A Complexity Analysis of an XML Update Framework

Mohammed Al-Badawi and Abdallah Al-Hamadani
The Department of Computer Science
Sultan Qaboos University
Muscat, Oman
mbadawi@squ.edu.om, abd@squ.edu.om

***Abstract*—XML update is problematic for many XML database techniques. The main issue tackled by these techniques is the cost reduction of updating the XML's hierarchal structure inside the underlying storage. PACD technique, introduced earlier, is an attempt in this direction. This paper mainly provides a complexity analysis of the PACD's updates primitives. The analysis, along with the comparative experimental results presented here, have shown that the cost of eight update primitives (out of nine discussed) leys under acceptable range of a constant 'c' where 'c' is an extremely small number comparing to the number of nodes 'n' in the underlying database. Such good performance is lacked in the compared techniques.**

Keywords-XML Databases;XML Update; Mapping

I. INTRODUCTION

Data stored in the extensible markup language (XML) containers (database) is subject to updates when circumstances change. Unfortunately, handling XML updates is a common problem in the existing XML storage models and optimization techniques. Relational approaches using node labeling techniques [6][12][13][14][16][22] 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 those approaches which use path summaries to encode the XML hierarchical structure, e.g., [3][4][7], an additional cost results from updating these summaries. In native XML approaches such as sequence based [10][15][17] and feature based techniques [19][23], the update problem is even worse. In the first case, the consequences of a single update operation, for example deleting a node, can affect hundreds or even thousands locations in the corresponding sequence depending on the node's 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 into what is called feature-based matrices [19].

PACD, an acronym for Parent-Ancestor-Child-Descendant, as an XML processing technique introduced in [2], brings the cost of updating the XML's hierarchal structure to the data representation level by encoding these structures into a set of structure-based matrices, which allow direct access to the information of the nodes affected by such update operations. This paper mainly introduces the PACD's Updates Query Handler (UQH) and provides a complexity analysis of its update primitives. The paper starts by revisiting the PACD's framework in Section II; then, it introduces the UQH framework in Section III. Section IV discusses the complexity of the different update primitives while Sections V and VI, respectively, summarize the complexity discussion and provides a supportive comparative experimental result. The paper is concluded in Section VII.

II. PACD'S XML PROCESSING MODEL

PACD, introduced in [1][2], is a bitmap XML processing technique consisting of two main components: the Index Builder (IB) and the Query Processor (QP). The IB (see Figure 1) shreds the XML's hierarchal structure (derived by the XPath's thirteen axes and their extension; the Next and Previous axes [1]) into a set of binary relations each of which is physically stored as an n×n bitmap matrix. An entry in any matrix is either '1' if the corresponding relationship is exists between the coupled nodes or '0' otherwise [8][23].

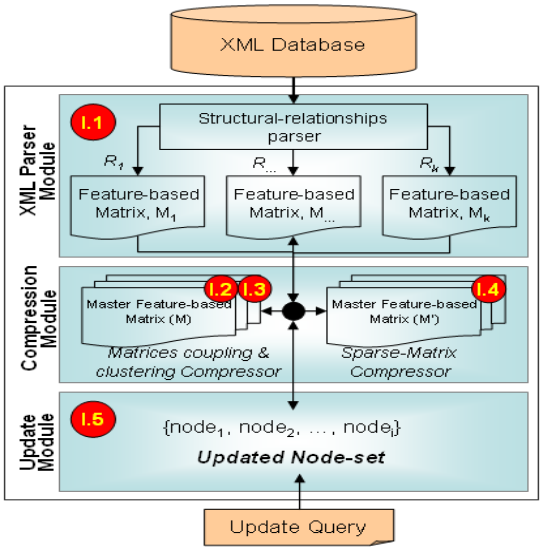


Figure 1. PACD's Index Builder (IB)

The IB component is also responsible to handle the XML updates. Once an update query is issued, the attached UQH determines the nodes affected by the update query and the type of the update operation itself. The following section introduces the PACD's UQH and its update primitives, while the primitives' complexity is discussed in Section IV.

 ISBN: 978-1-61208-280-6

III. THE UPDATE HANDLER

PACD's Update Query Handler (UQH), represented by operation I.5 of Figure 1, is responsible for all update tasks targeted by any ML update operation. These tasks include the translation of the update query, and the determination and the execution of necessary update primitive.

