
World Wide Web, 4: 101–119 (2001)

XStorM: A Scalable Storage Mapping Scheme for XML Data

WEN QIANG WANG, MONG LI LEE, BENG CHIN OOI and KIAN-LEE TAN
{wangwq,leeml,ooibc,tankl}@comp.nus.edu.sg
Department of Computer Science, National University of Singapore, 3 Science Drive 2, Singapore 117543

Abstract

With the increasing ubiquity of XML, an eXtensible Markup Language, the industry is racing to provide XML infrastructure for e-commerce, information interchange, effective query of diverse sources and yet more integration of diverse data. It is anticipated that large volumes of XML data will be created manually from HTML documents or generated using some WWW tools and electronic data interchange (EDI). In this paper, we examine how large amounts of XML data can be stored in a relational database. Our scheme considers the unique irregular features of XML, including missing elements or multiple occurrences of the same element, and elements which may have atomic values in some data items and structured values in others. A detailed experimental study demonstrates good query performance, effective space utilization and scalability.

Keywords: XML data management, relational database, scalable storage

1. Introduction

As XML expands beyond its document markup origins to become the basis for data interchanges on the Internet, the industry is racing to provide XML infrastructure for e-commerce, information interchange, effective query of diverse sources and yet more integration of diverse data. Once XML becomes pervasive, many information sources will structure their external view as a repository of XML data, regardless of their internal storage mechanisms. One highly anticipated application of XML is that XML will turn the Web into a database system, thereby making it possible to pose SQL-like queries and get better results than from today's Web search engines.

There are various possible ways to store and query XML data: ranging from a primitive file system, a relational database, an object-oriented database such as Excelon, to a special-purpose (or semi-structured) system such as Stanford's Lore [1,6]. The file system is the most straightforward option although there is no support for querying the XML data. An object-oriented database system has rich data modeling capabilities, which are useful for clustering XML elements and subelements. Unfortunately, the current generation of object-oriented database systems is not fully developed to process complex queries on large databases. Theoretically, special-purpose systems should work best as specially designed structures, indexes and query optimization techniques are customized to store and query XML data. However, such systems are not common and take time to mature.

The current trend is to leverage the robust and widespread technology by using a relational database system. Relational stores are great at providing multiple distinct logical views on the same data with very good scaling and transactional characteristics. Even then, there are many different ways to store XML data. Oracle 8*i* lets the user or system administrator decide how XML elements are stored in relational tables. [7] infers from the DTDs of the XML document how the XML elements should be mapped into tables. *STORED* [3] analyzes the XML data and expected query workload to obtain a set of schemas. Any data that cannot be accommodated in these schemas are stored in overflow graphs. This involves integration of the relational storage with a semistructured overflow, raising yet to be resolved system issues. Furthermore, if the data instance has a very irregular structure, then the schema extracted may not cover a large percentage of the data. A lot of overflow graphs will be generated leading to performance degradation.

Florescu and Kossman [5] takes the graph representation of an XML document and studies various schemes to map the edges and nodes into relational tables. One simple approach is to store all the edges in the XML graph into a single *edge* table containing information such as the *sourceNode, targetNode, name* etc. Separate value tables are created for each data type. This edge approach performs poorly for heavy queries because joins with the large edge table become very expensive. In any case, most of the data in the edge table is irrelevant for a specific query. Another approach is to generate a single *universal* table to sore all the edges. Such a table not only have many empty fields (i.e., set to NULL) given the irregularity of XML data, but it may also contain a lot of undesirable redundancy. The universal approach does not perform well for heavy queries for the same reasons as the edge approach. The *binary* approach which creates a relational table for each attribute (XML tag) and stores the value accordingly gives the best query performance. This scheme is similar to the binary storage scheme proposed to store semistructured data in [9] and is essentially a horizontal partitioning of the edge table. There are as many binary tables created as there are different subelement and attribute names in an XML document. The values of the attributes can be stored together (inlined) in the same table. Unfortunately, the number of join operations needed to answer a query is proportional to the number of attributes involved. This becomes very expensive when reconstructing large XML documents. In addition, the original XML document cannot be exactly reconstructed as the structure information has been lost.

In this paper, we address the above-mentioned drawbacks of mapping XML data to relational tables. We propose a mapping scheme, XStorM, to store XML data in relational databases. The motivations behind our approach are:

1. XML elements that represent entities in the real world (*objects*) are differentiated from XML elements that represent properties of entities (*attributes*). Excessive fragmentation of XML data is avoided when each object is mapped to a *core* relational table together with the majority of its attributes. To achieve this, a data-mining algorithm is used to find frequent patterns in the XML dataset.
2. Irregularities or data instances that deviate from the core schemas are stored in separate relational tables, or *overflow* tables.
3. Structural information of the XML document is embedded in the overflow table names for fast reconstruction of the original XML document.

4. Data integrity is guaranteed as the entire XML data instances are stored in the relational database.

Our experiments demonstrate that XStorM gives good query performance, uses minimal space requirements, enables reconstruction of original XML documents, and is scalable.

The rest of the paper is organized as follows. Section 2 illustrates the various ways XML data can be stored and introduces our mapping scheme XStorM. Section 3 explains the algorithms involved in mapping XML data to the relational tables. The performance results are shown in Section 4 and we conclude in Section 5.

2. The big picture

In this section, we informally discuss the intuition behind XStorM. We shall also illustrate with an example XML document, which will be as the running example in this paper, the proposed scheme.

The eXtensible Markup Language, XML, is initiated by the World Wide Web Consortium (W3C) as a simplified subset of SGML specially designed for Web applications. The key features in XML are that information providers can define new tags and attribute names at will, document structures can be nested to any level of complexity, and Document Type Definition (DTD) can be used to constrain the structure and data values of a class of XML documents. XML models data as a tree of elements with attributes composed of name–value pairs.

