A Comprehensive Evaluation of a Bitmapped XML Update Handler

Mohammed Al-Badawi, Abdallah Al-Hamdani, and Youcef Baghdadi
Department of Computer Science
Sultan Qaboos University
Muscat, Oman
{mbadawi, abd, ybaghdadi}@squ.edu.om

***Abstract*—XML (eXtensible Markup Language) update is problematic for many XML databases. The main issue tackled by the existing (and new) XML storages and indexing techniques is the cost reduction of updating the XML's hierarchal structure inside these storages. PACD (an acronym for Parent-Ancestor/Child-Descendent), as bitmapped XML processing technique introduced earlier, is an attempt in this direction. The technique brings the cost of updating the XML structure to the data representation level by introducing the 'next' and 'previous' axes as a mechanism to preserve the document order, and then using well-established matrix-based operations to manipulate the database transactions. This paper mainly provides a complexity analysis of the PACD update framework and presents a novel experimental evaluation method (in terms of comprehensiveness and completeness) for its update primitives. The outcomes of this evaluation have shown that the cost of eight update primitives (out of nine provided by PACD) locates under an acceptable range of a constant 'c', where 'c' is an extremely small number comparing to the number of nodes 'n' in the XML tree. Such good performance is lacked in the comparable techniques.**

Keywords-XML Databases; XML/RDBMS Mapping; XML Update; XML Indexing; Complexity Analysis; Experimental Design.

I. Introduction

Data stored in the extensible markup language (XML) containers (databases) is subject to update when circumstances change [1]. Unfortunately, handling XML updates is a common problem in the existing XML storages and optimization techniques. Relational approaches using node labeling techniques [2][3][4][5][6][7][8][9][10][11][12][13] require a large number of renumbering operations in order to keep the node labels updated whenever a node is inserted, deleted or moved from one location to another in the XML tree. For the approaches that use path summaries to encode the XML hierarchical structure [14][15][16][17][18], an additional cost results from updating these summaries. In native XML approaches such as sequence based [19][20][21][22][23] and feature based techniques [24][25][26], the update problem is even worse. In the first case, the consequences of a single update operation (for example deleting a node) can affect thousands locations in the corresponding sequence depending on the node location in the XML tree. A similar problem occurs in the case of feature based techniques, which rely on encoding the relationship between the nodes and the different ePaths of the XML tree inside what is called feature-based matrices [24].

PACD is XML processing technique introduced in [28][29] that brings the cost of updating the XML hierarchal structure to the data representation level by encoding these structures into a set of structure-based matrices each of which encodes a specific XPath [27] axis, plus two more axes specifically introduced by PACD to preserve the document order. Thus, PACD architecture combines some matrix-based operations along with the bit-wise operations to reduce the cost of querying and updating the structure of underlying XML file. This paper extends our previous work [1] by providing a detailed complexity analysis of the PACD Updates Query Handler (UQH). Unlike many existing studies, this paper presents a comprehensive evaluation process, which provides 1) a full algorithmic listing of all XML update primitives so that they can be re-used, 2) a detailed cost-analytical procedure of the XML update primitives, and 3) a supportive comprehensive experimental procedure that considers several testable aspects of the XML databases. Such evaluation method could be adopted by the XML research and development community to evaluate XML database processing techniques.

The paper starts by revisiting the PACD's framework in Section II. Then it introduces the UQH framework in Section III, while Section IV puts forward assumptions to facilitate the discussion of complexity analysis in the subsequent sections. Sections V to VII provide a detailed discussion of three types of update primitives: the insertion, deletion and change primitives, respectively. The overall complexity analysis and a supportive experimental evaluation are given in Sections VIII and IX, respectively. Section X concludes the paper.

II. Background: PACD's XML Processing Model

PACD, introduced in [28][29], is a bitmap XML processing technique consisting of three main components: the Index Builder (IB; operations I.1-I.4), the Query Processor (QP) and the Update Query Handler (UQH). The IB (see Figure 1) shreds the XML hierarchal structure (derived by the XPath's thirteen axes and their extension; the Next and Previous axes [28]) into a set of binary relations each of which is physically stored as an n×n bitmap matrix. An entry in any matrix is '1' if there is a corresponding relationship between the coupled nodes or '0' otherwise [30][25]. The IB operations I.2-I.4 are responsible to reduce

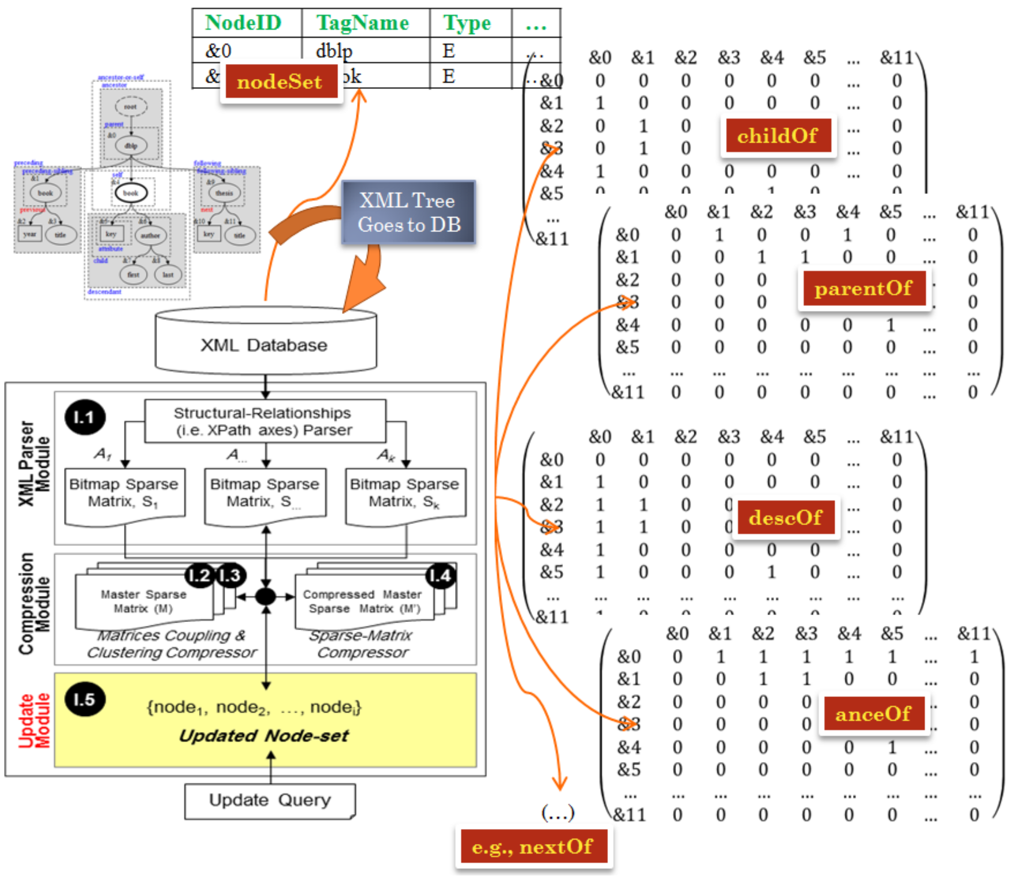


Figure 1. PACD Framework

the size of storing the XML structure by applying three levels of compression: the matrix-transformation level, the matrix-coupling level and the sparse-matrix compression level. More details about the data compression mechanism in particular and the IB in general can be found in [28][29].

On the other hand, the QP performs all operations related to the search-query execution. The full architecture of the QP was described in [29] but in brief, the process starts by analyzing the search-query statement to identify the affected nodes based on the twig structure. The process also identifies the query base matrices and draws an execution plan for the entire query, which eventually returns the results into a tabular-format (i.e., sub-matrices) and then converted to an XML data layout.

The next section describes the PACD's third component, that is the UQH, the core subject discussed in this paper.

III. THE UPDATE HANDLER

The PACD's UQH is responsible for all update operations, which includes the translation of the update query, the identification of update primitive(s), and the primitive execution.

Once the query is translated (e.g., from XQuery syntax to an SQL statement), the UQH starts identifying the node(s) that are affected by the update command/query. It navigates through the finite-state-machine (FSM) version of the update query in order to identify the affected node-set. Once the target node-set is known, the UQH determines and calls the appropriate update primitive (see Table I). PACD supports update primitives for single node insertion and deletion, twig insertion and deletion, and textual and structural contents changes.

The update primitive acts on all PACD's components including the NodeSet container and the structure based matrices (i.e., childOf, descOf and nextOf). Each update primitive executes certain instructions over each component such as adding new columns and rows and changing the bitmapped entries within the matrices. The cost of the update query execution will be the lump sum of the costs of executing all derived update primitives over each PACD's component. For example, an 'insert' primitive will involve adding one or more rows and columns to the bitmapped matrices, as well as adding one or more entries to the NodeSet container. Thus, the cost of the 'insert' operation becomes the cost of inserting the node information inside the NodeSet container plus the cost of inserting one row and column inside the childOf, descOf and nextOf matrices. More examples on using update primitives will be given later during the discussion of the update primitives.

