A Performance Evaluation of Storing XML Data in Relational Database Management Systems

Latifur Khan
Department of Computer Science
University of Texas at Dallas
Richardson, TX 75083-0688
Email: lkhan@utdallas.edu

Yan Rao
Department of Computer Science
University of Texas at Dallas
Richardson, TX 75083-0688
Email: yanrao@utdallas.edu

ABSTRACT

XML is an emerging standard for the representation and exchange of Internet data. Along with document type definition (DTD), XML permits the execution of a collection of queries, using XPath to identify data in XML documents. In this paper we examine how XML data can be stored and queried using a standard relational database management system (RDBMS). For this, we propose a technique for automatic mapping from an XML document to relations within the RDBMS. We demonstrate that our novel approach preserves the nested structure of the XML documents. By hiding database details we devise a seamless, transparent framework for user access to XML data. In order to achieve this, we propose a novel mechanism for translating an XPath query into an SQL statement. Furthermore, we propose efficient techniques for the construction of an XML document on the fly from the result set of the SQL statement. We also present findings in terms of query response time on the comparative performance of different techniques for the construction of an XML document on the fly.

Keywords

XML, Relational DBMS, DTD, XPath, SQL

1. INTRODUCTION

XML [3] is an emerging standard for the representation and exchange of Internet data. It is obvious that relational, object-relational, or object-oriented data models, do not suffice to integrate data from several data sources in the web. To support this, semi-structured data models have been proposed. The nature of this semi-structured data is that it is self-descriptive, and that it incorporates an optional XML definition (DTD).

One of three alternative data models can be deployed for the persistent storage of semi-structured data (i.e., XML documents). First, the development of specialized data management systems can be noted, such as Rufus [11], Lore [1, 9], and Strudel [7]. These are tailored to store and retrieve XML documents using special purpose indices and techniques of query optimization. Second, for an object-oriented database management system, an O_2 [2], or Object store, can be used to store XML documents because of the rich capability of this database system. Third, when a relational database management system (RDBMS) is employed XML data is mapped into relations and queries posed in a semi-structured query language which is then translated into SQL queries.

It is not possible to reliably predict which of these three approaches will be widely accepted. The first, the use of a specialized or special purpose database system, may work best, once needs are met concerning scalability and the level of maturity required for the handling of huge amounts of data. The second, an object-oriented database system, seems well-suited to complex data like XML, but vulnerable in the area of evaluating queries addressed to a very large database.

The third approach, (RDBMS), provides maturity, stability, portability, and scalability [12]. Furthermore, since a majority of the data on the web currently resides in and will continue to be stored in RDBMS, the opportunity arises for constructing a system using a RDBMS to store XML documents, making it possible to seamlessly query of data with one system and one query language. Given all these advantages, we believe that a RDBMS will be a viable option.

We have proposed a novel and innovative approach for mapping XML documents into RDMBS relations. This mapping technique takes into account nested structure, and stores this information into the relation in an encoded form. Furthermore, the mapping technique relies on both DTD and content (the document itself). Our goal is to provide the user with transparent, seamless data access. In other words, RDBMS storage and query processing will be hidden from the user. XPath is used for querying. Since XPath

WIDM 2001, Atlanta, GA, USA
© ACM 2001 1-58113-444-4/01/11 ...$12.00

query is not supported by RDBMS (which supports SQL), we propose an efficient translation mechanism for the conversion of XPath to SQL statements.

The main contributions of this work are as follows:

(1) We propose an automatic mapping technique from an XML document to RDBMS. We demonstrate that our novel approach preserves the nested structure of XML documents.

(2) By hiding database detail we have devised a seamless, transparent framework for user access to XML data. For this, we propose a novel mechanism for the translation of XPath queries into SQL statements. Furthermore, we propose efficient techniques for the construction of an XML document on the fly from the result set.

The remainder of this paper is organized as follows: Section 2 covers related work. Section 3 covers XML basics. Section 4 states the problem. Section 5 describes a novel technique for the mapping of an XML document to relational databases by keeping the nested structure of XML documents. Section 6 describes the implementation of our system, including details about the performance of various approaches. Finally, section 7 presents our conclusions, and a comment about future work.

2. RELATED WORK

To store semi-structured data (i.e., XML documents) into persistence storage, three alternative approaches can be proposed: a special purpose database management system, an object-oriented database management system, and a relational database management system.