XML is fundamentally different from relational and object-oriented data. The key distinction between data in XML and data in traditional models is that XML is not rigidly structured. In the relational and object-oriented models, every data instance has a schema, which is separate from and independent of the data. In XML, the schema exists with the data. Thus, XML data is self-describing and can naturally model irregularities that cannot be modeled by relational or object-oriented data. For example, data items may have missing elements or multiple occurrences of the same element; elements may have atomic values in some data items and structured values in others; and collections of elements can have heterogeneous structure. Figure 1 shows an XML representation of articles in SIGMOD Record. An *article* element consists of *issueNumber, title, startPage, endPage, authors* and *description*. The *authors* element consists of a set of *author*.

We can use the Document Object Model (DOM) [4] to give a graphical representation of an XML document. Figure 2 shows a partial DOM representation of the XML document in Figure 1. DOM has two types of nodes: *element* nodes and *text* nodes. All nodes that have children are element nodes. All leaf nodes are text nodes. An element node is labeled using the object name or attribute name, which are tags in the text representation of an XML document. A text node is labeled with the value of an attribute. To simplify our discussion, we assume that DOM is ordered from left to right and generate a unique identifier for each element node in DOM.

From the existing schemes, we observe that it is crucial to strike a balance between clustering attributes in a table and putting them in separate tables. We observe that there are two types of XML elements: one that denotes *objects* or entities in the real world and

```
<article>
        <issueNo> 16 </issueNo>
        <title> Interoperable Database Systems </title>
        <startPage> 365 </startPage>
        <endPage> 376 </endPage>
        <authors>
                <author> Gillian Ram </author>
                <author> James Braun </author>
        </authors>
        <abstract> Proliferation of database systems ... </abstract>
 </article>

<article>
        <issueNo> 18 </issueNo>
        <title> Parallel Query Processing</title>
        <startPage> 123 </startPage>
        <endPage> 133 </endPage>
        <authors>
                <author> Jacob Linz </author>
                <author> Paul Tan </author>
                <author> Kelvin Goh </author>
        </authors>
        <abstract> Large amounts of data ... </abstract>
</article>
```

Figure 1. Example of an XML document.

one that denotes *attributes* or the properties of entities. Note that in XML, an attribute can also be defined within the start tag of an element, for example, in the following element

```
<book btype = "textbook> myBook </book>
```

where *btype* is an attribute of the *book* element. Such attributes bear textual information instead of structural information and we have to mark these attributes when we store them in a relational database so that we can reconstruct the XML data instance correctly. In addition, we can differentiate between collections of attributes (as in *authors*) versus collections of objects. Figure 3 shows how *authors* becomes a collection of objects when the XML document records more information about an author such as *institution, country*, etc. in addition to *name*. Note that *authors* can also be a *mixed* collection of attributes (author just has name) and objects (author has name, institution and country or just name and institution) as shown in Figure 4.

It is important that different types of objects should be stored in separate tables since they are likely to be referenced elsewhere. For example, if *authors* is a complex object consisting of *name, institution, country*, then it should be stored in a separate table since a

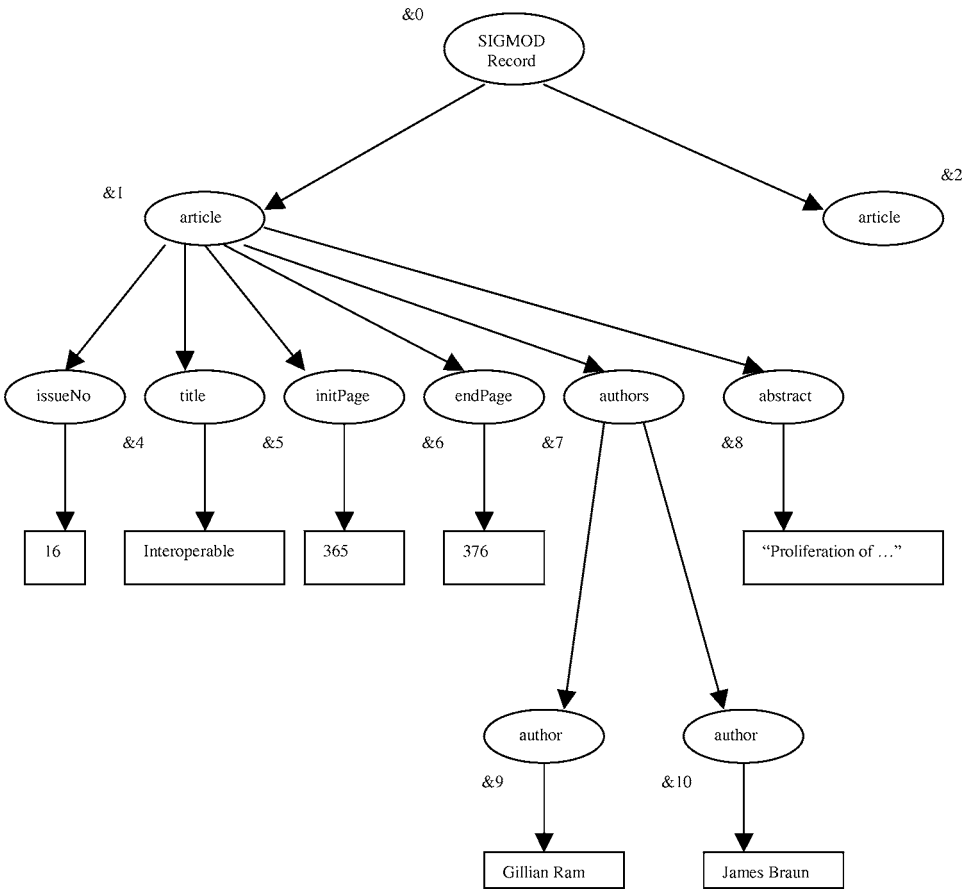

Figure 2. DOM representation of the XML document in Figure 1.

person is likely to write more than one article and repeating his information for each article leads to redundancy and updating anomalies. However, we have a number of options when *authors* is just a collection of author names. For example, if the articles almost always have two authors, then we can consider storing all the attributes of *article*, including $author_1$ and $author_2$ in the same table. In practice, however, this may not be possible as the XML data is often irregular. An article A may have 5 authors while another article B may have 2. If we map the article element together with all its attributes (including fields $author_1$, $author_2$, $author_3$, $author_4$, $author_5$) to a relational table, then all the author fields for article B will be NULL except for $author_1$ and $author_2$. This situation is worsened if the majority of the articles have two authors. Adding more authors to an article becomes problematic without expanding the table.