The above steps are summarized in the algorithm provided in Figure 2, whereas Table I lists out the update primitives that are currently supported by PACD's UQH.

```
INPUT: update-query
OUTPUT: none
Construct the FSM execution plan of the corresponding twig
node-set = the returned node-set from the FSM execution
Using the update-query syntax, determine the update-primitive(s)
Call the update-primitive(s) with the obtained node-set:
    Alter the NodeSet container;
    Alter the childOf matrix;
    Alter the descOf matrix;
    Alter nextOf matrix;
End;
```

Figure 2. PACD Update Handler Algorithm

TABLE I: PACD UPDATE HANDLER PRIMITIVES

Insertion	insertLeaf	adds a leaf node
	insertNonLeaf	adds an internal node
	insertTwig	adds a single-rooted, connected sub-tree
Deletion	deleteLeaf	removes a leaf node
	deleteTwig	removes a single-rooted, connected sub-tree
Updating	changeName	renames an element or attribute name
	changeValue	edits the value (text) of an attribute (element)
	shiftNode	moves a node from one place to another
	shiftTwig	moves a single-rooted, connect sub-tree from one place to another

IV. ASSUMPTIONS AND AN ANALYTICAL PROCEDURE

This section lists some assumptions that are considered during the complexity and experimental results analysis. The analytical procedure of the experimental results is also described here.

A. Assumptions During the Analysis

During the analysis of the above XML update primitives, the cost of any update primitive counts the number of work-units done by the underlying system in order to update every PACD's component. So, each of the following operation is counted as a *single* work-unit:

- Operations on the NodeSet container:
  - ◊ Insert new record/row
  - ◊ Delete a record/row
  - ◊ Change one (or more) attributes/fields within the record/row
- Operations on a matrix-based component (e.g., childOf):
  - ◊ Insert a complete row or column
  - ◊ Delete an entire row or column
  - ◊ Change an entry of a matrix (i.e., change the status from '0' to '1' or vice versa)

As for illustration, inserting a leaf-node requires the insertion of a new record inside the NodeSet container (1 unit), the addition of one row and column to the childOf, descOf and nextOf matrices (6 units), and may change at most one entry in the nextOf matrix (1 unit). So the leaf-node insertion process costs 8 work-units (or hits).

In addition, the analyses provided in this paper were done based on the following assumptions:

- When a row or column is inserted into a matrix, its entries are set to zero by default with no extra cost.
- The cost of 'search' operations (locating the records) inside the PACD storage components; for example, fetching the node ID among the NodeSet container, is set to zero assuming that a very efficient lookup algorithm is used.
- The number of children at any arbitrary node in the XML tree is 'α', where α is a small number comparing to the number of nodes 'n' for very large XML databases
- The number of descendants at any arbitrary node in the XML tree can be estimated by multiplying the number of nodes 'n' by a fraction 'f', where 0≤f≤1. The value of 'f' decreases exponentially as the context node goes from the root (where f=1) towards the leaf nodes (where f=0) [31].
- The given algorithms and their analyses are based on using the uncompressed PACD storage. Updating compressed PACD storage (which discussion is outside the scope of this paper) may involve additional steps and extra cost depending on the compression technique used.

Generally speaking, the above assumptions were made in order to simplify the analyses provided in the subsequent sections (Section V, VI and VII). The same assumptions also applied during the experimental result discussion in Section IX.

B. An Anlytical Procedure

During the discussion of each update primitive in the following sections, the usage of the primitive (including the function prototype), its pseudo-code, the complexity discussion, and one or two examples will be provided in separate subsections. Furthermore, all examples are based on the XML tree illustrated in Figure 3.

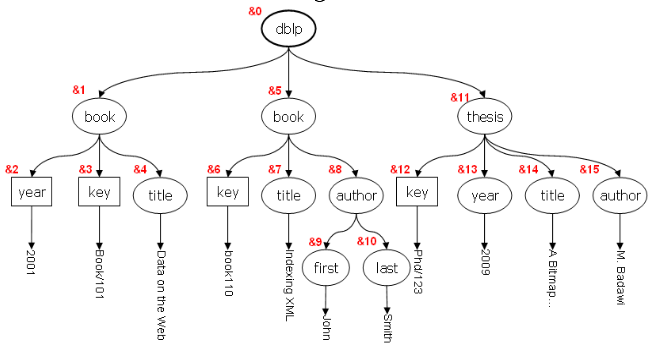


Figure 3. An XML Tree Example

V. INSERTION PRIMITIVES

This section discusses the three insertion primitives shown in Table I.

A. Leaf Node Insertion

1) Usage:

Syntax:	insertLeaf(node_info, parentID [,precID])
Description:	Inserts a node at the bottom-most level of the tree under

	the *parentID* node and next to *precID* node. Both the *parentID* and *precID* are identified by the UQH
Argument(s):	◊ node_info: all necessary information to fill the NodeSet record including the nodeID, tag/attribute name, node_type, and the value/textual content ◊ parentID: the ID of the parent node where the new node to be inserted ◊ precID: the ID of the preceding node. Must be specified in case of the order-preserving storage

2) *Algorithm:*

```
1   PROGRAM insertLeaf(node_info: nodeType, parentID:
    nodeIDType, precID: nodeIDType)
2     Get the next nodeID;
3     Insert the node information into NodeSet;
4     *-- update the childOf matrix:
5     Add a row and column to the ‘childOf’;
6     Set:
7          childOf[nodeID,parenID] = ‘1’,
8     *--update the descOf matrix:
9     Add a row and column to the ‘descOf’;
10    Let: anceSet = {node(i), where descOf[parentID,i]
    = ‘1’} ∪ parentID;
11    For each i ∊ anceSet:
12         Set: descOf[nodeID,i] = ‘1’;
13    *--update the nextOf matrix:
14    Add a row and column to the ‘nextOf’;
15    If precID ≠ null:
16         Let: temp = node(i), where nextOf[i,precID] =
    ‘1’;
17         Set: nextOf[nodeID,precID] = ‘1’;
18         If temp ≠ null:
19              Set: nextOf[temp,precID] = ‘1’;
20  PROGRAM_END.
```

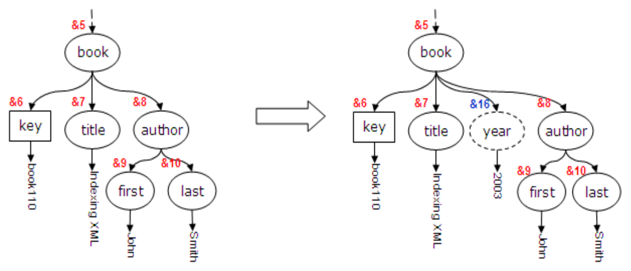
3) *Complexity Analysis:*

Based on the assumption given above, inserting the node’s information into the NodeSet container requires one hit (line 3), whereas updating the childOf matrix requires three hits: two to add a row and column (line 5) and one to set the child/parent relationship between the new node and the parentID (line 7). Similarly, updating the descOf matrix requires 2+h hits: two to add a row and column (line 9) and a maximum of ‘h’ hits (*where ‘h’ is the maximum height of the XML tree*) to set the descendant/ancestor relationship between the new node and its ancestor list, which is calculated in Line 10 (see Lines 11-12). In terms of the nextOf matrix, besides the two hits that are required to insert a row and column to the matrix (line 14), the program makes two additional hits to update the previous/next relationship (lines 17 and 19). So the total work-units required to insert a leaf node in an XML tree of height ‘h’ is 10+h. This is a very small number ‘c’ comparing to the number of nodes ‘n’; thus the complexity is of order O(c).

4) *Example:* Using the database in Figure 3, insert the ‘year’ information (e.g., 2003) to the book identified by the key ‘book/110’, where the ‘year’ information must precede the ‘author’ information (result given in Figure 4).

The cost breakdown is:

NodeSet	childOf	descOf	nextOf	**Total**
1	3	4	4	**12 hits**



(a) before insertion (b) after insertion of &16

Figure 4. A Leaf Node Insertion Example

B. *Non-Leaf Node Insertion*

1) *Usage:*

Syntax:	insertNonLeaf(node_info, parentID [,precID])
Description:	Inserts a node at any level of the tree except the lowest level. The *parentID* and the *precID* are identified by the UQH prior calling the primitive. At this stage, this primitive is only used to add additional level between a parent and the complete set of its children. Subdividing the parentID’s children between the existing parent and the new node is left to further investigation.
Argument(s):	◊ node_info: all necessary information to fill the NodeSet record including the nodeID, tag/attribute name, node_type, and the value/textual content ◊ parentID: the ID of the parent node where the new node to be inserted ◊ precID: the ID of the preceding node. Must be specified in case of the order-preserving storage

2) *Algorithm:*

```
1   PROGRAM insertNonLeaf(node_info:nodeType,
    parentID:nodeIDType,precID: nodeIDType)
2    Get the next nodeID;
3    Insert the node information into NodeSet;
4    *-- update the childOf matrix:
5    Add a row and column to the ‘childOf’;
6    Let: childSet = {node(i), where chilOf[i,parentID]
    = ‘1’}
7    For each i ∊ childSet:
8         Set: chilOf[i,nodeID] = ‘1’;
9    Set: childOf[nodeID,parentID] = ‘1’;
10   *--update the descOf matrix:
11   Add a row and column to the ‘descOf’;
12   Let: anceSet = {node(i), where descOf[parentID,i]
    = ‘1’} ∪ parentID;
13   Let: descSet = {node(j), where descOf[j,parentID]
    = ‘1’};
14   For each i ∊ anceSet:
15        Set: descOf[nodeID,i] = ‘1’;
16   For each j ∊ descSet:
17        Set: descOf[j,nodeID] = ‘1’;
18   *--update the nextOf matrix:
19   Add a row and column to the ‘nextOf’;
20   If precID ≠ null:
21        Let: temp = {node(i), where nextOf[i,precID] =
    ‘1’};
22        Set: nextOf[nodeID,precID] = ‘1’;
23        If temp ≠ null:
24             Set: nextOf[temp,precID] = ‘1’;
25  PROGRAM_END.
```

3) *Complexity Analysis:*

This primitive also requires one hit to insert inside the NodeSet container (line 3). However, more work is required to update the childOf matrix because the children of the parental node 'parentID' have to be assigned to the new node. So the number of hits required to update the childOf matrix is '1+α', where 'α' is the number of children of the context node at an arbitrary level in the XML tree.

