Efficient Storage and Retrieval of XML Documents Using XQuery

Huei-Huang Chen Chu-Yen Liu
Department of Computer Science and Engineering
Tatung University
40 Chungshan N. Road, 3rd sec.,Taipei, 104, Taiwan, R.O.C.
hhchen@ttu.edu.tw *g9106033@ms2.ttu.edu.tw*

Abstract- *In the last decade of the 20th century, because of the popularity of Internet, the trend is towards e-solutions for businesses. Not only apply on the electronic commerce but also on the information exchange to decrease time from material in the manufacturer to products brought by customers. However, the problem we confront today is that there are full of e-documents in businesses.*

XML has already been the standard of data interchange on the Internet. In the future, a large amount of data will be represented in XML format. However, most of the critical data in businesses are still stored in relational database management systems. It is difficult to query XML databases because of its textual format. In this paper we proposed a system to manage XML documents that could be queried by the query language XQuery. XML documents are stored in relational format and the XQuery expressions are translated into appropriate SQL queries. The results of the SQL queries are transformed into XML documents.

1. Introduction

With the growing popularity of the Internet, more and more transactions are carried out on-line [1] [2]. So far, businesses have adopted XML as de facto format for document communications. The increasing frequency of transactions between and within enterprises produces a huge amount of XML documents. There are two approaches to save and process XML documents: One is to save them in relational databases; the other is to save them in recently-developed native XML databases.

XML (eXtensible Markup Language) [11] [12] is now a universal standard for information exchanging because of a simple and flexible text format. A XML storage system must provide efficient manipulations (e.g., data storing and retrieval) of a large number of XML documents. Many researchers have proposed various strategies in implementing XML storage systems. In this paper, we investigated the translation and manipulation of XML documents in relational database systems. We propose an efficient algorithm to translate XML documents into tables in a relational database. The essential issues on query processing are included. The data manipulations can be translated into a set of simple SQL commands. In terms of those SQL commands, we can easily manipulate XML data in relational database systems. Figure 1 sketches the XML-to-relational mappings [4]. First we focus on the XML part on the right side of the figure. From a user's point of view, XML documents are stored as pairs of Uniform Resource Identifiers (URIs) and black-box XML documents, which may only be accessed through XQuery but not SQL. Internally, the documents are stored in a shredded schema that allows fast associative access with the relational algebra of the SQL engine; to achieve this we chose to implement the approach of [3] [10]. Additionally, the native SQL tables on the left side of the figure are exported as XML views and may be queried in XQuery just like the XML documents on the right side.

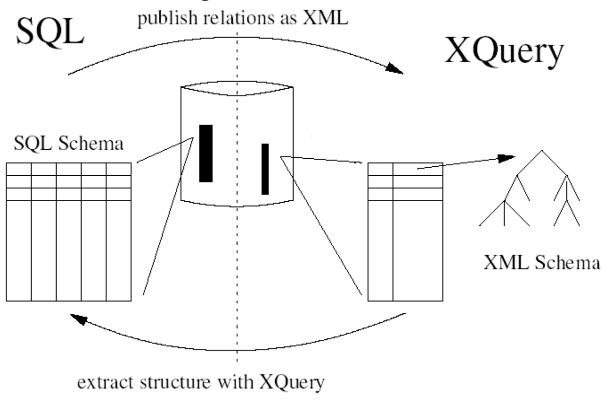


Figure 1: XML-to-relational mappings

2. Related Works

Many technologies have been developed to query documents that take advantage of XML's loose, yet incredibly flexible structure rules. One such language, XQuery, is quickly being accepted as the standard XML Query language because of its ability to format the resulting output. As powerful and popular as XML currently is, it is still a new language that is constantly evolving. This, coupled with the fact that most businesses' data is stored in the well-established relational database format, creates the need for a system that can combine the best of both technologies [5]. XQuery languages execute database-style queries over XML files. The first language for querying XML documents was Quilt but XQuery is the W3C standard language [14] [15] for querying XML documents. XQuery is a functional language this means that the queries are expressions. XQuery is powerful and easy to use when querying and retrieving XML data [6]. There are 7 expressions in XQuery, examples are: Path Expression, FLWR Expression, Primary Expression, Arithmetic Expression, Element Constructor, Conditional. Expression, Quantified Expression. The path expression is the simplest expression which is based on the syntax of the XPath language [13]. A path expression selects a part of the XML document and it is used in many other expressions like the FLWR expression. The FLWR expression, also called flower expression, consists of the clauses FOR-LET-WHERE-RETURN. It is the most used expression and it selects fragments from a specified XML document. The overall flow of information through an FLWR expression is illustrated in Figure 2.

