

Enumerating XML Data for Dynamic Updating

Lau Hoi Kit and Vincent Ng

Department of Computing,
The Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong
cstyng@comp.polyu.edu.hk

Abstract

In this paper, a new mapping model, called *n-INode*, is presented and it is used to support dynamic updating of XML data flexibly. One key feature of *n-INode* is the use of multi-dimensional numbering scheme to verify the containment relationships between XML nodes. The preliminary experiments show that *n-INode* supports dynamic updating of XML documents effectively and efficiently.

Keywords: XML, database, updating.

1 Introduction

There are different mapping models to store and manage XML data using relational databases. The two general approaches are the structure-mapping approach and the model-mapping approach as mentioned by YoshiKawa, M.A. T. (2001). The design of database schemas of the structure-mapping approach is based on the Document Type Definition (DTD) of XML documents. Mostly, they use the structural information to design the database schemas. Examples include STORED (Deutsch, A., Fernandez, M., and Suci, D. 1999), and XStorM (Wang, W.Q., Lee, M.L., Ooi, B.C., and Tan, K.-L. 2001). However, this design is not suitable for dynamic structure data because different structures of XML documents have different database schemas. Unlike structure-mapping approach, the database schemas of the model-mapping approach are fixed for different structures of XML documents. Examples include Edge (Florescu, D. and Kossmann, D. 1999), XRel (YoshiKawa, M.A. T. 2001), XParent (Jiang, H., Lu, H., Wang, W., and Yu, J.X. 2002), XNode (Ng, V., Lau, H.K., and Chun, S.W. 2002), INode (Lau, H.K. and Ng, V. 2003), and INode* (Lau, H.K. and Ng, V. 2004). It is capable to support dynamic structure data, as no structural information is required to design the database schemas. Therefore, it is more flexible and convenient to manage XML documents in relational databases.

To process XML documents, besides retrievals, three basic operations should be included. They are insertion,

updating and deletion. However, in many mapping methods, these kinds of operations are not considered.

In addition, structurally recursive XML queries are an important query class that follows the structure of XML data. In Park, C.-W., Min, J.-K., and Chung, C.-W. (2002), the authors defined structurally recursive query as a query involves an ancestor-descendant relationship (“//”), a filtering with ancestor-descendant relationships, and function calls to user-defined structurally recursive functions. Based on several use cases as discussed in Chamberlin, D., Fankhauser, P., Marchiori, M., and Robie, J. (2001) and the requirements of XQuery, structurally recursive queries are frequently used and they meet different requirements such as abilities to query the XML data without knowing the logical structure. For example, the following query “//title[//subject='business']” often uses to retrieve *title* elements, where *subject* descendant is ‘business’, regardless of the structure of data and the locations of *title* and *subject* elements in the data.

This paper is structured as follows. Section 2 gives a review to the related works in model-based mapping and briefly describes INode. We will discuss the supports of dynamic XML documents in Section 3. Section 4 introduces our new mapping method, *n-INode*. In Section 5, experimental results are shown. Finally, we conclude our work in Section 6.

2 Related Works

Edge, XRel, XParent, XNode, INode and INode* are six model-mapping approaches for storing different structures of XML documents in relational databases. Edge stores the XML data graph in a single table called *Edge*. Each node in the data graph is assigned a sequential number. Each tuple in the table corresponds to an edge in the data graph.

XRel uses a schema of four tables to store the XML documents. The tables are *Path*, *Element*, *Text* and *Attribute*. Each node has a region which is represented by the start and end position of the corresponding node in the XML document. By using the region, XRel does not require recursive queries to verify the ancestor-descendant relationship. Instead of using recursive queries, XRel uses ?-joins to identify the containment relationship. However, it is not effective to use ?-joins for queries with more ancestor-descendant relationships and no special index mechanism, which supports containments, is provided by off-the-self database management systems.

XParent uses a four-table schema to store XML data. Instead of using regions as in XRel, XParent uses a table, called *DataPath*, to maintain parent-child relationship. To verify the ancestor-descendant relationship, it requires

joining the *DataPath* table itself. It uses equijoins instead of ?-joins, which are considered as more costly than equijoins.

INode and INode* uses the concept of UID, which is proposed in Kha, D.D., M. Yoshikawa, and S. Uemura (2002), to assign an identifier (id) to each node in an XML document. It represents an XML document as an n_c -ary complete tree where n_c is the maximum number of child nodes of a node in the structure. Given a node n , the path of n will be denoted as $(a_1, a_2, a_3, \dots, a_k)$, where k is the depth of n and $1 = a_i = n_c$. With the concept of UID, for a node with UID i , the parent's UID of this node can be calculated directly by Equation (1). Furthermore, given n_c and the corresponding path sequence $(a_1, a_2, a_3, \dots, a_k)$ of a node, it calculates the node id using Equation (2) (All equations can be found at the Appendix page).

Given the node ids of i and j , where i is the ancestor of j and the depth difference between these two nodes is d , INode and INode* confirm the ancestor-descendant relationship by calculations.

INode uses a three-table schema. They are *Path*, *Element* and *Attribute*. The key feature of INode is that it uses node ids to maintain the parent-child relationship instead of using region as in XRel or using another table as in XParent. This further reduces the number of table joins when performing queries with the parent-child relationship or the ancestor-descendant relationship. However, it requires recursive queries to retrieve the ancestor-descendant relationship.