To update the descOf matrix, the primitive has to assign the ancestors of the 'parentID' to the new node 'nodeID' (lines 14-15) and the descendants of the 'parentID' as descendant from the new node (lines 16-17). The first process requires no more than 'h' hits, while the cost of the second process may extend to 'n' hits; but in reality it only requires a factor of 'n' hits depending on the insertion level (see Section IV). Finally, the cost of updating the nextOf matrix is the same for updating the nextOf matrix in the previous primitive (lines 22 and 24).

4) *Example:* Using the database in Figure 3, assign the current author of the book titled 'Indexing XML' to be the first author of the book so that other authors can be added later. This requires adding a parent node called 'au_det' for the 'first' and 'last' nodes under the original 'author' node (result given in Figure 5).

The cost breakdown is:

NodeSet	childOf	descOf	nextOf	**Total**
1	5	6	0	**12 hits**

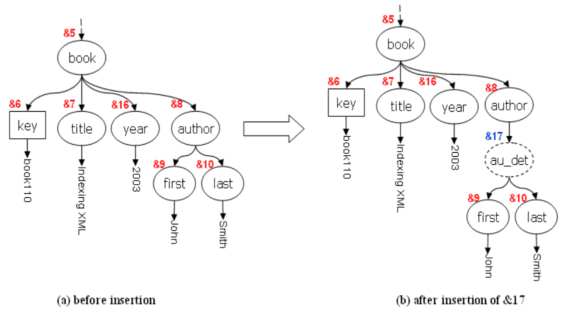


Figure 5. An Non-leaf Node Insertion Example

C. *Twig Insertion*

1) *Usage:*

Syntax:	insertTwig(twig_info, parentID [,precID])
Description:	Inserts a sub-tree of 'm' nodes under the *parentID* and after the *precID*. Both the parentID and the precID are determined by the UQH, and the twig is only inserted at bottom-most nodes
Argument(s):	◊ twig_info: all necessary information to fill the NodeSet record including the nodeID, tag/attribute names, node types, and the value/textual contents ◊ parentID: the ID of the parent node where the new twig to be inserted ◊ precID: the ID of the preceding node. Must be specified in case of the order-preserving storage

2) *Algorithm:*

The twig insertion can be modeled as inserting multiple-connected nodes. In other words, inserting a twig of 'm' nodes requires 'm' times the cost of inserting a single leaf-node and can be performed by the same algorithm in Section V(C) starting at the twig root node.

3) *Complexity Analysis:*

The cost of this primitive is 'm' times the cost of inserting a single leaf-node, where 'm' is the number of nodes inside the inserted twig.

4) *Example:* Using the database in Figure 3, add second author information (i.e., including the 'first' and 'last' name) to the book titled 'Indexing XML' (result given in Figure 6)

The cost breakdown is:

NodeSet	childOf	descOf	nextOf	**Total**
3	9	14	8	**34 hits**

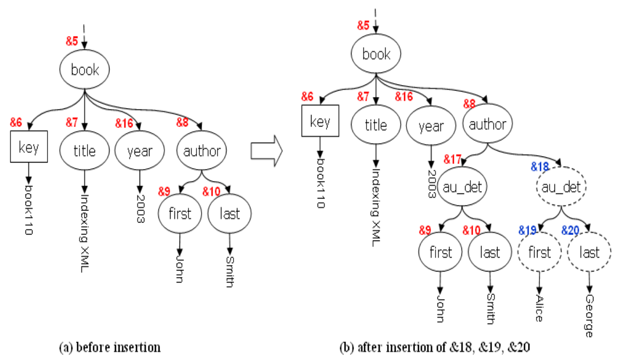


Figure 6. A Twig Insertion Example

D. *Insertion Primitives Summary*

Table II summarizes the number of work-units required to conduct the insertion primitives.

VI. Delete Primitives

PACD currently supports the 'deleteLeaf' and 'deleteTwig' primitives. These are discussed below.

A. *Leaf Node Deletion*

1) *Usage:*

Syntax:	deleteLeaf(nodeID)
Description:	Deletes a node from the lowest level of the tree labeled with *nodeID* that is returned by the UQH
Argument(s):	◊ nodeID: the unique node ID of the deleted node

2) *Algorithm:*

```
PROGRAM deleteLeaf(nodeID: nodeIDType)
  *-- update the childOf matrix:
  Locates the corresponding row and column of the nodeID inside the 'childOf';
  Remove the row and column from the 'childOf';
  *--update the descOf matrix:
  Locates the corresponding row and column of the nodeID inside the 'descOf';
  Remove the row and column from the 'descOf';
  *--update the nextOf matrix:
  Let:
     next = {node(i), where nextOf[i,nodeID] = '1'};
     prev = {node(j), where nextOf[nodeID,j] = '1'};
  Locates the corresponding row and column of the nodeID inside the 'nextOf';
  Remove the row and column from the 'nextOf';
  If next ≠ null AND prev ≠ null:
```

```
15       Set: nextOf[next,prev] = '1';
16    *--update the NodeSet container:
17    Locate the corresponding record of the nodeID
   inside the 'NodeSet';
18    Delete the nodeID;
19 PROGRAM_END.
```

3) *Complexity Analysis:*

Deleting a leaf node is simple and straightforward. In the childOf and descOf matrices, after locating the row and column IDs of the target node, the update process simply removes that row and column. Thus, the process involves two work units for each matrix. Regarding the deletion from the nextOf matrix, a special consideration is required when the target node has previous (line 11) and next (line 10) siblings. In this case, an extra hit is required to assign the *next* node of the target node to be the *next* node of the *previous* node of the target node. Finally, to remove the node from the NodeSet container, the system performs one work unit after locating the record of the target node (line 15). So the 'deleteLeaf' primitive does not do more than *eight* work units to remove a node from the PACD's storage.

4) *Example:* Using the database in Figure 3, remove the author's last-name from the book identified by the key 'book/110' (result given in Figure 7).

The cost breakdown is: (Note: the node ID &10 will be recycled)

childOf	descOf	nextOf	NodeSet	**Total**
2	2	2	1	**7 hits**

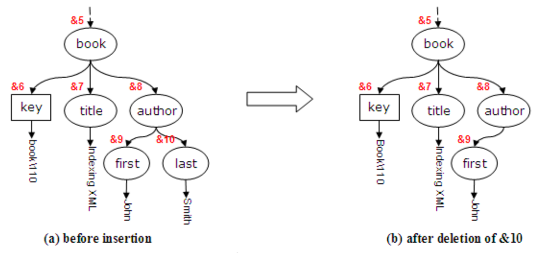


Figure 7. A Leaf Node Deletion Example

B. *Twig Deletion*

1) *Usage:*

Syntax:	deleteTwig(twigRootNodeID)
Description:	Deletes a connected sub-tree rooted at 'twigNRootNodeID' from the XML tree. The twigRootNodeID is returned by the UQH process
Argument(s):	◊ twigRootNodeID: the node ID of twig's root node

2) *Algorithm:*

```
1  PROGRAM deleteTwig(twigRootNodeID: nodeIDType)
2    *-- reconnect the next_of list of the nextOf
   matrix:
3    Let:
4       next = {node(i), where nextOf[i,twigRootNodeID]
   = '1'};
5       prev = {node(j), where nextOf[twigRootNodeID,j]
   = '1'};
6    If next ≠ null AND prev ≠ null:
7       Set: nextOf[next,prev] = '1';
8    *--identify all the node inside the deleted twig:
9    Let: descSet = {node(i), where descOf[i,
   twigRootNodeID] = '1'} ∪ twigRootNodeID;
10   *--remove row and columns from all matrices, and
   the node_info from the NodeSet :
11   For each i ∈ descSet:
12      Locates the corresponding row and column of the
   nodeID inside the 'childOf';
13      Remove the row and column from the 'childOf';
14      Locates the corresponding row and column of the
   nodeID inside the 'descOf';
15      Remove the row and column from the 'descOf';
16      Locates the corresponding row and column of the
   nodeID inside the 'nextOf';
17      Remove the row and column from the 'nextOf';
18      Locate the corresponding record of the nodeID
   inside the 'NodeSet';
19      Delete the nodeID;
20 PROGRAM_END.
```

3) *Complexity Analysis:*

Deleting a twig of 'm' nodes is very similar to deleting a leaf-node except that the cost is multiplied by 'm'. Furthermore, deleting a twig will involve only one reconnection process over the previous/next relationship. This process is performed to rearrange the previous/next relationship of the *previous* and the *next* nodes of the *root* node of the target twig (lines 3-7). So the maximum cost of the 'deleteTwig' primitive is '1+[m×(2+2+2+1)]' work units, where 'm' is the number of nodes inside the deleted twig.