One solution is to store collections of attributes in a separate table. Hence, we can have a *core* relational table for an object containing all the single-valued attributes and separate

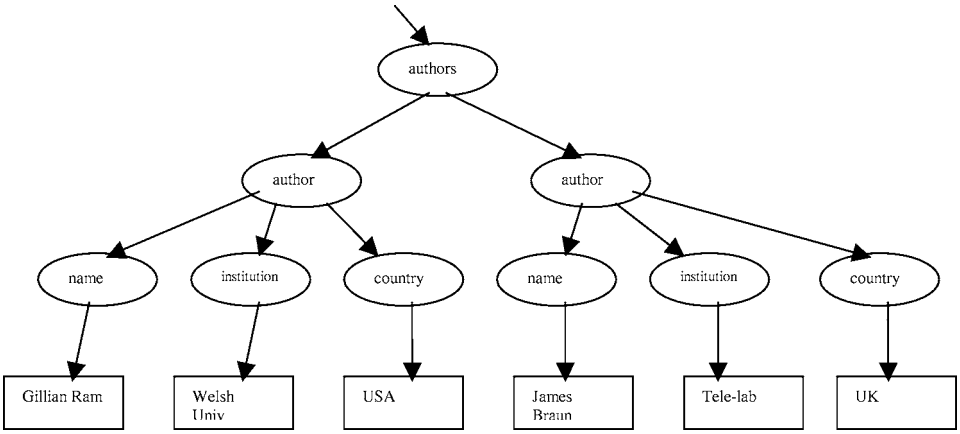

Figure 3. Authors as a collection of objects.

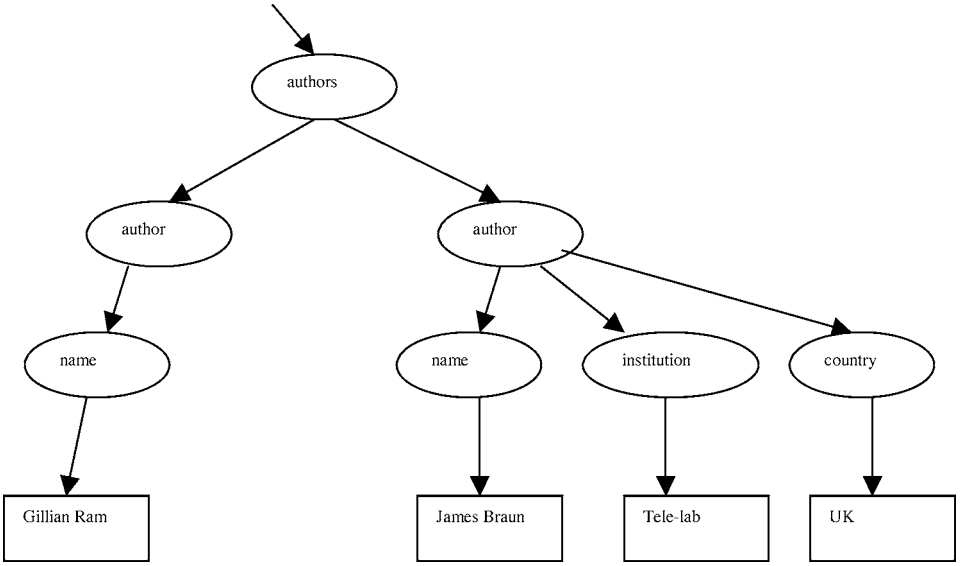

Figure 4. Authors as a mixed collection of attributes and objects.

overflow tables for collections of attributes. This scheme resembles how multivalued attributes are handed in relational data model. However, this may not work too well if an XML element has many different small collections of attributes, leading to many overflow tables and subsequent joins for heavy queries.

On the other hand, suppose we know that the majority of the articles have two authors, with a few outliers that have more than two authors or even one author. In this case, we can incorporate $author_1$ and $author_2$ into the core relational table for *article*. Additional

Table 1. Sample relation tables from XML data instance

Table Name: Article_Core

articleID	issueNo	title	initPage	endPage	authors.author$_1$	authors.author$_2$	abstract
1	16	Interoperabler Database	365	376	Gillian Ram	James Braun	...
2	18	Parallel Query Processi	123	133	Jacob Linz	Paul Tan	...

Table Name: Article.authors.author

articleID	index	author
2	1	Kelvin Tan

authors will be stored in an overflow author table. If an article has only one author, then we simply set the *author*$_2$ field to NULL. In the next section, we will explain how we can employ a data mining algorithm for semistructured data [8] to find frequent *tree patterns* in XML graph.

Table 1 shows the relational tables obtained assuming that the majority of the articles have two authors. Note that we have embedded the structural information of the XML document in the attributes authors.*author*$_1$ and authors.*author*$_2$ and the overflow table name article.*authors.author*. The subscripts are used to ensure uniqueness of names. This embedded structural information facilitates fast reconstruction of the original XML document as we will demonstrate in our experiments.

3. The XStorM mapping scheme

In this section, we describe a mapping scheme for storing XML data into relational database. The steps involved are:

1. Identification of XML objects.
2. Identification of frequent tree patterns in XML graph.
3. Generate core relational tables.
4. Generate overflow tables.

We will elaborate on each of these steps in the following subsections.

