

Storing and Querying XML Data in Object-Relational DBMSs*

Kanda Runapongsa
University of Michigan
krunapon@eecs.umich.edu

Jignesh M. Patel
University of Michigan
jignesh@eecs.umich.edu

Abstract

As the popularity of eXtensible Markup Language (XML) continues to increase at an astonishing pace, data management systems for storing and querying large repositories of XML data are urgently needed. In this paper, we investigate an Object-Relational DBMS (ORDBMS) for storing and querying XML data. We present an algorithm, called XORator, for mapping XML documents to tables in an ORDBMS. An important part of this mapping is assigning a fragment of an XML document to a new XML data type. We demonstrate that using the XORator algorithm, an ORDBMS is usually more efficient than a Relational DBMS (RDBMS). Based on an actual implementation in DB2 V.7.2, we compare the performance of the XORator algorithm with a well-known algorithm for mapping XML data to an RDBMS. Our experiments show that the XORator algorithm requires less storage space, has much faster loading times, and in most cases can evaluate queries faster. The primary reason for this performance improvement is that the XORator algorithm results in a database that is smaller in size, and queries that usually have fewer number of joins.

1 Introduction

As the popularity of XML (eXtensible Markup Language) [5] for representing easily sharable data continues to grow, large repositories of XML data are likely to emerge. Data management systems for storing and querying these large repositories are urgently needed. Currently, there are two dominating approaches for managing XML repositories [13]. The first approach is to use a native XML database engine for storing and querying XML data sets [1, 19]. This approach has the advantage that it can provide a more natural data model and query language for XML data, which is typically viewed using a hierarchical or graph representation. The second approach is to map the XML data and queries to constructs provided by a Relational DBMS (RDBMS) [12, 14, 24, 25]. XML data is mapped to relations, and queries on the XML data are converted into SQL queries. The results of the SQL queries are then converted to XML documents before returning the answer to the user. If the mapping of the XML data and queries to relational constructs is automatic, then the user does not need to be involved in the complexity of mapping. One can leverage many decades of research and commercialization efforts by exploiting existing features in an RDBMS. An additional advantage of an RDBMS is that it can be used for querying both XML data and data that exists in the relational systems. The disadvantage of using an RDBMS is that it can lower performance since a mapping from XML data to the relational data may produce a database schema with many relations. Queries on the XML data when translated into SQL queries may potentially have many joins, which would make the queries expensive to evaluate.

In this paper, we investigate a third approach, namely using an Object-Relational DBMS (ORDBMS) for managing XML data sets. Our motivations for using an ORDBMS are threefolds: First, most database vendors today offer *universal* database products that combine their relational DBMS and ORDBMS offerings into a single product. This implies that the ORDBMS products have all the advantages of an RDBMS. Second, an ORDBMS has a more expressive type system than an RDBMS, and as we will show, can be used to produce a more efficient mapping from an XML data model to constructs in the ORDBMS type system. Third, an ORDBMS is better suited for storing and querying XML documents that may use a richer set of data types.

We present an algorithm, called *XORator* (**X**ML to **OR** Translator), that uses Document Type Definitions (DTDs) to map XML documents to tables in an ORDBMS. An important part of this mapping is the assignment of a fragment of an XML document to a new XML data type, called XADT (**X**ML **A**bstract **D**ata **T**ype). Among

*This paper is based upon work supported by an IBM CAS Fellowship, an IBM Faculty Award, and a research grant from NCR.

several recently proposed XML schema languages, in this paper, we use DTDs since real XML documents that conform to DTDs are readily available today. Although we focus on using DTD, the XORator algorithm is applicable to any XML schema language that allows defining elements composed of attributes and other nested subelements. In this paper, we also explore alternative storage organizations for the XADT. Storing a large XML fragment as a tagged string can be inefficient as repeated tags can occupy a large amount of space. To reduce this space overhead, we also explore the use of an alternative *compressed* storage technique for the XADT.

We have implemented the XORator algorithm and the XADT in DB2 UDB V.7.2, and used real and synthetic data sets to demonstrate the effectiveness of the proposed algorithm. In the experiments, we compare the XORator algorithm with the well-known Hybrid algorithm for mapping XML data to relational databases [24]. Our experiments demonstrate that compared to the Hybrid algorithm, the XORator algorithm requires less storage space, has much faster loading times, and in most cases can evaluate queries faster. In many cases, query evaluation using the XORator algorithm is faster by an order of magnitude, primarily because the XORator algorithm produces a database that is smaller, and results in queries that usually have fewer number of joins.

The remainder of this paper is organized as follows. We first discuss related work in Section 2. Section 3 describes the XORator algorithm for mapping XML documents to relations in an ORDBMS using a DTD. We then compare the effectiveness of the XORator algorithm with the Hybrid algorithm in Section 4. Finally, we present our conclusions and discuss future work in Section 5.

2 Related Work

In this section we discuss and compare previous work on mapping XML data to relational data. Several commercial DBMSs offer some support for storing and querying XML documents [10, 11, 22]. However, these engines do not provide automatic mappings from XML data to relational data, thus the user needs to design an appropriate storage mapping. A number of previous works have been proposed for automatic mapping from XML documents to relations [12, 14, 16, 23–25].

Deutsch, Fernandez, and Suciu [12] proposed the STORED system for mapping between the semistructured data model and the relational data model. They adapted a data mining algorithm to identify highly supported patterns for storage in relations. Along the lines of mapping XML data sets to relations, Florescu and Kossmann [14] proposed and evaluated a number of alternative mapping techniques. From their experimental results, the best overall approach is an approach based on separate *Attribute* tables for every attribute name, and inlining values into these Attribute tables. While these approaches require only an instance of XML data in the transformation process, Shanmugasundaram et al. [24] used the DTD to find a "good" storage mapping. They proposed three strategies to map DTDs into relational schemas and identified the *Hybrid inlining* algorithm as being superior to the other ones (in most cases). Most recently, Bohannon et al. [2] introduced a cost-based framework for XML-to-relational storage mapping that automatically finds the best mapping for a given configuration of an XML Schema, XML data statistics, and an XML query workload. Like [2, 24], we also use the schema of XML documents to derive a relational schema. However, unlike these previously discussed algorithms, we leverage the data type extensibility feature of ORDBMSs to provide a more efficient mapping. We compare the effectiveness of the XORator algorithm (using an ORDBMS) with the Hybrid algorithm (using an RDBMS), and show that the XORator algorithm generally performs significantly better.

