
&0	1	&1	catalogue	ref	
&1	1	&2	univ	string	ABC
&1	1	&3	course	ref	
&3	1	&4	@cno	string	291
&3	1	&5	title	string	Database Systems
&3	1	&6	sections	ref	
&6	1	&7	section	ref	
&7	1	&8	@sno	string	H1
&7	1	&9	instructor	string	Dr. Lin
&6	2	&10	section	ref	
&10	1	&11	@sno	string	H2
&10	1	&12	instructor	string	Dr. Dean
&1	2	&13	course	ref	
&13	1	&14	@cno	string	539
&13	1	&15	title	string	Programming
&13	1	&16	sections	ref	
&16	1	&17	section	ref	
&17	1	&18	@sno	string	H1
&17	1	&19	instructor	string	Dr. Hanks
&13	1	&20	TA	ref	
&20	1	&21	@sid	string	123
&20	1	&22	lab	string	D01
&13	2	&23	TA	ref	
&23	1	&24	@sid	string	112
&23	1	&25	lab	string	D02

Table 2.1: A relational storage example using the Edge approach.

Independent of XML DTDs, the Edge mapping approach can be applied to a wide range of XML documents or other semi-structured documents that have arbitrary graph structures. However, such a decomposition method makes the query evaluation very inefficient, in that a number of self-join operations

are needed to restore the hierarchical structure of the XML data at query processing time, due to the high fragmentation of the data in relations. The following example illustrates how XML queries are translated into SQL query statements for the Edge approach.

Example 3 List the information of TAs for all courses.

Q1:/catalogue/course/TA

SQL 1 A translated SQL query statement for Q1 using Edge.

```
SELECT  edge3.tgt
FROM    edge edge1, edge edge2, edge edge3
WHERE   edge1.label='catalogue'
AND     edge2.label='course'
AND     edge3.label='TA'
AND     edge1.tgt=edge2.src
AND     edge2.tgt=edge3.src;
```

For such a simple node selection query, the Edge approach requires three selections on the *label* field and two self-joins in order to ensure the following edge connection: *catalogue*→*course*→*TA*. The number of joins and selections involved are determined by the depth of the desired nodes. Because all data are stored in the Edge table, a large amount of data irrelevant to a query has to be scanned at query processing time. Furthermore, since the size of the Edge table is proportional to the size of the input XML document, those join operations will be very expensive for querying a large XML document.

In addition to the effect on query processing, the process for constructing the XML query result (XML result publishing) is also affected by this kind of data fragmentation. For example, at least two more joins and two label selections will be involved for the result publishing of Q1, whose expected query result is shown below.

The expected query result of Q1:

```
<result>
    <TA sid="123"> <lab>D01</lab> </TA>
    <TA sid="112"> <lab>D02</lab> </TA>
</result>
```

In order to reduce the volume of data irrelevant to a query that has to be processed during querying time, Florescu and Kossmann [12] also proposed a variant of Edge mapping, called the Binary approach. For the Binary mapping schema, they propose the creation of separate tables for each different label in the XML data graph, and then store all edges with the same label in one table, identified by the label name. Each Binary table has the following structure:

Binary_{label}(source, ordinal, target, flag, value).

Since some existing database systems still have limits on the total number of tables permitted, the Binary approach is probably not feasible for storing large collections of XML documents. Based on the above observations, we decide not to use the Binary approach in our experiment, which will be discussed in Chapter 7.

2.3.2 XParent

XParent [14] is a four-table path-based mapping system, which uses fixed schemas to store various XML documents according to the XPath model. **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 identified by its node identifier *Did* in the XML tree. The parent-child relationships among nodes are stored in the **DataPath** table by recording the pairs of *Did*s of parent and child nodes. Another table, the **LabelPath** table, stores all distinct label-paths in the *Path* field, and their depth in the *Len* field. These four relational tables are as follows:

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

Table 2.2 shows the LabelPath table for the example XML document (Figure 2.1); other tables are listed in Figure 2.4.

LabelPath table

Id	Len	Path
1	1	/catalogue
2	2	/catalogue/course
3	3	/catalogue/course/@cno
4	3	/catalogue/course/title
5	3	/catalogue/course/sections
6	4	/catalogue/course/sections/section
7	5	/catalogue/course/sections/section/@sno
8	5	/catalogue/course/sections/section/instructor
9	3	/catalogue/course/TA
10	4	/catalogue/course/TA/@sid
11	4	/catalogue/course/TA/lab
12	2	/catalogue/univ

Table 2.2: The XParent LabelPath table for the example XML document

Element table

PathID	Ordinal	Did
1	1	&1
12	1	&2
2	1	&3
3	1	&4
4	1	&5
5	1	&6
6	1	&7
7	1	&8
8	1	&9
6	2	&10
7	1	&11
8	1	&12
2	2	&13
3	1	&14
4	1	&15
5	1	&16
6	1	&17
7	1	&18
8	1	&19
9	1	&20
10	1	&21
11	1	&22
9	2	&23
10	1	&24
11	1	&25

DataPath table

ParentID	ChildID
&0	&1
&1	&2
&1	&3
&3	&4
&3	&5
&3	&6
&6	&7
&7	&8
&7	&9
&6	&10
&10	&11
&10	&12
&1	&13
&13	&14
&13	&15
&13	&16
&16	&17
&17	&18
&17	&19
&13	&20
&20	&21
&20	&22
&13	&23
&23	&24
&23	&25

Data table

PathID	Ordinal	Did	Value
12	1	&2	ABC
3	1	&4	291
4	1	&5	Database Systems
7	1	&8	H1
8	1	&9	Dr. Lin
7	1	&11	H2
8	1	&12	Dr. Dean
3	1	&14	539
4	1	&15	Programming
7	1	&18	H1
8	1	&19	Dr. Hanks
10	1	&21	123
11	1	&22	D01
10	1	&24	112
11	1	&25	D02

Figure 2.4: XParent schemas for the example XML document

Compared with the Edge mapping approach, 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, such as the query
/catalogue/course[sections/section/instructor="Dr. Hanks"]/TA,
a number of joins or self-joins are still needed to check node connections.

Two paths are involved in the above query, */catalogue/course/TA*, pointing to the desired nodes, and */catalogue/course/sections/section/instructor= "Dr. Hanks"*, specifying the condition. Although, with the aid of the LabelPath table, XParent can easily locate the nodes represented by those two paths, several joins are required to ensure that the pairs of nodes, "TA" and "Instructor", are connected by the same "course" nodes, which are their nearest common ancestor. The nearest common ancestor of two nodes is the node that connects them.

Figure 2.5 illustrates the tracing route of this ancestor searching process. Because only parent-child relationships are stored in XParent, three joins are needed to locate the courses that Dr. Hanks taught, as well as an additional join operation to find the TAs for the same courses.

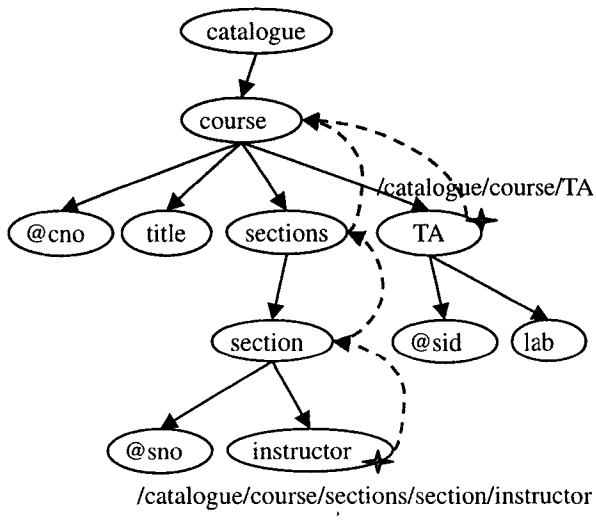


Figure 2.5: Ancestor tracing route using XParent

The number of joins needed in XParent for queries such as the one shown above is determined by the distance between related paths. We define the distance of two nodes as the sum of their distances from the nearest common ancestor, and the distance of a node from its ancestor as the depth difference between them. In the above example, the distance between the "instructor" node and the "TA" node is four, and the number of required joins is five.

The main reason for this inefficiency is that XML data are scattered in relation schemas with a very high fragmentation degree. The hierarchical structure of an XML document is flattened to binary parent-child relationships. This type of mapping strategy makes searching ancestor information very expensive.

As an example, SQL 2 demonstrates how XParent processes the above XML query.