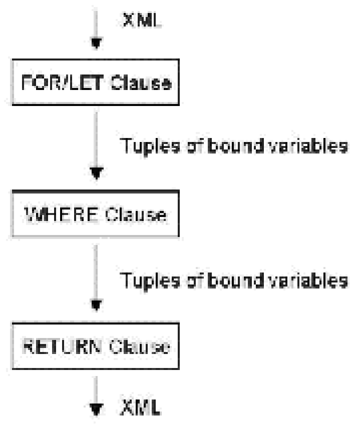


Figure 2: Flow of data in an FLWR expression

A FLWR expression [5] [6] iterates over a sequence of items and binds variables that can be used in the scope of the current expression. If the item sequence is empty, the result of the FLWR expression is an empty sequence. A FLWR expression consists of one or more for and let clauses in any combination, followed by an optional where clause, and a return clause. Briefly, these clauses are interpreted as follows:

- A "for" clause binds one or more variables to each value of the result of the following expression.
- A "let" clause binds one or more variables to the complete result of the expression.
- A "where" clause retains only those intermediate results that satisfy the following condition.
- A "return" clause evaluates the following expression and returns the result.

3. Storage and Retrieval System Architecture

Our system is a prototype implemented with the architecture of LegoDB [8] [9] and Silkroute [7] at mind. It does not follow the architecture of LegoDB and Silkroute to a point, but the main components and basic tasks are similar. The architecture will be discussed in section 3.1. In section 3.2 an XML example will be introduced, there will also be discussion about what type of documents the system can handle. Our system, as part of a larger combination system, translates Xqueries made by the user over XML documents into SQL queries to be run on a local relational database. The results are then formatted according to the XQuery.

3.1 System Architecture

The system architecture diagram is presented in Figure 3. The boxes represent the modules in our system and the text outside the boxes represents the data that flows on the edges between the respective modules.

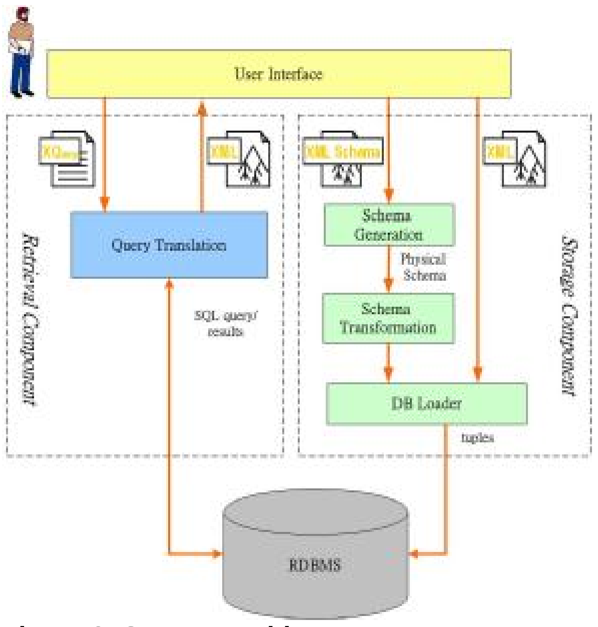


Figure 3: System Architecture

The architecture has two components that consist of four different modules. From the Figure 3, the system components are also depicted in Table 1.

Table 1: Storage and Retrieval System components

Components	Function
Storage Components	
Schema Generation	This module takes as input an XML schema and outputs a physical schema.
Schema	In this module the physical

Transformation	schema which was output from the Schema generation module will be used as input. The physical schema will be transformed into a create table statement written in SQL.
DB Loader	This module takes apart the XML document/XML Schema, creates insert statements using the information in the document and loads the statement/statements into the tables.
Retrieval Components	
Query Translation	The Query Translation module is used to translate XQuery questions to SQL and the result of the SQL queries to XML document. The document is then returned to the user.