3.1. Object identification

As noted earlier, element nodes in DOM can be differentiated into object nodes and attribute nodes. The goal of this step is to find all the XML objects in an XML data instance. For example, if we have an XML data instance containing 1000 articles, then we need to identify all nodes that represent the object *article*. However, to automatically identify nodes that represent a specific object is a daunting task—if we know nothing about the XML instance, then it is difficult, if not impossible, to know whether a node represents an object or not.

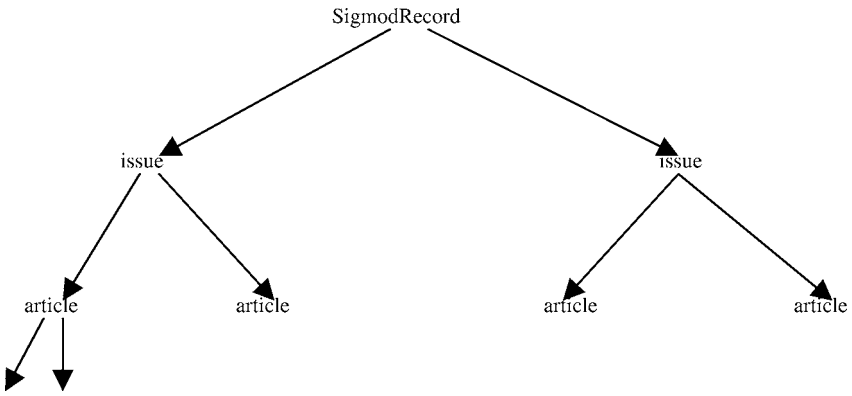

Figure 5. Partial data tree of an XML document.

In this paper, we adopt a three-step approach. First, we determine the number of paths corresponding to a *prefix*.[1] We shall refer to this as the *support* of the prefix. Next, we identify the minimal prefix which is the shortest prefix whose support is greater than or equal to some certain predetermined threshold. Finally, the node at the lowest level of the minimal prefix is the target object that we are looking for. This scheme can be implemented efficiently using the well-known breadth first search (BFS) algorithm. We shall illustrate this object identification process.

Figure 5 shows the partial data tree of a sample XML document. The object we would like to identify in this example is "article". We first carry out a BFS on the data tree. During the search process, we will discover the prefixes. If the support of a prefix exceeds a certain threshold or minimal support, then we record it. Nodes that support this prefix will not be expanded, that is, their children will not be pushed into the queues in the BFS algorithm. Suppose we set the minimal support to be 3, which is a heuristic value. The prefix we will discover is: *SigmodRecord* $\rightarrow$ $issue$. However, the support of this prefix is only 3 (i.e., only 2 "issue" tags), which is less than the minimal support. The next prefix found is *SigmodRecord* $\rightarrow$ *issue* $\rightarrow$ *article*. The support of this prefix is 4, which is greater than the minimal support and then we record this prefix. As mentioned previously, the children of article node will not be pushed into the queue. The search process stops. With this prefix, it is straightforward to identify the four article objects in the sample. Note that choosing an appropriate minimal support value is crucial. In the above example, if the minimum support value we choose is 2, then the object identified will be issue, not article.

3.2. Frequent tree patterns

Given the "schema-less" semi-structured XML data, it is impossible to find a general schema that covers the whole XML data instance. Deutsch et al. [3] showed that generating a storage schema for semistructured data that minimizes cost is NP-hard in the size of the input data. Dynamic programming algorithms are not feasible. Instead, we use a data-mining algorithm to identify frequent tree patterns in an XML graph. This enables

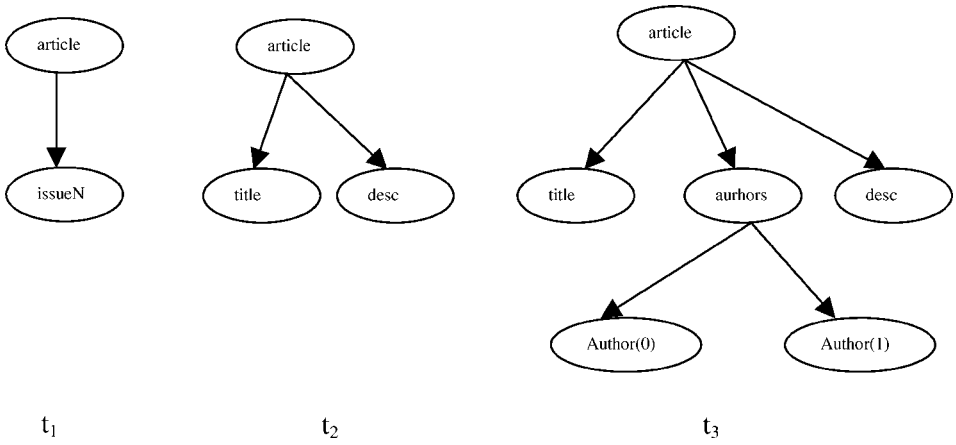

Figure 6. Examples of tree expressions.

us to generate a schema that covers a major portion of the data. Our aim is to incorporate as much small attribute collections into an object's core relational table as possible in order to minimize the overflow tables. Query performance is improved when excessive fragmentation is avoided because joins are reduced.

We adapt the data-mining algorithm for semistructured data described in [8] for our purpose here. We will first review some of the concepts used.

A *tree-expression* is a tree-like structure for representing patterns in the DOM graph. A *k-tree-expression* is a tree-expression containing k leaf nodes. Examples of tree-expressions are shown in Figure 6: t_1 is a 1-tree-expression, t_2 is a 2-tree-expression and t_3 is a 4-tree-expression. A node N in the DOM graph *supports* a tree-expression *TE* if and only if we can find *TE* in the sub-tree rooted at N. For example, the article node in Figure 2 supports all the tree-expressions t_1, t_2 and t_3 in Figure 6. The *support of a tree-expression TE* is defined as the number of nodes that support *TE*.