4) *Example:* Using the database in Figure 3, remove the complete author's information from the book identified by the key 'book/110' (result given in Figure 8). Note: this will remove the nodes '&8' and '&9'.

The cost breakdown is:

childOf	descOf	nextOf	NodeSet	**Total**
4	4	4	2	**14 hits**

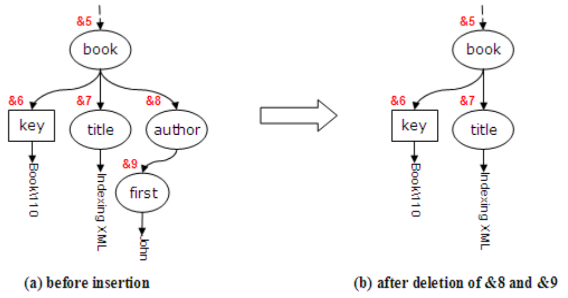


Figure 8. A Twig Deletion Example

C. *Deletion Primitives Summary*

Table III summarizes the number of work-units required to conduct the deletion primitives.

VII. CHANGE PRIMITIVES

Table I introduced four change primitives that can be used to rename node description (i.e., element and attribute names), change the value of a node, and move a node or a twig from one place to another inside the XML tree.

A. Node Description Change

1) Usage:

Syntax:	changeName(nodeID|oldName,newName)
Description:	Renames a node (identified by the *nodeID*) or a set of nodes (that have the same name identified by *oldName*) to the new name *newName*
Argument(s):	◊ nodeID: the node ID of a particular node ◊ oldName: the element or attribute name of a set of nodes ◊ newName: the new name to be assigned to the changed nodes

2) Algorithm:

```
1  *-----Case1: changing particular node's name:
2  PROGRAM changeName(nodeID: nodeIDType, newName:
   string)
3    *-- update the NodeSet container:
4    Locates the corresponding record of the nodeID in
   the 'NodeSet';
5    Replace the 'name' attribute by the 'newName';
6  PROGRAM_END;
7
8  *-----Case2: changing a set of  nodes' name:
9  PROGRAM changeName(oldName: string, newName: string)
10   *-- update the NodeSet container:
11   Let: updateSet = {node(i), where NodeSet.Name =
   oldName};
12   For each node ∈ updateSet:
13       Replace the 'name' attribute by the 'newName';
14 PROGRAM_END;
```

3) Complexity Analysis:

PACD separates the XML textual content representation from the structural content representation and manage them in a different storage component. The former (which includes the node ID, tag/attribute name, type and value) are arranged in the NodeSet container that stores the node information in a separate record. This arrangement makes it easier for the *textual-based* change operations such as the 'changeName' to alter node's record regardless the complexity of the XML's hierarchal structure. So, to change the name of a particular node, it will be sufficient to allocate that node in the NodeSet container and change the 'Name' attribute (lines 4-5). In the case of changing multiple node names such as changing an attribute or element name, the whole nodes labelled with that name class have to be changed. So, the 'changeName' primitive initially identifies all the nodes that share the same name (line 11) and then alters the 'Name' attribute of all identified nodes (lines 12-13). The complexity of this process depends on the distribution of the tag/attribute name in the XML tree, which might be estimated or obtained from the XML schema.

4) Example1: Using the database in Figure 3, change the name of the node 'thesis' to be 'phdthesis'.

This query changes the tag name of the node &11 from 'thesis' to 'phdthesis' with the cost of one work-unit.

B. Node Value Change

1) Usage:

Syntax:	changeValue(nodeID|oldName,newValue)
Description:	change the textual contents of a node (identified by the *nodeID*) or a set of nodes (that have the same name identified by *oldName*) to the new value *newValue*
Argument(s):	◊ nodeID: the node ID of a particular node ◊ oldName: the element or attribute name of a set of nodes ◊ newValue: the new textual content to be assigned to the nodes

2) Algorithm:

The algorithm of this primitive is identical to the one in Section VII-A(2).

3) Complexity Analysis:

The cost of this primitive is similar to the 'changeName' primitive, see Section VII-A(3).

4) Example: Using the database shown in Figure 3, change the publication year for the book labelled with 'Book/101' to be '2000' instead of '2001'.

This query changes the value of the node &2 from '2001' to '2000' with the cost of one work-unit.

5) Example: Using the database in Figure 3, change the 'title' of all publications to the uppercase.

In this query, the 'oldName' parameter is 'title' and the 'newValue' parameter is a function that converts its argument to the uppercase. The query will perform three work units in total.

C. Single Node Shifting

In the context of XML tree, single node shifting is only meaningful when the node is a leaf node. This can be used to transfer information from one block to another, for example to swap the first and second books' ID as in Figure 3. The NodeSet information is not affected by this primitive.

1) Usage:

Syntax:	shiftNode(nodeID,newParentID[,leftID])
Description:	Moves the node labeled with *nodeID* to under the node *newParentID*. If the exact location is required, the preceding node at the new location (i.e., *'leftID'*) must be specified
Argument(s):	◊ nodeID: the node to be moved ◊ newParentID: the parent node at the new location ◊ leftID: the preceding node at the new location

2) Algorithm:

```
1  PROGRAM shiftNode(nodeID: nodeIDType, newParentID:
   nodeIDType, leftID: nodeIDType)
2    *-- update the childOf matrix:
3    Let: oldParentID = {node(i), where
   childOf[nodeID,i] = '1'};
4    Set:
5        childOf[nodeID,newParentID] = '1';
6        childOf[nodeID,oldParentID] = '0';
7    *--update the descOf matrix:
8    Let:
9        oldAnceSet = {node(i), where descOf[nodeID,i]
   = '1'};
10       newAnceSet = {node(j), where
   descOf[newParentID,j] = '1'} ∪ newParentID;
11   For each node i ∈ newAnceSet:
12       Set: descOf[nodeID,i] = '1';
13   For each node i ∈ oldAnceSet:
14       Set: descOf[nodeID,i] = '0';
15   *--update the nextOf matrix:
16   Let:
17       next_of_nodeID = {node(i), where
```

```
   nextOf[i,nodeID] = '1'};
18       prev_of_nodeID = {node(j), where
   nextOf[nodeID,j] = '1'};
19       next_of_leftID = {node(i), where
   nextOf[i,leftID] = '1'};
20       prev_of_leftID = {node(j), where
   nextOf[leftID,j] = '1'};
21   Set (if any combination is not null):
22       nextOf[next_of_nodeID,prev_of_nodeID] = '1';
23       nextOf[nodeID,prev_of_nodeID] = '0';
24       nextOf[nodeID,leftID] = '1';
25       nextOf[next_of_leftID,nodeID] = '1';
26       nextOf[leftID,prev_of_leftID] = '0';
27       nextOf[next_of_leftID,leftID] = '0';
28 PROGRAM_END.
```

3) Complexity Analysis:

Moving a leaf node from one place to another releases the child/parent relationship between the node and its original parent and creates a new child/parent relationship between the node and the new parent. This requires two hits (lines 5 and 6). Similarly, in the descOf matrix, the shifting process releases the descendant/ancestor relationship between the node and its original ancestor list, and creates a new set of descendant/ancestor relationships between the node and the ancestors of the new parent. This requires no more than '2×h' hits, where 'h' is the maximum height of the XML tree (lines 11-14).

Updating the previous/next relationship for the 'shiftNode' is a bit complicated but it requires no more than six hits to release the old previous/next relationships and to set up the new ones (lines 22-27).

4) Example: Using the database in Figure 3, move the publication year of book 'book/101' to be the publication year for the book 'Book/110' (see Figure 9).

The cost breakdown is:

childOf	descOf	nextOf	**Total**
2	4	4	**10 hits**

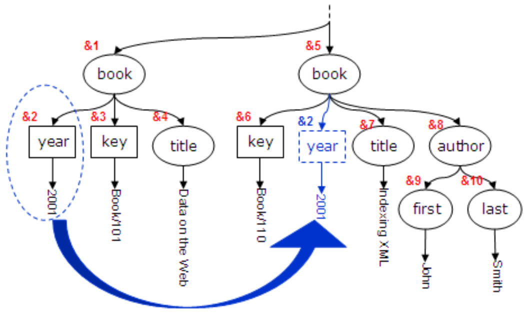


Figure 9. A Leaf Node Shifting Example

D. Twig Shifting

Twig shifting operations are useful when a sub-tree is moved from one parent to another without deleting the sub-tree and creating it again under the new parent.