3.2 Scenario XML Example and Limitations

3.2.1 The XML example

When describing the different modules of our system a simple example consisting of an XML schema and an XML document will be used. The XML example is a simple cars data. In this section it will also be discussed what type of documents the system can handle. The XML schema which follows bellow is called cars.xsd show in Figure 4.

```
<?xml version=”1.0” encoding=”ISO-8859-1” ?>
<xs:schema
xmlns:xs=”http://www.w3.org/2001/XMLSchema”>
<xs:element name=”cars”>
<xs:complexType>
<xs:sequence>
<xs:element name=”car” maxOccurs=”unbounded”>
<xs:complexType>
<xs:sequence>
   <xs:element name=”owner” type=”xs:string”/>
   <xs:element name=”model” type=”xs:string”/>
   <xs:element name=”color” type=”xs:string”/>
   <xs:element name=”displacement”
     type=”xs:integer”/ >
   <xs:element name=”seats” type=”xs:integer”/>
   <xs:element name=”year” type=”xs:integer”/>
</xs:sequence>
<xs:attribute name=”license” type=”xs:string”/>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>
```

Figure 4: cars.xsd

The XML document which follows is called car.xml (see in Figure 7) and corresponds to the schema above.

3.2.2 Limitations on the input documents

Limitations have been made on what type of XML schemas in our system can handle. There are many different types of semantic and syntax in the W3C standard for XML schemas but for now the system only handles the simplest forms of XML schemas. The system can not handle user defined data types, the only data types that are implemented is text/string, date and integer. When you transform the XML schema into a create table statement the data types in the statement must be compatible with the data types in the database, MS SQL server 2000. The user defined data types and the simple data types do not correspond to any of the data types in the database and therefore can not be used in the create statement. Complex types create the levels in the XML documents and the root element is a complex type that all the other types are included in. Attributes are included and they can be used at different levels. Limitations on what types of XML documents handled are set by the limitations on the XML schemas.

3.3 Overview the Components

3.3.1 Storage Components

In this subsection, we describe how the fixed mapping is implemented. Our system generates a space of possible schema mappings by repeatedly transforming the original XML Schema, and for each transformed schema, applying a fixed mapping from XML Schema into relations. In order to guarantee the existence of a fixed mapping, we define the notation of “Physical schema”. In a physical schema, each type name defines a structure that can be directly mapped to a relation. We now sketch the fixed-mapping algorithm used to generate a physical schema from an XML Schema. The algorithm is applied top-down on the structure of the type, and for each type in the original schema. For a type definition define type X { T }, where X is the name of the type, and T is the structure defining that type, the fixed-mapping algorithm is first applied on T. This returns a new type T’ along with a set of new type definitions define type X1 { T1 } ... define type Xn { Tn }, where all of the type T’, T1, ..., Tn are normalized. A given type structure T is normalized, recursively, as follows:

- If the type is an atomic type (e.g., string), return the type unchanged.
- If the type is an (optional) element declaration, then return the element declaration with its content normalized.
- If the type is a sequence of two types, return the sequence of their normalized types.

- If the type is a repetition (e.g., element a*), then insert a new type name with the normalized content of the original type (e.g., X1* with define type X1 { element a }).
- If the type is a union (e.g., element a | element b), then create a new type name for each component of the union with the contents of the original type normalized (e.g., X1 | X2 with define type X1 { element a } and define type X2 {element b }).

After the original schema is normalized, transformations can be applied on the resulting physical schema. These transformations always result in a physical schema which can then be mapped into a relational table. The algorithm to map a physical schema into a set of relations is as follows:

- Create one relation *R*X for each type name X;
- For each relation *R*X, create a key that stores the id of the corresponding element;
- For each relation *R*X, create a foreign key parent *P*X to all the relations *R*PX such that *P*X is a parent type of X;
- Create a column in *R*a for each element a inside the type X that contains a value;
- If the data type is contained within an optional type then the corresponding column can contain a null value.

The mapping procedure follows the type stratification: elements in the physical type layer are mapped to columns, elements within the optional types layer are mapped to columns that allow null values, and named types are used only to keep track of the child-parent relationship and for the generation of foreign keys.