We observe that a k-tree expression, $k \geqslant 1$, can be constructed by "gluing" a sequence of k 1-tree expressions that are not prefixes of each other. A 1-tree expression is actually a simple path from a root node to a leaf node. For example, suppose we have three 1-tree expressions p_1, p_2, p_3 as shown in Figure 7. Then, a 3-tree expression p_4 can be constructed from p_1, p_2, p_3 and the sequence is $\langle p_1, p_2, p_3 \rangle$. We say that $p_4 = \langle p_1, p_2, p_3 \rangle$. Note that if the sequence of the 1-tree expression is different, then a different k-tree expression is constructed.

Let p_i denote a 1-tree-expression, for $i \geqslant 1$. Then a k-tree-expression $\langle p_1, p_2, \ldots, p_k \rangle$ is constructed from two $(k-1)$-tree-expressions $\langle p_1, p_2, \ldots, p_{k-2}, p_{k-1} \rangle$ and $\langle p_1, p_2, \ldots, p_{k-2}, p_k \rangle$. We call these two $(k-1)$-tree-expressions a *matching pair.*

The above k-tree expression property is very useful as it prunes our search. We do not need to consider a k-tree expression if it has some "subtree-expression" that is known to be infrequent.

In order to determine frequent k-tree-expressions, we use the depth first search algorithm to discover all the 1-tree-expressions starting from the object nodes found in step 1.

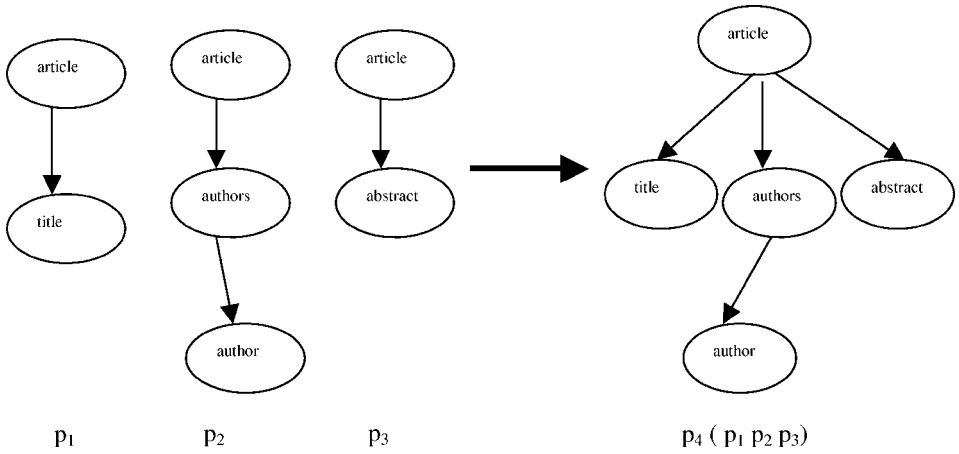

Figure 7. Example of how a k-tree-pattern can be constructed from k 1-tree-expressions.

The supports of these 1-tree-expressions are tracked. Frequent 1-tree-expressions are used to generate 2-tree-expressions; frequent 2-tree-expressions are used to generate 3-tree-expressions, and so on. Finally, the algorithm will generate frequent k-tree-expressions for a given k. A large k should be used to find a schema with maximal data coverage. Figure 8 gives the details of the algorithm.

3.3. Generate core and overflow tables

The frequent k-tree-expression obtained in step 2 creates a schema for the XML data. Relation tables can now be generated from it. Root nodes in the k-tree-expressions represent objects. Each object node n in the schema is mapped to a core relational table R. Leaf nodes in the tree rooted at n become attributes of R. In addition, R has an attribute that stores the object identifiers, for example, articleID in Table 1. The XML data can now be loaded into these relational tables. Nulls are used for any missing data.

Since XML is semi-structured, not all the data can fit into the core tables. In contrast to STORED which uses overflow graphs in external devices, we store the extraneous data in overflow tables in the relational database as shown in Table 1. The names of overflow tables are given by *ObjectName.collectionName*. The overflow table names embed the XML structural information necessary for pattern matching queries and reconstruction of the XML document. For example, the overflow table name for the article object is *article.authors*, indicating the path in the XML graph. Note that there is an additional attribute, *objectID*, in the overflow tables that contains the identifier of the object to which these overflow data belongs to.

```
Algorithm Frequent Tree Patterns (k)
/* find frequent 1-tree-expressions */

Ti, 1 ≥ i ≥ k : sets of i-tree-expressions, initially empty;
Fi, 1 ≥ i ≥ k : set of frequent i-tree-expressions, initially empty;

For each object node n do {
        Depth-First-Traverse (n) ;
        For each 1-tree-expression t found do {
                If (t IN T1)
                Then addSupport(t);     /* increment the support of t */
                Else T1 = T1 UNION {t}; /* add t to the list of 1-tree-expressions */
        }
}
For each t in T1 do
        If (getSupport(t) ≥ THRESHOLD)
                F1 = F1 UNION {t};
/* generate k-tree-expression */
For i =2 to k do {
/* generate i-tree-expression from matching (i-1)-tree-expression */
        Ti = combineMatchingPairs (Fi-1);
        For each t in Ti do
                If (getSupport(t) ≥ THRESHOLD)
                        Fi = Fi UNION {t};
}
End.
```

Note: "THRESHOLD" is a user-specified value and may vary on different XML data instances. In our experiments, if a tree-expression has a support of 60% of the total number of objects, we consider it as a frequent tree expression.

Figure 8. Algorithm to find frequent tree pattern.

4. Performance study

We carried out a series of experiments on a Pentium 233PC with 64 MB RAM to evaluate the performance of XStorM. Two metrics are used: the size of the relational database generated and the response time of different classes of queries. We compared our storage scheme with the binary approach in [5] and STORED [3]. All the mapping algorithms are implemented in Java and calls to the Oracle database are made using the JDBC. Figure 9 shows the system architecture.