The execution of the update query starts by identifying the node(s) that are affected by the update command/query. The handler navigates through a finite-state-machine (FSM) version of the update-query [2] in order to locate the affected node-set. Once the target node-set is identified, the handler 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-based changes.

The update primitives act on all PACD's components including the NodeSet container [2] and the structure based matrices. Each update primitive has to execute certain instructions over each component such as adding new columns and rows (over the bitmapped matrices). The cost of the update-query is the sum-cost of executing all generated update primitives over the all PACD's components, i.e., matrices and NodeSet container. For example, the 'insert' primitive can involve adding one or more rows and columns to the bitmapped matrices, as well as adding one or more entries to the NodeSet container. So, the cost of the insert operation will be the cost of inserting the node information inside the NodeSet container plus the cost of inserting one row and column inside the 'child', 'desc' and 'next' matrices respectively. The general update algorithm is given in Table II.

TABLE I. XML UPDATE 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	remove a single-rooted, connected sub-tree
Updating	changeName	rename an element or attribute name
	changeValue	edit the value (text) of an attribute (element)
	shiftNode	move a node from one place to another
	shiftTwig	move a single-rooted, connect sub-tree from one place to another

TABLE II. XML UPDATES EXECUTION ALGORITHM

```
INPUT: update-query
OUTPUT: none
Construct the FSM execution plan corresponding to query's twig
node-set = the returned node-set from the FSM execution
Using the update-query syntax, determine the 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;
```

IV. DISCUSSION OF UPDATE PRIMITIVES

This section discusses, through examples, the complexity of the update primitives. The complexity is counted as the number of the 'change' actions performed on the NodeSet, child matrix, descendant (desc) matrix and next matrix. Due to space limitation, the paper only presents sample algorithms for some update primitives and then provides an example on how the algorithm works based on the XML tree in Figure 2.

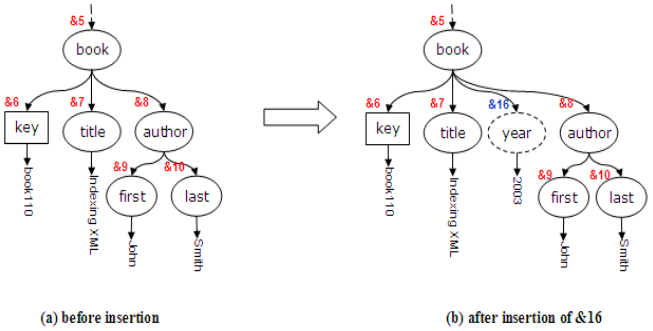


Figure 2. An Example XML Tree (includes an insertion case)

A. *Insert Primitives*

Table I has shown three insertion primitives that can be triggered against XML databases. The prototypes of the methods implementing these primitives are:

insertLeaf(node_info, parentID [,precedingID]);
insertNonLeaf(node_info, parentID [,precedingID]);
insertTwig(twig_info, parentID [,precedingID])

In the above prototypes, the 'node_info' indicates the information of the node(s) to be inserted which includes the nodeID, tag_name and the optional textual contents. The 'parentID' refers to the node ID of the parent node under which the insertion will take place. The 'precedingID' must be specified if the document order [6][16] is to be preserved, and it indicates the node ID of the node that must precede the new node.

1. *Leaf Node Insertion*

This method inserts a node at the bottom-most level of the tree under parentID node and next to precedingID node. Both parentID and precedingID are identified by the UQH during the query translation process.

Example: Add the 'year' information, i.e., 2003, to the book identified by the key 'book/110', where the 'year' information must precede the 'author' information (Figure 2).

The cost breakdown of the above operation is:

NodeSet	child	desc	next	Total
1	3	4	4	12 hits

2. *Non-Leaf Node Insertion*

This method can insert a node at any level of the tree except the bottom-most level. ParentID and precedingID are

identified by the UQH prior calling the primitive. In this paper, the analysis assumes that the primitive is only creating an additional level between a parent and its children, making these children as the children of the inserted node.