In regard to a special purpose database system, Rufus [11], Lore [1, 9], and Strudel [7] report the development of research prototypes, while Lotus Notes was developed as a commercial product [15]. These are tailored to store and retrieve XML documents using special purpose indices and techniques of query optimization.

Insofar as an object-oriented database management system is concerned, the rich capability of such a database system permits the use of O_2 [2] or Object store for the storage of XML documents (e.g. the MONET project [14]).

For a relational database management system one of two techniques can be considered. First, schema are extracted from XML documents based on semi-structured data [4, 6, 10]. By analyzing this semi-structured data, and the workload of a target application, efficient schema can be constructed. Thus, performance will be little concerned with the matter of how semi-structured data is stored in RDBMS. Second, rather than extracting a schema, different techniques are studied for storing XML documents in relational databases. Examination of how XML data can be mapped into tables or relations can be found in [8, 12, 13, 18, 19, 20]. Besides the pure relational case [13], an object-oriented approach is also proposed. Furthermore, all of these use XML-QL [5] from XML documents to extract data, while simply ignoring the restructured element, (i.e., the result of SQL could be a XML document).

Our approach is similar to [8, 13, 17]. However, it differs from these in the following ways. First, our mapping technique relies on both content (the XML document itself) and the DTD of the document. Furthermore, our mapping technique preserves the nested structure, and handles query effectively by reducing the number of join operations. In addition, we propose two alternative techniques for addressing the matter of the restructuring aspect of the mapping (i.e., map of database result set into XML document) which has not been addressed by any previous works. Moreover, Yoshikawa et al. [17] decompose an XML document into a set of nodes and store these nodes in several tables along with encoded path information from the root to each node. However, this path does not serve as a primary key (in our case it is). One of the shortcoming of their path encoding is that it does not facilitate the construction of XML document on the fly from the result set of database SQL query.

3. XML PRIMERS

An element of XML is simply a type declaration or a set of elements. Document type definitions (DTDs) define the structure of an XML document (e.g., what elements, attributes etc. are permitted in the document). XML documents contain data, and must contain exactly one root element which will contain all the other elements. It might happen that an element contains a set of sub-elements in addition to the data. For example, a sample XML document might be:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE transactionSet SYSTEM "transactionsetrules.dtd">
<!-- Revision 1.5.0 data/OBOE, docs, August 29, 2000, -->
<transactionSet name="Request for Quotation" id="840"
 revision="004010"  functionalGroup="RQ" description="This
 Draft Standard for Trial Use contains the .."
 xmlTag="RequestForQuotation">
  <table section="header">
    <segment name="Transaction Set Header" id="ST"
     description="To indicate the start of a transaction set and to
     assign a control …" sequence="10" occurs="1"
     required='M' xmlTag="TransactionSetHeader">
      <dataElement name="Transaction Set Identifier Code"
       id="143" sequence="1" description="Code uniquely
       identifying a Transaction Set" type="ID" required="M"
```

Figure 1. A Portion of an XML Document