Example 4 Find the TAs who work with Dr. Hanks.
Q2:/catalogue/course[sections/section/instructor="Dr. Hanks"]/TA
SQL 2: 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;
```

2.3.3 XRel

XRel, proposed by Yoshikawa and Amagasa et al. [28], is a generic mapping approach that 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 2.3 are (0, 448) and (315, 323), respectively. The **Element** table, **Attribute** table and **Text** table are created for storing element nodes, attributes and text contents, respectively. As in the Edge and XParent approaches, each tuple in these relations represents one node in the XML tree. All path expressions are stored in the **Path** table. The basic XRel schemas are as follows:

Path(PathID, Pathexp)
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 the authors introduced a new format for representing paths. They proposed the use of 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. For example, a simple XPath expression, */book//price*, which queries the price of all kinds of books, can be translated correctly into an SQL query with a condition clause

```
WHERE pathexp LIKE '#/book#%/price'
```

to find all tuples with the expected path expressions, such as '#/book#/price' (/book/price) or '#/book#/novel#/price' (/book/novel/price). However, if only one character is used as a step separator in the path string, the SQL clause

```
WHERE path LIKE '/book%/price'
```

may return some unwanted tuples, such as '/bookcase/color/price', '/bookshelf/price', and so on.

In XRel schemas, the containment relationships among nodes in an XML document can be captured by comparisons between start and end positions. Therefore, XRel sometimes does not need to verify all the intermediate edge connections, one by one, between two nodes, e.g., *node* $a \rightarrow b \rightarrow c \rightarrow$ *node* d, and needs only to check whether one node (d) is reachable from the other node (a). Thus, XRel will use fewer 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 deteriorates, especially for large documents. Different from other approaches, the join operations involved in checking containments are non-equijoins, which are executed as hash joins at runtime in most RDBMSs. Our experiments showed that, for some complicated queries, a Full Table Scan (FTS) and Merge Join Cartesian are involved when querying data in XRel schemas. An example for processing query Q2 is shown below.

Example 5 Find the TAs who work with Dr. Hanks.
Q2:/catalogue/course[sections/section/instructor="Dr. Hanks"]/TA
SQL 3: A translated SQL query statement for Q2 using Xrel.

```
SELECT  eta.start, eta.end
FROM    element eta, element ecrs, text inst,
        path pta, path pinst, path pcrs,
WHERE   pinst.pathexp =
             '#/catalogue#/course#/sections#/section#/instructor'
        AND pta.pathexp = '#/catalogue#/course#/TA'
        AND pcrs.pathexp = '#/catalogue#/course'
        AND inst.pathid = pinst.pathid
        AND eta.pathid  = pta.pathid
        AND ecrs.pathid = pcrs.pathid
        AND inst.value  = 'Dr. Hanks'
        AND ecrs.start < inst.start
```