Shimura et al. [25] proposed the method that decomposed XML documents into the nodes, and stored them in relational tables according to the node types. They defined a user data type to store a region of each node within a document. This data type keeps positions of nodes, and the methods (associated with the data type) determine ancestor-descendant and element order relationships. Schimdt et al. [23] proposed the Monet XML data model, which is based on the notion of binary associations, and showed that their approach had better performance than the approach proposed by Shimura et al. [25]. Since the Monet approach uses a mapping scheme that converts each distinct edge in DTD to a table, their mapping scheme produces a large number of tables. The Shakespeare DTD maps to four tables using the XORator algorithm, while it maps to ninety-five tables using the algorithm proposed in [23].

Techniques for resolving the data model and schema heterogeneity difference between the relational and XML data models have been examined [15]. The problem of preserving the semantics of the XML data model in the mapping process has also been addressed [17]. These techniques are complementary to the XORator algorithm of mapping based on the structural information of the XML data.

Our work is closest to the work proposed by Klettke and Meyer [16]. While their mapping scheme uses a combination of DTD, the statistics of sample XML documents, and the query workload to map XML data to ORDBMS data, the XORator algorithm only examines the DTD. Whereas there is no implementation or experimental evaluation presented in [16], we implement the XORator algorithm and compare it with the Hybrid algorithm. Furthermore, their mapping assumes the existence of the following type constructors: set-of, and list-of in ORDBMSs (which are not available in current commercial products), and requires that the user set a threshold specifying which attributes should be assigned to an XML data type. However, there are no guidelines provided in choosing a threshold. On the other hand, the XORator algorithm requires neither user input nor query workload. The XORator algorithm is a practical demonstration of the use of an XML data type and the advantage of using an ORDBMS over an RDBMS.

To the best of our knowledge, this paper is the first one that presents the implementation of an XML data type and the experimental results on the storage mappings with and without the XML data type. We also identify the causes for the limitations in the use of the XML data type and propose certain modifications to the relational engine that would make it better exploit the XML data type.

3 Storing XML Documents in an ORDBMS

In this section, we describe the XORator algorithm for generating an object-relational schema from a DTD. In our discussions below we will graphically represent a DTD using the DTD graph proposed by Shanmugasundaram et al. [24]. A sample DTD for describing Plays is shown in Figure 1, and the corresponding DTD graph is shown in Figure 3.

<!ELEMENT PLAY	(INDUCT?, ACT+)>
<!ELEMENT INDUCT	(TITLE, SUBTITLE*, SCENE+)>
<!ELEMENT ACT	(SCENE+, TITLE, SUBTITLE*, SPEECH+, PROLOGUE?)>
<!ELEMENT SCENE	(TITLE, SUBTITLE*, (SPEECH SUBHEAD)+)>
<!ELEMENT SPEECH	(SPEAKER, LINE)+>
<!ELEMENT PROLOGUE	(#PCDATA)>
<!ELEMENT TITLE	(#PCDATA)>
<!ELEMENT SUBTITLE	(#PCDATA)>
<!ELEMENT SUBHEAD	(#PCDATA)>
<!ELEMENT SPEAKER	(#PCDATA)>
<!ELEMENT LINE	(#PCDATA)>

Figure 1: A DTD of a Plays Data Set

Figure 1 shows a DTD which states that a PLAY element can have two subelements: INDUCT and ACT in that order. Symbol “?” followed INDUCT indicates that there can be zero or one occurrence of INDUCT subelement nested in each PLAY element. Symbol “+” followed ACT indicates that there can be one or more occurrences of ACT subelements nested in each PLAY element. At the second ELEMENT definition, symbol “*” followed SUBTITLE indicates that there is zero or more occurrences of SUBTITLE subelements nested in each INDUCT element. A subelement without any followed symbol represents that there must be only one occurrence of that subelement. For example, an ACT element must contain one and only one TITLE subelement. For more details about DTD, please refer to [4].

3.1 Reducing DTD Complexity

The first step in the mapping process is to simplify the DTD information to a form that makes the mapping process easier. We start by applying the set of rules proposed in [24] to simplify the complexity of DTD

element specifications. These transformations reduce the number of nested expressions and the number of element items. Examples of these transformations are as follow:

- Flattening (to convert a nested definition into a flat representation): $(e_1, e_2)^* \rightarrow e_1^*, e_2^*$
- Simplification (to reduce multiple unary operators into a single unary operator) : $e_1^* \rightarrow e_1^*$
- Grouping (to group subelements that have the same name): $e_0, e_1^*, e_1^*, e_2 \rightarrow e_0, e_1^*, e_2$

In addition, e^+ is transformed to e^* .

The simplified version of the DTD shown in Figure 1 is depicted in Figure 2.

<!ELEMENT PLAY	(INDUCT?, ACT*)>
<!ELEMENT INDUCT	(TITLE, SUBTITLE*, SCENE*)>
<!ELEMENT ACT	(SCENE*, TITLE, SUBTITLE*, SPEECH*, PROLOGUE?)>
<!ELEMENT SCENE	(TITLE, SUBTITLE*, SPEECH*, SUBHEAD*)>
<!ELEMENT SPEECH	(SPEAKER*, LINE*)>
<!ELEMENT PROLOGUE	(#PCDATA)>
<!ELEMENT TITLE	(#PCDATA)>
<!ELEMENT SUBTITLE	(#PCDATA)>
<!ELEMENT SUBHEAD	(#PCDATA)>
<!ELEMENT SPEAKER	(#PCDATA)>
<!ELEMENT LINE	(#PCDATA)>

Figure 2: A DTD of a Plays Data Set (Simplified Version)

3.2 Building a DTD Graph

After simplifying the DTD using the simplification rules [24], we build a DTD graph to represent the structure of the DTD. Nodes in the DTD graph are elements, attributes, and operators. Unlike the DTD graph proposed by Shanmugasundaram et al. [24] where each element below a * node appears exactly once, in our DTD graph, elements that contain characters are duplicated to eliminate the sharing. To illustrate the application of this rule, consider the the SUBTITLE element which is an element of type PCDATA (contains characters). In the DTD graph [24], the SUBTITLE element appears only once, as shown in Figure 3.