```
<!ELEMENT transactionSet (table+)>
<!ATTLIST transactionSet name CDATA #REQUIRED>
…
<!ELEMENT table (segment*)>
<!ATTLIST table section (header | detail | summary) "header">
<!ELEMENTsegment (segment*,(compositeDE?, dataElement?)*)*>
<!ATTLIST segment name CDATA #REQUIRED>
<!ATTLIST segment id CDATA #REQUIRED>
…
<!ELEMENT compositeDE (dataElement)+>
<!ATTLIST compositeDE name CDATA #REQUIRED>
<!ATTLIST compositeDE id CDATA #REQUIRED>
…
<!ELEMENT dataElement EMPTY>
<!ATTLIST dataElement name CDATA #REQUIRED>
<!ATTLIST dataElement id CDATA #REQUIRED>
```

Figure 2. A Portion of DTD for an XML Document in Figure 1

For query purposes XPath can be used. XPath is a language for finding information in an XML document. Using XPath, we can specify the locations of document structures or data in an XML document, and then process this information using XSLT.

4. PROBLEM STATEMENT

XML is an emerging technology which is hierarchical in nature. It has been proposed for the purpose of data exchange in the Web. Its nested, self-describing nature provides simple, flexible means for applications to exchange data. However, it is not designed to facilitate efficient retrieval of data or data storage.

Database management systems facilitate the persistent storage of data. The relational database management system is one of the successful DBMS which have dominated applications for more than 30 years because of its reliability, scalability, tools and performance. Therefore, XML documents are stored into a RDBMS. In this case, in order to handle XML queries, we need to come up with a mechanism for translating from XPath to SQL statements. Furthermore, we would like to render the RDBMS transparent to the user. In other words, the RDBMS will be hidden from the user. For this, we need to address the following two problems:

SampleTable

PathId	DataItem	ParentId
.transactionSet		
.transactionSet.table0		.transactionSet
.transactionSet.table0.segment0		.transactionSet.table0
.transactionSet.table0.segment0.dataElement0		.transactionSet.table0.segment0

Table 1. A Portion of SampleTable

AttributeTable

PathId	AttributeName	AttributeValue
.transactionSet	Name	Request for Quotation
.transactionSet	Id	840
…	…	…
.transactionSet.table.segment0	Description	To indicate the start of a transaction set and to assign a control number
…	…	…
.transactionSet.table.segment1.dataElement8 sequence9	Name	Security Level Code
.transactionSet.table0.segment1.dataElement8	Id	786
.transactionSet.table0.segment1.dataElement8	Sequence	9
…	…	…

Table 2. A Portion of AttributeTable

- We need an automatic mapping technique that effectively converts XML documents into relations of the RDBMS.
- We need techniques to translate XPath queries into SQL statements and which will then finally convert the result set of these SQL statements to an XML document.

5. OUR APPROACHES

In this section, we first address the initial problem, then present a solution to the second.

5.1 X-R Conversion

Our approach to converting an XML document to the relations of the RDBMS (X-R) is as follows:

First, we create a relation named, SampleTable in the RDBMS with the columns *PathId, DataItem, and ParentId*. Note that PathId is the primary key. Each element along the data item of an XML document will be mapped as a tuple in the table. For the time being, we will ignore the element's attribute, as well as its value and Ids (see Section 5.1.2 for further details). However, each element's tag will participate in the generation of PathId (see Section 5.1.1). Furthermore, each element's data will be stored as a value under the column, DataItem. ParentId is used to keep the element's parent information (see Section 5.2.2).

An element may be structured rather than atomic. In other words, an element may contain a set of sub-elements. In this case, besides a tuple for this element, a tuple or row will be created for each of the sub-elements. For example, in an XML document at line 4 (see Figure 1) an element “transactionSet” will be mapped as a tuple in the SampleTable along with the content. Furthermore, at line 9, sub-element “table” of this “transactionSet” will be mapped as a tuple in the SampleTable relation.

It is important to note that even if an element only contains a set of sub-elements (i.e., no DataItem associated with the element tag itself) we will still create a tuple for this element along with a null value in the DataItem. The rational behind this is to keep the element in the relation so that this will prove helpful during the extraction of nested structure information on the fly. For example, the sub-element “table” does not have any content, but it is nevertheless associated with a set of attributes and their values. Thus, DataItem for this row in the relation is empty (see second row in Table 1).

5.1.1 PATHID Generation

