XStorM: A Scalable Storage Mapping Scheme for XML Data

(Extended Abstract)

Wen Qiang Wang Mong Li Lee Beng Chin Ooi Kian-Lee Tan

Department of Computer Science, National University of Singapore
{wangwq, leeml, ooibc, tankl}@comp.nus.edu.sg

1. INTRODUCTION

As XML becomes pervasive, many information sources are beginning to 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.

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. [3] infers from the DTDs of the XML document how the XML elements should be mapped into tables. *STORED* [1] 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. [2] takes the graph representation of an XML document and studies various schemes to map the edges and nodes into relational tables. However, these schemes perform poorly for complex queries involving joins.

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. 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.

2. THE XStorM MAPPING SCHEME

In this section, we describe XStorM, a novel mapping scheme for storing XML data into relational database. The scheme comprises 3 phases as detailed in the following subsections.

2.1 Object Identification

In XStorM, we represent the XML document using DOM [4]. We observe that element nodes in DOM can be differentiated into object nodes and attribute nodes. The goal of this phase 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 especially if we know nothing about the XML instance.

We propose a three-step approach. First, we determine the number of paths corresponding to a prefix. We shall refer to this as the *support* of the path prefix. Next we identify the minimal path prefix which is the shortest path prefix whose support is greater than or equal to a certain predetermined threshold. Finally, the node at the lowest level of the minimal path prefix is the target object that we are looking for. It turns out that this scheme can be implemented efficiently using the well-known breadth first search (BFS) algorithm. Note that choosing an appropriate minimal support value is crucial as it may lead to different objects being identified.

2.2 Frequent Tree Patterns Identification

Given the "schema-less" semi-structured XML data, it is impossible to find a general schema that covers the whole XML data instance. [3] shows that generating a storage schema for semistructured data which minimizes cost is NP-hard in the size of the input data. Our approach is to use a data-mining algorithm to identify frequent tree patterns in the XML graph. This enables us to generate a schema that covers a major portion of the data. Our aim is to incorporate as much small attribute collections in an object's core relational table as possible to minimize the number of overflow tables. Query performance is improved when excessive fragmentation is avoided, reducing joins. We adapt the data-mining algorithm for semistructured data described in [4]. We shall first explain some concepts.

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. 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*. The *support of a tree-expression TE* is defined as the number of nodes that support *TE*.

We observe that a k-tree expression, $k \geq 1$, 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. Note that if the sequence of the 1-tree expression is different, then a different k-tree expression is constructed.

Let p_i denote 1-tree-expression, $i \geq 1$. Then a k-tree-expression $<p_1, p_2, ..., p_k>$ is constructed from two (k-1)-tree-expressions $<p_1, p_2, ..., p_{k-2}, p_{k-1}>$and $<p_1, p_2, ..., p_{k-2}, p_k>$. 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 "sub-tree-expression" that is known to be infrequent.

In order to determine frequent k-tree-expressions, we first use the depth first search algorithm to discover all the 1-tree-expressions starting from the object nodes found in phase 1. The support 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. The algorithm will eventually generate frequent k-tree-expressions for a given k. A large k should be used to find a schema with maximal data coverage.

2.3 Generate Core and Overflow Tables

The frequent k-tree-expression obtained in phase 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. 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. Overflow table names are given by *ObjectName.collectionName*. The overflow table names embed the XML structural information which is necessary for pattern matching queries and reconstruction of the XML document. There is also an additional attribute, *objectID*, in the overflow tables that contains the identifier of the object to which these overflow data belongs to.

3. DISCUSSION

We implemented XStorM, and evaluated its performance against the Binary approach in [2] and STORED [1]. We conducted experiments on XML documents whose sizes range from 1 MB to 100 MB containing 1000 to 100,000 objects. We also run a large set of queries, ranging from simple ones that retrieve objects from a single table to complex ones that involve multiple tables. Due to space constraint, we shall summarize our findings here. Readers are referred to [6] for details.

Theoretically, schemes which map XML data into relational tables based on schematic information should work better than just storing XML data in attribute tables. The most expensive operation in query processing is the 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's 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 today's RDBMS has very powerful query optimizer which 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, such as 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 gain in XStorM becomes obvious. Among the three schemes, XStorM gives the best performance for all the queries. The gain can be as much as 10 times over the Binary approach, and 4 times over STORED.

4. REFERENCES

[1] A. Deutsch, M. Fernandez and D. Suciu. Storing Semistructured Data with STORED. *Proc. ACM SIGMOD,* pp 431-442, 1999.

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

[3] J. Shanmugashundaram et al. Relational database for querying XML documents: Limitations and opportunities. *Proc. VLDB*, 1999.

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

[5] Document Object Model Level 1 Specification. **http://www.w3.org/TR/REC-DOM-Level-1**.

[6] W.Q. Wang, M.L. Lee, B.C. Ooi, K.L. Tan, "XStorM: A Scalable Storage Mapping Scheme for XML Data", available at http://www.comp.nus.edu.sg/~wangwq.