3.3.1.1 Schema Generation

The Schema Generation module takes an XML schema (see Figure 4) as input and transforms it into a physical schema. When the XML schema was input into our system it resulted in the following physical schema is show in Figure 5:

```
type cars = car [ owner [ string ], model [ string ], color [ string ], displacement [ integer ], seats [ integer ], year [ integer ], license [ integer ] ] ;
```

Figure 5: Physical schema

The root of the XML schema cars.xsd is cars and therefore car becomes the type in the physical schema. Later on it will be shown that car will also be the name of the table that the XML document will be translated into.

3.3.1.2 Schema Transformation

In this module the physical schema which was output from the Schema generation module will be used as input. The physical schema will be transformed into a create table statement written in SQL. When using the Physical Schema (see Figure 5) created in the Schema generation module as input. The following Create table statement (see Figure 6) will be the output of the Schema transformation module.

```
CREATE TABLE cars (car id serial PRIMARY KEY, owner STRING, model STRING, color STRING, displacement INTEGER, seats INTEGER, license INTEGER) ;
```

Figure 6: Create table statement

When the create table statement is successfully created it will be passed to the database and the following module will start working on filling the table with information.

3.3.1.3 DB Loader

The DB loader takes the XML document that corresponds to the XML schema used in earlier modules as input and creates one or several insert statements written in SQL. This depends on the structure of the XML document (see Figure 8). When using the XML document as input the result is only one insert statement which will be passed to the database. This is the case when the document car.xml (see Figure 7) is loaded to the database, because the document contains several cars with information about their owners and other details. A part of the car XML document with two car elements are here presented:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<cars
xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"xsi:noNamespaceSchemaLocation="cars.xsd">
<car license="AA-1111">
   <owner>Kevin </owner>
   <model>Peugeot</model>
   <color>Silver</color>
   <displacement>2000</displacement>
   <seats>4</seats>
   <year>2002</year>
</car>
<car license="BB-2222">
   <owner> Helen</owner>
   <model>Jeep</model>
   <color>Black</color>
   <displacement>3000</displacement>
   <seats>4</seats>
   <year>2003</year>
</car>
   …
</cars>
```

Figure 7: car.xml (A part of the car XML document)

Consider a DOM tree fragment rooted at the cars element. The DOM tree is illustrated by Figure 8.

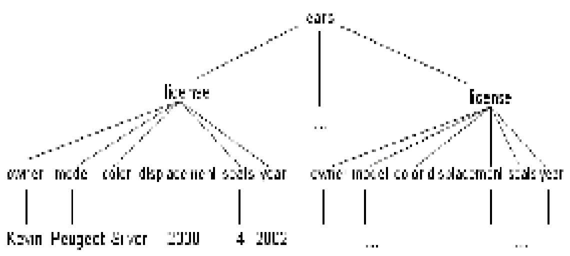


Figure 8: Document Object Model (DOM) representation of a simple XML document

The output, with two car elements, is the following two insert statements (see Figure 9):

```
Insert into cars ( car_owner, car_model, car_color, car_displacement, car_seats, car_license )
values ('Kevin ', 'Peugeout', 'Silver', '2000' , '4', 'AA-1111', '2002');
Insert into cars ( car_owner, car_model, car_color, car_displacement, car_seats, car_license )
values (' Helen', 'Jeep', 'Black', '3000' , '4', 'BB-2222', '2003');
```

Figure 9: Insert statements

The DB Loader also uses the create table statement which was output from the Physical schema transformation module. The reason for this is to find out what data types the elements are to be able to build correct insert statements. Some elements require around them because the data type requires it in SQL like the string and date data types. The system is restricted to manage only the data types string and integer.

3.3.2 Retrieval Components

In this subsection, we describe how the user submits an XQuery question as input, translates it into the SQL query and sends it to the database. The result set returned from the database is then converted into an XML document and returned to the user.

3.3.2.1 Query Translation

The Query Translation module takes an XQuery question as input, translates it into the corresponding SQL query and sends it to the database. The result set returned from the database is then converted into an XML document which is the output of the module.

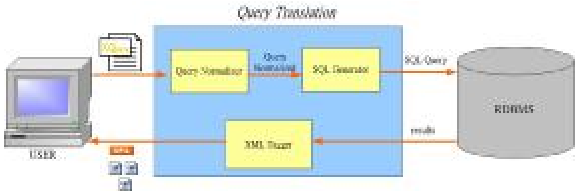


Figure 10: Query Translation Diagram

3.3.2.1.1 XQuery to SQL translation

In our system, the mapping of XQuery to SQL (see Figure 10) is done in two phase. The first phase user submits an XQuery question (see Figure 11) to the Query Nomoralizer then rewriting an XQuery XQ in a normal form XQnf is shown in Figure 12.