PathId of an element, the primary key in the table, is used to capture the nested position of the element in the XML document. Later this encoded knowledge concerning the nested position will be helpful for the formation of an XML document from the table. For the formation of PathId for an element, first the parent element, and then the grand parent of this element are identified. Finally, the process is stopped all the way back to the root element. Tags of these elements will be concatenated separated by dot (.), where the first tag is the root element's tag, and the last tag is the element's tag itself. We can formally define PathId of an element as the concatenation of the PathId of the parent element and its element tag. For example, at line 4 in Figure 1 we see that the element transactionSet maps into a tuple in the SampleTable. Its PathId will be transactionSet. This is because the element is the root element.

One important observation is that this PathId generation mechanism depends entirely on an element's tag, but not the content or data values. However, it might happen that some elements occur multiple times. If we follow the above strategy for PathId generation, we will end up with the same PathId for more than one element, which violates the definition of primary key.

To solve this problem, we need to address two questions. First, we need to identify which elements may occur more than once in a given XML document. Second, for these elements we need to generate a unique PathId. With regard to the first question, we will rely on the DTD. From the DTD we can identify which elements may occur more than once based on *, +. In this way we can identify which elements require special treatment in the generation of PathId. With regard to the second question, the PathId of each of these elements cannot be generated merely through the addition of the element tag itself at the end of the sequence. Instead, some sequence numbers (started from 0) will be padded with the tag itself so that PathId for each element will be unique. For example, in an XML document a sub-element “table” occurs multiple times (conforms by DTD) under element “transactionSet.” Hence, element “table” along with sequence number (started with 0) will be added to its immediate parent's PathId. Note that the parent's PathId in this case is “transactionSet.” Therefore, the first and second sub-element table's PathId will be “transactionSet.table0”, and “transactionSet.table1” respectively.

5.1.2 Attribute

An element in an XML can have any number of attributes. An attribute consists of an attribute name and its value. In our mapping technique we use a separate relation named *AttributeTable*. Since some elements may not have an attribute, this separate relation is more effective than would be the case with a single relation as per as storage concerned. The self-explanatory column names of the AttributeTable are as follows: *PathId, AttributeName, AttributeValue*. All three columns together will form a primary key. In addition, PathId is a foreign key associated with PathId in the SampleTable. Each attribute name, along with its value for an element, will be stored as a tuple in the separate relation, AttributeTable used for mapping. If an element has 3 attributes, 3 tuples will be created in the relation, AttributeTable. For example, in XML document element “transactioSet” has a set of attributes, and their values. The AttributeName of these are “Name,” “Id,” “Description” which have values “Request For…,” “890,” “To indicate…” respectively (see Table 2). Note that ID, IDREF of XML document will also be handled similarly as Attribute.

5.2 Query Mechanism

With regard to the second problem, conversion of relational data to XML (R-X), we will first present a technique for translating XPath query to query path Id (similar to PathId), the latter denominated *QPathId.* Next, we can construct an SQL statement using this QPathId. Finally, we will present alternative techniques for R-X conversion.

In this paper we will address location path, that is, basic XPath query. The basic XPath syntax is similar to file system addressing. If the path starts with the slash / it represents an absolute path to the required element. Furthermore, a location path consists of a sequence of one or more location steps separated by /. For example, location path /A/B/C consists of 3 location steps, A, B and C.

5.2.1 Translation of xpath to an sql statement

For translating XPath to an SQL statement we first need to construct QPathId from XPath and then use an SQL statement to retrieve rows that match this QPathId. Since each location step simply corresponds to an element each will simply participate in the formation of QPathId from left to right separated by dot. Thus, the XPath of /A/B/C will be mapped into the QPathId .A.B.C. Furthermore, an element with * or + in the DTD requires special treatment. Recall that for a case of multiple occurrence an element's tag is padded with some sequence number. With tag names of this type, for elements which occur more than once, a wild character will be added. For example, the XPath of /A/B/C will be mapped into the QPathId .A.B?.C where element, B might occur multiple times as defined in a DTD.

5.2.2 R-X (Restructuring)