```
AND ecrs.end  > inst.end
AND ecrs.start < eta.start
AND ecrs.end > eta.end;
```

Another drawback of the XRel mapping approach is the use of absolute positions in the document to describe a node. This makes some update operations very expensive, although it is effective for XML query results publishing. A single change of value might result in the update of all the relational tuples of a document, which is not trivial for a large XML document. For example, if the value of an attribute node is changed from "291" to "c291", the end positions of all nodes in the example document are one byte forward.

2.3.4 Monet Model

Monet [22] is another path based XML relational storage model proposed by Schmidt et al. The basic idea is similar to that of the Edge mapping, which identifies parent-child relations from the XML data graph. At the mapping stage, a different approach is applied, which creates 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 relatively small, compared to the Edge approach. However, as with the Binary approach to Edge mapping, this approach might not be viable for large collections of XML documents due to the limit on the total number of tables in database systems. For example, the Monet approach created 2587 tables for a single XML document for Webster's Dictionary [22].

2.4 Other Mapping Approaches

In addition to generic mapping approaches, which consider situations in which DTDs are not available, several other XML-to-Relation mapping techniques have also recently been investigated.

Some of these approaches use XML DTDs or XML schemas information to generate relational schemas [25] [15][18]. Considering that numerous sophisticated Web applications are based on flexible usage of XML documents and the demand for storage of various kinds of XML documents (which are well-formed but may have no associated DTDs or whose DTDs are not known at the processing stage), obviously, mapping strategies using DTDs or XML schemas are not appropriate for storing a large number of such structurally-variant XML documents.

Various other approaches, such as the STORED system [10] and a cost based system LegoDB [6], design database schema according to the analysis and statistics on frequent structure and query work load. However, 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 workload, are also required.

Cooper, Sample, Franklin et al. pursue a different direction to improve the performance of querying XML data in databases. To facilitate the navigation and selection of nodes on the XML trees, they build a special index, Index Fabric [9], on top of RDBMSs, for storing path information (raw path), instead of using B-tree indexes. Their experimental results show that the fast index improves performance, but mainly for refined paths, which are specialized paths for tuning frequently occurring queries.

2.5 Chapter Summary

In this chapter, we continued to be motivated by the importance of designing database schemas for storing and querying XML data in RDBMSs. In the case that XML documents are very large, or occur in huge numbers, it is imperative to convert those data to a format where they can be retrieved effectively. Therefore, we analyzed the existing generic mapping techniques in

terms of effectiveness of query processing.

The generic mapping techniques do not rely on the DTD or XML Schema information to design relational database schema for XML documents, and can be applied to a wide range of XML documents. Most existing generic mapping techniques create relational schemas based on a local granular structure, such as an edge or a node in the XML tree, so that their relational schemas are general enough for storing arbitrary XML documents. However, this simplicity also results in high fragmentation of XML data in relations, which makes query processing and XML query result publishing very expensive.

Chapter 3

Xregion Data Model

In this chapter, we formally define the XML data tree, XML structure tree, region and nested level, which make up 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 slightly different from the XPath tree models [11] discussed in Chapter 2. We model the text of an element as the value of that element node, rather than as a separate text node on the tree. For this reason, we adapt the formal definition of an XML data tree from XNF [3], by modifying the total function "ele" and adding a new function "val".

Definition 1 [3] 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.

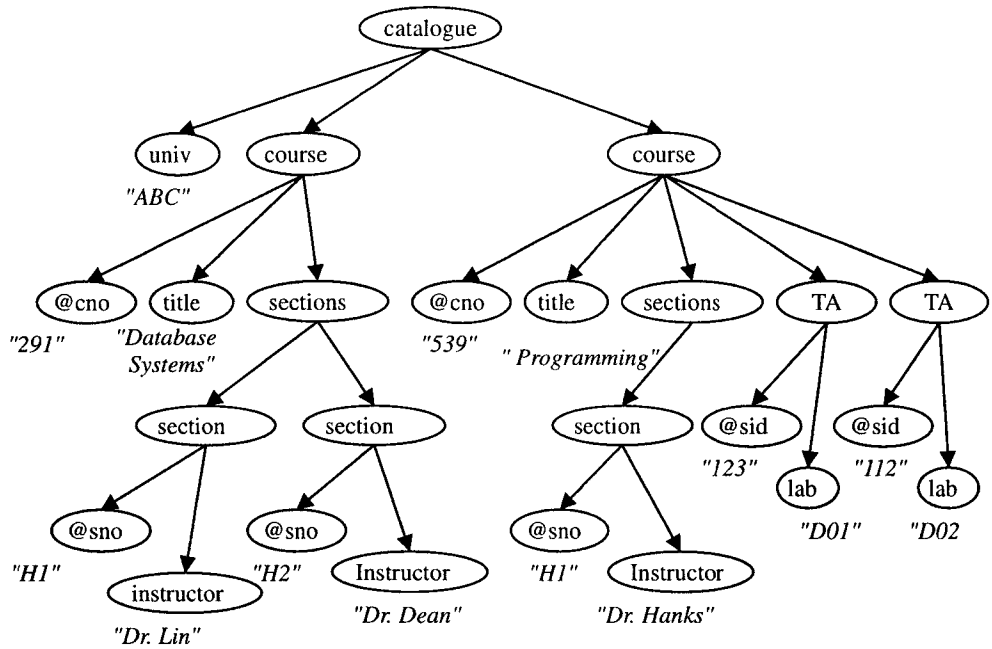


Figure 3.1: An XML data tree for the example XML document

- $lab: V \rightarrow El.$
- $ele: V \rightarrow V^*$
- $val: V \rightarrow Str \cup null$
- $attr$ is a partial function $V \times Attr \rightarrow Str$. For each $v \in V$, the set $\{al \in Att \mid attr(v, al)\ is\ defined\}$ 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 3.1 is a graphic depiction of the data tree for the sample XML document in Figure 2.1.

3.2 XML Structure Tree

An XML document contains both meta data and data itself, and its meta data, including all paths and other information, can be described by the structure tree.

The structure tree summarizes the hierarchical structure of an XML document by combining repeating structures in the XML data tree and marking all set-valued nodes explicitly. All nodes that have the same path information in a data tree are represented by exactly one node in the structure tree of the document.

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

- $Vs{:}Pt \rightarrow El \cup Attr$.
- $multi{:}Pt \rightarrow 1|n$.
- $r \in Vs(Pt)$ is the root of XML structure tree S.

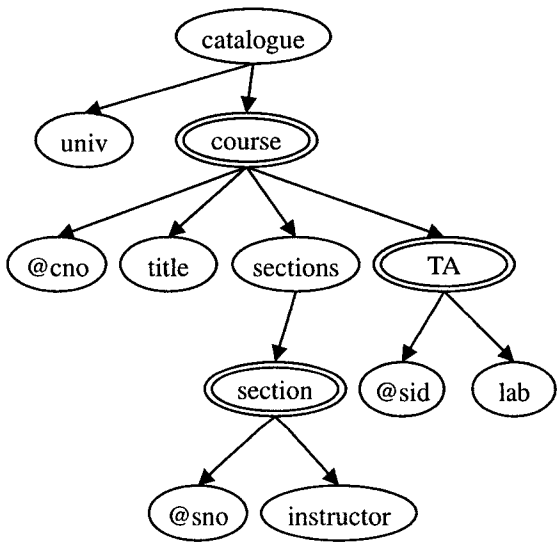


Figure 3.2: An XML Structure tree

multi(p_{t1}) shows the maximum cardinality of nodes identified by a path $p_{t1} \in Pt$, in the data tree. The hierarchical structure of XML data is captured by paths. Figure 3.2 shows an XML structure tree for the example XML document. For simplicity, all multi-valued nodes that occur repeatedly under their parent nodes, $\{p_{t1} \in Pt | multi(p_{t1})=\text{n}\}$, are identified by a double ellipse in this structure tree.

For example, each course 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 structure information for an XML document includes not just path information, but also the 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 S is a subtree R such that

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

2. all the subtrees of R rooted with a 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). Therefore, we can map all nodes in a region into one relation with every node in this region represented by a separate column in the relation. Another feature of a region is that the ancestors and cousins of all nodes in a region belong to the same region. Consequently, storing all the regions in separate tables may significantly reduce the number of joins needed for query evaluation. Figure 3.3 shows all the regions for the sample document, and the corresponding relations for storing regions. For example, the relation for the region "course" is:

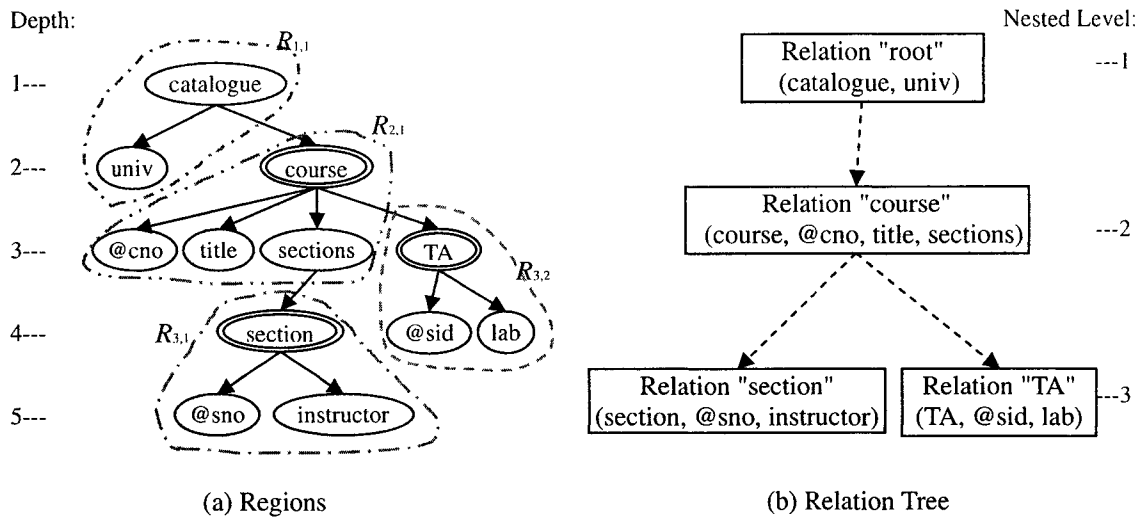
Relation_course (course, @cno, title, sections)

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 in one relation, a region tree is also called a relation tree. Figure 3.3 (b) describes the relation tree for the sample document.

Given a region tree Tr, the nested level of a region is then defined as the depth of its corresponding region node in the region tree.

All nodes in a given region belong to the same nested level, although they are at different depths of the XML structure tree. For example, in Figure 3.3,



(a) Regions

(b) Relation Tree

Figure 3.3: An example for *regions*. The structure tree is divided into four regions by its three set-valued nodes *course*, *section* and *TA*.

while the node "section","instructor" and "TA" are at different depths in the structure tree, they are at the same nested level —nested level 3.

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 leftmost as 1. Consider Figure 3.3 (b) again. Relation "root" is labeled as $R_{1,1}$, Relation "course" as $R_{2,1}$, Relation "section" as $R_{3,1}$, and Relation "TA" as $R_{3,2}$.

3.5 Region Instance

Given an XML data tree T, once all regions of T are identified, T can then be partitioned into a set of subtrees, each of which is an instantiation of a region. More formally, we define a region instance as follows:

Definition 4 Let T be an XML data tree, and R be a region of the XML structure tree of S. Then, an instance I of R is defined as a subtree $I(T, R)$ of T, such that each node (or an edge) in $I(T, R)$ is an instantiation of a node (or an edge) in R.

On the data tree T, each occurrence of the root node of region R, and all its non-set-valued descendants forms an instance of R. For example, the I_{course_1} and I_{course_2}, marked on the XML data tree shown in Figure 3.4, are two instances of the region "course".

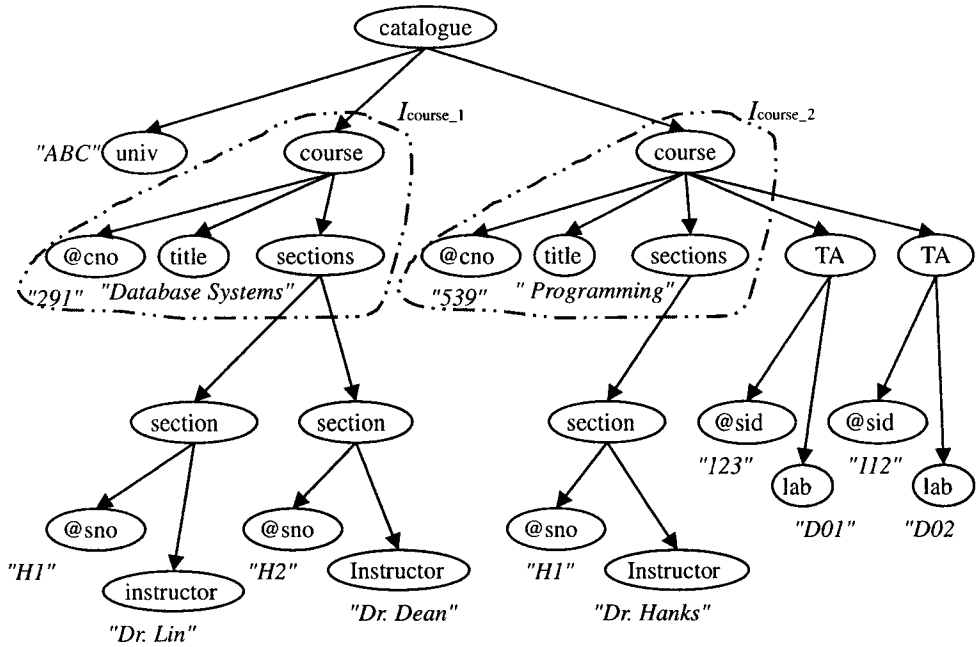


Figure 3.4: Two instances of region "course".

Because the XML structure tree S characterizes the meta data of T, it is not difficult to see that T can be partitioned into the set of region instances. Furthermore, since all nodes (except the region root) in a region, are single-valued, each instance of a region can be stored as a tuple in the table for the region.

By partitioning an XML document into a set of region instances, the document can be then stored in a set of relational tables, one for each region.

3.6 Chapter Summary