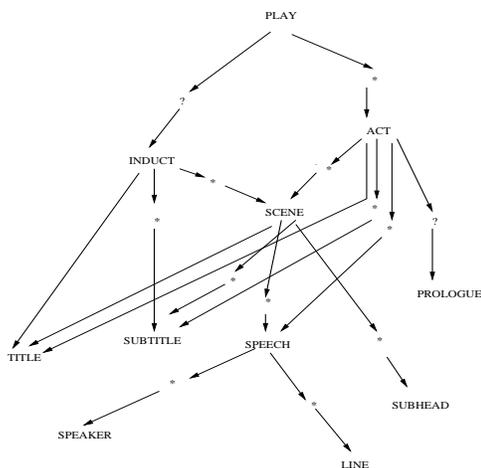


Figure 3: The DTD Graph for the Plays DTD

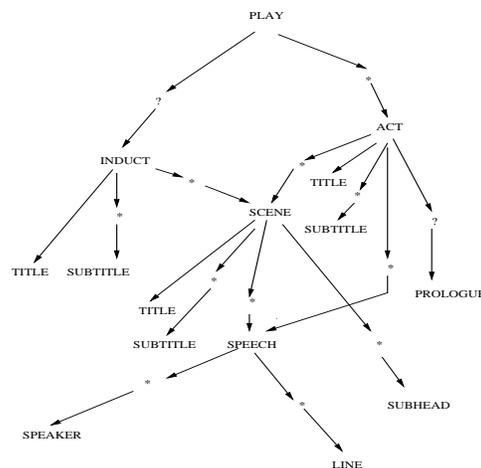


Figure 4: The Revised DTD Graph for the Plays DTD

We choose to decouple the shared SUBTITLE element by rewriting the DTD graph, as shown in Figure 4. The advantage of this approach is that fewer joins are required for queries that involve the SUBTITLE element and its parent elements in the DTD graph (such as the INDUCT or the ACT elements) when the SUBTITLE element is represented directly in the table corresponding to the parent attribute. The disadvantage of this

approach is that queries on the SUBTITLE elements must now query *all* tables that contain data corresponding to the SUBTITLE element. In the future, we plan to take the query workload (if it is available) into account during the transformation.

3.3 XORator: Mapping DTD to an ORDBMS Schema

The next step is to map this DTD graph to constructs in an ORDBMS schema. For this purpose, the XORator algorithm builds on the procedure used in the Hybrid algorithm [24]. The procedure used in the Hybrid algorithm is summarized as follows. After creating a DTD graph, the Hybrid algorithm creates an element graph which expands the relevant part of the DTD graph into a tree structure. Given an element graph, a relation is created for nodes that satisfy any of these following conditions: 1) nodes that have an in-degree of zero, 2) nodes that directly below a * operator, 3) recursive nodes with in-degree greater than one, and 4) one node among mutually recursive nodes with in-degree one. All remaining nodes (nodes not mapped to a relation) are inlined as attributes under the relation created for their closest ancestor nodes (in the element graph).

On the other hand, the XORator algorithm creates only a DTD graph, and not all nodes below a * operator are mapped to a relation. The XORator algorithm allows mapping an entire subtree of the DTD graph to an attribute of the XADT. An XADT attribute can store a fragment of an XML document, and its interfaces are described in Section 3.4. The implementation details of the XADT in DB2 are described in Section 4.1.

Using the XADT, the XORator algorithm applies the following rules:

1. If a non-leaf node N in the DTD graph is accessed by only one node, and if there is no link incident any descendant of the node, then node N is assigned to an XADT attribute. If node N is assigned to a relation, then queries on this node and its parent require a join. This rule identifies maximal subgraphs that are connected to the remaining nodes in the graph by a single node. Each subgraph is mapped to an XADT attribute.
2. If a non-leaf node below a * node is accessed by multiple nodes, then it is assigned to a relation. For nodes that are mapped to relations, the ancestors of these nodes must also be assigned as relations.
3. If a leaf node is below a * node, then it is assigned as an attribute of the XADT. Otherwise, it is assigned as an attribute of string type.

The schemas of the relations produced by the two algorithms are shown in Figures 5 and 6 respectively.

play	<i>(playID:integer)</i>
act	<i>(actID:integer, act_parentID:integer, act_childOrder:integer, act_title:string, act_prologue:string)</i>
scene	<i>(sceneID:integer, scene_parentID:integer, scene_childOrder:integer, scene_title:string)</i>
induct	<i>(inductID:integer, induct_parentID:integer, induct_childOrder:integer, induct_title:string)</i>
speech	<i>(speechID:integer, speech_parentID:integer, speech_parentCode:string, speech_childOrder:integer)</i>
subtitle	<i>(subtitleID:integer, subtitle_parentID:integer, subtitle_parentCode:integer, subtitle_childOrder:integer, subtitle_value:string)</i>
subhead	<i>(subheadID:integer, subhead_parentID:integer, subhead_childOrder:integer, subhead_value:string)</i>
speaker	<i>(speakerID:integer, speaker_parentID:integer, speaker_childOrder:integer, speaker_value:string)</i>
line	<i>(lineID:integer, line_parentID:integer, line_childOrder:integer, line_value:string)</i>

Figure 5: The Relational Schema transformed using the Hybrid Algorithm

Fields shown in *italic* are primary keys. As introduced in the mapping algorithms proposed in [24], each relation has an ID field to serve as the primary key for that relation; and all relations corresponding to element nodes having a parent also have a parentID field to serve as a foreign key to the parent tuple. Moreover, all relations corresponding to element nodes that have multiple parents have a parentCODE field to identify the corresponding parent tables. In this paper, we add a childOrder field to serve as the order number of the element among its siblings.

play	(<i>playID:integer</i>)
act	(<i>actID:integer, act_parentID:integer, act_childOrder:integer, act_title:string, act_subtitle:XADT, act_prologue:string</i>)
scene	(<i>sceneID:integer, scene_parentID:integer, scene_childOrder:integer, scene_title:string, scene_subtitle:XADT, scene_subhead:XADT</i>)
induct	(<i>inductID:integer, induct_parentID:integer, induct_childOrder:integer, induct_title:string, induct_subtitle:XADT</i>)
speech	(<i>speechID:integer, speech_parentID:integer, speech_parentCode:string, speech_childOrder:integer, speech_speaker:XADT, speech_line:XADT</i>)

Figure 6: The Relational Schema transformed using the XORator Algorithm

3.4 Defining an XML Data Type (XADT)

There are two aspects in designing the XADT: choosing a storage format for the data type, and defining appropriate methods on the data type. We discuss each of these aspects in turn.

3.4.1 Storage Alternatives for the XADT

A naive storage format is to store in the attribute the text string corresponding to the fragment of the XML document. Since a string may have many repeated element tag names, this storage format may be inefficient. An alternative storage representation is to use a *compressed* representation for the XML fragment. The approach that we adopt in this paper is to use a compression technique inspired by the XMill compressor [18]. The element tags are mapped to integer codes, and element tags are replaced by these integer codes. A small dictionary is stored along with the XML fragment to record the mapping between the integer codes and the actual element tag names.