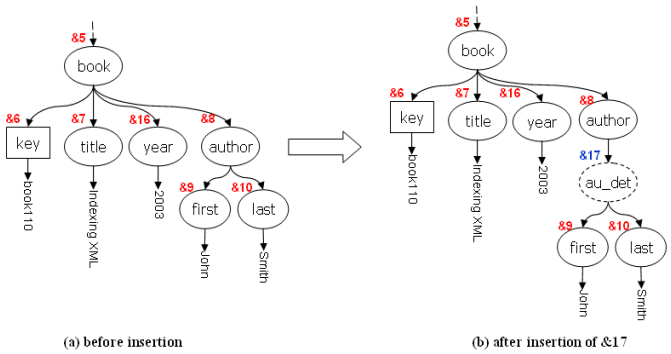
TABLE III. INSERTING NON-LEAF NODE ALGORITHM

```
insertNonLeaf(node_info:nodeType,parentID:nodeIDType,precID:
nodeIDType)
   Get the next nodeID;
   Insert the node information into NodeSet;
   *-- update the child matrix:
   Add a row and column to the 'child';
   Let: childSet = {node(i), where child[i,parentID] = '1'}
   For each i ∈ childSet:
       Set: child[i,nodeID] = '1';
   Set: child[nodeID,parentID] = '1';
   *--update the desc matrix:
   Add a row and column to the 'desc';
   Let: anceSet = {node(i), where desc[parentID,i] = '1'} ∪
       parentID;
   Let: descSet = {node(j), where desc[j,parentID] = '1'};
   For each i ∈ anceSet:
       Set: desc[nodeID,i] = '1';
   For each j ∈ descSet:
       Set: desc[j,nodeID] = '1';
   *--update the next matrix:
   Add a row and column to the 'next';
   If precID ≠ null:
       Let: temp = {node(i), where next[i,precID] = '1'};
       Set: next[nodeID,precID] = '1';
       If temp ≠ null:
           Set: next[temp,precID] = '1';
END.
```

Example: Make the current author of the book titled 'Indexing XML' to be the FIRST author of the book so that other authors can be added. This requires adding a parent node called 'au_det' for the 'first' and 'last' nodes under the original 'author' node (Figure 3).

The cost breakdown of the above operation is:

NodeSet	child	desc	next	Total
1	5	6	0	12 hits



(a) before insertion (b) after insertion of &17

Figure 3. Non-Leaf Node Insertion

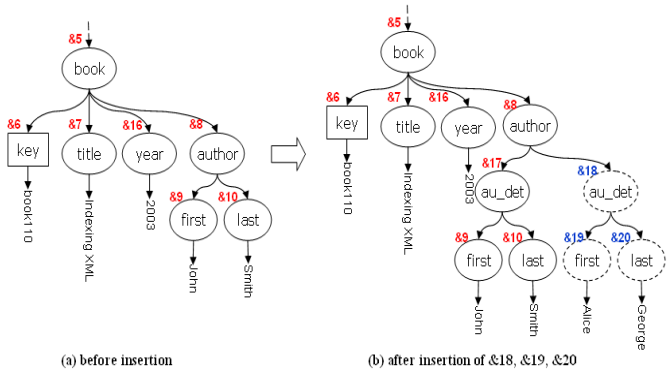
3. *Twig Insertion*

This method 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. 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 Table III starting at the twig's root node.

Example: add a second author sub-tree, i.e., including the 'first' and 'last' name, to the book titled 'Indexing XML' (Figure 4).

The cost breakdown of the above operation is:

NodeSet	child	desc	next	Total
3	9	14	8	34 hits



(a) before insertion (b) after insertion of &18, &19, &20

Figure 4. Twig Insertion

4. *Complexity Analyses*

Due the space limitation the full complexity analyses of the operations is omitted. The following table only summarizes the number of work-units required to conduct the insertion primitives in general.

TABLE IV. COMPLEXITY ANALYSES 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 any node, usually $0 \le f \le ½$
c= is very small number comparing to 'n' in large XML databases such that $\lim_{n\to\infty} \frac{c}{n} = 0$

B. *Deletion Primitives*

Table I has shown two deletion primitives that can be triggered against XML databases. The prototypes of the methods implementing these primitives are:

deleteLeaf(nodeID);

deleteTwig(twigRootNodeID);

In the above prototypes, the 'nodeID' indicates the node ID of the node that to be deleted while the 'twigRootNodeID' indicates the node ID of the root node of the targeted twig. The discussion in this section assumes that deleting a non-leaf node results in a cascade deletion of its children, therefore the 'deleteTwig' operation will be applied in the case of deleting a non-leaf node.

1. *Leaf Node Deletion*

This method deletes a node from the bottom-most level of the tree labeled with nodeID, which is returned by the UQH during the query translation process.