In this chapter, we formally defined the data model used in our proposed XML relational mapping approach, Xregion.

An XML data graph is a tree structure that represents the topological structure and data of an XML document. The XML structure tree of an XML document is built by summarizing the structure of its XML data tree. It represents the complete hierarchical structure of the document, and identifies all multi-valued nodes in the document.

In the process of creating the database schema for an XML document, its structure tree will be partitioned into disjoint regions, with each region represented by a set-valued node. Each region is stored in a separate table, and every node within the region is represented by a separate field. Each occurrence of a region structure in the XML data tree is an instance of the region and is stored as a tuple in the table that represents the region.

Chapter 4

Xregion Storage Schema

In this chapter, we present the storage schema for Xregion, which specifies how to store the set of all region instances of any given XML document into a relational database system.

The basic idea of Xregion is to store all the instances of one region into one table. Because an XML database system will be used to store all types of XML documents, we shall establish a schema to assign an existing table (or to create a new table if necessary) to store a region from any given XML document.

4.1 Basic Database Schema for Xregion

The database schema for Xregion consists of one *meta table*, for storing all the meta information, such as edges, paths and document identifiers, and a limited number of *data tables*, one for a region.

Since a region instance will be stored as a tuple, i.e., a list of node values, all structured information in a region is not preserved in data tables. Thus, we use the meta table to store all such information.

Unlike all other generic mapping approaches, the Xregion storage schema separates the structured information from the value information. Rather than explicitly storing the edge and path information for each and every region

instance, we use the meta table to designate the structured information for all region instances. The advantages are two-fold. First, by using a meta table to store the structure information, we eliminate unnecessary redundancy. Second, without the structure information, one data table can be used to store different regions from different XML documents.

For simplicity, we assume that the number of nodes in any region is limited to n, say $n = 20$. Should the number of nodes exceed this limit, we can either increase the number of columns of the table for the region, or 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 region can be represented by a set of tuples of n columns, one for the value of each node in the region.

To uniquely identify each tuple in the region table, 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 the 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 a given tuple in a region.

In order to preserve the order among all sibling nodes, we will use one 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 need a column to store the document name. The *tuple_id*, together with the *doc_name* field, will be the primary key for each data table.

In summary, each table designated as *table_i_j* is defined as a table with $n + 4$ columns, that is,

table_i_j(doc_name, tuple_id, p_id, ordinal, col_1, ..., col_{n-1}, 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 of the region root, *ordinal* is used to store the ordinal position of the tuple, and col_i, for $1 \leq i \leq n$, is 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. Therefore, the structure of the root table is:

table_1_1(doc_name, tuple_id, col_1, ..., col_{n-1}, 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 mapped into a distinct column of its region table, for each path, we keep the names of its region table, its parent region table and its column name, as a tuple in the *meta_table*. We shall also store, for each tuple, the name of the XML document.

In summary, the meta table consists of five columns and its structure is:

meta_table(doc_name, path, table_name, col_name, 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, the Xregion will first construct the XML structure tree. Once the structure tree is constructed, the Xregion will assign (or create if necessary) a table schema for each region, and then store the table assignment and path information in the meta table.

After the meta table is populated, by traversing the XML data tree in depth-first-order the Xregion will partition the XML data tree into region instances and store them in appropriate tables.

All the data tables and meta table for the sample document described in Figure 2.1 are shown in Tables 4.1 - 4.5.

doc_name	Path	table_name	col_name	p_table
course.xml	/catalogue	table_1_1	col1	
course.xml	/catalogue/univ	table_1_1	col2	
course.xml	/catalogue/course	table_2_1	col1	table_1_1
course.xml	/catalogue/course/@cno	table_2_1	col2	table_1_1
course.xml	/catalogue/course/title	table_2_1	col3	table_1_1
course.xml	/catalogue/course/sections	table_2_1	col4	table_1_1
course.xml	/catalogue/course/sections/section	table_3_1	col1	table_2_1
course.xml	/catalogue/course/sections/section/@sno	table_3_1	col2	table_2_1
course.xml	/catalogue/course/sections/section/instructor	table_3_1	col3	table_2_1
course.xml	/catalogue/course/TA	table_3_2	col1	table_2_1
course.xml	/catalogue/course/TA/@sid	table_3_2	col2	table_2_1
course.xml	/catalogue/course/TA/lab	table_3_2	col3	table_2_1

Table 4.1: The meta_table.

The *meta_table* contains complete mapping information for all the element types and attributes of an XML documents. Each record of the *meta_table* is identified by a path and a document name. For example (Table 4.1), for the XML document "course.xml", the "TA" element node, its attribute "@sid" and sub element node "lab" are stored in *col1*, *col2* and *col3* fields of *table_3_2*, respectively, and their parent table is *table_2_1*.

Every data table stores all region instances of the region represented by

the table. For example, both instances of the "course" region, I_{course_1} and I_{course_2}, illustrated in Figure 3.4, are stored as two tuples in *table_2_1*, which is shown in Table 4.2.

doc_name	tuple_id	p_id	ordinal	col1	col2	col3	col4
course.xml	2.0	1	1	*course1*	291	Database System	*sections1*
course.xml	5.0	1	2	*course2*	539	programming	*sections2*

Table 4.2: Table_2_1(*course*).

In this example, the values of the *tuple_id* of all the region instances are assigned based on the document order. Figure 4.1 describes the tuple orders for the example XML document. The *tuple_id*, together with the *doc_name* field, serves as the primary key for each data table. The "p_id" and "doc_name" fields work like a foreign key referring to the parent table in the upper level.

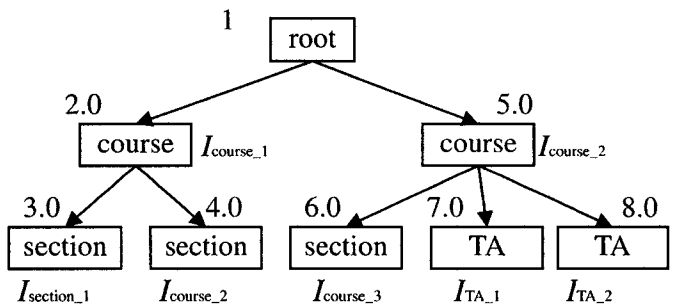


Figure 4.1: Tuple order.

The corresponding tables for region "section" and region "TA" are shown in Figure 4.3 and 4.4, respectively.

doc_name	tuple_id	p_id	ordinal	col1	col2	col3
course.xml	3.0	2.0	1	*section1*	H1	Dr. Lin
course.xml	4.0	2.0	2	*section2*	H2	Dr. Dean
course.xml	6.0	5.0	1	*section3*	H1	Dr. Hanks

Table 4.3: Table_3_1 (*section*)

doc_name	tuple_id	p_id	ordinal	col1	col2	col3
course.xml	7.0	5.0	1	*TA1*	123	D01
course.xml	8.0	5.0	2	*TA2*	112	D02

Table 4.4: Table_3_2 (*TA*)

The data table for the root region of the example XML document is shown in Figure 4.5.

doc_name	tuple_id	col1	col2
course.xml	1	*catalogue*	ABC

Table 4.5: Table_1_1 (*root*)

4.3 Discussion

4.3.1 Storing Different XML Documents

It is not feasible to create different schemas for each individual XML document in a system that stores a large number of XML documents, due to the overhead and system limitations.

Despite the heterogeneous structures of different XML documents, the Xregion uses one specified database schema to store all types of XML documents. This is largely due to the separation of the the value content and structured information in the Xregion schema.

Table 4.6 is a snapshot of the table, *table_3_1*, in which the data of "course.xml" and "test.xml" are stored.

4.3.2 Query Evaluation

The *meta_table* of Xregion provides sufficient information to users to enable the development of a standard interface for query processing and XML query result publishing. The mapping information of all components of an XML

doc_name	o_id	p_id	ord	col1	col2	col3
...	...	...	...	...	..	
course.xml	3.0	2.0	1	*section1*	H1	Dr. Lin
course.xml	4.0	2.0	2	*section2*	H2	Dr. Dean
course.xml	6.0	5.0	1	*section3*	H1	Dr. Hanks
test.xml	4.0	3.0	1	h1	j1	
test.xml	6.0	3.0	2	h2	j2	
...	...	...	...	...	..	

Table 4.6: Two different XML documents share the table_3_1.

document is 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 an instance is stored in the same tuple with its parent, while 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. Compared with other existing generic mapping approaches, therefore, Xregion is considerably more efficient in evaluating queries. First, queries on nodes within a region are simplified to one or a limited number of selections on one region table, without the need for join operations, which are otherwise required by other existing mapping approaches. For example, SQL 1, 2 and 3 are translated SQL statements of Xregion, Edge and XParent, respectively, for the XML query given below.

Example 6 Given the example XML document in Figure 2.1, find the course title of the course with a course number "291".

Q1: /catalogue/course[@cno="291"]/title