3 Dynamic Updating of XML Documents

Three basic operations are needed to support dynamic updating of XML documents. They are insertion, modification and deletion. To modify or delete a node, it requires locating the desired node and performs modification or deletion. The total cost of modification (deletion) is the sum of the cost of locating the desired node and the cost to modify (delete) the node. Here, these total costs are dominated by the costs of locating the desired node. Therefore, it is important to have a good query performance for the location step for the modify and delete operations.

For the insertion of Edge, it requires to locate the parent node because one of the database attribute is the source ID, which is pointing to the parent node. Then it may require updating the ordinal of the sibling nodes. Finally, it inserts the node in the Edge table.

Unlike Edge, XRel uses the concept of region to identify the nodes. For every insertion, it will change the start and end positions of existing nodes. Therefore, it requires updating the start and end positions of existing nodes before inserting a node.

As XParent uses a table, *DataPath*, to maintain the parent-child relationship, a new record of this table is required for each insertion. It also requires retrieving the parent node in order to get the corresponding *Pid* of the parent node. Like Edge, it requires updating the ordinal of the sibling nodes when necessary. Finally, it inserts the

node in *Element* table and a record of parent-child relationship in *DataPath* table.

To insert a node using INode or INode*, it does not require retrieving the parent node likes Edge and XParent because it uses node id to maintain the parent-child relationship. It calculates the node id with path sequence only. Therefore, it just inserts the node and updates the ordinal of the sibling nodes. However, it has fixed the maximum number of child nodes in order to calculate the node id. This limited the number of nodes that can be inserted.

4 n-INode

A new mapping method, n -INode, is developed to enhance the flexibility to support dynamic updating of XML documents. In the n -INode method, a multi-dimensional node id is introduced. The multi-dimensional node id enables the ability to insert new nodes to a node when it has n_c child nodes and all existing node ids are not to be re-calculated. For node ids with k dimensions, each dimension has a pair of numbers, (id^x, io^x) where $1 \leq x \leq k$. The first number is calculated based on the path sequence with Equation (2). The second number is the insertion order. Therefore, for a node with k dimensions, the node id is defined as $(id^1, io^1, id^2, io^2, id^3, io^3, \dots, id^k, io^k)$.

Since id^x is calculated, it is possible that the id^x of a newly inserted node is the same as the id^x of an existing node. Therefore, the insertion order, io^x , is introduced to distinguish nodes with the same id^x . The insertion order is a sequence of numbers starting from zero. For all descendants of a node that the corresponding insertion order is larger than zero, a new dimension is used in order to identify the nodes.

By using the multi-dimensional node id, n -INode can continue to insert nodes as child nodes to a given node when it has n_c child nodes. In addition, it does not require maintaining the original XML documents. The handling of different types of insertions will be discussed next.

Case 1. Insert a child node under a non-full node.

In Fig. 2, since the number of child nodes of A is less than n_c , B can use the value of id_1 calculated by Equation (2). Hence, the value of the virtual node id is 10 and the node id of B is (10,0).

Case 2. Insert a child node at the leftmost under a full node.

In Fig.3, the id_1 of C must be used by another node as the number of child nodes of A is n_c . Hence, the value of io_1 of C is the increment of the maximum value of io_1 amongst the existing nodes with the same id_1 values. Since the path sequence of C is (1,2,1), the parent node id is 3, the id_1 calculated by Equation (1) is 8, and the maximum of io_1 for the nodes with 8 as id_1 is 0. Hence, the node id of C is (8,1).

Case 3. Insert a child node at the rightmost under a full node.

Similar to case 2, there must a node with the same id_I of D existed, which is B shown in Fig. 4. Here, io_I is used again to distinguish the nodes and io_I is assigned to a value larger than n_c . If the maximum of the insertion order is not larger than n_c , n_c+1 is used as the value of io_I instead. In this case, the id_I of D is 10, which is the same as B while the maximum of io_I for the nodes with 10 as id_I is 0. As a result, n_c+1 will be the value of io_I for D, which is 4. Finally, the node id of D is (10,4). Based on this assignment method, the number of nodes that can be inserted at the left-hand side of B is limited by n_c . All node ids are re-calculated when io_I of any node is equal to n_c and a new and larger value of n_c is used to re-calculate the node ids.

Case 4. Insert a child node in the middle under a full node.

In Fig. 5, id_I of E is the same as that of node Q, which is the right hand side of the newly inserted node. Suppose the path sequence of node Q before any insertion is (1, 2, 2). After the insertion, the path sequence of E is also (1, 2, 2). The ids of E and Q are the same, which are 9. Therefore, the io_I is used to distinguish the nodes and the io_I value of E is the increment of the maximum of io_I of existing node with the same id_I . In our example, the maximum of io_I for the nodes with 9 as id_I is 0. Therefore, the io_I of E is 1 and the node id of E is (9,1).