4.1. Experimental setup

We created synthetic XML documents based on the ACM SIGMOD Record XML data for our experiments. The data sets have the following characteristics:

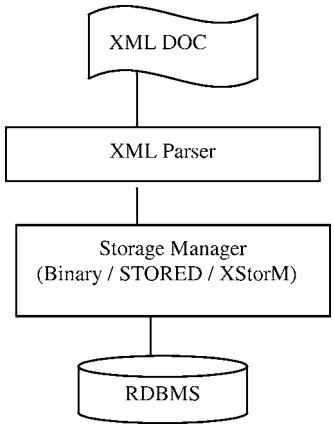

Figure 9. System architecture for experiments.

Table 2. Benchmark query templates

Query	Description	Feature
Q1	Retrieving information to reconstruct XML object	Select by object id
Q2	Find objects that have attribute a_1 with value in certain range	Select by value
Q3	Find objects that have attributes a_1 and a_2 with certain values	Two predicates
Q4	Find objects that have a_1 and a_2 with certain value or just a_1 with certain value	Optional predicate
Q5	Find objects that have a_1 or a_2 or a_3 with certain value	Predicate on attribute name
Q6	Find objects that match a certain pattern	Pattern matching

1. Five sets of XML documents with varying sizes are used: 1 MB, 10 MB, 20 MB, 40 MB, and 100 MB. They contain 1274, 11500, 22890, 44200, and 113754 articles, respectively.
2. Each article has the following information: issue number, title, starting page, ending page, a number of authors, and a short abstract. The number of authors varies from 1 to 17. We observe that although the majority of the articles have 2 or 3 authors.

In order to minimize space wastage, the attributes are divided into two classes: large and small attributes. For example, the name of an author is a small attribute while the abstract is a large attribute.

Table 2 describes the query template used in our experiments. These query templates test a variety of features, including simple selections by object identifiers (oid) and attribute values, optional predicates, predicates on attribute names and pattern matching. All the attribute values are stored as strings in the database. For predicates involving numeric comparisons, we convert the string value to a number using the *to_number* () function provided in SQL.

In order to obtain reproducible experimental results, we carry out all the benchmark queries as follows: Each query is run once to warm up the database buffers and then ten times subsequently to get the average running time of the query. Warming up the buffers

will have an impact on light queries that operate on data that fits in the main memory although heavy queries are not affected.

4.2. Storage requirements

We first investigate the amount of storage required by the various mapping schemes to store XML data. Table 3 shows the size of the XML document and the resulting relational databases. The binary approach produces a much larger relational database compared to STORED and XStorM. Indeed, the binary approach requires double of the actual data size for storage, while STORED and XStorM requires a storage space of about 70–80% of the original XML documents. The high storage space requirement by the binary approach is because it stores the structural information for each object, even when the objects have similar structure. On the contrary, both STORED and XStorM only store common structures extracted from the documents. While both STORED and XStorM use about the same amount of storage for the base data, the overflow data obtained in XStorM is very minimal compared to STORED. Furthermore, XStorM keeps both the base data and overflow data in the relational database instead of using auxiliary data structures as in STORED. Apart from consistency, keeping the overflow data in the database is amenable to faster reconstruction.

Table 3. Comparison of database sizes generated by different schemes

	XML MB	Binary MB	STORED MB	XStorM MB
		Data set 1		
Base data	1.1	2.25	0.84	0.84
Overflow data	–	–	0.1	0.01
Total	1.1	2.25	0.94	0.85
		Data set 2		
Base data	10.1	20.2	7.6	7.6
Overflow data	–	–	0.85	0.08
Total	10.1	20.2	8.45	7.68
		Data set 3		
Base data	20.2	40.3	15.2	15.2
Overflow data	–	–	1.7	0.16
Total	20.2	40.3	16.9	15.36
		Data set 4		
Base data	40	82.7	30.4	30.4
Overflow data	–	–	3.4	0.32
Total	40	82.7	33.8	30.72
		Data set 5		
Base data	100	202.2	77.4	77.4
Overflow data	–	–	7.4	0.8
Total	100	202.2	84.8	78.2

4.3. Response time of queries

Query running time is a very important measure when evaluating these mapping schemes. We will elaborate the results in the following subsections.

4.3.1. Retrieve information to reconstruct XML document. This experiment queries information needed to reconstruct a XML object from the relational database. This query involves selection by object id and for every table, there are indexes built on object id (e.g., articleID, source) column because it is the primary key or part of primary key. Figure 10 shows the results. We observe that in most cases, there is no significant difference among all three mapping schemes since index lookup is very fast on relational databases. For the 100 MB dataset, the binary scheme performs badly because it requires the retrieval of data from all the attribute tables compared to STORED or XStorM which retrieves data from just the core table and overflow graphs/tables.

4.3.2. Selection queries. In this experiment, we tested queries to retrieve objects that have attribute with values in a certain range on the various databases. An example of the query used is

Query: Select articles that have "initpage" between 500 and 600.

The results are shown in Figure 11. The running times for all three mapping schemes are almost the same. The reason is that the query only involves one attribute. Therefore, no join operation is needed to answer this query. For the 100 MB dataset, the binary scheme performs better since its single attribute table is smaller than the core table in the STORED scheme and XStorM scheme.

4.3.3. Join queries. This experiment examines the situation where the predicates are involved in a query. A typical query is shown below.

Query: Select articles with issueNumber 15 and whose 10th author is 'Pinar Koksal'.

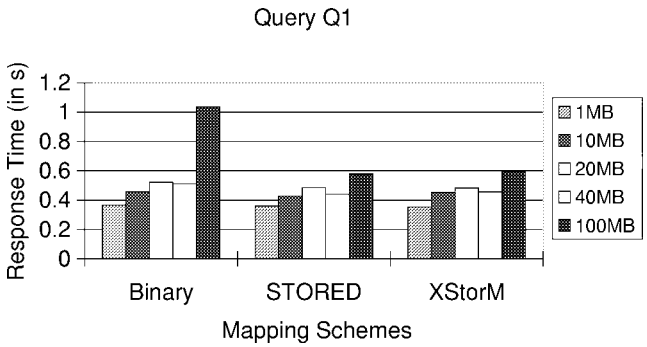

Figure 10. Results of reconstructing XML document experiment.