SQL 1: A translated SQL query statement for Q3 using Xregion.

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

SQL 2 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 3 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 belong to the same "course" elements. For Xregion, the "@cno" and its corresponding "title" are stored in the same tuple as "course", therefore, only a single selection on "@cno" and a projection on "title" are required.

Furthermore, Xregion is more efficient than other approaches in searching

for ancestor information. Xregion transforms the hierarchical structure of an XML document into a relatively simple nested structure among regions (relations), which are normally much shallower than the structure tree. Therefore, the process for searching ancestor of nodes is converted to searching 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 4, 5 is a translated SQL statement of Xregion and XParent, respectively, for the example XML query discussed in Chapter 2.

Example 7 Find the TAs who work with Dr. Hanks.

Q2:/catalogue/course[sections/section/instructor="Dr. Hanks"]/TA

SQL 4: 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 5: 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 the same instances of their parent region *course*. XParent, however, requires a number of joins to check the connection *instructor*←*section*←*sections*← *course*→*TA*, in order to ensure that the pairs of nodes, "TA" and "Instructor", are connected by the same "course" nodes. Figure 4.2 is a graphical depiction of tracing the nearest common ancestor "course" under Xregion and XParent schemas, and shows that XParent is required to check two more steps than is necessary with Xregion, for the example query.

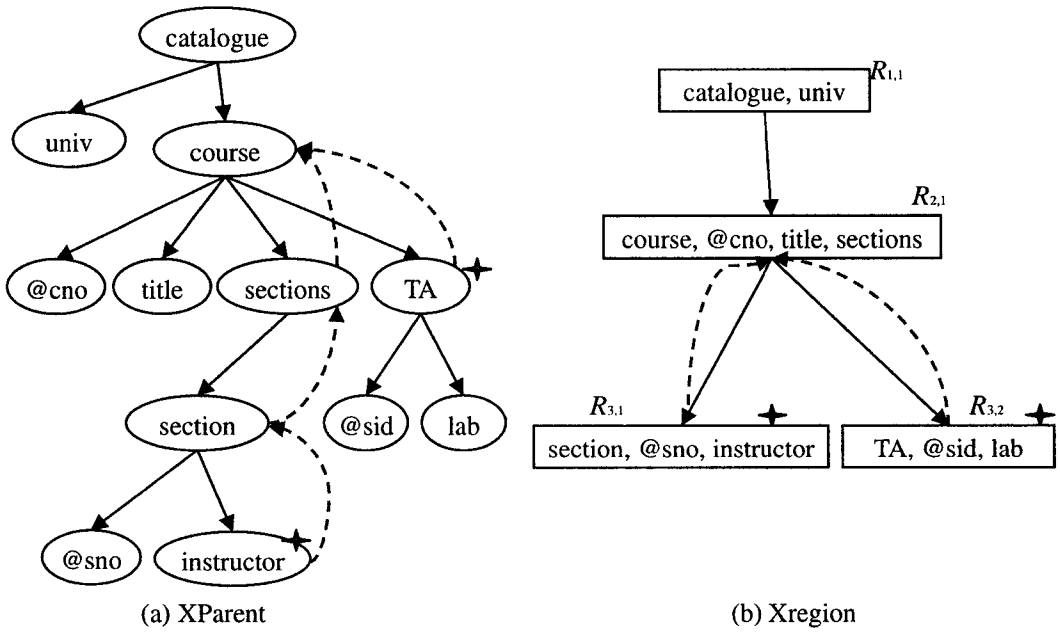


Figure 4.2: Ancestor tracing route for XParent vs Xregion

XML query result publishing is another important aspect in evaluating an XML-to-Relation mapping approach. 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 descendants.

Xregion also speeds up the XML query result publishing process. Because all children nodes (except for the set-valued child nodes) of an element are stored in the same relation as the element, under Xregion schema, transforming

the answer from the relational database to XML format, does not involve expensive operations. For example, shown in SQL 4, all the contents of "TA" elements can be retrieved from a single table. However, for other existing approaches, a number of join operations are still needed to construct the query result in XML format, due to the high fragmentation of the XML data stored in the database. For example, four more joins on the table Data and the table DataPath are involved for answering the above query using the XParent approach.

The worst case of the Xregion approach is that each region of an XML documents contains only one structure node. In this case, the Xregion schema is similar to those of existing generic mapping approaches (i.e. every tuple in a Xregion table represents only one node), therefore, the performance of the Xregion is comparable to and not worse than those of the Edge and XParent approaches.

4.3.3 Processing Updates

Xregion stores XML documents based on their document structures. When updates to a document occur, the cardinality of node occurrences may also change, e.g., some nodes become set-valued nodes.

One solution for the problem is not to modify the storage schema for new set-valued nodes, and store the data according to their original mapping schema. This method is simple, but it will bring another problem of redundancy.

The other solution is to revise the partitions of regions and their corresponding storage schema dynamically according to the updates. Although the problem of updating Xregion storage schema is non-trivial, it is manageable. For each new set-valued node, at most one region (in which the node resides) will be affected. First, the region is split into two regions by the new set-valued node, and a new table is assigned to the new region represented by the new set-valued node. Then the meta_table is modified, and the data belonging to

the new region will be migrated to the new table.

4.4 Chapter Summary

In this chapter, we presented the mapping strategy of Xregion and its basic database schemas, as well as a detailed mapping example. We also discussed query evaluation, query result publishing and the issue of storing a large number of different XML documents using Xregion schema in one system.

Chapter 5

Implementation

5.1 System Outline

We have implemented an XML loading system, called XML Loader, based on the Xregion storage schema described in the previous chapter, for storing XML documents into relational databases. Figure 5.1 describes the architecture of our XML loading system.

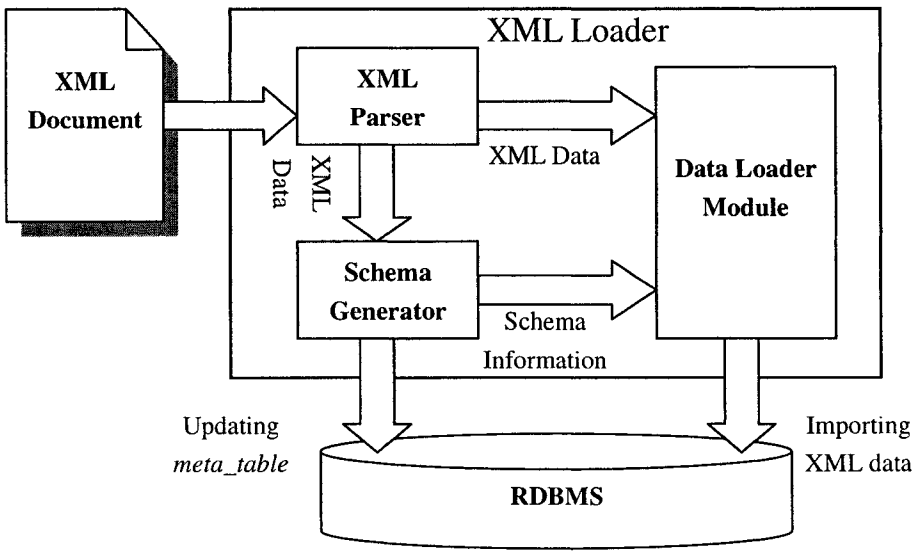


Figure 5.1: The prototype of the XML importing system.

There are three main modules: XML Parser, Schema Generator and XML Data Loader. The XML Parser reads the input XML document(s), extracts the XML tag name and value of each node in the document, materializes the path for each node, and detects all the set-valued nodes. The Schema Generator identifies regions for each set-valued node and creates the corresponding mapping schemas for the XML document. The Data Loader module takes the schema information generated in the Schema Generator module, composes tuples according to the relation assignment, and loads those tuples into their corresponding tables in the underlying database. The programs have been entirely written by us, using the JAVA programming language(Java^{TM} 2 Platform, Standard Edition, v 1.4.1) and JDBC (Java Database Connectivity 2.0).

5.2 XML Parser Module

It is very important to parse XML documents efficiently, especially in applications proposing to handle large volumes. Several types of XML parsing techniques are available, of which *Document Object Model* (DOM) and *Simple API for XML* (SAX) are two popular parsing mechanisms.

A DOM parser converts the entire XML document into a tree stored in memory, provides good navigation support, and allows the user to access an XML document at random positions. However, the problem of lack of enough memory resource arises when processing a very large XML document, since typically the DOM tree is an order of magnitude larger than the document.

In contrast to the DOM, a SAX parser works incrementally and fires off a series of events as it reads through the input XML document. For example, when the SAX encounters the start tag of an element, it will invoke the callback method, **startElement().** The disadvantage of SAX is that it is more complicated than the DOM. The user is required to implement the callback methods

for handling incoming events, such as **Start_Element**, **End_Element** and **CHARACTERS**.

In comparison with the DOM, the SAX parsing technique offers a great performance benefit, especially for large XML document. We therefore implement our XML parser module based on the SAX interface.

The major 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 retrieve the ancestor information of a given element node. Therefore, in implementation, we use a path stack to trace the local hierarchical structure and node occurrence of an XML document. At any given time, the top of the stack is the parent information of the incoming node.

5.3 Schema Generator Module

The Schema Generator module is the core component of our XML Loader system. Figure 5.2 shows a simplified view of the schema generation procedure. First, the XML document structure tree obtained from the 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 existing in the underlying database, a new table representing the region will 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.

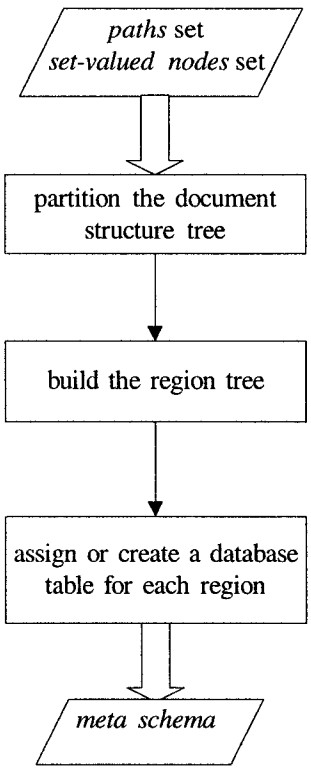


Figure 5.2: Schema Generator

In order to avoid unnecessary checking for cardinalities of children nodes, and speed up the region partitioning process, we process those set-valued element nodes in order, from bottom to top, and update the structure tree automatically after processing each set-valued node, by removing the subtree (region) represented by the node from the whole structure tree.

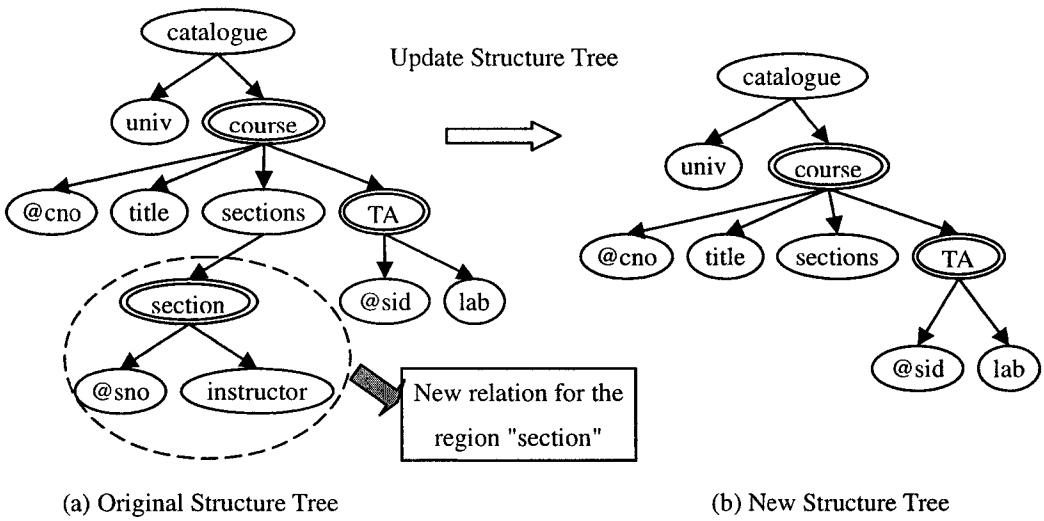


Figure 5.3: Generating a relation for the deepest set-valued node *section* and updating the structure tree by cutting the *section* subtree off.

Figure 5.3 illustrates this *cutting* and *updating* schema generating process. For example, at the start we select the "section" node, which is one of the deepest set-valued nodes in the document structure tree, and we create a relation for "section" and all its descendants. The structure tree is then updated by removing the subtree rooted by "section" node.

The advantage of this *cutting* and *updating* method is that no extra operation, such as checking for the cardinality of descendants, is required for partitioning. Because any descendant of the deepest set-valued node cannot be a set-valued node, all descendants can be included in its region and no checking is involved. Our cutting and updating algorithm maintains the document structure tree dynamically, and guarantees that, at any time, the set-valued node being processed is the deepest set-valued node.

5.4 Data Loader Module

The Data Loader module reads the schema meta data generated by the Schema Generator Module, identifies region instances, 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 the database incrementally by a one-pass sequential scan. It can, therefore, process very large XML documents, as long as the size of the document is supported by the underlying operating system.