Example: Remove the author's last-name from the book identified by the key 'book/110'.

The cost breakdown of the above operation is:

child	desc	next	NodeSet	Total
2	2	2	1	7 hits

2. *Twig Deletion*

This method deletes a connected sub-tree rooted at 'twigRootNodeID' from the XML tree. The twig root node ID is returned by the UQH during the query translation process.

TABLE V. INSERTING NON-LEAF NODE ALGORITHM

```
deleteTwig(twigRootNodeID: nodeIDType)
  *-- reconnect the next_of list of the nextOf matrix:
  Let:
      next = {node(i), where nextOf[i,twigRootNodeID] = '1'};
      prev = {node(j), where nextOf[twigRootNodeID,j] = '1'};
  If next ≠ null AND prev ≠ null:
      Set: nextOf[next,prev] = '1';
  *--identify all the node inside the deleted twig:
  Let: descSet = {node(i), where descOf[i, twigRootNodeID] =
     '1'} ∪ twigRootNodeID;
  *--remove row and columns from all matrices, and the node_info
  *-- from the NodeSet :
  For each i ∈ descSet:
      Locates the corresponding row and column of the nodeID
          inside the 'childOf';
      Remove the row and column from the 'childOf';
      Locates the corresponding row and column of the nodeID
          inside the 'descOf';
      Remove the row and column from the 'descOf';
      Locates the corresponding row and column of the nodeID
          inside the 'nextOf';
      Remove the row and column from the 'nextOf';
      Locate the corresponding record of the nodeID inside the
          'NodeSet';
      Delete the nodeID;
END.
```

Example: Remove the complete author's information from the book identified by the key 'book/110' (Figure 5). Note: this will remove the nodes '&8' and '&9'.

The cost breakdown of the above operation is:

child	desc	next	NodeSet	Total
4	4	4	2	14 hits

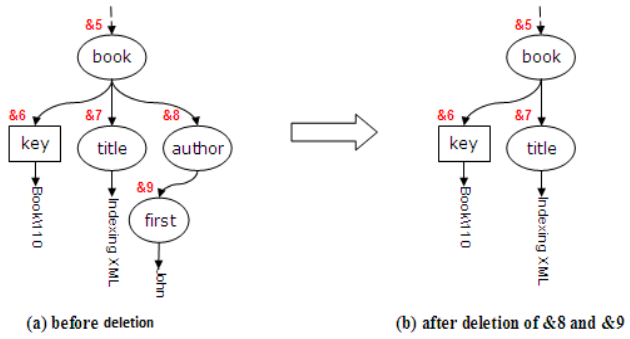


Figure 5. Twig Deletion

3. *Complexity Analyses*

The following table only summarizes the number of work-units required to conduct the deletion primitives in general.

TABLE VI. COMPLEXITY ANALYSES 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' in large XML databases such that $\lim_{n\to\infty} \frac{c}{n} = 0$							

C. *Change Primitives*

Table I has shown four change primitives that can be triggered against XML databases. The prototypes of the methods implementing these primitives are:

changeName([nodeID|oldName],newName);
changeValue([nodeID|oldName],newValue);
shiftNode(nodeID,newParentID[,leftID]);
shiftTwig(twigRootID,newParentID[,leftID]);

In the above prototypes, the 'nodeID' indicates the node ID of the targeted node targeted. The 'oldName' and 'newName' indicate the tag-name of the targeted node. The 'newValue' is the textual content of the node to be altered. The 'parentID' and the 'leftID' are the node ID of the parent node and left node of the targeted node. Finally, the 'twigRootID' is the node ID of the twig to be shifted.

1. *Tag-Name Change*

This method 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.

Example: 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.

Example: Change the name of all nodes labeled with 'key' to be 'pub_id'.

 ISBN: 978-1-61208-280-6

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

2. *Textual-Value Change*

This method changes 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.

Example: Change the publication year for the book labeled 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.

Example: 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.

3. *Single Node Shifting*

This method moves the node labeled with nodeID to be under the node newParentID. If the exact location is required, the preceding node at the new location, i.e., 'leftID', must be specified.

Example: Move the publication year of book 'book/101' to be the publication year for the book 'Book/110' (Figure 6).