In some cases where there are few repeated tags in the XADT attribute, the compression increases the storage size because of the dictionary space. Consequently, we have two implementations of the XADT: one that uses compression, and the other one that does not. The decision to use the “correct” implementation of the XADT is made during the document transformation process by monitoring the effectiveness of the compression technique. This is achieved by randomly parsing a few sample documents to obtain the storage space sizes in both uncompressed and compressed versions. Compression is used only if the space efficiency is above a certain threshold value.

3.4.2 Methods on the XADT

In addition to defining the required methods for input and output on the XADT, we also define the following methods:

1. **XADT getElm**(*XADT inXML, VARCHAR rootElm, VARCHAR searchElm, VARCHAR searchKey, INTEGER level*):

This method examines the XML fragment stored in *inXML*, and returns all *rootElm* elements that have *searchElm* within a depth of *level* from the *rootElm*. A default value for *level* indicates that the level information is to be ignored. If *searchKey* and *searchElm* are specified, this method only considers the *searchElm* that contains the *searchKey* keyword. If only *searchKey* is an empty string, then it returns all *rootElm* elements that have *searchElm* as subelements. If only *searchElm* is an empty string, then it returns all *rootElm* elements. If both *searchElm* and *searchKey* are empty strings, this method returns all *rootElm* elements in the *inXML* fragment.

The above function answers a simple path query with two element tag names, but more complex path queries can be answered by a composition of multiple calls to this function. This function takes an

XADT attribute as input and produces an XADT output which can then be an input to another call of this function.

2. **INTEGER findKeyInElm**(XADT *inXML*, VARCHAR *searchElm*, VARCHAR *searchKey*):

This method examines all elements with the tag name *searchElm* in *inXML*, and searches for all *searchElm* elements with content that matches the *searchKey* keyword. As soon as the first *searchElm* element that contains *searchKey* is found, the function returns a value of 1 (true). Otherwise, the function returns a value of 0 (false). If only *searchKey* is an empty string, this method simply checks whether *inXML* contains any *searchElm* elements. If only *searchElm* is an empty string, this method simply checks whether *searchKey* is part of the content of any element in *inXML*. Both *searchElm* and *searchKey* cannot be empty strings at the same time.

This function is a special case of the *getElm* method defined above, and is implemented for efficiency purposes only.

3. **XADT getElmIndex**(XADT *inXML*, VARCHAR *parentElm*, VARCHAR *childElm*, INTEGER *startPos*, INTEGER *endPos*): This method returns all *childElm* elements that are children of the *parentElm* elements and with the sibling order from *startPos* to *endPos* positions. If only *parentElm* is an empty string, then *childElm* is treated as the root element in the XADT. Note that *childElm* cannot be an empty string.

In this paper, we only use the three methods described above, however, more specialized methods can be implemented to improve the performance using the XADT even further.

Sample queries in both algorithms posed on a data set describing Shakespeare Play are depicted in Figures 7 and 8. The DTD for this data set is shown in Figure 10. Figure 7 shows query QE1, which retrieves lines that are spoken in acts by the speaker HAMLET and have the keyword ‘friend’ in the line. Figure 7(a) shows the uses of the XADT methods: *getElm* and *findKeyInElm*, and Figure 7(b) shows the query QE1 executed over the database produced by the Hybrid algorithm.

```
SELECT    getElm(speech_line, 'LINE', 'LINE', 'friend')
FROM      speech, act
WHERE     findKeyInElm(speech_speaker,
                      'SPEAKER', 'HAMLET') = 1
AND       findKeyInElm(speech_line, 'LINE',
                      'friend') = 1
AND       speech_parentID = act_ID
AND       speech_parentCODE = 'ACT'
```

(a) Using the XORator Algorithm

```
SELECT    line_val
FROM      speech, act, speaker, line
WHERE     speech_parentID = act_ID
AND       speech_parentCODE = 'ACT'
AND       speaker_parentID = speech_ID
AND       speaker_val = 'HAMLET'
AND       line_parentID = speech_ID
AND       line_val like '%friend%'
```

(b) Using the Hybrid Algorithm

Figure 7: Query QE1 in Both Algorithms

Figure 8 shows query QE2, which returns the second line in each speech. Figure 8(a) shows the uses of the XADT method: *getElmIndex*, and Figure 8(b) shows the query QE2 executed over the database produced by the Hybrid algorithm.

```
SELECT    getElmIndex(speech_line, '', 'LINE', 2, 2)
FROM      speech
```

(a) Using the XORator Algorithm

```
SELECT    line_val
FROM      speech, line
WHERE     line_parentID = speech_ID
AND       line_childOrder = 2
```

(b) Using the Hybrid Algorithm

Figure 8: Query QE2 in Both Algorithms

3.5 Unnest Operator

In addition to the functionality provided by the methods described above, to answer queries posed on the XML data we also need an *unnest* operator. As described in Section 3.3, using the XORator algorithm, it is possible to map an entire subtree of a DTD graph below a * node to an XADT attribute. One can then view the XADT attribute as a set of XML fragment trees. When a query needs to examine individual elements in the set, an *unnest* operator is required. For example, for the Plays DTD (of Figure 1), consider the query that requests a distinct list of all speakers who speak in at least one play. In our approach, speakers are stored as an XADT attribute. It is possible that one speech has many speakers. Thus the speaker attribute, which is of type XADT, of a speech tuple can store the XML fragment, such as `<speaker>s1</speaker><speaker>s2</speaker>`, while another speech tuple could have a single speaker stored as `<speaker>s1</speaker>`. To evaluate the query, we need to first *unnest* the speakers and then retrieve distinct speakers. In our implementation, we define such an *unnest* operation using a table User-Defined Function (UDF). A table UDF is an external UDF which delivers a table to the SQL query in which it is referenced. A table UDF can be invoked in the FROM clause of a SQL statement, and produces a table with tuples in an unnested form.

Figure 9 shows the content of table `speakers` before we *unnest* the `speaker` attribute and the result of the query that *unnests* the `speaker` attribute.