Here, we would like to address the conversion of the relation (result set of SQL statement) to XML (R-X). For this, we would first like to construct the SQL statement(s) using QPathId, and then generate a tree/forest from the result set of SQL statement(s). Finally, we will construct the DOM tree from this tree/forest. At this point, it is very trivial to construct the XML document from this DOM tree. Therefore, we will be concerned with the construction of the tree/forest in the main memory. This is a challenging job because the tree/forest will be constructed on the fly from the result set of SQL statement(s). In addition, we would like to avoid a proliferation of SQL statements. In other words, our overall goal is to reduce query response time. There are several ways for constructing the tree in main memory from the database.

5.2.2.1 Naïve method (NV): The first method of tree construction to be considered is the *Naive Method* (NM). The notion of the NM is as follows: First, we issue the following SQL statement to identify the rows that match with QPathId (say, .A.B?C).

Select PathId, DataItem

From SampleTable

Where PathId **Like** '.A.B?.C'

After the execution of this SQL statement we will get a set of rows where each simply corresponds to an element and its value. Furthermore, some of these elements might contain sub-elements (nested elements). We need to retrieve these sub-elements to achieve R-X. Therefore, we can issue a SQL statement (similar to the above except that QPathId will be changed) for each element of the result set. This entire process will be repeated for the grand-sub elements and so on. The problem with this approach is the explosion of SQL statements. If the tree/forest contains n number nodes we are required to issue n SQL statements.

5.2.2.2 Effective Method (EM): An alternative approach is the *Effective Method* (EM). In the EM, in order to retrieve all the elements along with sub-elements, only one SQL statement is issued. Since the PathIds of all the sub-elements and grand sub-elements of this element are prefixed with the PathId of this element, we can simply use the "Like" operator to look for this pattern or prefix in the SampleTable. For example, we can consider the case in which the QPathId is "A.B?.C." Hence, we would like to find all elements' PathIds (including sub-elements) which start with "A.B?.C%." The percentage sign (%) tells us that we are interested only the pattern or prefix. The following SQL statement will be used:

Select PathId, ParentId, DataItem

From SampleTable

Where PathId **Like** '.A.B?.C%'

Order By PathId

The ParentId will be used to construct the tree effectively. Order by clause ensures that parent nodes will appear earlier as compared to children nodes in the result set. We have not covered how we can retrieve each element's associated attributes. Since attributes are stored in the relation AttributeTable, we need to join the two relations, SampleTable, and AttributeTable, in order to retrieve elements along with attributes. However, some elements may not have an attribute. Even so, we would like to retrieve this set of elements without attributes along with the set of elements with attributes. For this, we can do a left outer join between SampleTable and AttributeTable. The following left outer join query will be issued:

Select PathId, ParentId, DataItem, AttributeName,

AttributeValue

From SampleTable s **Left Outer Join** AttributeTable a

ON s.PathId = a.PathId

Where s.PathId **Like** '.A.B?.C%'

Order By s.PathId

After retrieving all the elements and sub-elements, we need to construct the forest in the main memory. For this, we will first partition the result set into a set of groups based on PathId. Each

selected element, along its associated attributes, will form a group. Next, we will form a new set of groups out of this set of groups based on an element's nested structure or hierarchy. Each member of this new set of groups will correspond to a tree where the root corresponds to an element and its children correspond to its sub-elements. Furthermore, the PathId for all nodes in the tree will be prefixed with the PathId of the root node. A collection of these trees will then form a forest.

Before going on to discuss the algorithm to be employed, we would like to formally define some terms.

Definition 1: A node-set (S) is a set of nodes returned by a SQL statement. Each element of S has attributes, such as PathId, ParentId, and DataItem.

Definition 2: Parent (x) is the parent node of a node, x.

Definition 3: A root node, r, of a tree is a node of S whose parent node does not appear in S (i.e., r | Parent (r) $\notin$ S.PathId).

Definition 4: A root-set, R, is a collection of root node (r) such that R = {r | r $\in$ S Λ Parent (r) $\notin$ S.PathId}. Each element of R has a PathId attribute.

It is clear from the definitions that parent node of each node of R will not appear as a node of S. Therefore, if we simply subtract all nodes' (of S) ParentIds from all nodes' (of S) PathIds, we will get a set of root nodes. Therefore,

R = { x| x $\in$ S Λ Parent(x) $\in$ (S.ParentId - S.PathId) }

To construct the tree in main memory from the selected nodes of the result set, the following pseudo-code is used:

For each node r of R

Construct a node with r.PathId

For each node s of S

If (s.ParentId=r.PathId) //whether a children node

Construct a node with s.PathId and set child link from r to s

Explore (s)

Explore (s)

For each node l of S

If (l.ParentId=s.PathId) //whether a children node