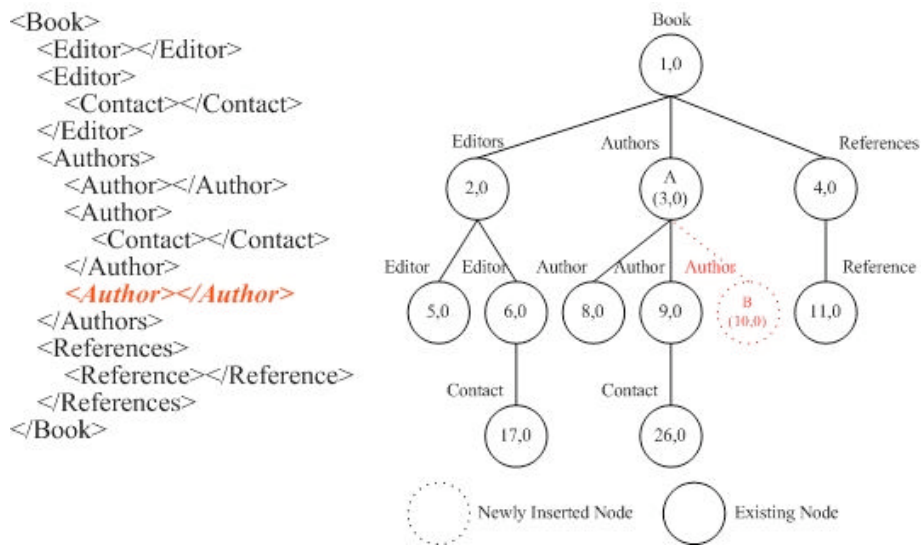


Fig. 2. Case 1 – Insertion.

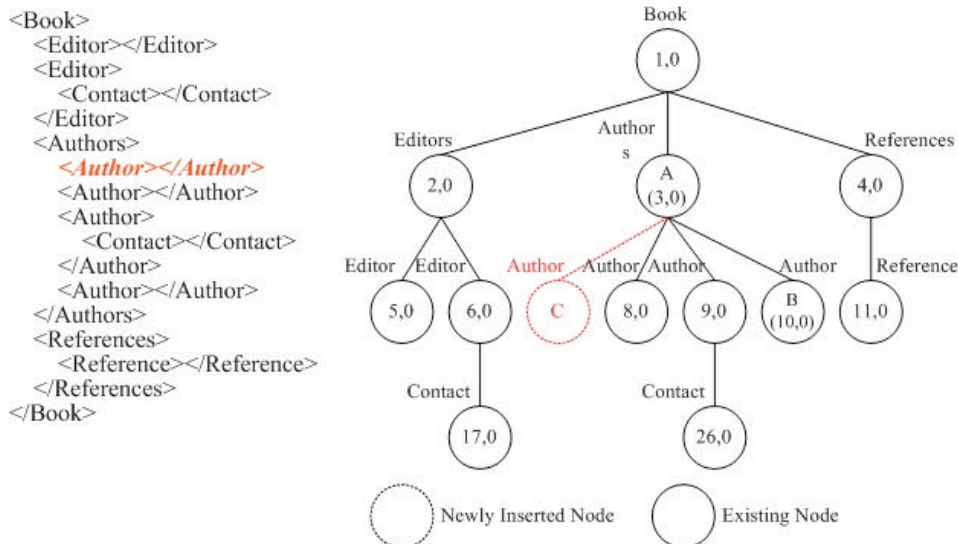


Fig. 3. Case 2 – Insertion.

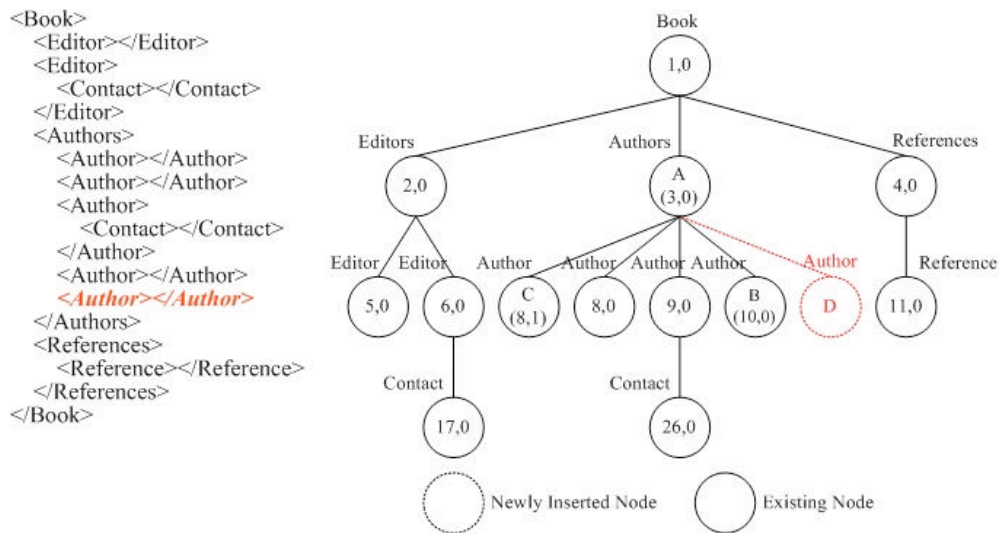


Fig. 4. Case 3 – Insertion.

Case 5. Insert a node F under node E where the insertion order of node E is larger than one as shown in Fig. 6.