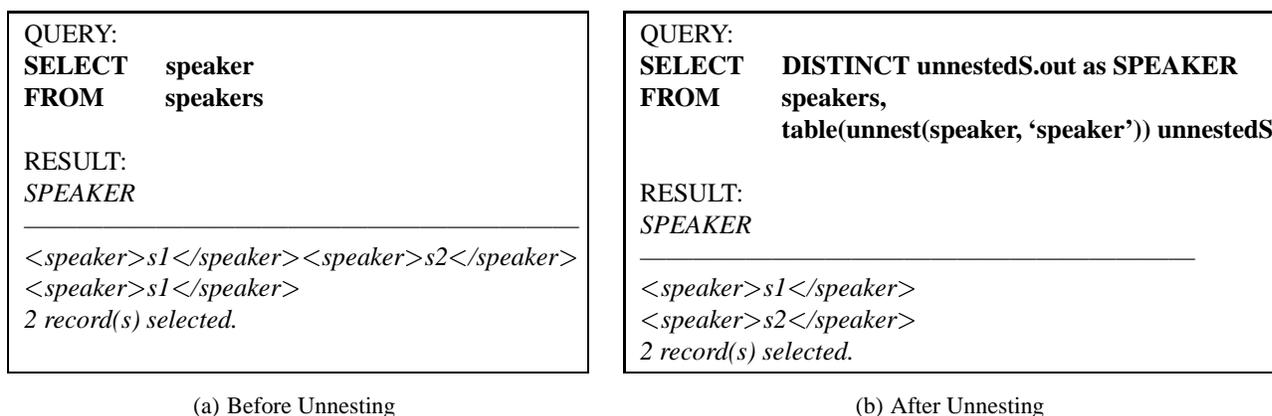


Figure 9: Before and After Unnesting `speaker` Attribute

The first parameter (`speaker`) of the table UDF `unnest` is the attribute name that contains nested elements. The second parameter of this function, `'speaker'`, is the tag name of elements to be *unnested*. `unnestedS` is the name of the table that is returned from this function, `unnest`. This table has one attribute, `out`, which contains the *unnested* elements.

4 Performance Evaluation

In this section, we present the results of implementing the XORator algorithm (along with the XADT), and the results comparing its effectiveness with the Hybrid algorithm [24].

We evaluated the effectiveness of the two algorithms using both real and synthetic data sets. For the real data set, we used the well-known Shakespeare plays data set [3]. In Section 4.3, we present the experimental results comparing the Hybrid and the XORator algorithms using this data set. We also tested the XORator algorithm using a data set that is “deep” and would force the XORator algorithm to map large XML fragments to XADT attributes. The deep DTD allows us to test how effective the XORator algorithm is when most of the data may be inside the XADT attribute. We used the SIGMOD Proceedings DTD [20] for this purpose. Since we wanted large data sets, we used an XML document generator [9] to generate data conforming to this DTD. The results of the experiment with this DTD are presented in Section 4.4.

4.1 Implementation Overview

We implemented the XADT in DB2 UDB V.7.2 for Windows. Two versions of the XADT were implemented based on the two storage alternatives discussed in Section 3.4.1. The first implementation stores all the element tag names as strings. The second one stores all the element tag names as integers and uses a dictionary to encode the tags as integers, thereby storing the XML data in a compressed format. In both cases, the XADT was implemented on top of the VARCHAR data type provided by DB2. We used the C string functions to implement the methods outlined in Section 3.4.2.

To parse the original XML documents, we used the IBM's Alphawork Java XML Parser Release (XML4J V.2.0.15) [8]. We modified the parser so that it reads the DTD and applies the XORator algorithm and the Hybrid algorithm, generating SQL commands to create tables in DB2.

The modified parser also chooses the storage alternative. The compressed format is chosen only if it reduces the storage space by at least 20%. In our current implementation all tuples in a table with an attribute of the XADT use the same storage representation. To decide which storage alternative to use, we randomly parse a few sample documents to obtain the storage space sizes in both uncompressed and compressed cases.

4.2 Experimental Platform and Methodology

We performed all experiments on a single-processor 550 MHz Pentium III machine running Windows 2000 V.5.0 with 256 MB of main memory. We used the IBM DB2 V.7.2 as the database system. All the UDFs on the XADT are run in a *NOT FENCED* mode in DB2. We chose to run in the *NOT FENCED* mode because the *FENCED* option causes the UDF run in an address space that is separated from the database address space, and this causes a significant performance penalty [7]. DB2 was configured to use a page size of 8MB, and the use of hash joins was enabled.

Before executing queries in both algorithms, we created indexes as suggested by the DB2 Index Wizard, and collected statistics. The execution times reported in this section are cold numbers. Each query was run five times, and we report the average of the middle three execution times.

4.3 Experiments Using the Shakespeare Plays Data Set

In this experiment, we loaded 37 XML Shakespeare play documents (size 7.5 MB) into DB2 using the two mapping algorithms. The DTD corresponding to the Shakespeare Plays data set [3] is shown in Figure 10.

<!ELEMENT PLAY	(TITLE,FM,PERSONAE,SCNDESCR,PLAYSUBT,INDUCT?,PROLOGUE?,ACT+,EPILOGUE?)>	<!ELEMENT ACT	(TITLE,SUBTITLE*,PROLOGUE?,SCENE+,EPILOGUE?)>
<!ELEMENT TITLE	(#PCDATA)>	<!ELEMENT SCENE	(TITLE,SUBTITLE*,(SPEECH STAGEDIR SUBHEAD)+)>
<!ELEMENT FM	(P+)>	<!ELEMENT PROLOGUE	(TITLE,SUBTITLE*,(STAGEDIR SPEECH)+)>
<!ELEMENT P	(#PCDATA)>	<!ELEMENT EPILOGUE	(TITLE,SUBTITLE*,(STAGEDIR SPEECH)+)>
<!ELEMENT PERSONAE	(TITLE,(PERSONA PGROUP)+)>	<!ELEMENT SPEECH	(SPEAKER+,(LINE STAGEDIR SUBHEAD)+)>
<!ELEMENT PGROUP	(PERSONA+,GRPDESCR)>	<!ELEMENT SPEAKER	(#PCDATA)>
<!ELEMENT PERSONA	(#PCDATA)>	<!ELEMENT LINE	(#PCDATA STAGEDIR)*>
<!ELEMENT GRPDESCR	(#PCDATA)>	<!ELEMENT STAGEDIR	(#PCDATA)>
<!ELEMENT SCNDESCR	(#PCDATA)>		
<!ELEMENT PLAYSUBT	(#PCDATA)>		
<!ELEMENT INDUCT	(TITLE,SUBTITLE*,(SCENE+ (SPEECH STAGEDIR SUBHEAD)+))>		

Figure 10: The DTD of Shakespeare Data Set

For this data set, the XORator algorithm chooses not to use the compressed storage alternative since the compressed representation actually increases the storage size. The schemas for these relations are presented in the extended version of this paper [21]. Table 1 shows the comparisons of the number of tables, database sizes, and index size between the two algorithms. The sizes of the database produced by the XORator algorithm is about 60% of the size of the database produced by the Hybrid algorithm.