```
For $cars in document ("car.xml")/cars
Where $cars/license = "AA-1111" and "BB-2222"
Return $cars/owner, $cars/model, $car/color ;
```

Figure 11: XQuery question

```
Let $cardb := document ("car.xml")/cars
For $cars in cardb/cars, $v_owner in $cars/owner,
$v_color in $cars/color, $v_license in $cars/license,
$v_model in $cars/model
Where $v_license = "AA-1111" and "BB-2222"
Return ($v_owner, $v_model, $v_color);
```

Figure 12: XQuery normal form XQnf

The second phase XQnf is passed to the SQL Generator to create SQL queries for the given Physical schema:

-***SELECT clause***. For each varable *v* in the XQuery *return* clause, if *v* refer to a type in the Physical schema, all attributes of the corresponding table are added to the clause. Otherwise, if *v* refers to an element with no associated type, the corresponding attribute is added to the clause.

-***FROM clause***. For each variable *v* mentioned in the XQuery (in both *where* and *return* clauses), if *v* refer to a type *T* in the physical schema, the corresponding table *RT* is added to the clause.

-***WHERE clause***. Conditions in *where* clause are translated in a straightforward manner, by replacing variable occurrences with the appropriate column name. In addition, for each variable in the XQuery, if the path expression defining the variable includes elements in that are mapped into separate tables, a condition must be added to enforce the key/foreign-key constraint. The query mapping algorithm generates the SQL Query shown in Figure 13:

```
SELECT   car_license, car_owner, car_model,
         car_color
FROM cars
WHERE car_license = 'AA-1111' AND 'BB-2222';
```

Figure 13: SQL Query

3.3.2.1.2 SQL result to XML conversion

Runs the SQL query on the RDBMS and passes the resulting tuples (see Figure 14) to the XML Tagger. Finally the XML Tagger will be converted into an XML documents (see Figure 15) based on resulting tuples; XML documents will be returned to the user.

Cars

car_license	car_owner	car_model	car_color
AA-1111	Kevin	Peugeout	Silver
BB-2222	Helen	Jeep	Black

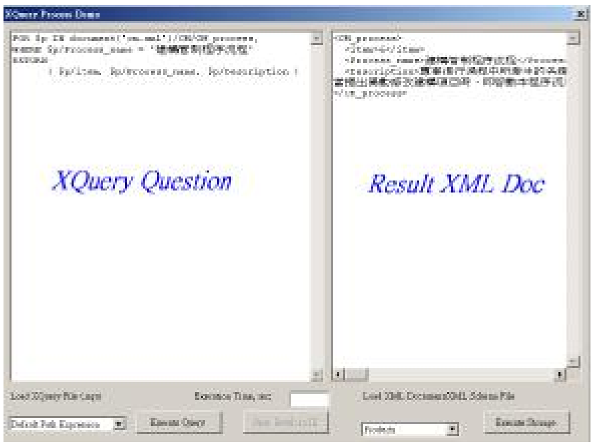
Figure 14: Resulting tuple

```
    <car licence="AA-1111">
    <owner>Kevin </owner>
    <model>Peugeot</model>
    <color>Silver</color>
    </car>
    <car licence="BB-2222">
    <owner> Helen</owner>
    <model>Jeep</model>
    <color>Black</color>
    </car>
```

Figure 15: Query Results of XML documents

4. System Implementation

A prototype system, called XQPD, has been designed for this research (see Figure 16). We implement the XQPD on the client-server environment and use MS SQL Server 2000 as the development tools. Users can submit an XQuery question as input. The results returned from the database server is then converted into an XML document and returned to the user.