1) Usage:

Syntax:	shiftTwig(twigRootID,newParentID[,leftID])
Description:	Moves a sub-tree (twig) rooted at the *twigRootID* to be a sub-tree under the node *newParentID*. If the exact location is required, the preceding node at the new location (i.e., *leftID'*) must be specified
Argument(s):	◊ twigRootID: the root of the twig to be moved ◊ newParentID: the parent node at the new location ◊ leftID: the preceding node of twig's root node at the new location

2) Algorithm:

```
1  PROGRAM shiftTwig(twigRootID: nodeIDType,
   newParentID: nodeIDType, leftID: nodeIDType)
2    *-- update the childOf matrix:
3    Let: oldParentID = {node(i), where
   childOf[twigRootID,i] = '1'};
4    Set:
5        childOf[twigRootID,newParentID] = '1';
6        childOf[twigRootID,oldParentID] = '0';
7    *--update the descOf matrix:
8    Let:
9        twigNodeSet = {node(1..m), where node(i) ∊
   twig};
10       oldAnceSet = {node(i), where
   descOf[twigRootID,i] = '1'};
11       newAnceSet = {node(j), where
   descOf[newParentID,j] = '1'} ∪ newParentID;
12   For each node i ∊ newAnceSet:
13       For each node j ∊ twigNodeSet:
14          Set: descOf[j,i] = '1';
15   For each node i ∊ oldAnceSet:
16       For each node j ∊ twigNodeSet:
17          Set: descOf[j,i] = '0';
18   *--update the nextOf matrix:
19   Let:
20       next_of_ twigRootID = {node(i), where
   nextOf[i, twigRootID] = '1'};
21       prev_of_ twigRootID = {node(j), where
   nextOf[twigRootID,j] = '1'};
22       next_of_leftID = {node(i), where
   nextOf[i,leftID] = '1'};
23       prev_of_leftID = {node(j), where
   nextOf[leftID,j] = '1'};
24   Set (if any combination is not null):
25       nextOf[next_of_twigRootID,prev_of_twigRootID]
   = '1';
26       nextOf[twigRootID,prev_of_twigRootID] = '0';
27       nextOf[twigRootID,leftID] = '1';
28       nextOf[next_of_leftID, twigRootID] = '1';
29       nextOf[leftID,prev_of_leftID] = '0';
30       nextOf[next_of_leftID,leftID] = '0';
31 PROGRAM_END.
```

3) Complexity Analysis:

Similar to the 'nodeShift' primitive, the 'twigShift' primitive makes two amendments to the structure of the childOf matrix: one to release the child/parent relationship between the twig's old parent and its root, and another to set up the child/parent relationship between the twig's new parent and its root (lines 5-6). When updating the descOf matrix, the cost is multiplied by 'm' during the 'twigShift' operation because the primitive has to deal with 'm' nodes rather than a single node as in 'nodeShift' primitive (lines 12-17). The cost of updating the nextOf matrix is same for both the 'nodeShift' and 'twigShift' primitives (lines 24-30).

TABLE II: COST SUMMARY OF THE INSERTION PRIMITIVES

Operation	**Growth in**		**# of Work-Units (Hits)**				
	NodeSet	***Matrix***	***NodeSet***	***childOf***	***descOf***	***nextOf***	***Max. Complexity***
insertLeaf	1 rec. more	1 row more 1 col more	1	2+1	2+h	2+2	O(c)
insertNonLeaf	1 rec. more	1 row more 1 col more	1	2+α	2+f×n	2+2	O(f×n)
insertTwig (m nodes)	m rec. more	m row more m col more	m	m.(2+1)	m.(2+h)	m.(2+2)	O(m.c)

n= total number of nodes in the XML tree
h= the maximum height of the XML tree (# of levels)
α= the maximum breadth-degree (i.e., number of children) of any XML node
f= a number between 0 and 1, where 'f×n' is the number of descendants at an arbitrary node
c= is very small number comparing to 'n' such that, for large XML databases, $\lim_{n\to\infty}\frac{c}{n}=0$

TABLE III: COST SUMMARY OF THE DELETION PRIMITIVES

Operation	**Growth in**		**# of Work-Units (Hits)**				
	NodeSet	***Matrix***	***NodeSet***	***childOf***	***descOf***	***nextOf***	***Max. Complexity***
deleteLeaf	1 rec. less	1 row less 1 col less	1	2	2	2+1	O(c)
deleteTwig (m nodes)	m rec. less	m rows less m cols less	m	m×2	m×2	m×(2+1)	O(m.c)

n= total number of nodes in the XML tree
c= is very small number comparing to 'n' such that , for large XML databases, $\lim_{n\to\infty}\frac{c}{n}=0$

TABLE IV: COST SUMMERY OF THE CHANGE PRIMITIVES

Operation	**Growth in**		**# of Work-Units (Hits)**				
	NodeSet	***Matrix***	***NodeSet***	***childOf***	***descOf***	***nextOf***	***Max. Complexity***
chnageName	none	none	1 or k	0	0	0	O(k)
changeValue	none	none	1 or k	0	0	0	O(k)
nodeShift	none	none	0	2	2×h	6	O(c+2.h)
twigShift (m nodes)	none	none	0	m×2	m×2×h	6	O(c+m×2×[h+1])

n= total number of nodes in the XML tree
k= the number of nodes per tag/attribute name (usually much smaller than 'n')
h= the height of the XML tree (# of levels)
c= is very small number comparing to 'n' such that , for large XML databases, $\lim_{n\to\infty}\frac{c}{n}=0$

4) *Example:* Using the database in Figure 3, move the author information of book 'book/110' to be the author for the book 'Book/101' (see Figure 10).

The cost breakdown is:

childOf	descOf	nextOf	**Total**
2	12	2	**16 hits**

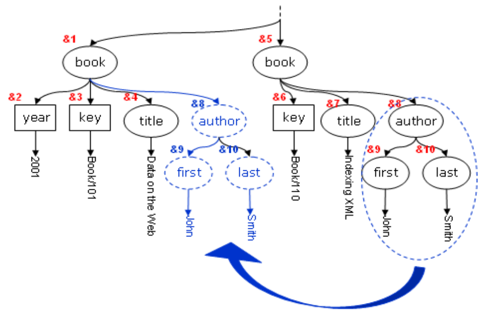


Figure 10. A Twig Shift Example

E. *Change Primitives Summary*

Table IV summarizes the number of work-units required to conduct each change primitives.

VIII. OVERALL COMPLEXITY DISCUSSION

The analysis provided in Sections V, VI and VII shows that the cost of all update-primitives over the PACD's uncompressed data representation locates in acceptable limits in general. Of the update primitives discussed, the highest update complexity is only a fraction of the number of nodes (i.e., 'n') and this only happens during the *rarely-used* operation 'insertNonLeaf'. The cost of other update operations ranges between a very small constant 'c' and 'm×c' in the case of manipulating a *twig* of size 'm' nodes.

From the technical point of view, the bitmapped XML structure (see Section II) and the introduction of the *previous/next* axes has played a major role in such cost reduction. Unlike node-labeling based techniques [32][33][34][8][12][13], the use of the nextOf matrix (to encode the document order) has narrowed the spread of label changes to consider only the adjacent nodes of the targeted node. Also encoding the basic XML structures (i.e., the child/parent and descendant/ancestor relationships) using the bitmapped *node-pairs* (i.e., the childOf and descOf matrices) has reduced the high cost and complexity that result from using: (1) *path-summaries* [35][36][37][16][17][18], and (2) *sequences* [20][19][22] to encode such structures. The analysis has shown that the number of changes in the childOf structure is bounded by a small constant 'c' (where 'c' is a very small number comparing to 'n', the total number of nodes in the XML tree) in most cases except the 'insertNonLeaf' primitive, which requires 'α' number of hits depending on the node's *breadth degree*. On the other hand, the same primitive may perform up to 'n' hits over the descOf matrix, however in real situations that number is fractioned by small number 'f', which ranges between '0' and '1' (usually $0 \leq f \leq ½$ for real, well designed XML databases).

Another source of cost reduction in the PACD's update transactions is the separation between the textual content representation and the XML hierarchal structure representation. The *content-based* primitives only affect the NodeSet container while the *structure-based* update primitives affect the bitmapped matrices. This is not valid in the case of path-summary and sequence-based techniques, where the underlying path-summary or sequence has to be changed. In general, the number of hits over the NodeSet container is limited by the number of targeted nodes except when amending a tag/attribute name or a node value for a set of nodes that share the same tag/attribute name. In this case, the cost is limited by the number of nodes that share the same tag/attribute name, which is also considered small comparing to the entire XML tree.

IX. A Comparative Study

To evaluate and support the analysis provided above, this section presents an experimental study conducted to compare the PACD's performance (in terms of the update handling) against two representative mapping techniques. The section initially provides the experiment setup, including the list of the used update queries, the structure of the compared techniques, and the underlying test databases. Then it presents the experimental results and their discussion for each query in a separate section counting the number of work-units done by the query over the test databases. Finally, the section lists out the main finding from the experiment.

A. The Experiment Setup

A comparative experiment between the performance PACD technique and two representative XML techniques from the literature is conducted to support the above complexity analyses. The experiment executes 6 update queries –as a representation of the above update primitives- translated over 3 XML databases for the 3 selected XML techniques. The 6 update queries are listed in Table V while the characteristics of the 3 XML databases are given in Table VI. Table VII shows the XML/RDBMS mapping schema of the three compared techniques, PACD, XParent [36] and Edge [38], while other specifications of these techniques can be found in [29], [36] and [38], respectively.

The experiment (see the result summary in Table VIII) counts the number of changes (hits) done over the technique data storage (2nd column of Table VIII), and lists them per query ID in separate columns over each XML database. The number of hits, over all components, is summed up in the last 3 rows of Table VIII.

Finally, the experiment was conducted using a stand-alone Intel Pentium-IV machine with 3.6GHz dual processor and 1GB of RAM. The machine was operated by MS Windows XP SP3, and the translation of all XML queries to the corresponding RDBMS queries was executed using MS FoxPro database engine. Furthermore, data indices were used whenever applicable over the three techniques to leverage their performance with the power of RDBMS. Such HW/SW setup was counted to have no influence on the generated results.