An important feature of identifying region instances is that the procedure for 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, as shown in Figure 5.4, the construction of a *course* tuple, is interrupted by that of two *section* tuples.

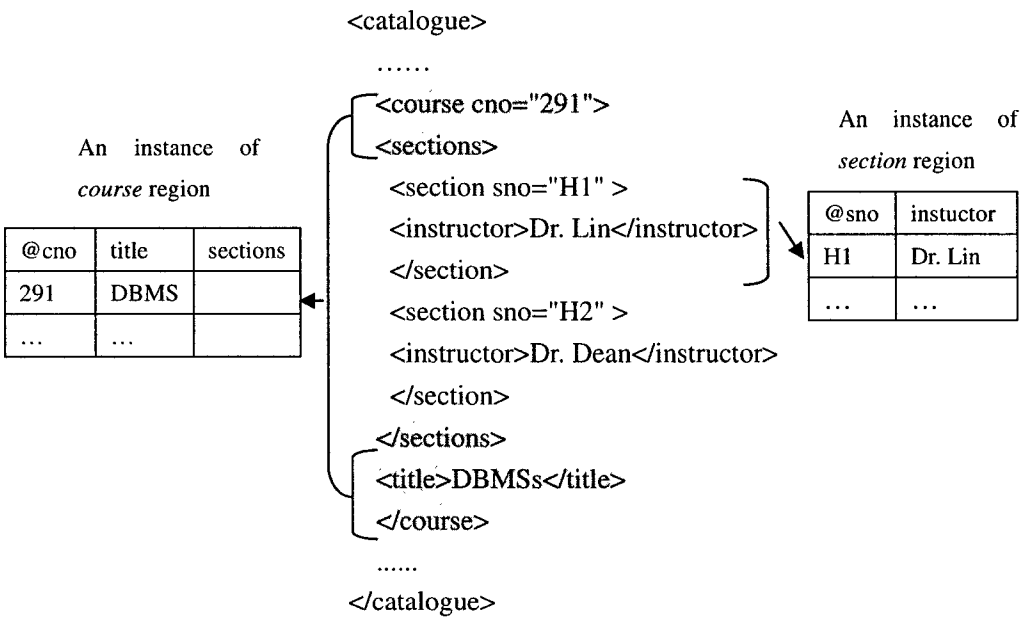


Figure 5.4: An instance of *course* region split by two instances of region *section*

In our implementation, we design a special stack to keep all ongoing region

instances, and pop out an instance from the stack only when the file reader encounters the end tag of its region root (the set-valued node), which indicates the completeness of this instance. As can be seen, the size of the stack is less than the depth of the region tree, which is much shallower than the XML data tree.

5.5 Chapter Summary

In this chapter, we first presented the architecture of our XML loading system, then briefly described three main modules—the XML Parser, Schema Generator, and XML Data Loader. The programs have been entirely written by us, using the JAVA programming language and JDBC.

Chapter 6

Experiments

To evaluate the effectiveness and scalability of the proposed XML generic mapping approach, Xregion, extensive experiments have been conducted. In this chapter, we compare the performance of Xregion with the Edge mapping and XParent approaches.

6.1 Experimental Setups

For the purpose of performance comparison, we implemented the Edge mapping and XParent mapping approaches using Java programming language and SAX parser API. All experiments were conducted on a PIII/1GHZ PC with 1G RAM, running Red Hat Linux release 7.1. The relational database system used in the experiments was Oracle 9i database standard edition release 2. We selected three different XML data collections, with sizes of 7.65M, 200MB and 2GB, respectively, as our data sets. In order to cover different aspects of XML queries over XML data, queries were selected for the corresponding data collections. Table 6.1 summarizes the features of the three XML collections in our experiments.

Name	Size	#paths
SHAKS	7.65MB	57
DBLP	200MB	156
SYN2G	2GB	156

Table 6.1: Data sets information

6.1.1 Data Sets

- SHAKS

  SHAKS consists of 37 Shakespeare plays in XML document format with an 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

  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 experiments contained the data up to January, 2004, listing more than 470,000 articles. The size of this DBLP XML document is 200M.

- SYN2G

  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 nine modified DBLPs 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 modify only letters. In order to simulate the real distribution of authors and the numbers of 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, each time creating a new version of DBLP. For example, a title "An Object-Based Approach to the B Formal Method" can be translated into "An Objezt-Based Ajjroazr to tre B Formal Metrod", based on the following 5-letter dictionary.

Letter	New Value
c	z
h	r
k	o
p	j
v	f

6.1.2 Query Set

For experiments on the DBLP and SYN2G XML documents, we used the queries from XParent [14] and those presented by F. Tian et al. [26] as query templates. The queries experimented on SHAKS data set were selected from XRel [28]. These queries test various aspects of query performance.

All benchmark XML queries in the experiment were translated by us into a set of SQL statements, one for each XML query. All SQL statements for each mapping approaches were tuned based on execution plans, and executed in the Oracle database under SQL trace mode (session setting SQLTRACE was enabled).

6.1.3 Performance Measurement

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

In all our experiments, the size of the database buffer is set to 32 MB, which is considerably less than the size of the DBLP and SYN2G XML documents, but four times larger than the size of SHAKS. Indexes are properly

built on relational tables for all three mapping approaches. For Edge and XParent, we created indexes as proposed in [12] [14]. For Xregion, composite indexes are built on *table_i_j*(doc_name, tuple_id), *table_i_j*(doc_name, p_id), and *meta_table*(doc_name, path). Indexes are also built on the value columns of data tables.

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

6.2 Experimental Results

In this thesis, we present experimental results of XML queries on three different XML collections with sizes of 7.65M, 200MB and 2GB, respectively. Experiments were conducted after warming up the database buffer by executing all the query templates once in random order. The experimental results and discussion for each data set are given below .

6.2.1 Experiment on SHAKS

The SHAKS data set has been used to test many XML Relational mapping approaches in literature. We experimented on SHAKS using the same queries presented in XRel [28], in order to check the correctness of our system, as well as to compare the performance of our proposed approach, Xregion, with other mapping approaches (Edge, XParent and XRel), for querying small XML documents.

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

By comparing the query results returned by these four mapping approaches with the results published by XRel [28], we conclude that both our system and our implementations for the Edge and XParent approaches are correct. The elapsed times of all queries for SHAKS are shown in the Figure 6.1 in logarithmic scale. 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 6.2 shows the number of logical I/Os, which are the database buffer cache reads, involved in each query for all four mapping approaches.

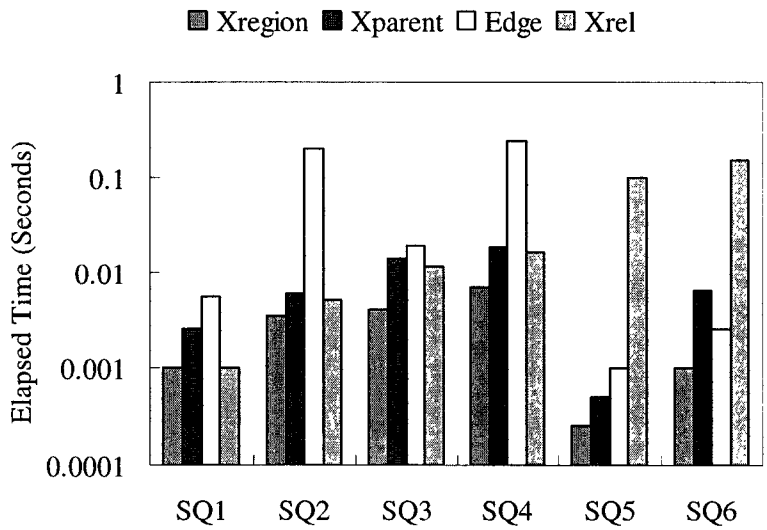


Figure 6.1: Query elapsed time for querying the SHAKS (size 7.65MB) using Xregion, XParent, XRel and Edge

For all queries, Xregion outperforms other mapping approaches, and the elapsed time for each query is less than 0.01 seconds. XParent and XRel per-

Query	Xregion	XParent	Edge	XRel	#tuples returned
SQ1	20	202	598	18	185
SQ2	47	387	21685	49	618
SQ3	73	705	709	658	750
SQ4	96	930	29599	876	951
SQ5	10	33	64	116	4
SQ6	29	427	183	3952	6

Table 6.2: Logical I/O blocks for querying the SHAKS using Xregion, XParent, XRel and Edge

form similarly for SQ1, SQ2, SQ3 and SQ4, each of which contains only one path expression. However, for SQ5 and SQ6, which contain more than one simple path expression, XRel is much slower than other mapping approaches, because non-equijoins on element start and end positions are involved for evaluating these two queries. The Edge approach consumes considerable time for SQ2 (a long path expression) and SQ4 (containing a "//" in the middle of the path expression), since a number of joins are needed to check the connection of all possible steps on the path expressions.

#of element nodes	179,689
#of attribute nodes	0
#of text nodes	147,442
#of simple paths	57

Table 6.3: Test data details for SHAKS XML collection(7.65MB)

The details for the SHAKS, and the size of the resulting database tables

Approach	Database size	#rows	#tables
XRel	10.1MB	327131	3
XParent	11.13MB	506820	4
Edge	8.45MB	179689	1
Xregion	8.37MB	177655	26

Table 6.4: Sizes of resulting database tables for Xregion, XParent, Edge and XRel schemas of SHAKS XML collection(7.65MB)

for each mapping schema are shown in Table 6.3 and 6.4, respectively.

6.2.2 Experiment on DBLP

In testing the DBLP, we adopted as query templates the queries from XParent as well as those presented by F. Tian et al. [26], which test a variety of aspects of query performance. The following are seven query templates for the DBLP XML document.

- Q1 Select titles of conference papers 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

We ran each query template multiple times with different constants; for example, with Q2, Q3, Q6 and Q7, we experimented on 100 different authors. All the results reported were the average elapsed times of random executions for all variants of each query template.

The elapsed times for all queries are shown in Figure 6.2 (logarithmic scale) and Table 6.6; the number of I/O blocks involved are shown in Table 6.5. These results show that Xregion dramatically improve the performance of query evaluation.

Query	Xregion		XParent		Edge	
	LIO	PIO	LIO	PIO	LIO	PIO
Q1	266	113	147142	47599	182288	16344
Q2	253	134	1102	333	2125	268
Q3	380	171	1974	637	3132	357
Q4	1426	1399	338703	279199	9730	1438
Q5	6	5	21	9	19	8
Q6	1063	328	96668014	31770	38809339	2511464
Q7	919	103	54164	66404	19728401	1347270

Table 6.5: Database Buffer I/O (LIO) Blocks and Disk I/O (PIO) blocks for querying the DBLP (size 200MB) using Xregion, XParent and Edge

Query	Xregion	XParent	Edge
Q1	0.16	20.34	25.52
Q2	0.505	1.87	2.321
Q3	0.621	3.02	1.57
Q4	0.83	28.035	12.32
Q5	0.03	0.08	0.05
Q6	0.652	1568.39	938.39
Q7	0.391	30.6	494.79

Table 6.6: Elapsed times for querying the DBLP (size 200MB) using Xregion, XParent and Edge

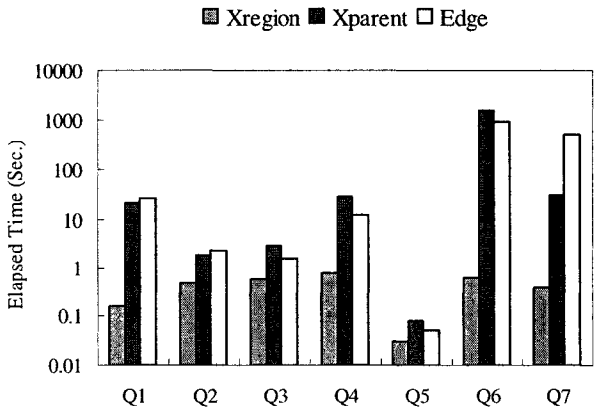


Figure 6.2: Query elapsed time: Xregion, XParent and Edge for the DBLP (size 200MB)

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

For query Q1 and Q4, which require 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 queries, such as Q6 and Q7, Xregion outperforms 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 require a number of joins tracing nearest common ancestors in order to process queries with multiple paths and predicates specified on different branches. For example, query Q6 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>
```