	Hybrid	XORator
Number of tables	17	7
Database size (MB)	15	9
Index size (MB)	30	3

Table 1: Comparisons between the Two Algorithms When Using the Shakespeare Data Set

To produce data sets that are larger than the base data set (size 7.5 MB), we took the original Shakespeare data set and loaded it multiple times, producing data sets that were two, four and eight times the original database size. We call these configurations DSx2, DSx4, and DSx8 respectively. We call the original configuration DSx1.

The query set in this experiment is described below:

QS1: Flattening List speakers and the lines that they speak.

QS2: Full path expression Retrieve all lines that have stage directions associated with the lines.

QS3: Selection Retrieve the lines that have the keyword “Rising” in the text of the stage direction.

QS4: Multiple selections Retrieve the speeches spoken by the speaker “Romeo” in the play “Romeo and Juliet.”

QS5: Twig with selection Retrieve the speeches in the play “Romeo and Juliet” spoken by the speaker “Romeo” and the lines in the speech that contain the keyword “love.”

QS6: Order access Retrieve the second line in all speeches that are in prologues.

We chose this simple set of queries because it allows us to study the core performance issues and is an indicator of performance for more complex queries. In this paper, we do not focus on automatically rewriting XML queries into equivalent SQL queries. We refer the reader to proposed query rewriting algorithms from XML query languages to SQL [6, 25]. The SQL statements corresponding to the above queries for both algorithms are presented in the extended version of this paper [21].

The data loading times and the result of executing queries QS1 through QS6 for the various data sets are shown in Figure 11. In this Figure, we plot the ratios of the execution times using the Hybrid and the XORator algorithms on a **log** scale. Since the size of the database produced using the XORator algorithm is about 60% of the size of the database produced using the Hybrid algorithm, the XORator algorithm results in much less loading times than the Hybrid algorithm. The XORator algorithm also results in significantly better execution times for all queries, except query QS6. **In most queries, the XORator algorithm is an order of magnitude faster than the Hybrid algorithm.** This is because all queries (on the database produced) using the XORator algorithm requested at least one fewer join than the corresponding query (on the database produced) using the Hybrid algorithm. In the cases of query QS6, the response times of the XORator algorithm are more than those of the Hybrid algorithm. This is because the database needs to scan the XADT attribute to extract elements in the specified order when using the XORator algorithm, while the database needs to only extract the value of the element order attribute when using the Hybrid algorithm.

As the data size increases, the Hybrid technique is generally less scalable than the XORator technique because the database using the Hybrid technique scans more data and uses fewer indexes. As shown in Figure 11, the ratios of the response times of some queries, such as query QS3, do not always increase as the data size increases. This is because the optimizer chooses different plans for different data set sizes. Note that the query optimizer has the most up-to-date statistics since we always ran the “runstats” command and created indexes as suggested by the DB2 Index Wizard before executing the queries.

4.4 Experiments Using the Synthetic Data Set

This section presents the experimental results from using the SIGMOD Proceedings data set. The DTD of this data set is an example of a deep DTD. With such a DTD, the XORator technique maps large fragments of XML data to the XADT attributes. So this DTD is representative of the worst-case scenario for the XORator algorithm. The DTD corresponding to the SIGMOD Proceedings data set has seven levels. Elements, such as “author”, that are likely to be queried often, are at the bottom-most level. The DTD is depicted in Figure 12

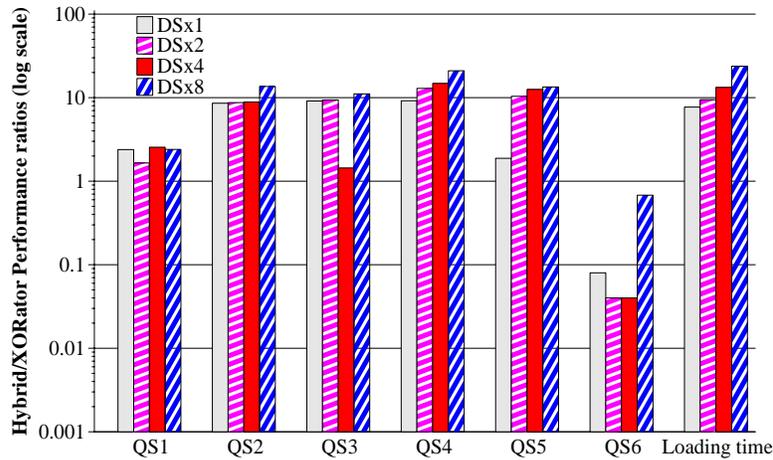


Figure 11: Hybrid/XORator Performance Ratios (log scale) as Database Sizes Increase

<!ELEMENT PP (volume,number,month,year,conference, date,confyear,location,sList)>	<!ELEMENT aTuple (title,authors,initPage,endPage, Toindex,fullText)>
<!ELEMENT volume (#PCDATA)>	<!ELEMENT title (#PCDATA)>
<!ELEMENT number (#PCDATA)>	<!ATTLIST title articleCode CDATA #IMPLIED>
<!ELEMENT month (#PCDATA)>	<!ELEMENT authors (author)*>
<!ELEMENT year (#PCDATA)>	<!ELEMENT author (#PCDATA)>
<!ELEMENT conference (#PCDATA)>	<!ATTLIST author AuthorPosition CDATA #IMPLIED >
<!ELEMENT date (#PCDATA)>	<!ELEMENT initPage (#PCDATA)>
<!ELEMENT confyear (#PCDATA)>	<!ELEMENT endPage (#PCDATA)>
<!ELEMENT location (#PCDATA)>	<!ELEMENT Toindex (index)?>
<!ELEMENT sList (sListTuple)*>	<!ELEMENT index (#PCDATA)>
<!ELEMENT sListTuple (sectionName,articles)>	<!ATTLIST Toindex %Xlink;>
<!ELEMENT sectionName (#PCDATA)>	<!ELEMENT fullText (size)?>
<!ATTLIST sectionName SectionPosition CDATA #IMPLIED >	<!ELEMENT size (#PCDATA)>
<!ELEMENT articles (aTuple)*>	<!ATTLIST fullText %Xlink;>

Figure 12: The DTD of the SIGMOD Proceedings Data Set

In this experiment, we loaded the 3000 documents (size 12 MB) into DB2. For this data set, the XORator algorithm chooses to use the compressed storage alternative since the compressed representation reduces database size by about 38%. Table 2 shows the comparisons of the number of tables, database sizes, and index size between the two algorithms. The sizes of the database produced by the XORator algorithm is about 65% of the size of the database produced by the Hybrid algorithm. Please refer to the extended version of this paper [21] for the schemas of these relations.