In this case, the insertion of E is similar to Case 4 and F is further inserted as a child of E. The path sequence of F is (1,2,2,1) and the calculated id of F is 26 by using Equation (2). However, a node with node id (26,0) already exists, which is Q, and the insertion order cannot be used to identify these two nodes. If the insertion order is used, the possible node id of F becomes (26,1). Based on this assignment, the parent-child relationships of nodes E, F, Q and R cannot be verified by using Equation (1). It is because Q can be the parent node of F and E can be the

parent node of R by using Equation (1). Therefore, it requires another way to distinguish F and R and the second dimension of node id is used. For all existing nodes, the second dimension node id is assigned with (0,0). For example, the node id of E becomes (9,1,0,0). For the newly inserted node F, the first dimension is assigned with (9,1), which is the same as the first dimension node id of its parent node. The id of the second dimension is the calculated value by Equation (2), which is 26. The insertion order of the second dimension is zero as no existing node with the same node id. Finally, the node id of F is (9,1,26,0).

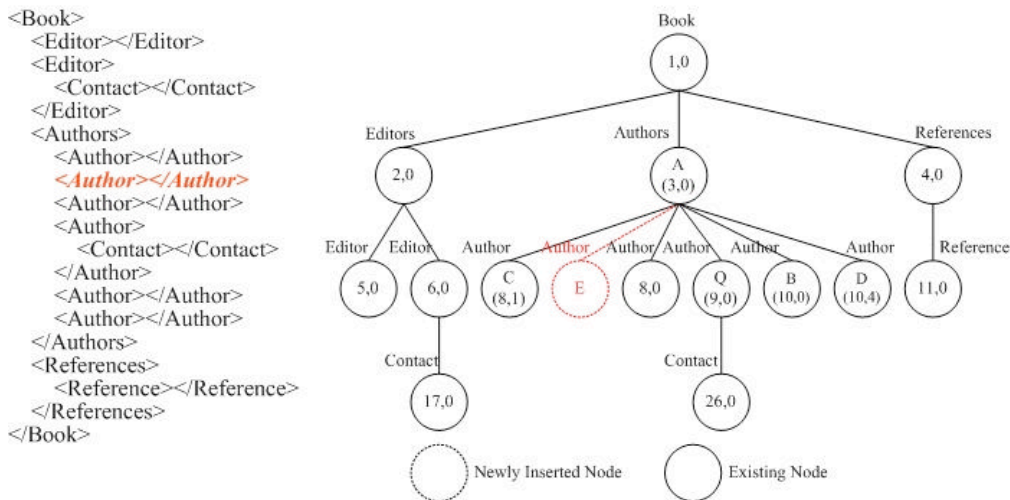


Fig. 5. Case 4 – Insertion.

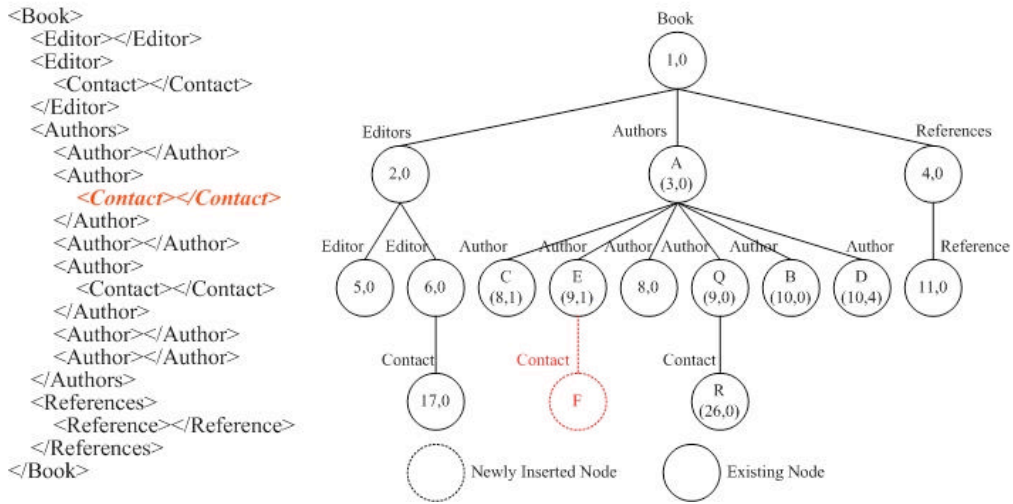


Fig. 6. Case 5 – Insertion.

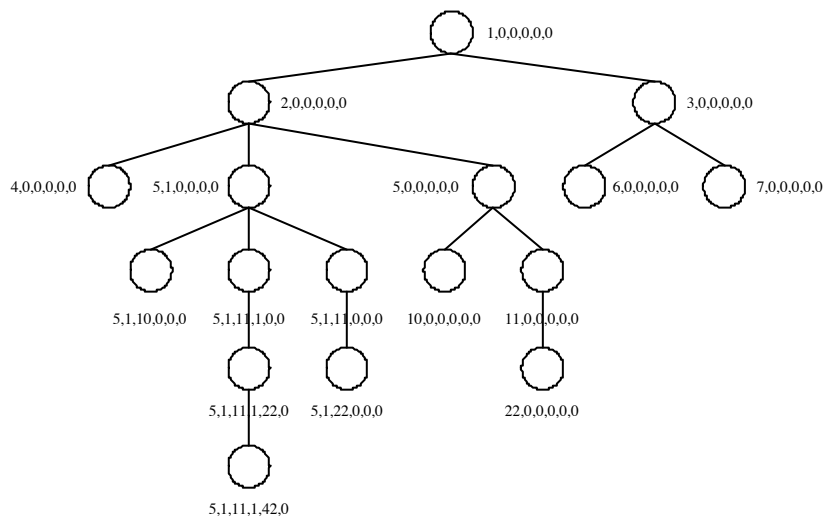


Fig. 7. An Example of Three Dimension Node ID.