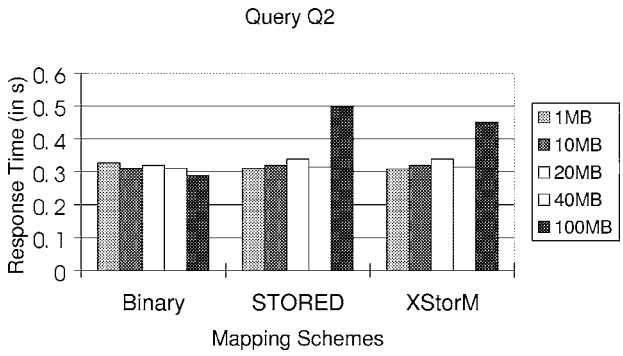

Figure 11. Results of selection query experiment.

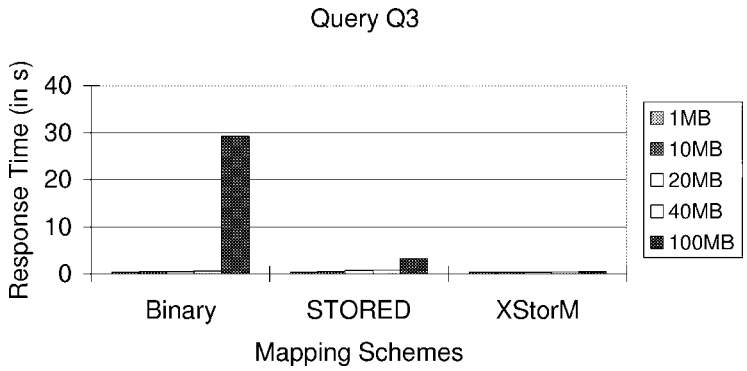

Figure 12. Results of join query experiment.

Note that one of the predicates involves overflow attribute for the STORED and XStorM schemes. Figure 12 shows the results of this experiment. We can see that the differences of running time becomes large enough to claim that XStorM performs the best among all three schemes when size of data sets increase. The reason is simple: joining a much smaller overflow table is much faster than joining two large attribute tables as required in the binary approach. The STORED scheme performs poorly in the 100 MB data set because there is too many overflow graphs and searching for values in them is a time consuming process.

4.3.4. Queries with optional predicates. We test the performance of queries containing optional predicates and predicates involving overflow data on the various storage mapping schemes. For instance,

> *Query: Select articles that have first author 'Dallan Quass' AND 7th author 'SvetlozarNestorov' OR just first author 'Kenneth A. Ross' (no 7th author).*

Figure 13 contains the results of the experiment. XStorM has the best overall performance. Furthermore, STORED performs poorly for the same reasons given in join query experiments.

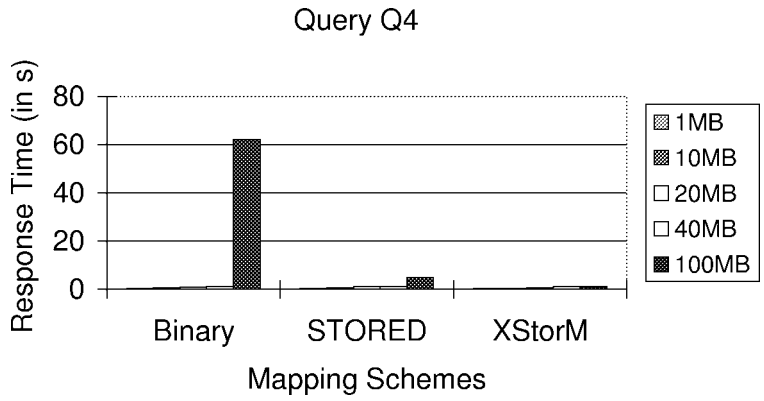

Figure 13. Results of optional predicate query experiments.

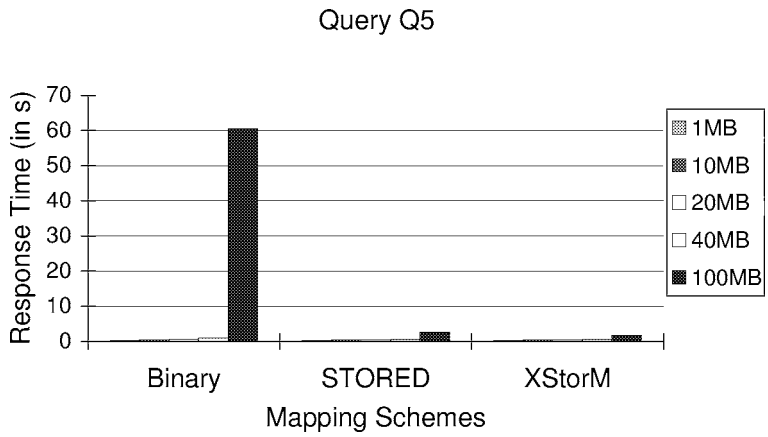

Figure 14. Results of query with attribute predicates experiment.

4.3.5. Queries with attribute predicates. This experiment evaluates the performance of queries involving predicates on attribute names. Figure 14 shows the results when the following query is issued.

> *Query: Select articles that have initpage = 388 or endpage = 2 or 7th author 'Svetlozar Nestorov'.*

The binary scheme performs poorly because we need to search matching tuples in three attribute tables and then take the union of the tuples returned. Although the STORED scheme only searches one table and XStorM searches two tables, XStorM performs better than STORED because accesses to the overflow graphs in STORED takes more time than retrievals from the overflow tables in XStorM.