TABLE V. The Experiment's Update Queries

Query ID	Query Description
U1	Insert an Atomic Value, i.e., leave node
U2	Insert a Non-atomic Value, i.e., non-leave or internal node
U3	Delete an Atomic Value , i.e., leave node
U4	Delete a Non-atomic Value , i.e., non-leave or internal node
U5	Change an Atomic Value, i.e., the textual content of a node
U6	Change a Non-atomic Value, i.e., tag-name

TABLE VI. Features of The used XML Databases

	DBLP [39]	XMark [40]	Treebank [41]
Size (#of nodes)	2,439,294	2,437,669	2,437,667
Depth(#of levels)	6	10	36
Min Breadth[†]	2	2	2
Max Breadth	222,381	34,041	56,385
Avg Breadth[†]	11	6	3
#of Elements	2,176,587	1,927,185	2,437,666
#of Attributes	262,707	510,484	1

TABLE VII. The Experimental Comparable XML Techniques

Technique	Components (XML/RDBMS Mapping Schema)

PACD	XMLNodes(nodeID, type, tagID)
	XMLSym(tagID, desc)
	XMLValues(nodeID, value)
	nextOf(nextID, prevID)
	childOf(childID, parented)
	descOf(descID, anceID)
	** childOF+descOf= OIMatrix(Source, Target, relType)*

Edge	Edge(source, target, ordinal, label, flag, value)

XParent	labelPath(pathID, length ,PathDesc)
	element(pathID, ordinal, nodeID)
	data(pathID, ordinal, nodeID, value)
	dataPath(nodeID, parented)
	ancestors(nodeID, anceID, level)

Query: U1_DBLP

Insert a new author called "New Author" of the *inproceedings* publication identified by the key "conf/ecai/BeeringerAHMW94". The new author must be the first in the author list of that publication.

Query: U1_XMark

Insert a new location called "New Location" for the item identified by "item38683" from South America.

Query: U1_TreeBank

Insert a new 'CC' element (textual-value is "New CC") under the 'NP' element which has a child called 'CC' that stores "IQXQwyLNRrdOEoHUpfWNbB==" and the existing 'CC' element can be reached by the path '/FILE/EMPTY/S/S/VP/NP/VP/PP/NP/CC'. The new 'CC' element must precede the existing 'CC' element.

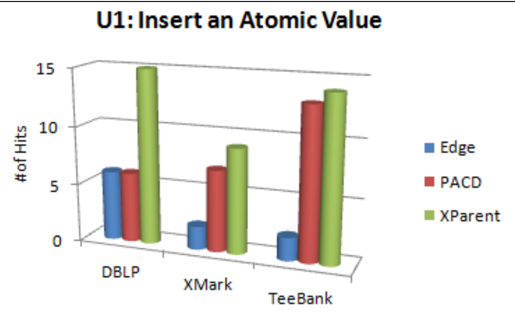


Figure 11. Performance of the "Insert an Atomic Value" Query

Query: U2_DBLP

Insert a new root element called "nRoot" to the DBLP100 database.

Query: U2_XMark

Insert a new root element called "nRoot" to the XMark100 database.

Query: U2_TreeBank

Insert a new root element called "nRoot" to the TREE100 database.

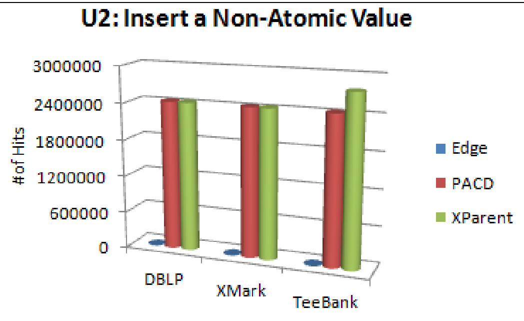


Figure 12. Performance of "Insert a Non-Atomic Value" Query

B. *Result Discussion*

This section discusses the experimental results. Each subsection discusses the results of a particular XML update query including the syntax of the executed query, a graphical representation of the results, and a brief analysis of each technique performance. The final remarks about these analyses are given in Section IX.C.

1) *Inserting an Atomic Value*

The three queries in Figure 11 insert a new leaf-node at levels 3, 5 and 10 of the DBLP, XMark and TreeBank databases, respectively. The queries were designed to act at a distance of 30% from the root-node.

The graph shows that PACD required six, seven and thirteen amendments to the underlying relational schema in order to execute this query. The number of amendments is mainly controlled by the level number where the insertion was applied. For example, the 6 operations required by PACD over DBLP are distributed as follows: 1 insertion to the 'XMLNodes' table and another insertion to the 'XMLValues' table because the node contains a textual value. Three operations were also required to update the 'OIMatrix' table while the last operation was required to link the new node to its next node at the 'nextOf' table. As discussed earlier, PACD requires at most two operations to update the 'nextOf' table for any leaf-node insertion, and at most 'h' to update the 'OIMatrix' table for the same process, where 'h' is the maximum number of levels in the XML tree.

Query: U3_DBLP

Delete the author named "Antje Beeringer" from the inproceedings record that is identified by the key "conf/ecai/BeeringerAHMW94".

Query: U3_XMark

Delete the location for the item identified by "item38683" from South America. The deleted element is storing the content "United States".

Query: U3_TreeBank

Delete the 'CC' element (textual-value is "IQXQwyLNRrdOEoHUpfWNbB==") which can be reached by the path '/FILE/EMPTY/S/S/VP/NP/VP/PP/NP/CC'.

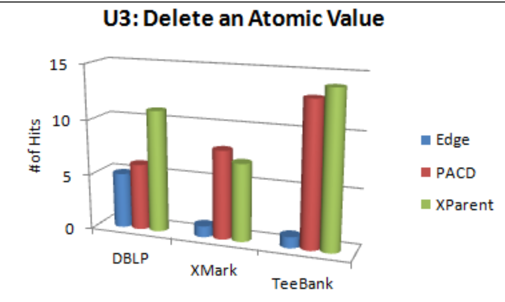


Figure 13. Performance of "Delete an Atomic Value" Query

Query: U4_DBLP

Delete the entire record of an article labeled with the KEY "tr/gte/TM-0332-11-90-165".

Query: U4_XMark

Delete the entire record of an item labelled with the ID "item7" from Africa.

Query: U4_TreeBank

Delete the entire record of the element 'PP' that is reachable by the path '/FILE/EMPTY/S/VP/S/VP/NP/VP/NP/PP' and its child element 'TO' has the value "6fc25UxSwWg9Pz+yyR6wi8==".

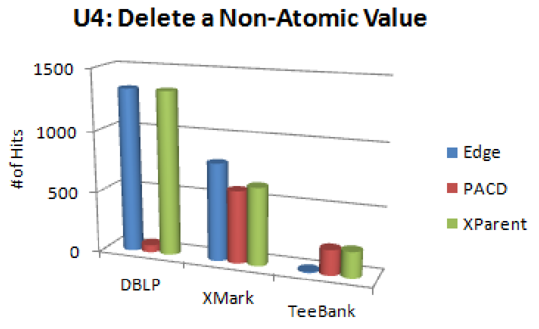


Figure 14. Performance of "Delete a Non-Atomic Value" Query

Compared to other techniques, PACD is always better than XParent because the later required one more insertion to the 'data' table and more *change* operations to keep the document order updated at the 'ordinal' attributes of the 'elem' and 'data' tables. On the other hand, Edge performance is either same as PACD or better. This superiority is determined by the fact that Edge encodes are far less of XML structure, which in turn affects its query performance. In summary, when linked with the queries-range coverage, PACD appears to have the best update performance for this type of query among the three techniques.

2) Inserting Non-Atomic Value

These queries (Figure 12) insert virtual root-nodes to the three XML databases, respectively. The virtual new root insertion is used here for three reasons. Firstly, the operation was chosen to represent the process of inserting non-leaf nodes, which may occur at any level in the XML tree. Secondly, inserting at the top most level can give a logical comparison between the number of operations required by the operation rather than inserting at lower locations in different XML databases. Finally, it will be easier to observe the XML updater behavior and judge its performance with relation to the database size and other XML features.

For this particular query, PACD required '2+n' amendments to the underlying relational schema, where 'n' is the number of nodes in the XML tree. The 'n' operations were required to insert the parent/child and descendant/ancestor relationships of the new node, whereas the other two operations were used to insert the

Query: U5_DBLP

Change the book title identified by the key "phd/Mumick91" from "Query Optimization in Deductive and Relational Databases" to be "Query Optimization in Deductive and Relational Databases: Modified".

Query: U5_XMark

Change the street name of a person's address by the key "person0" from "85 Geniet St" to be "85 Geniet St: Modified".

Query: U5_TreeBank

Change the value stored in 'IN' element which is reachable by the path "/FILE/S/VP/SBAR/S/NP/SBAR/S/VP/S/VP/VP/NP/PP/NP/PP/NP/VP/PP/IN" from "CrtsNRQX7cNqOsWbpvPMgO==" to "CrtsNRQX7cNqOsWbpvPMgO==:Modified".

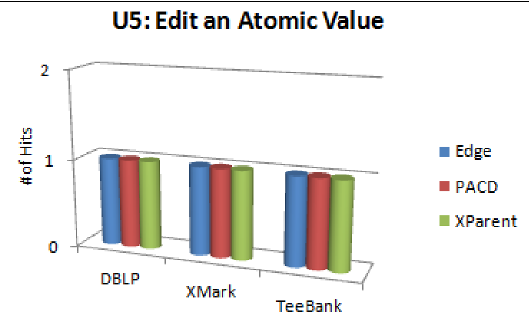


Figure 15. Performance of "Edit an Atomic Value" Query

Query: U6_DBLP

Change the description of the element/attribute name 'key' records to 'ID_NUM'.