Fig. 7 shows an XML data graph with three dimensions node ids assigned and the value of n_c is three. According to the assignment of node ids, any two nodes have parent-child relationship if and only if the numbers of dimensions are the same or the dimension difference is one. Therefore, two cases are considered for the verification of parent-child relationship.

Case A. IDs with the same number of non-zeros id_x .

Consider the nodes n_1^3 and n_2^3 with ids (5,1,11,0,0,0) and (5,1,22,0,0,0) as shown in Fig. 7, respectively. The ids are divided into three parts as follow:

| 1 st part | 2 nd part | 3 rd part |
|----------------------|----------------------|----------------------|
| 5 1 | 11 0 | 0 0 |
| 5 1 | 22 0 | 0 0 |

For the first part, it requires to verify that the nodes belong to the same sub-tree. Therefore, the ids must be the same for the first part. For the second part, the relationship is verified by calculation. Since the ids are 2 dimensions only, the third parts of them must be zero. In general, for k

dimensions, the parent-child relationship is verified using Equation (3), where j is the number of dimensions of the ids.

Case B. IDs with one dimension difference.

Consider the nodes n_1^3 and n_2^3 with ids (5,1,11,1,0,0) and (5,1,11,1,22,0) as shown in Fig. 7, respectively. The ids are divided as three parts again. However, there is no zero id_x in the id of n_2 . The third part is skipped below.

| 1 st | 2 nd |
|-----------------|-----------------|
| 5 1 | 11 1 0 0 |
| 5 1 | 11 1 22 0 |

Like Case 1, the first parts of ids are the same in order to verify that the nodes are belong to the same sub-tree. Since there is a dimension difference of the ids, the second part is required to verify two sub-conditions. One sub-condition is the nodes belong to the same sub-tree. It works similarly as the verification of the first part. The other sub-condition is that the parent-child relationship to be verified. The verification of the third part is the same as in Case 1. In general, for k dimensions, the parent-child relationship is

verified using Equation (3), where j is the number of dimensions of the parent node id.

For the verification of ancestor-descendant relationship, three cases are considered. The first two cases are similar to the cases of parent-child relationship by replacing the *Parent* function with *Ancestor* function. Another case is considered below.

Case 3. IDs with more than one dimension difference.

Consider the nodes n_1^3 and n_2^3 with ids (5,1,0,0,0,0) and (5,1,11,1,22,0) as shown in Fig. 7, respectively. The ids again are divided into three parts. However, the non-zero dimension of n_1^3 is one and there is no zero pair for the id of n_2^3 . The first and the third part are omitted here.

| 2 nd part | | | | | |
|----------------------|---|----|---|----|---|
| 5 | 1 | 0 | 0 | 0 | 0 |
| 5 | 1 | 11 | 1 | 22 | 0 |

Let j is the number of non-zero dimensions of n_1 and l is the number of non-zero dimensions of n_2^3 . With the second part, the relationship is verified by calculations with j dimension node id of n_1 and j to l dimension node ids of n_2^3 . For k dimensions, the ancestor-descendant relationship is verified by Equation (4).

5 Implementation and Experimental Results

For the implementation of multi-dimensional node id, all zero pairs of id_x and io_x are replaced with the last non-zero pairs. Fig. 7 shows an example with 3 dimensions. In the figure, node (5,1,0,0,0,0), is represented as (5,1,5,1,5,1). This replacement aims to eliminate the checking of zero pairs.

For the modify operation, the node is selected and modified. Similarly, the node is selected to perform deletion and the node is marked as deleted. In addition, the node id of the deleted node is not used again for newly inserted nodes.

In our design, a database attribute, *Dim*, is added to *Element* and *Attribute* table, which is the number of dimensions used of the node id. Finally, n -INode adopts a four-table schema. The tables include Path, Element, Attribute and Ancestor. A database schema example is shown in Table 1 (found at the Appendix page). When the maximum size of an XML file is available, we can estimate the number of dimensions needed but this is not often provided.

To evaluate the query performance of using n -INode, experimental studies have been conducted. Edge, XRel,

XParent, INode* and n -INode are studied. All experiments were conducted on a 933MHz Pentium III with 256MB RAM, 20GB hard disk. The RDBMS used was Oracle 9i Enterprise Edition and the size of SGA is 37MB. The Bosak Shakespeare collection (2002) and the XML benchmark project (2001) are used as the data sources. For the second data source, different scale factors are used to generate the data. Four different sizes of data, known as Set1, Set2 Set 3 and Set4, are generated using the generator version 0.92. Set4 is used to evaluate the support of dynamic updating of XML documents as shown in. Table 2.