Construct a node with l.PathId and set child link from s to l

Explore (l)

First we identify all root nodes. Then for each root node, r, we can create a tree in the main memory. For this, we will explore all the immediate children nodes within the root node. Then, for each immediate children node, we will explore its children nodes (grand children) until we reach leaf nodes. The above technique is implemented by recursively calling the routine "Explore." First, the left most child of each node is explored. Then its immediate left most child (grand child) is explored and so on.

6. IMPLEMENTATION

As the RDBMS, the Oracle database management system is used in a client server setting. Note that tuple shipment (i.e., communication cost) affects the response time of queries. To speed retrieval, the primary index is built on the PathId attribute of SampleTable. For mapping an XML document into the database we need to traverse the XML document. For this we rely on an XML document object model (DOM), the use of which facilitates the construction of a tree structure in the main memory. This tree structure will contain the document's elements, attributes, and text etc. A DOM-based parser exposes the data (i.e., makes available), along with a programming library-called the DOM Application Programming Interface (API), which will allow data in an XML document to be accessed and manipulated. This API is available for many different programming languages, including Java, which is used here.

6.1 Results

In order to test the process we have designed we used a couple of XML documents. Due to limitations of space, results will be reported for only one of these documents. A portion of this document is shown in Figure 1. 1709 tuples were inserted into the SampleTable by employing the X-R technique. In addition, 14,819 tuples were inserted into the Attribute table. This indicates that on average each row of the SampleTable is associated with 7.6 rows of the AttributeTable. In other words, on average each element has 7.6 attributes (see Table 2 for further details).

For XPath query we used 8 sample queries such as:

/transactionSet/table/segment/segment

/transactionSet/table/segment/segment/dataElement

/transactionSet/table/segment/dataElement

/transactionSet/table/segment/compositeDE

/transactionSet/table/segment/compositeDE/dataElement

/transactionSet/table/segment/segment/segment/segment/compositeDE/dataElement

/transactionSet/table/segment/compositeDE/dataElement/@name= 'Report Action Code'

/transactionSet/table/segment/segment/dataElement/@name='Time Code'

The first three queries are related to broad formulation and the next three are related to narrow formulation and the last two related to attribute queries. It is important to note that attribute queries are simply narrow queries where a certain element's attribute contains value. Furthermore, note that in the narrow query case more specific elements are addressed and fewer sub-elements selected. Each of these queries was tested for NM, EM, Ascii File (AF) methods. With AF, XML document is stored as an ASCII file. Then it is loaded entirely in main memory, DOM tree is constructed and XPath queries are performed in the tree. With NM, we assume the secondary index is available on ParentId attribute of SampleTable. Thus, we conducted a number of experiments in order to demonstrate the superiority of EM over NM and AF. In these experiments, we reported the response time (the *Y*-axis of the reported graphs) of eight queries for each of these methods.

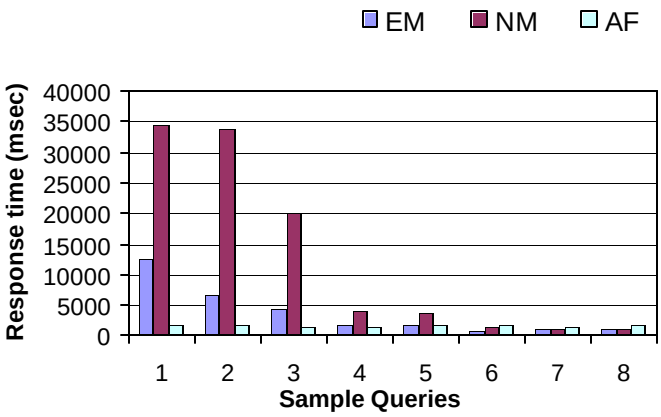

Figure 3. Response Time of EM, NM, and AF

In Figures 3, the X axis represents sample queries. The first three queries are related to broad query formulation, the next three queries are related to narrow query formulation, and the last two queries to attribute query formulation. Thus, results are reported for only above 8 queries. In Figure 3 for each query the first, second, and third bars represent EM, NM, and AF respectively. Furthermore, the data demonstrates that the response time of EM outperforms NM. Note that this pattern is pronounced related to broader query cases. For example, in query 1, 12.71 sec versus 34.35 sec response time is achieved for EM as opposed to NM where query 4, 1.9 sec versus 4.2 sec response time is obtained. This is because in the case of a broader query, it is necessary to retrieve more sub-elements from a selected element. Recall that a broad query many contain many sub-elements as nested form. In NM case more SQL statements will be issued, causing a high response time. On the other hand, in EM case only a single SQL statement will be issued. Furthermore, in the case of narrow query formulation, fewer sub-elements need to be retrieved from a selected element. Therefore, a smaller number of SQL statements will be issued in NM case, and the gap between EM and NM will be less pronounced.