	Hybrid	XORator
Number of tables	7	1
Database size (MB)	23	15
Index size (MB)	34	2

Table 2: Comparisons between the two Algorithms When Using the SIGMOD Proceedings Dataset

To produce data sets that are larger than the base data set (size 12 MB), we took the original SIGMOD Proceedings data set and loaded it multiple times, producing data sets that were two, four and eight times the original database size. We call these configurations DSx2, DSx4, and DSx8 respectively. We call the original configuration DSx1.

The query set in this experiment is described below:

QG1: Selection and extraction Retrieve the authors of the papers with the keyword “Join” in the paper title.

- QG2: Flattening** List all authors and the names of the proceeding sections in which their papers appear.
- QG3: Flattening with selection** Retrieve the proceeding section names that have the papers published by authors whose names have the keyword “Worthy.”
- QG4: Aggregation** For each author, count the number of proceeding sections in which the author has a paper.
- QG5: Aggregation with selection** Count the number of proceeding sections that have papers published by authors whose names have the keyword “Bird.”
- QG6: Order access with selection** Retrieve the second author of the papers with the keyword “Join” in the paper title.

The SQL statements corresponding to the above queries for both algorithms are presented in the extended version of this paper [21].

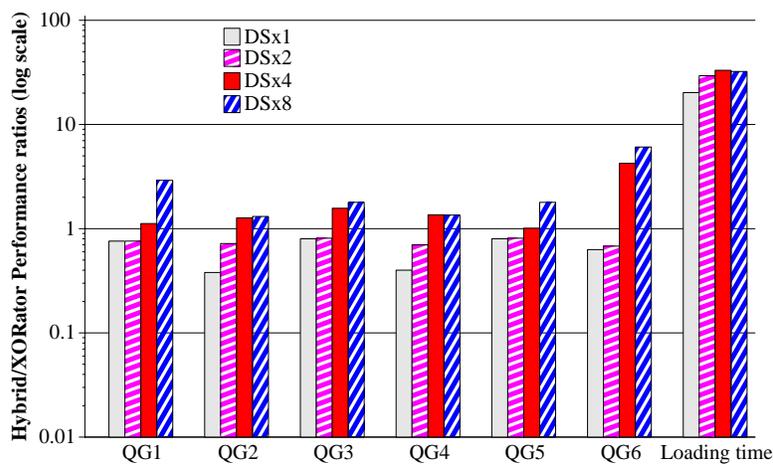


Figure 13: Hybrid/XORator Performance Ratios (log scale) as Database Size Increases

The performance comparison between the Hybrid technique and the XORator technique (with the XADT implemented in compressed version) for the various data sets is shown in Figure 13. Since the size of the database produced using the XORator algorithm is about 65% of the size of the database produced using the Hybrid algorithm, the XORator algorithm results in much less loading times than the Hybrid algorithm. Two observations can be made based on Figure 13: a) when the size of data is small (DSx1 and DSx2), the XORator algorithm performs worse than the Hybrid algorithm; and b) when the size of data becomes large (DSx4 and DSx8), the XORator algorithm outperforms the Hybrid algorithm.

When the amount of data is small, the XORator algorithm results in worse execution times for all queries. With the SIGMOD Proceedings data set, all data is mapped to a single table. Consequently, there is no table join in the query, but each query has four to eight calls of UDFs to extract subelements or to join elements inside the XADT attribute. The cost of invoking UDFs is a significant component of the query evaluation of the XORator algorithm. To investigate if a UDF incurs a higher performance penalty than an equivalent built-in function, we conduct the following experiment. We implemented two string functions, namely “return length” and “return substring”, using built-in string functions and using UDFs. The experiment consists of two queries:

QT1: Return the length of string in the SPEAKER attribute

QT2: Return a substring of string in the SPEAKER attribute from the fifth position to the last position

Both queries were run on the Shakespeare data set and returned 31,028 tuples. In the built-in function case, we used `length` and `substr` functions in queries QT1 and QT2 respectively. In the UDF case, we used UDFs that called the C functions `strlen` and `strncpy` in queries QT1 and QT2 respectively. The results of these experiments are shown in Figure 14 (the exact time taken to run this query is deliberately not shown

in this Figure). Compared to using a built-in function, using the equivalent UDF is approximately 40% more expensive.

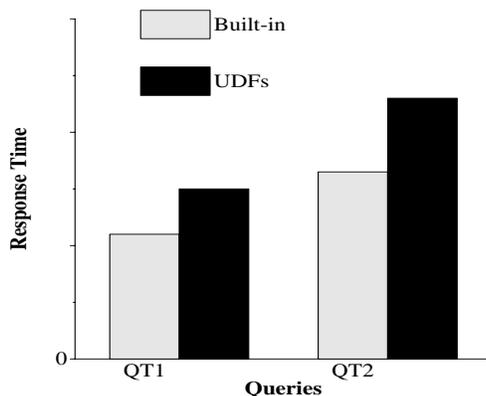


Figure 14: Overhead in Invoking UDFs

Invoking UDFs are expensive for two reasons. First, our implementation of the methods on the XADT use string compare and copy functions on the VARCHAR. This sometimes requires scanning a large amount of data. Perhaps, if we have the metadata associated with each XADT attribute to help us quickly access the starting position of each element stored inside the XADT data, the performance may be improved. We plan on investigating this issue further in the future.

Second, the cost of evaluating a UDF is actually higher compared to the cost of evaluating an equivalent built-in function, as shown in Figure 14. If the XADT were implemented as a native data type (by the database vendors), we would expect the overhead in invoking the methods associated with the XADT would be reduced significantly, making the XORator technique more competitive.

Regarding the second observation in Figure 13, as the data size increases, the ratios of the response times between the Hybrid and the XORator algorithms become more than one (i.e., the XORator technique starts to outperform the Hybrid technique). The reason is that queries using the XORator algorithm typically have no join and thus the response times grow at $O(n)$ rate (scan cost). However, the queries using the Hybrid algorithm, which typically have many joins, grow at either $O(n \log n)$ rate (merge sort join cost), or $O(n^2)$ rate (nested loop join cost), where n is the number of tuples.