XParent uses four path selections and ten joins to locate and check the connections among nodes involved in the query. The Edge approach requires seven selections on edge labels and six self-joins for checking edge connections in order 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 as "article"), it uses only two joins for connecting "author" relation with "article" relation.

Database Size

The resulting database sizes of mapping schemas are also a critical issue, when storing large XML documents into RDBMSs. Table 6.7 shows the size of the resulting relational database tables and indexes for three mapping schemas.

Approach	Size of tables	#rows	#tables	Size of index
XParent	323MB	16472978	4	416MB
Edge	237MB	5643462	1	296MB
Xregion	176MB	2777916	47	115MB

Table 6.7: Sizes of resulting database tables and indexes for Xregion, XParent and Edge schemas of DBLP XML document.

The size of the DBLP XML document is 200 MB. We see that Xregion uses even less space than the original DBLP file. This is because all non-set-valued nodes of the XML document are stored in the same tuples as their parents. The total number of rows in the database of Edge schema shows that there are 5,643,462 nodes in the XML document, while that of the Xregion schema (2,777,916), shows that more than 50% of the nodes are inlined with their parent nodes.

The size of the database tables for XParent 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 node are bundled with position information. In addition, XParent also use another table—the DataPath table—to record the parent-child relationships between element nodes. The database size of XParent is therefore the largest of all three mapping approaches.

Xregion also uses less space for indexes, while XParent consumes more than twice the document size for indexes.

6.2.3 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 the original DBLP XML document to 2GB. In generating the new test XML document, we keep the ratio of different elements and attributes in the original DBLP document.

We used the same query templates, Q1 to Q5 of the DBLP data set, and the same set of author names, year and paper types for experiments on the SYN2G XML document using Xregion, XParent and Edge approaches. The size of the resulting database of each mapping approach scaled about 10 times the size of its corresponding database for DBLP data set.

Table 6.8 shows the query elapsed times ratio for Xregion on DBLP and SYN2G XML documents. The ratios for all queries except Q4 are around 10, which is the ratio of the size of DBLP and SYN2G.

Query	DBLP(200MB)	SYN2G(2GB)	Ratio
Q1	0.16	2.01	12.5
Q2	0.505	4.74	9.4
Q3	0.621	7.93	12.7
Q4	0.83	13.42	16
Q5	0.03	0.17	5.6

Table 6.8: Ratios of the elapsed times (Seconds) for querying the DBLP vs SYN2G for Xregion schemas

The elapsed times of Q1 to Q5 for all three mapping approaches are shown in Figure 6.3 (logarithmic scale) and Table 6.9, and the corresponding I/Os involved are displayed in Table 6.10.

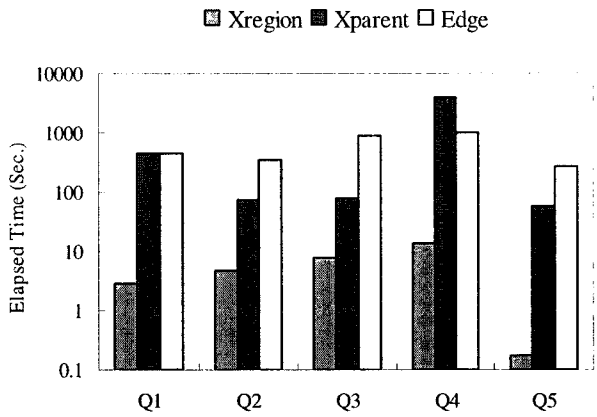


Figure 6.3: Query elapsed time: Xregion, XParent and Edge using SYN2G

Query	Xregion	XParent	Edge
Q1	2.01	444.52	433.5
Q2	4.74	72.98	350.72
Q3	7.93	79.63	904.54
Q4	13.42	3839.85	1028.9
Q5	0.17	55.74	268.64

Table 6.9: Query elapsed time for the SYN2G (size 2GB) using Xregion, XParent and Edge

Query	Xregion		XParent		Edge	
	LIO	PIO	LIO	PIO	LIO	PIO
Q1	3435	397	1439117	438082	12100222	548497
Q2	3560	667	113921	111354	6181545	465415
Q3	5214	1120	115773	112317	17040575	1375609
Q4	17439	1839	9333852	2327792	18676595	1373851
Q5	26	24	109665	109599	7728209	464890

Table 6.10: Database Buffer I/O (LIO) Blocks and Disk I/O (PIO) blocks for querying the SYN2G (size 2GB) using Xregion, XParent and Edge

These results show that the scalability of the Xregion approach is superior to that of XParent and Edge. Most of the queries evaluated in the Xregion schema were running within 15 seconds, whereas the other two methods required several minutes to execute a single query.

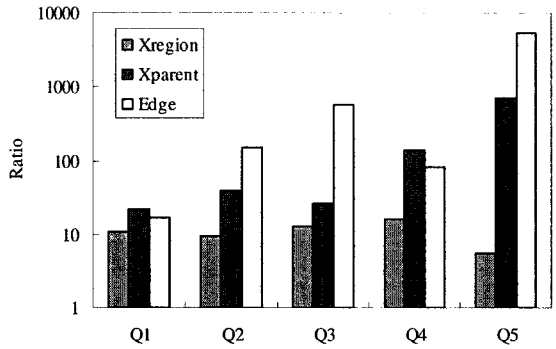


Figure 6.4: Query elapsed time ratios for DBLP and SYN2G using Xregion, XParent and Edge.