Query: U6_XMark

Change the description of the element/attribute name 'id' records to 'ID_NUM'.

Query: U6_TreeBank

Change the description of the element/attribute name 'PP' records to 'PPPP'.

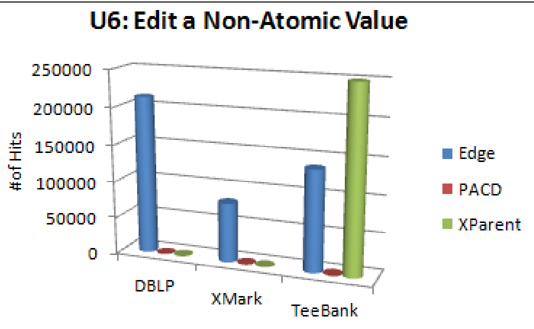


Figure 16. Performance of "Edit a Non-Atomic Value" Query

corresponding record in the 'XMLNodes' and 'XMLSym' tables because the new node was assumed to have distinct tag name from the existing tag/attribute list.

Due to its restricted XML mapping algorithm, Edge had only one amendment to execute this update operation. This amendment was required to insert the new node's record into the underlying mapping schema without affecting the 'ordinal' attribute because the root node logically has no siblings. XParent workload on the other hand was slightly higher than PACD. XParent required extra "s" operations to update the 'labelPath' table where "s" is the number of records in that table, which stores the corresponding XML schema summary. In general, XParent's number of operations exceeds PACD ones by the number of the records affected inside the underlying XPath summary, and also the relative position of the inserted node amongst its siblings.

3) *Delete an Atomic Value*

These queries (Figure 13) delete a single leaf node from each XML database. The deleted nodes were located at levels 3, 5 and 10 of the DBLP, XMark and TreeBank databases, respectively; and they were 30% away from the root node each database. In addition, the deleted nodes were chosen to have at least one sibling node of the same tag-name so that the impact of the deletion process on the document order can be calculated.

PACD performed '2+p+2' amendments to the underlying mapping schema of all XML database types where 'p' is the level number of the node deleted. The first 2 operations were required to remove the corresponding records from the

TABLE VIII. THE EXPERIMENTAL RESULTS

Tech. Name	Query / Tables	DBLP						XMark						Treebank					
		U1	U2	U3	U4	U5	U6	U1	U2	U3	U4	U5	U6	U1	U2	U3	U4	U5	U6
Edge	edge	6	1	5	78815	1	213634	2	1	1	792	1	80316	2	1	1	9	1	136545
PACD	XMLSym	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	0	0	1
	XMLNodes	1	1	1	13	0	0	1	1	1	48	0	0	1	1	1	8	0	0
	XMLValues	1	0	1	12	1	0	1	0	1	24	1	0	1	0	1	4	1	0
	OIMatrix	2	n	2	26	0	0	4	n	4	514	0	0	9	n	9	186	0	0
	nextOf	2	0	2	11	0	0	1	0	2	2	0	0	2	0	2	5	0	0
XParent	elem	6	1	4	78815	0	0	2	1	1	48	0	0	2	1	2	9	0	0
	data	6	0	4	12	1	0	2	0	1	24	1	0	2	0	2	4	1	0
	labelpath	0	146	0	0	0	8	0	252	0	0	0	9	0	338750	0	0	0	248480
	dataPath	1	1	1	13	0	0	1	1	1	48	0	0	1	1	1	9	0	0
	ancestor	2	n	2	26	0	0	4	n	4	514	0	0	9	n	9	186	0	0
Edge	Total	6	1	5	78815	1	213634	2	1	1	792	1	80316	2	1	1	9	1	136545
PACD	Total	6	n+2	6	62	1	1	7	n+2	8	588	1	1	13	n+2	13	203	1	1
XParent	Total	15	n+148	11	78866	1	8	9	n+254	7	634	1	9	14	n+338752	14	208	1	248480

n=2437669, is the number of nodes inside each XML database and it should be unified for all databases

'XMLNodes' and 'XMLValues', respectively, and last 2 operations were required to update the 'nextOf' table while the 'p' operations were conducted on the 'OIMatrix' table to remove the node's corresponding records. In general, the number of operations for this type of update is determined by the level number of the target node in the XML tree with a maximum cost of '2+h+2' where 'h' is the maximum number of levels in the XML tree.

The performance of Edge and XParent in update was close to PACD's. In XParent, the update handler requires '2×rs' more operations (where 'rs' is the number of the right-hand side siblings of the node) to update the document-order inside the 'elem' and 'data' tables, while Edge's processor required '1+2×rs' operations to remove the node from the list and amend the siblings' ordinal attribute.

In summary, PACD appears more efficient for this type of queries due the document order preserving mechanism.

4) *Delete a Non-Atomic Value*

The action of these queries (Figure 14) was conducted at levels two, five and ten of the DBLP, XMark and TreeBank databases, respectively. These queries were included to test the performance of deleting a sub-tree from the master XML tree. The number of nodes in the target sub-trees was selected to be very small compared to the master XML tree so that identifying the number of records affected became easy. The DBLP's sub-tree consisted of 13 nodes distributed over 2 levels, and the XMark's sub-tree had 48 nodes distributed over 7 levels while the number of nodes and levels in the TreeBank's sub-tree were 8 and 4, respectively. All sub-tree nodes were combinations of atomic and non-atomic nodes.

PACD required 13 and 12 operations to remove the DBLP's sub-tree from the nodes and values lists, respectively, 26 operations to update the parent/child and descendant/ancestor relationships, and 11 operations to update the 'nextOf' container. These figures were determined by three factors. Firstly, the sub-tree size determined the number of 'delete' operations from both the 'XMLNodes' and 'XMLValues' tables. Secondly, the breadth and the depth as well as the level of the sub-tree's root node all controlled the number of update operations of the 'OIMatrix' table. Finally, the number of update operations at the 'nextOf' table was mainly controlled by the breadth of the sub-tree including, at most, 2 operations to re-link the left and right hand side nodes for the previous/next relationship. In general, PACD generates a manageable number of changes for this type of queries especially when the update happens at the low levels of the XML tree.

On the other hand, Edge and XParent performed 1270 times more operations compared to PACD for the DBLP's query, and the three techniques were close to each other for the XMark's query while PACD and XParent were 22 times higher than Edge for the TreeBank's query. These figures support the above conclusion that the number of operations is determined by the size of the deleted sub-tree and its location in the master XML tree. In general, PACD's document-order encoding mechanism had a clear impact in reducing the number of changes that are required to conduct sub-tree deletion operations.

5) *Edit an Atomic Value*

This is the cheapest update query (Figure 15) that can be ever conducted by any technique tested. All techniques over all database types have made exactly one amendment to their relational schema storage. In this case, PACD needs to update the 'XMLValues' table, Edge also updates the corresponding record in its orphan table while XParent needs to change the record inside the 'data' table.

6) *Edit a Non-Atomic Value*

These queries (Figure 16) can be used to alter the tags and attributes names without affecting the document's hierarchal structure. The experiment has chosen to alter the name of some elements/attributes, which were widely repeated in each XML database to show the importance of minimizing the cost of such update queries. The DBLP's query was designed to change all 'KEY' attributes, and the XMark's query was designed to change all 'ID' attributes, while the TreeBank's query was deigned to change the name of the recursive element 'PP'. The 'KEY', 'ID' and 'PP' tokens were repeated 213¸634, 80¸316 and 136¸545 times, respectively inside the corresponding XML databases.

In general, the statistics show that the number of amendments conducted by PACD was always 1 because PACD stores all database tokens only once. On the other hand, the number of amendments in Edge's table was determined by the number of elements/attributes that hold the same name, while the number of amendments in XParent environment was determined by the number of XPath expressions that contain the element/attribute name. So, for Edge, the number of changes was 213¸634, 80¸316 and 136¸545 over the DBLP, XMark and TreeBank databases respectively, while XParent performed 8, 9 and 248¸480 changes over the same set of XML databases. The high number of changes produced by XParent over the TreeBank database was due the recursive properties of the element 'PP' inside the XML schema.

C. *Main Findings*

The experiment discussed here has evaluated the PACD's update primitives by executing six XML update queries over three different XML databases. The evaluation process examined the performance of PACD over each XML database and compared it with Edge's and XParent's performance over the same database set.

Comparing to other techniques, and taking into account the queries-range coverage, PACD appeared having the best performance for most of the queries in all situations. The experiment has also shown that the performance of XParent and Edge was delayed by the cost of the document order persevering mechanism. PACD eliminates this cost by encoding the previous/next relationship that requires at most 2 changes for any type of query/operation that concerns about document-order.

X. Conclusion

This paper has discussed the PACD's updating framework, which is managed by a set of low cost update primitives. Once an update query is issued, the Update Query Handler (UQH) process identifies the target node-set and the necessary update primitive(s). The translation of an update query may generate one or more update primitives each of which may alter one or more XML nodes. The UQH currently can generate nine update primitives divided into three categories; the insert, delete, and change primitives.

This paper has provided a comprehensive complexity analysis of the PACD's update primitives supported by illustrative examples for each update primitive. The paper also presented an experimental evaluation process to support the analysis and generalize conclusions based on the generated results.