5 Conclusion and Future Work

We have proposed and evaluated XORator, an algorithm for mapping XML document with DTDs to relations in an Object-Relational DBMS. Using the type-extensibility mechanisms provided by an ORDBMS, we added a new data type called XADT that can store and query arbitrary fragments of an XML document. The XORator algorithm maps a DTD to an object-relational schema that may have attributes of type XADT. Using an implementation in DB2, we show that the XORator algorithm generally outperforms the well-known Hybrid algorithm that is used for mapping XML documents to relations in an RDBMS. The primary reason for this superior performance is that queries in the XORator algorithm usually execute fewer number of joins. We also show that it is important to pay close attention to the implementation and evaluation of the UDF. Perhaps, if the database vendors implemented the XADT as a native data type, the overhead in invoking the methods associated with the XADT be reduced, making the XORator algorithm more effective.

For future work, we will expand the mapping rules to accommodate additional factors, such as the query workload, and the statistics of XML data, including the number of levels and the size of the data that is in an XML fragment. We will also investigate storing of metadata with the XADT attribute to improve the performance of the methods on the XADT.

6 Acknowledgements

We would like to thank H.V. Jagadish and Atul Prakash for providing insightful comments. We also would like to thank Kelly Lyons and Chun Zhang for providing information about UDFs. Finally, we would like to thank Hamid Pirahesh and Berthold Reinwald for allowing Kanda Runapongsa to work on this paper during her summer internship at IBM Almaden Research Center.

References

- [1] Software AG. Tamino - The Information Server for Electronic Business, 2000. <http://www.softwareag.com/tamino/>.
- [2] P. Bohannon, J. Freire, P. Roy, and J. Simeón. From XML Schema to Relations: A Cost-Based Approach to XML Storage. In *Proceedings IEEE International on Data Engineering*, San Jose, California, February 2002.
- [3] J. Bosak. The Plays of Shakespeare in XML, July 1999. <http://metalab.unc.edu/xml/examples/shakespeare/>.
- [4] J. Bosak, T. Bray, D. Connolly, E. Maler, G. Nicol, C.M. Sperberg-McQueen, L. Wood, and J. Clark. W3C XML Specification DTD, June 1998. <http://www.w3.org/XML/1998/06/xmlspec-report-19980910.htm>.
- [5] T. Bray, J. Paoli, and C.M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0, February 1998. <http://www.w3.org/TR/REC-xml.html>.
- [6] M.J. Carey, J. Kiernan, J. Shanmugasundaram, E.J. Shekita, and S.N. Subramanian. XPERANTO: A Middleware for Publishing Object-Relational Data as XML Documents. In *Proceedings International Conference Very Large Data Bases*, pages 646–648, Cairo, Egypt, September 2000.
- [7] D. Chamberlin. *Using The New DB2: IBM's Object-Relational Database System*. Morgan Kaufmann Publishers, Inc., San Francisco, California, 1996.
- [8] IBM Corporation. XML Parser for Java., Febuary 1998. <http://www.alphaworks.ibm.com/>.
- [9] IBM Corporation. IBM XML Generator, September 1999. <http://www.alphaworks.ibm.com/tech/xmlgenerator>.
- [10] IBM Corporation. IBM DB2 UDB XML Extender Administration and Programming, March 2000. <http://www-4.ibm.com/software/data/db2/extenders/xmlext/docs/v71wrk/dxx%awmst.pdf>.
- [11] Oracle Corporation. Oracle XML SQL Utility. http://otn.oracle.com/tech/xml/oracle_xsu/content.html.
- [12] A. Deutsch, M. F. Fernandez, and D. Suciu. Storing Semistructured Data with STORED. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 431–442. ACM Press, 1999.
- [13] D. Florescu, G. Graefe, G. Moerkotte, H. Pirahesh, and H. Schning. Panel: XML data management: Go Native or spruce up Relational Systems? In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Santa Barbara, California, May 2001. (Panel Chair: Per-Ake Larson).

- [14] D. Florescu and D. Kossmann. A Performance Evaluation of Alternative Mapping Schemes for Storing XML Data in a Relational Database. In *Rapport de Recherche No.3684*, INRIA,Rocquencourt,France, March 1999.
- [15] G. Kappel, E. Kapsammer, S. Raush-Schott, and W. Retschegger. X-Ray - Towards Integrating XML and Relational Database Systems. In *International Conference on Conceptual Modeling (ER)*, pages 339–353, Utah, USA, October 2000.
- [16] M. Klettke and H. Meyer. XML and Object-Relational Database Systems - Enhancing Structural Mappings Based on Statistics. In *International Workshop on the Web and Databases*, Dallas, Texas, May 2000. <http://dbms.uni-muenster.de/events/webdb2000/>.
- [17] D. Lee and W. W. Chu. Constraints-preserving Transformation from XML Document Type Definition to Relational Schema. In *International Conference on Conceptual Modeling (ER)*, pages 323–338, October 2000.
- [18] H. Liefke and D. Suciu. XMill: an Efficient Compressor for XML Data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 153–164, Dallas, Texas, May 2000.
- [19] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A Database Management System for Semistructured Data. *SIGMOD Record*, 26(3):54–66, September 1997. <http://www-db.stanford.edu/lore/pubs/lore97.ps>.
- [20] Sigmod Record. Sigmod Record: XML Edition. <http://www.dia.uniroma3.it/Araneus/Sigmod/Record/DTD/>.
- [21] K. Runapongsa and J. M. Patel. Storing and Querying XML Data in ORDBMSs. <http://www.eecs.umich.edu/~krunapon/pubs/xorator.pdf>.
- [22] M. Rys. State-of-the-art Support in RDBMS:Microsoft SQL Server’s XML Features. *Bulletin of the Technical Committee on Data Engineering*, 24(2):3–11, June 2001.
- [23] A.R. Schmidt, M.L. Kersten, M. Windhouwer, and F. Waas. Efficient Relational Storage and Retrieval of XML Documents. In *WebDB’2000 Third International Workshop on the Web and Databases*, Dallas, Texas, May 2000. <http://dbms.uni-muenster.de/events/webdb2000/>.
- [24] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D.DeWitt, and J.Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *Proceedings International Conference Very Large Data Bases*, pages 302–314, Edinburgh, Scotland, September 1999.
- [25] T. Shimura, M. Yoshikawa, and S. Uemura. Storage and Retrieval of XML Documents Using Object-Relational Databases. In *International Conference on Database and Expert Systems Applications*, pages 206–217, Florence, Italy, September 1999.