Figure 6.4 shows the ratio of query elapsed time for DBLP and SYN2G using Xregion, XParent and Edge. The performances of XParent and Edge degrade dramatically on large XML documents. This is because data are scattered with a high fragmentation degree in relations, a number of joins are needed to trace ancestor information, and the data involved in the join operations are in very large volume.

6.2.4 Schema Generation Time

In addition to query performance, the time required to create database schema for a given XML document is also important for evaluating the proposed approach, Xregion. Different from other existing approaches, which use fixed schemas for storing XML documents, Xregion needs an additional process to create or assign database schema to the input XML document according to its structure.

The schema generation times of the Xregion approach for all data sets in

our experiments are shown in Table 6.11.

Data set	Size	Schema Time	Total Loading
SHAKS	7.65MB	4.965 Sec.	167.831 Sec.
DBLP	200MB	66.868 Sec.	1514.123 Sec.
SYN2G	2GB	412.569 Sec.	10080.69 Sec.

Table 6.11: The elapsed time for the schema generation process for all data sets using our XML Loader system running on a PIII/1GHZ PC with 1G RAM.

The data shown in the "Total Loading" column represents the fastest XML data loading time among the three mapping approaches in our experiments, i.e., Xregion, XParent and Edge. All the data were loaded using the Direct Path loading method of the Oracle SQL*Loader utility.

The results show that the process of storage schema creation for XML documents using the Xregion approach consumes less than 5% of the total XML data loading time, and can be done within several minutes, even for large documents.

6.3 Chapter Summary

In this Chapter, we presented the experimental results on three different XML data sets, which show that Xregion significantly outperforms other approaches. The test on SYN2G XML documents demonstrated that the scalability of Xregion is superior to XParent and Edge, and Xregion enjoyed high performance, even with XML data of large size.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this thesis, we have presented a new generic mapping approach, called Xregion, for storing XML data in relational database systems. Different from existing generic mapping approaches, Xregion takes into account the cardinality of node occurrences in XML documents. We first partition the XML document into several disjoint regions according to the cardinality of element nodes, and then store these regions, together with their parent references, in separate database tables. Each region is stored as a set of tuples in one table, with each column of the table representing a different element type or attribute belonging to the region. Xregion therefore lowers the fragmentation level of decomposed XML data in databases.

Our experiments show promising results indicating 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 the Xregion schema consumes less than 15 seconds, while XParent or Edge require 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 required for complicated queries in query processing.

The proposed mapping method is a meta-data driven approach and no

relational schema assignments are hard-coded. The mapping information of all components of an XML document is recorded as a meta-data identified by paths and document name. 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 transparent to the query processing module, since the interface for query processing is the meta-data stored in the *meta_table*.

We have implemented an XML loading system based on our 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.

7.2 Future Work

The first future research topic to be investigate is the development of an efficient query translation technique to translate XML queries into corresponding SQL statements for the Xregion schema. Such a query translator should support the core syntax of XQuery [7].

We will work on the design of XML benchmark databases and a well-designed class of XML queries that are able to investigate various aspects of performance of XML storage models.

In addition to tuning the database schema, another research direction, which focuses on creating more efficient indexes for paths, can be explored to further improve the query performance of Xregion.

Because the path of a node in an XML document is a sequence of tag names, starting from the root to the node, a large number of paths share the same prefix, a practice which are not favored by the traditional indexes of RDBMS, such as B-tree. Cooper, Sample, Franklin et al. proposed the Index

Fabric approach [9], to build a special index for path information (raw path) instead of using a RDBMS B-tree index. Based on their experimental results, we expect that the performance of Xregion can be further improved by using the Index Fabric to store our meta data.

Bibliography

[1] http://www.w3.org/xml/.

[2] Sihem Amer-Yahia and Divesh Srivastava. A mapping schema and interface for XML stores. In *Proceedings of the fourth international workshop on Web information and data management*, pages 23–30. ACM Press, 2002.

[3] Marcelo Arenas and Leonid Libkin. A normal form for XML documents. *ACM Transactions on Database Systems (TODS)*, 29(1):195–232, 2004.

[4] Anders Berglund, Scott Boag, Don Chamberlin, and et. al. *XML Path Language (XPath) 2.0.* W3C Working Draft, http://www.w3.org/TR/xpath20/, November 2003.

[5] Philip Bohannon, Juliana Freire, Jayant R. Haritsa, Maya Ramanath, Prasan Roy, and Jerome Simeon. LegoDB: Customizing relational storage for XML documents. In *the 28th International Conference on Very Large Data Bases*, pages 1091–1094, 2002.

[6] Philip Bohannon, Juliana Freire, Prasan Roy, and Jme Simeon. From XML schema to relations: A cost-based approach to XML storage. In *ICDE*, 2002.

[7] Don Chamberlin. Xquery: An XML query language. *IBM Systems Journal*, 41(4):597–615, 2002.

[8] Sophie Cluet, Pierangelo Veltri, and Dan Vodislav. Views in a large scale XML repository. In *Proceedings of the 27th International Conference on Very Large Data Bases*, pages 271–280, 2001.

[9] Brian Cooper, Neal Sample, Michael J. Franklin, Gisli R. Hjaltason, and Moshe Shadmon. A fast index for semistructured data. In *Proceedings of the 27th International Conference on Very Large Data Bases*, pages 341–350. Morgan Kaufmann Publishers Inc., 2001.

[10] Alin Deutsch, Mary Fernandez, and Dan Suciu. Storing semistructured data with STORED. In *Proceedings ACM SIGMOD International Conference on Management of Data*, pages 431–442, 1999.

[11] Mary Fernandez, Ashok Malhotra, and et al. *XQuery 1.0 and XPath 2.0 Data Model.* W3C Working Draft, http://www.w3.org/TR/xpath-datamodel, 2003.

[12] Daniela Florescu and Donald Kossmann. Storing and querying XML data using an RDMBS. *IEEE Data Eng. Bull.*, 22(3):27–34, 1999.

[13] Torsten Grust. Accelerating xpath location steps. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 109–120. ACM Press, 2002.

[14] Haifeng Jiang, Hongjun Lu, Wei Wang, and Jeffrey Xu Yu. Path materialization revisited: An efficient storage model for XML data. In Xiaofang Zhou, editor, *Thirteenth Australasian Database Conference (ADC2002)*, Melbourne, Australia, 2002. ACS.

[15] Gerti Kappel, Elisabeth Kapsammer, S. Rausch-Schott, and Werner Retschitzegger. X-ray - towards integrating XML and relational database systems. In *International Conference on Conceptual Modeling / the Entity Relationship Approach*, pages 339–353, 2000.

[16] Latifur Khan and Yan Rao. A performance evaluation of storing XML data in relational database management systems. In *Proceeding of the third international workshop on Web information and data management*, pages 31–38. ACM Press, 2001.

[17] Atakan Kurt and Mustafa Atay. An experimental study on query processing efficiency of native-xml and xml-enabled database systems. In *Proceedings of the Second International Workshop on Databases in Networked Information Systems*, pages 268–284. Springer-Verlag, 12 2002.

[18] S. Lu, Y. Sun, M. Atay, and F. Fotouhi. A new inlining algorithm for mapping XML DTDs to relational schemas. In *Proc. of the 1st International Workshop on XML Schema and Data Management*, Lecture Notes in Computer Science, Chicago, Illinois, USA, October 2003.

[19] Jason McHugh, Serge Abiteboul, Roy Goldman, Dallan Quass, and Jennifer Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3):54–66, 1997.

[20] Neoklis Polyzotis and Minos Garofalakis. Structure and value synopses for XML data graphs. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 466–477, 2002.

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

[22] Albrecht Schmidt, Martin Kersten, Menzo Windhouwer, and Florian Waas. Efficient relational storage and retrieval of XML documents. *Lecture Notes in Computer Science*, 1997:137+, 2001.

[23] Albrecht Schmidt, Florian Waas, Martin Kersten, Daniela Florescu, Michael J. Carey, Ioana Manolescu, and Ralph Busse. Why and how to benchmark XML databases. *SIGMOD Rec.*, 30(3):27–32, 2001.

[24] Jayavel Shanmugasundaram, Eugene J. Shekita, Jerry Kiernan, Rajasekar Krishnamurthy, Stratis Viglas, Jeffrey F. Naughton, and Igor Tatarinov. A general techniques for querying XML documents using a relational database system. *SIGMOD Record*, 30(3):20–26, 2001.

[25] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *The VLDB Journal*, pages 302–314, 1999.

[26] F. Tian, D. DeWitt, J. Chen, and C. Zhang. The design and performance evaluation of alternative XML storage strategies. *SIGMOD Record SPECIAL ISSUE: Data management issues in electronic commerce*, 31(1):5–10, 2002.

[27] Paul J. Wagner and Thomas K. Moore. Integrating XML into a database systems course. In *Proceedings of the 34th SIGCSE technical symposium on Computer science education*, pages 26–30. ACM Press, 2003.

[28] M. Yoshikawa, T. Amagasa, T. Shimura, and S. Uemura. Xrel: A path-based approach to storage and retrieval of XML documents using relational databases. *ACM Transactions on Internet Technology*, 1(1):110–141, August 2001.