Table 2. Data Sets of the XML Benchmark Project

| Name | Parameters Used | Size (MB) | Maximum depth |
|------|-----------------------------|-----------|---------------|
| Set1 | scale factor=0.05, split=10 | 5.61 | 12 |
| Set2 | scale factor=0.1, split=10 | 11.3 | 12 |
| Set3 | scale factor=0.2, split=10 | 22.8 | 12 |
| Set4 | scale factor=0.1 | 11.3 | 12 |

As discussed in Section 3, the performance of modification and deletion is directly related to the query performance. Therefore, our evaluation will be focused on insertion. In order to have a set of nodes for insertion, Set4 is divided into two parts. The first part is loaded into the database and this part is inserted again to evaluate the performance of inserting nodes to the loaded data at different locations. It includes the first two hundred child nodes of each parent node. The second part includes the rest of the nodes that are not loaded in order to evaluate the performance of appending nodes to the loaded data. Since XRel is based on the concept of region, it requires the original XML documents or reconstructs the XML files from relational tables to determine and update the start and end positions. Therefore, we studied Edge, XParent and n -INode. For each insertion, three methods require retrieving the parent node. The insertion cost is defined as the sum of query elapsed time and the insert elapsed time. In addition, two query types are used to retrieve the parent node. They are structurally recursive query and non-structurally recursive query. The numbers of nodes that are inserted are 50, 100, 200 and 400. The average insertion cost is taken. The values of n_c is 20 for n -INode with three dimensions.

5.1 Storage Usage

Fig. 8 shows the storage usage of n -INode in comparison with Edge and XParent. The data loaded is the first part of Set4. After implemented the multi-dimension node id, the storage required of n -INode is increased. However, it is lower than that of XParent.

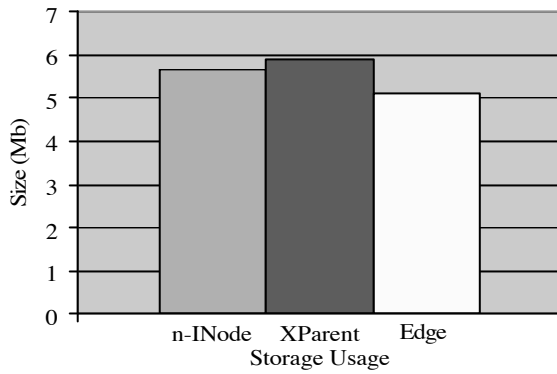


Fig. 8. Storage Usage with *n*-INode, XParent and Edge.

5.2 Insertion Performance

Fig. 9 and Fig. 10 show the average insertion cost of inserting a node. The average insertion cost is in log scale. In most cases, *n*-INode performs similarly to XParent and it performs better than Edge for insertion with structurally recursive queries and *n*-INode further improves the performance when a large number of nodes are inserted because it does not require using recursive query.

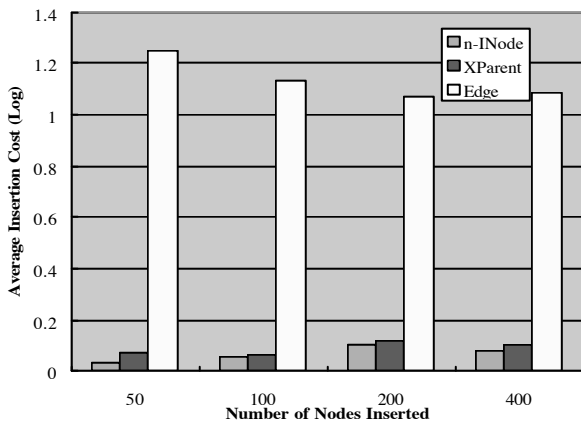


Fig. 9. Average Inserted Cost of Insertion of a Node in the Second Part of Set4.

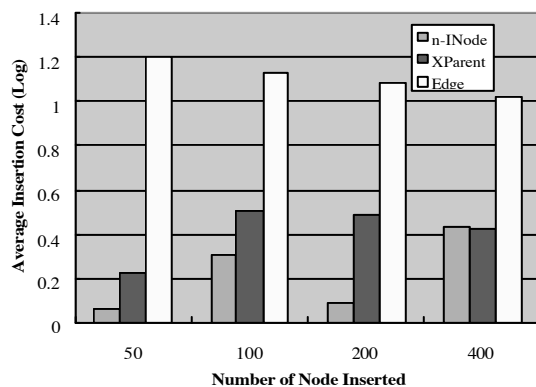


Fig. 10. Average Inserted Cost of Insertion of a Node in the First Part of Set4.

5.3 Query Performance

The Bosak Shakespeare collection is used for the evaluation of query performance. To load the data set of the Bosak Shakespeare collection, the value of n_c is 50 with three dimensions for *n*-INode. The set of queries is shown in Fig. 11.

The query elapsed times are shown in Fig. 12. Every query is run ten times for each method and the average elapsed time is taken. *n*-INode performs similarly to INode* as they use the same calculation to verify the relationships between different nodes. However, *n*-INode performs not as well as INode* in some cases because more calculations are used to verify the relationships due to more dimensions. Even then, its performance is comparable to XParent and XRel.