```
FOR $p IN document('[illegible].xml')/CM/CM_process,
WHERE $p/process_name = '[illegible]'
RETURN
      { $p/item, $p/process_name, $p/description }
```

```
<CM_process>
  <item>[illegible]</item>
  <Process_name>[illegible]</Process_name>
  <description>[illegible]
</CM_process>
```

Figure 16: System Implementation

5. Conclusions and Future Works

In this paper, we propose a structure to build a storage and retrieval integrated architecture for XQuery. The contributions of this paper are related to the following aspects:

- Using fixed-mapping algorithm to store XML document/XML Schema into RDBMS.
- Using XQuery to retrieve XML document from RDBMS.

There are still some problems unsolved or not well solved:

- In our paper, we limit the scope of document in semi-structured. As we know, documents which could include documents images spreadsheets, voice or video, etc. How about these types of document to management are the topic to discuss.
- We proposed is focusing on Storage and Retrieval. Deletion and revision function will be added in our system if the future versions of XQuery support deleting and revising.
- Our future version of system can support all XML schema and user-defined document.

6. References

[1] James H. Cook, "XML Sets Stage for Efficient Knowledge Management", *IT Professional*, pp.55-57, May –June 2000.

[2] Takeya Kasukawa, Hideo Matsuda, Michio Nakanishi, and Akihiro Hashimoto, "A New Method for Maintaining Semi-Structured Data Described in XML", *Proceedings of IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing*, pp.258-261, August 1999.

[3] Elisa Bertino, and Barbara Catania, "Integrating XML and Databases", *IEEE Internet Computing*, Vol: 5 Issue: 4, pp.84-88, July-Aug, 2001.

[4] R. Bourret, C. Bornhövd, and A. Buchmann, "A Generic Load/Extract Utility for Data Transfer between XML Documents and Relational Databases", *Second International Workshop on Advance Issues of E-Commerce and Web-Based Information Systems (WECWIS 2000)*, pp.134-143, June 2000.

[5] James McGovern, Per Bothner, Kurt Cagle, James Linn, and Vaidyanathan Nagarajan, *XQuery Kick Start*, ISBN: 0672324792, SAMS, 2003.

[6] Don Chamberlin, Jonathan Robie, and Daniela Florescu, "Quilt: An XML Query Language for Heterogeneous Data Sources", *Proceedings of WebDB 2000 Conference*, in Lecture Notes in Computer Science, Springer-Verlag, 2000.

[7] Mary Fernandez, Wang-Chiew Tan, and Suciu Dan. "SilkRoute: Trading between Relations and XML", *Proceedings of the Ninth International World-wide Web Conference* (WWW'9), Amsterdam, May 2000.

[8] P. Bohannon, J. Freire, J. R. Haritsa, M. Ramanath, P. Roy, and J. Simeon, "LegoDB: Customizing Relational Storage for XML Documents", *Proceedings of the 28th VLDB Conference*, Hong Kong, China, 2002.

[9] P. Roy, J. Simeon, P. Bohannon, and J. Freire, "From XML schema to relations: A cost-based approach to XML storage", *Proceedings of the 18th International Conference on Data Engineering*, pp 64 - 75, Feb-March 2002.

[10] I. Manolescu, D. Florescu, and D. Kossmann. "Answering XML queries on heterogeneous data sources. In *Proc. of the 27th Int. Conf. on Very Large Data Bases (VLDB 2001)*, pages 241–250, 2001.

[11] W3C, *XML*, http://www.w3.org/XML, 1998.

[12] W3C, *XQuery 1.0 and XPath 2.0 Formal Semantics,* W3C Working Draft, http://www.w3.org/TR/xquery-semantics/, February 2004.

[13] W3C, *XML Path Language (XPath)* Version 1.0. http://www.w3.org/TR/xpath, November 1999.

[14] W3C, *XML Query Data Model*, W3C Working Draft, http://www.w3.org/TR/xpath-datamodel/, November 2003.

[15] W3C, *XQuery 1.0: An XML Query Language*, W3C Working Draft, http://www.w3.org/TR/xquery/, November 2003.