One important point is that response time of AF is independent of types of queries. In other words, it does not fluctuate too much with types of queries (1.6~1.8 sec range). This is because the dominating cost is here fetching of XML document from server site to client site's main memory as a DOM tree. Furthermore, for broad queries AF outperforms EM and NM. For example, in query 1 1.8 sec versus 12.7 sec is achieved for AF as opposed to EM. This is because in the former case the document is simply fetched through http protocol from the server site. On the other hand the last two observe several common implementation overheads: database SQL operations are performed at server site, tuples are selected, assembled and sent to the client site over the network as a block and finally, tuples are disassembled at the client site. In our case, JDBC has been employed to connect to database server in order to handle all implementation overheads. It is apparent that in the last two cases, communication cost will be less as compared to communication cost of the former. However, the implementation overheads may simply offset this communication cost saving. For example, in query 6, 1.6 sec versus 0.8 sec is achieved for AF as opposed to EM. This is because here communication cost savings plays a dominating role over implementation overheads.

It is important to note that we cannot generalize the broader query result. It entirely depends on the data set.

7. CONCLUSIONS AND FUTURE WORK

We have proposed an automatic mapping technique from an XML document to relations within an RDBMS. We have demonstrated that our novel approach preserves the nested structure of the XML documents. We have also devised a seamless, transparent framework for user access to XML data by hiding database details. For this, we have proposed a novel mechanism for the translation of an XPath query to an SQL statement. Furthermore, we propose efficient techniques for the construction of an XML document on the fly from the result set.

We would like to extend this work in the following directions. First, we would like to support XML-QL and evaluate the performance of various approaches. Next, we would like to consider other databases (e.g., object-oriented and object-relational) and measure the performance of the relational model over various alternatives.

8. REFERENCES

[1] S. Abiteboul, D. Quass, J. MeHugh, J.Widom, J.Wiener. *The Lorel Query Language for Semi-structured Data*, International Journal on Digital Libraries, 1(1), pp. 68-88, April 1997

[2] F. Bancihon, G. Barbedette, V. Benzaken, C. Delobel, S. Gamerman, C. Lecluse, P. Pfeffer, P.Richard, F. Velez. *The design and implementation of O2, an object-oriented database system*,

Proc. of the Second International Workshop on Object-oriented Database, 1988, ed. K Dittrich

[3] T. Bray, J. Paoli, C.M. Sperberg-McQueen, *Extensible Markup Language (XML) 1.0*,

http://www.w3.org/TR/REC-xml

[4] P. Buneman, S. Davidson, M. Fernandez, and D. Suciu, Adding Structure to Unstructured Data, Proc. of The Internation Conference on Database Theory (ICDT), Delphi, Greece, 1997.

[5] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu, *XML-QL: a Query Language for XML*, Proc. of the Int. WWW Conf., 1999

[6] A. Deutsch, M. F. Fernandez, D. Suciu, *Storing Semi-structured Data with STORED*, SIGMOD Conference 1999: 431-442

[7] M. Fernandez, D. F. Florescu, J. Kang, A. Levy and D. Suciu, Catching the Boat with Strudel: Experiences with a Web-Site Management System, Proc. of ACM SIGMOD Conference on Management of Data, WA, 1998

[8] D. Florescu, D. Kossman, *A Performance Evaluation of Alternative Mapping Schemes for Storing XML Data in a Relational Database*, Rapport de Recherche No. 3680 INRIA, Rocquencourt, France, May 1999

[9] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, J. Widom, *Lore: A Database Management System for Semi-structured Data*, SIGMOD Record 26(3): 54-66 (1997)

[10]S. Netorov, S. Abiteboul, and R. Motwani, Extracting Schema for Semistructured Data, Proc. of ACM SIGMOD Conference on Management of Data, Seattle, WA, 1998

[11] K. Shoens, A. Luniewski, P. Schwarz, J. Stamos, and J. Thomas, *The Rufus system: Information organization for semi-structured data*, Proc. of the Int. Conf. On VLDB, pages 97-107, Dublin, Ireland, 1993

[12] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, J. F. Naughton, *Relational Databases for Querying XML Documents: Limitations and Opportunities*, VLDB 1999: 302—214

[13] F. Tian, D. DeWitt, J. Chen, and C. Zhang, The Design and Performance Evaluation of Various XML Storage Strategies. *Submitted for publication ,Computer Science, University of Wisconsin, Madison*

[14] R.V. Zwol, P. Apers, and A. Wilschut, Modeling and Querying Semi-structured Data With MOA, Workshop on Query Processing for Semi-structured Data and Non-standard Data Formats, 1999.

[15] http://www.lotusnotes.com/, 1998.

[17] 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, Vol. 1, No. 1, June 2001.

[18] A. Schmidt, M. Kersten, M. Windhouwer, F. Waas, Efficient Relational Storage and Retrieval of XML Documents, Proceedings of WEBDB 2000.

[19]M. F. Fernandez and J Simeon and P. Wadler, An Algebra for {XML} Query",Foundations of Software Technology and Theoretical Computer Science", pp.11-45, 2000.

[20]M. J. Carey and D. Florescu and Z. G. Ives and Y. Lu and J. Shanmugasundaram, E. J. Shekita and S. N. Subramanian",XPERANTO: Publishing Object-Relational Data as (XML)",WebDB (2000), pp. 105-110.