5.4 Scalability Test: XRel vs XParent vs *n*-INode

In the scalability test, Set1, Set2 and Set3 of the XML benchmark project are used. The settings of *n*-INode to load the data are the same as that of the evaluation of insertion performance. Fig. 13 shows the result of the scalability test of XRel, XParent, INode* and *n*-INode, respectively. The query elapsed time ratio is defined as t_2/t_1 where t_1 is the elapsed time of a query using Set1 and t_2 is the elapsed time of the same query using Set2 or Set3. By implementing the multi-dimension node id and Ancestor table, the scalability of *n*-INode is improved and it is superior to XRel and XParent.

| | |
|------|---|
| QS1: | /PLAY/ACT |
| QS2: | /PLAY/ACT/SCENE/SPEECH/LINE/STAGEDIR |
| QS3: | //SCENE/TITLE |
| QS4: | //ACT/TITLE |
| QS5: | /PLAY/ACT[2] |
| QS6: | (/PLAY/ACT)[2]/TITLE |
| QS7: | /PLAY/ACT/SCENE/SPEECH[SPEAKER='CURIO'] |
| QS8: | /PLAY/ACT/SCENE//SPEAKER='Steward']/TITLE |
| QS9: | /PLAY/ACT//SPEAKER='Steward']/TITLE |

Fig. 11. Queries Used with the Bosak Shakespeare Collection.

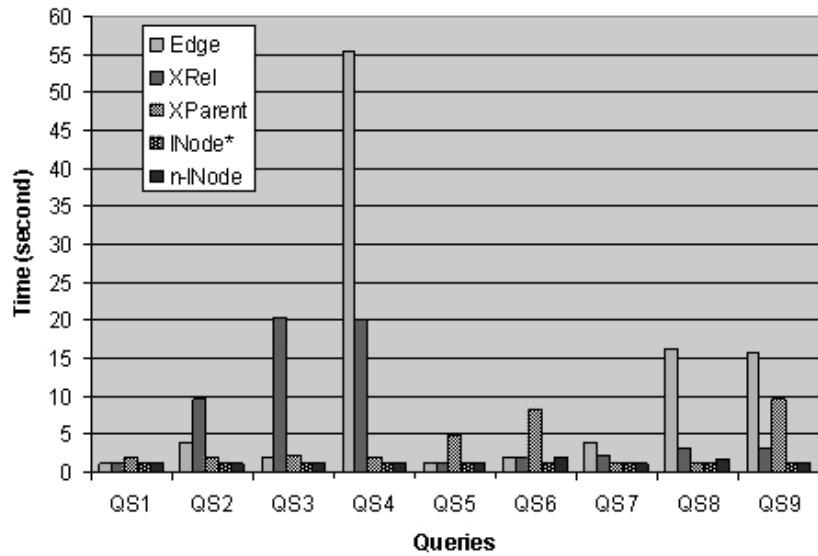
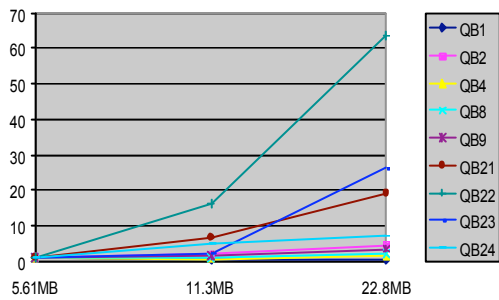
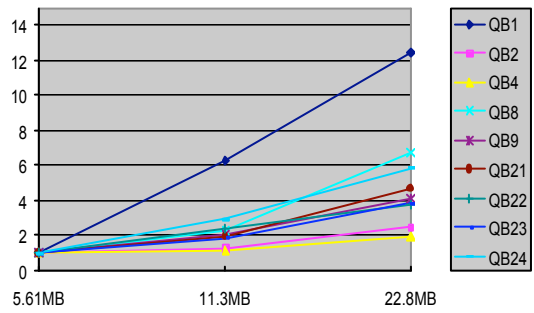


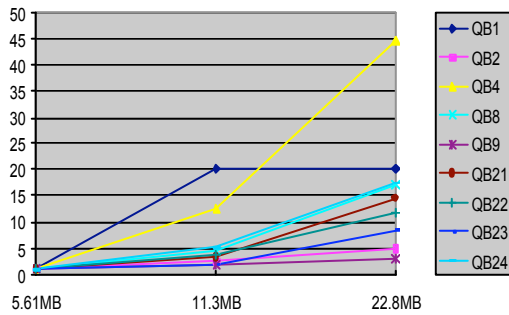
Fig. 12. Average Query Elapsed Time: using the Bosak Shakespeare Collection



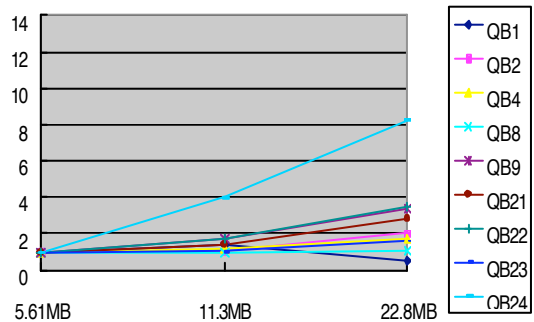
(a) Average Query Elapsed Time Ratio for XRel.



(c) Query Elapsed Time Ratio for INode*.



(b) Average Query Elapsed Time Ratio for XParent.



(d) Average Query Elapsed Time Ratio for n-INode.

Fig. 15. Scalability Test with XRel, XParent, INode and n-INode.

6 Conclusions

In this paper, a new mapping scheme, *n-INode*, is introduced to support dynamic updating of XML documents. In the experimental study, the query performance and the effectiveness of supporting dynamic updating XML documents are studied. *n-INode* outperforms or has similar insertion and query performance in most cases in comparison with other methods.

Acknowledgement

The work of the authors are supported in part by the Central Grant of The Hong Kong Polytechnic University, research project code H-ZJ89.