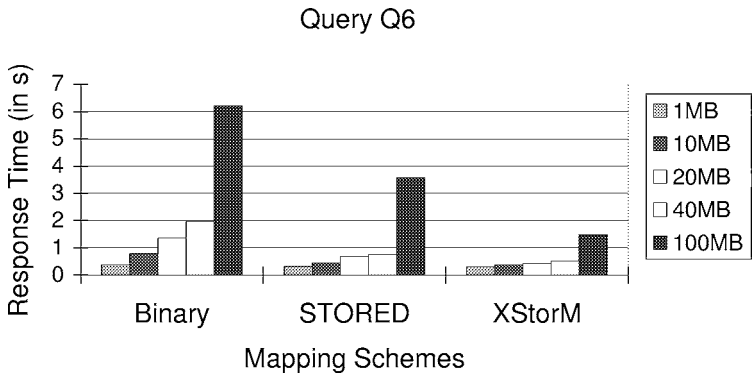

Figure 15. Results of pattern matching query experiment.

4.3.6. Pattern matching queries. Figure 15 shows the performance results of the following pattern matching query:

> *Query: Select articles that have attributes issuenumber, title, initpage, and 9 authors.*

XStorM performs significantly better than the binary scheme. For example, when the XML data is 40 MB, XStorM is about 4 times faster than the binary scheme. The binary scheme performs poorly in this query because it needs to join many attribute tables to find the matching tuples, especially when the pattern involves many attributes. The STORED scheme performs much better than the binary scheme because most of the attributes are contained in the core table.

4.3.7. Discussion. Theoretically, schemes that maps XML data into relational model based on schematic information should work better than just storing XML data in attribute tables. The most expensive operation in query processing is join operation and to answer most queries, we need information about several attributes of an object. In the binary scheme, if we want information from several attributes, we have to join the corresponding attribute tables to form the query result. If the attribute tables are large, then the join operation will be expensive. On the other hand, storing XML data according to schema not only saves disk space, but also reduces the number of join operations needed to answer a query.

For example, let us consider the query to find objects with attributes a_1 and a_2 with certain values. For binary scheme, to answer this query we need to join two attribute tables table a_1 and table a_2. On the other hand, STORED and our proposed scheme, XStorM, only search one table for tuples that satisfy the selection condition. While in most cases, selection operation is much faster than the join operation, there are situations that involve overflow data. Suppose attribute a_2 is not included in the schema of the table. In this case, we will need to join the core table with the overflow table that stores a_2 to get the complete answer. The cost of this join operation is tolerable because overflow tables are typically much smaller than the core table. In addition, storing overflow data in

relational tables have a better query performance than storing overflow data in local disk. The reason is because relational databases have very powerful query optimizers that will find the optimal plan for most queries. If we store overflow data on local disk, we cannot make use of this conventional tool and have to find a way to efficiently retrieve and update them. Furthermore, if the size of overflow data is too large to fit the main memory, then we have to fetch them from hard disk, which is also a time consuming process.

When the XML data set is small, for example, 1 MB, we observe that there is not much significant difference in the running times for the three schemes. However, when the data set increase, the performance grain in XStorM becomes obvious. Among the three schemes, XStorM gives the best performance for all the queries.

5. Conclusion

In this paper, we have examined how XML data can be stored using a relational database. The semistructured feature in XML introduces complications in the mapping. Our proposed scheme, XStorM overcomes the problems by making a distinction between XML elements that represent entities in the real world, that is, objects, and XML elements that represent properties of entities, that is, attributes. A breadth-first-search algorithm is used to identify objects in the XML data.

XStorM avoids excessive fragmentation of XML data by mapping each object together with the majority of its attributes to a core relational table. A data mining algorithm is employed to find frequent patterns in the XML dataset. These frequent patterns are used to generate the core relations for objects. Irregularities or data instances that deviate from the core schemas are stored in separate overflow tables. The names of these overflow tables also contain the structural information of the XML document for fast reconstruction of the original XML document. Our performance study has demonstrated that XStorM gives good query performance, minimizes storage space and is scalable.

We plan to extend this work in several directions. First, the object identification algorithm can only identify objects whose paths are of the same length. For objects that are represented by multiple paths of different length, new methods have to be designed. We are looking into this. Second, recently, WWW Consortium has released a recommendation on schema definition of XML data. We plan to examine how our results can be applied there.

Acknowledgement

This work is funded by the National University of Singapore Academic Research Fund RP082112.

Note

1. The prefix of a node is simply a chain of its ancestor nodes starts from root node and ends at its parent node.

References

[1] S. Abiteboul, D. Quass, J.Widom, and J. Wiener, "The lorel query language for semistructured data," *Internat. J. Digital Libraries* 1(1), 1997.

[2] T. Bray, J. Paoli, and C. Sperberg-McQueen, "Extensible markup language (XML) 1.0. W3C," Recommendation, available at `http://www.w3.org/TR/1998`, 1998.

[3] A. Deutsch, M. Fernandez, and D. Suciu, "Storing semistructured data with STORED," in *Proc. of ACM SIGMOD*, 1999, pp. 431–442.

[4] Document Object Model (DOM) level 1 specification, `http://www.w3.org/TR/REC-DOM-Level-1`.

[5] D. Florescu and D. Kossman, "Storing and querying XML data using an RDBMS," *Bulletin of IEEE Computer Society Technical Committee on Data Engineering*, 1999.

[6] J. McHugh, S. Abiteboul, R. Goldman, and J. Widom, "Lore: A database management system for semisturctured data," *SIGMOD Record* 26(3), 1997.

[7] J. Shanmugashundaram et al. "Relational databased for querying XML documents: Limitations and opportunities," in *Proc. of VLDB*, 1999.

[8] K. Wang and H. Liu, "Discovering typical structures of documents: a road map approach," in *ACM SIGIR Conf. on Research and Development in Information Retrieval*, 1998.

[9] R. Zowl, P. Apers, and A. Wilschut, "Modeling and querying semistructured data with MOA," in *Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats*, 1999.