Both analysis and experimental results provided in this paper have shown that the computation cost of the XML updates can be improved using the PACD's update primitives, which specifically act on its data-storage. The summary of the complexity discussion is given in Tables II, III and IV, while the full experimental result summary is depicted in Table VIII.

Besides, the paper has supplied a full algorithmic listing of the XML update primitives under the PACD environment, along with a comprehensive evaluation method (and the results), which can be recycled by the XML research community to test and evaluate the XML database developments. Such level of details is rarely found in the existing literature.

References

[1] M. Al-Badawi and A. Al-Hamadani, "A Complexity analysis of an XML update framework," in Proceedings of ICIW 2013, Rome, Italy, 2013, pp. 106-113, ISSN: 2308-3972, ISBN: 978-1-61208-280-6.

[2] T. Härder, M. Haustein, C. Mathis, and M. Wagner, "Node labelling schemes for dynamic XML documents reconsidered," International Journal of Data Knowledge Engineering, vol. 60, issue 1, 2007, pp. 126-149.

[3] P. O'Neil, E. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury, "ORD-PATHs: insert-friendly XML node labels," In proceeding of ACM/SIGMOD international conference on Management of Data, 2004, pp. 903-908.

[4] W. Shui, F. Lam, D. Fisher, and R. Wong, "Querying and marinating ordered XML data using relational databases," in Proceedings of the 16th Australasian database conference - vol. 39, Newcastle, Australia, 2005, pp. 85-94.

[5] I. Tatarrinov, S. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang, "Storing and querying ordered XML using a relational database system," ACM/SIGMOD Record, Madison, Wisconsin, 2002, pp. 204-215.

[6] H. Wang, H. He, J. Yang, P. Yu, and J. Yu, "Dual labeling: Answering graph reachability queries in constant time," in Proceedings of the International conference of Data Engineering, 2006, pp. 75-86.

[7] C. Zhang, J. Nsughton, D. DeWitt, Q. Luo, and G. Lohman, "On supporting containent queries in relational database management systems," in Proceedings of the 2001 ACM SIGMOD international conference on Management of Data, California, USA, 2001, pp. 425-436.

[8] J. K. Min, J. Lee, and C. W. Chung, "An efficient XML encoding and labeling method for query processing and updating on dynamic XML data," Advance in Databases: Concepts, Systems and Applications, LNCS, vol. 4443, 2009, pp. 715-726.

[9] S. Sakr, "A prime number labeling scheme for dynamic ordered XML trees," in Proceedings of the Intelligent Data Engineering nd Automated Learning, LNCS, vol. 5326, 2008, pp. 378-386.

[10] J. Lu, X. Meng, and T. W. Ling, "Indexing and querying XML using extended Dewey labeling scheme," Journal of Data & Knowledge Engineering, vol. 70, issue 1, 2011, pp. 35-59.

[11] L. Xu, T. Wang Ling, and H. Wu, "Labeling dynamic XML documents: an order-centric approach," IEEE Transactions on Knowledge and Data Engineering, vol. 24, issue 1, 2012, pp. 100-113.

[12] J. Liu, Z. M. Ma, and L. Yan, "Efficient labeling scheme for dynamic XM trees," Information Sciences, vol. 221, 2013, pp. 338-354.

[13] R. Lin, Y. Chang, and K. Chao, "A compact and efficient labeling scheme for XML documents," Database Systems for Advanced Applications, LNCS, vol. 7825, 2013, pp. 269-283.

[14] Q. Chen, A. Lim, and K. Ong, "D(K)-Index: An adaptive structural summary for graph-structured data," in Proceedings of the 2003 ACM SIGMOD international conference on Management of data, CA, USA, 2003, pp. 134-144.

[15] C. Chung, J. Min, and K. Shim, "APEX: An adaptive path index for XML data," in Proceedings of the 2002 ACM SIGMOD international conference on Management of data, Madison, Wisconsin, 2002, pp. 121-132.

[16] S. Haw and C. Lee, "Extending path summary and region encoding for efficient structural query processing in native XML databases," Journal of Systems and Software, vol. 82, issue 6, 2009, pp. 1025-1035.

[17] A. Arion, A. Bonifati, I. Manolescu, and A. Pugliese, "Path summaries and path partioning in modern XML databases," World Wide Web, vol. 11, issue 1, 2008, pp. 117-151.

[18] M. Sadoghi, I. Burcea, and H. A. Jacobsen "A gneric boolean predicated XPath expression matcher," in Proceedings of the 14th Int. Conf. on Extending Database Technology, 2011, pp. 45-56.

[19] J. Kwon, P. Rao, B. Moon, and S. Lee, "Fast XML document filtering by sequencing twig patterns," ACM Transactions on Internet Technology (TOIT), vol. 9, issue 4, Article 13, 2009, pp. 13.1-13.51.

[20] H. Wang and X. Meng, "On sequencing of tree structures for XML indexing," in Proceedings of the 21st international conference on Data Engineering, 2005, pp. 372-383.

[21] H. Wang, X. Wang, and W. Zeng, "A research on automaticity optimization of KeyX index in native XML database," in Proceedings of the 2008 international conference on Computer Science and Software Engineering, 2008, pp. 700-703.

[22] W. Li, J. Jang, G. Sun, and S. Yue "A new Sequence-Based approach for XML data query," in Proceedings of the 2013 Chinese Intelligent Automation Conf., LNEE, vol. 256, 2013, pp. 661-670.

[23] H. Al-Jmimi,A. Barradah, and S. Mohammed "Sibiling labeling scheme for updating XML dynamically," in Proceedings of the 4th Int. Conf. on Computer Engineering and Technology, vol. 40, 2012, pp. 21-25.

[24] J. Yoon, S. Kim, G. Kim, and V. Chakilam, "Bitmap-based indexing for multi-dimensional multimedia XML document," in Proceedings of the 5th International Conference on Asian Digital Libraries-ICADL2002, Singapore, 2002, pp. 165-176.

[25] N. Zhang, M. Özsu, I. Ilyas, and A. Aboulnaga, "FIX: feature-based indexing technique for XML documents," in Proceedings of the 22nd international conference on VLDB, vol. 32, Seoul, Korea, 2006, pp. 259-270.

[26] R. Senthilkumar and A. Kannan, "Query and update support for indexing and compressed XML (QUICX)," Recent Trends in wireless and Mobile Networks Communication in computer and Information Science, vol. 162, 2011, pp. 414-428.

[27] J. Clark and S. DeRose, XML Path Language (XPath)-Version 1.0, [Online] Available online: http://www.w3.org/TR/xpath/, [Accessed on: 25/05/2014].

[28] M. Al-Badawi, H. Ramadhan, S. North, and B. Eaglestone, "A performance evaluation of a new bitmap-based XML processing approach over RDBMS," Int. J. of Web Engineering and Technology, vol. 7, no. 2 , 2012, pp. 143 – 172.

[29] M. Al-Badawi, B. Eaglestone, and S. North, "PACD: A bitmap-based approach for processing XML data," WebIST'09, Lisbon, Portugal, 2009, pp. 66-71.

[30] H. He, H. Wang, J. Yang, and P. Yu, "Compact reachability labeling for graph-structured data," in Proceedings of the 14th ACM international conference on Information and knowledge management, Bremen, Germany, 2005, pp. 594-601.

[31] T. Bray, J. Paoli, C. Sperbeg-McQueen, E. Maler, and F. Yergeau, Extensible Markup Language (XML) 1.0 (Fourth Edition), [Online] Available online: http://www.w3.org/TR/REC-xml/, [Last accessed on: 25/05/2014].

[32] J. Yun and C. Chung, "Dynamic interval-based labelling scheme for efficient XML query and update processing," Journal of Systems and Software, vol. 81, issue 1, 2008, pp. 56-70.

[33] J. Lu, T. Ling, C. Chan, and T. Chen, "From region encoding to extended dewey: on efficient processing of XML twig pattern matching," in Proceedings of the 31st International Conference on VLDB, Trondheim, Norway, 2005, pp. 193-204.

[34] X. Wu, M. Lee, and W. Hsu, "A prime number labeling scheme for dynamic ordered XML trees," in Proceedings of the 20th international conference on Data Engineering, 2004, pp. 66-78.

[35] R. Goldman and J. Widom, "DataGuides: enabling query formulation and optimaization in semistructured database," in Proceedings of the 23rd international conference on VLDB, 1997, pp. 436-445.

[36] H. Jiang, H. Lu, W. Wang, and J. Yu, "XParent: an efficient RDBMS-based XML database system," International conference on Data Engineering, CA, USA, 2002, p. 2.

[37] 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/IT., vol. 1, issue 1, NY, USA, 2001, pp. 110-141.

[38] D. Florescu and D. Kossmann "A Performance Evaluation of alternative Mapping Schemas for Storing XML Data in a Relational Database," TR:3680, May 1999, INRIA, Rocquencourt, France, pp. 1-24.

[39] DBLP, The DBLP Computer Science Biblography, [Online] Available at http://dblp.uni-trier.de/, [Last accessed on 24/05/2014].

[40] A. Schmidt, F. Waas, M. Kersten, D. Carey, I. Manolescu, and R. Busse. "XMark: a benchmark for XML data management," International conference on Very Large Data Bases, Hong Kong, China, 2002, pp. 974-985.

[41] PennProj, The Penn Treebank Project, [Online] Available online at http://www.cis.upenn.edu/~treebank/, [Last accessed on: 25/05/2014].