7 References

YoshiKawa, M.A. T. (2001). XRel: a path-based approach to storage and retrieval of XML documents using

- relational databases. *ACM Transactions on Internet Technology (TOIT)*, 1(1): pages 110-141.
- Deutsch, A., Fernandez, M., and Suci, D. (1999). Storing Semistructured Data with STORED, Proceedings of the 1999 ACM SIGMOD international conference on Management of Data, pages 431-442, ACM Press.
- Wang, W.Q., Lee, M.L., Ooi, B.C., and Tan, K.-L. (2001). XStorM: A Scalable Storage Mapping Scheme for XML Data. *World Wide Web*, 2001. 4(1-2): pages 101-119.
- Florescu, D. and Kossmann, D. (1999). A Performance Evaluation of Alternative Mapping Schemes for Storing XML Data in a Relational Database. INRIA. pages 31.
- Jiang, H., Lu, H., Wang, W., and Yu, J.X. (2002), Path materialization revisited: an efficient storage model for XML data. Proceedings of the thirteenth Australasian conference on Database technologies. pages 85-94, Australian Computer Society, Inc.
- Ng, V., Lau, H.K., and Chun, S.W (2002). XNode: Fast Retrieval of XML Data from Relational Tables. IASTED International Conference Information Systems and Databases (ISDB 2002), Tokyo, Japan.
- Lau, H.K. and Ng, V. (2003). INODE An Enumeration Scheme for Efficient Storage of XML Data. Cooperative Internet Computing. 2003, Kluwer Academic Publisher, pages 165-184.
- Park, C.-W., Min, J.-K., and Chung, C.-W. (2002). Structural Function Inlining Technique for Structurally Recursive XML Queries. The 28th International Conference on Very Large Data Bases, pages 83-94, Hong Kong, China.
- Kha, D.D., M. Yoshikawa, and S. Uemura (2002). A Structural Numbering Scheme for XML Data. EDBT 2002 Workshops, Lecture Notes in Computer Science 2490, Springer-Verlag Heidelberg, pages 91-108.
- Chamberlin, D., Fankhauser, P., Marchiori, M., and Robie, J. (2001). XML query use cases. Working Draft. <http://www.w3.org/TR/2001/WD-xmlquery-use-cases-20011220>. 20 December 2001.
- The Bosak Shakespeare collection.* (2002) <http://metalab.unc.edu/bosak/xml/eg/shaks200.zip>.
- Schmidt, A.R., et al. (2001). *The XML Benchmark Project*.
- Lau, H.K. and Ng, V. (2004), INode*: An Effective Approach for Storing XML using Relational Database. *IJWET Journal Special Issue on INTERNET AND DATABASE*. (To be published)

Appendix

$$Parent(i) = \left\lfloor \frac{(i-2)}{n_c} + 1 \right\rfloor \quad (1)$$

$$f(path) = \begin{cases} \sum_{i=1}^2 a_i & , k \leq 2 \\ n_c^{k-2} \sum_{i=1}^2 a_i - n_c^{k-2} + \sum_{i=0}^{k-3} n_c^i a_{k-i} + 1 & , k > 2 \end{cases} \quad (2)$$

| | |
|--|-----|
| $Parent(n_1^k, n_2^k) \equiv \exists j(1 \leq j \leq k) \cdot \left[\bigvee_{i=1}^{j-1} (id_1^i = id_2^i) \wedge (io_1^i = io_2^i) \right] \wedge$ $\left\{ \left[[id_1^j = Parent(id_2^j) \wedge (io_1^j = 0)] \wedge \left[\bigvee_{i=j+1}^k (id_1^i = id_2^i = io_1^i = io_2^i = 0) \right] \right] \vee \right.$ $\left. \left[[id_1^j = Parent(id_2^{j+1}) \wedge (id_1^j = id_2^j) \wedge (io_1^j = io_2^j)] \wedge \left[\bigvee_{i=j+1}^k (id_1^i = io_1^i = 0) \right] \wedge \left[\bigvee_{i=j+2}^k (id_2^i = io_2^i = 0) \right] \right] \right\}$ | (3) |
| $Ancestor(n_1^k, n_2^k) \equiv \exists j(1 \leq j \leq k) \exists l(j \leq l \leq k) \left[\bigvee_{i=1}^{j-1} (id_1^i = id_2^i) \wedge (io_1^i = io_2^i) \right]$ $\wedge \left[\left[\bigvee_{i=j+1}^l id_1^i = Ancestor(id_2^i) \wedge (id_1^j = id_2^j) \wedge (io_1^j = io_2^j) \right] \vee \left[\bigvee_{i=j}^l Ancestor(id_1^i, id_2^i) \wedge (io_1^i = 0) \right] \right]$ $\wedge \left[\bigvee_{i=j+1}^k (id_1^i = io_1^i = 0) \wedge \bigvee_{i=l+1}^k (id_2^i = io_2^i = 0) \right]$ | (4) |

Table 1. Database Schema of *n*-INode Using Two Dimensions Node ID.

| Table | Database Attributes |
|-----------|--|
| Path | PathID, Len, PathExp |
| Element | NodeID1, Order1, NodeID2, Order2, DocID, PathID, Ordinal, Value, Dim |
| Attribute | NodeID1, Order1, NodeID2, Order2, DocID, PathID, Value, Dim |
| Ancestor | Difference, Component1, Component2 |