The cost breakdown of the above operation is:

childOf	descOf	nextOf	Total
2	4	4	10 hits

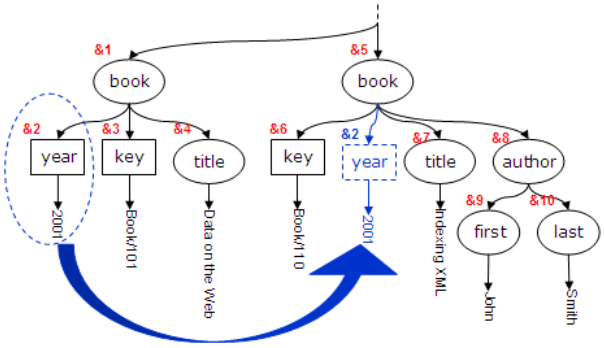


Figure 6. Single Node Shifting (Parent Change)

4. *Twig Shifting*

This method 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.

TABLE VII. INSERTING NON-LEAF NODE ALGORITHM

```
shiftTwig(twigRootID: nodeIDType, newParentID: nodeIDType,
                leftID: nodeIDType)
  *-- update the childOf matrix:
  Let: oldParentID = {node(i), where childOf[twigRootID,i] =
       '1'};
  Set:
      childOf[twigRootID,newParentID] = '1';
      childOf[twigRootID,oldParentID] = '0';
  *--update the descOf matrix:
  Let:
      twigNodeSet = {node(1..m), where node(i) ∈ twig};
      oldAnceSet = {node(i), where descOf[twigRootID,i] = '1'};
      newAnceSet = {node(j), where descOf[newParentID,j] = '1'}
          ∪ newParentID;
  For each node i ∈ newAnceSet:
      For each node j ∈ twigNodeSet:
          Set: descOf[j,i] = '1';
  For each node i ∈ oldAnceSet:
      For each node j ∈ twigNodeSet:
          Set: descOf[j,i] = '0';
  *--update the nextOf matrix:
  Let:
      next_of_ twigRootID = {node(i), where
          nextOf[i, twigRootID] = '1'};
      prev_of_ twigRootID = {node(j), where
          nextOf[twigRootID,j] = '1'};
      next_of_leftID = {node(i), where nextOf[i,leftID] = '1'};
      prev_of_leftID = {node(j), where nextOf[leftID,j] = '1'};
  Set (if any combination is not null):
      nextOf[next_of_twigRootID,prev_of_twigRootID] = '1';
      nextOf[twigRootID,prev_of_twigRootID] = '0';
      nextOf[twigRootID,leftID] = '1';
      nextOf[next_of_leftID, twigRootID] = '1';
      nextOf[leftID,prev_of_leftID] = '0';
      nextOf[next_of_leftID,leftID] = '0';
END.
```

Example: Move the author information of book 'book/110' to be the author for the book 'Book/101' (Figure 7).

The cost breakdown of the above operation is:

child	desc	next	Total
2	12	2	16 hits

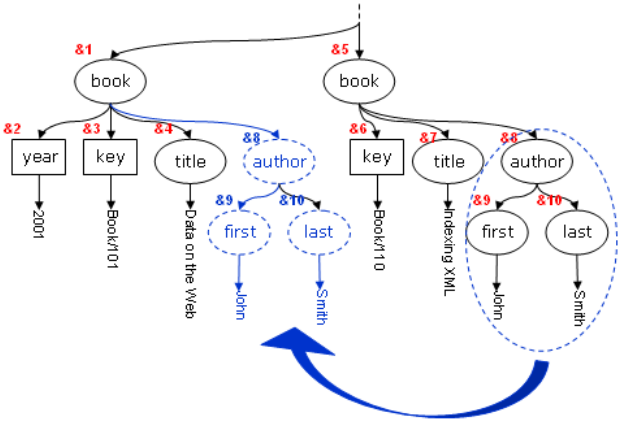


Figure 7. Twig Shifting (Parent Change)

5. *Complexity Analyses*

The following table only summarizes the number of work-units required to conduct the change primitives in general.

TABLE VIII. COMPLEXITY ANALYSES SUMMARY 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' in large XML databases such that $\lim_{n\to\infty} \frac{c}{n} = 0$

V. OVERALL COMPLEXITY

The analysis provided above has shown that the cost of all update-primitives over the PACD's *uncompressed* data representation lies 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 infrequently accessed operation 'insertNonLeaf'. The cost of other update operations ranges between a very small constant 'c' (where 'c' is an extremely small number comparing to the number of nodes 'n' in the database) and 'm×c' in the case of manipulating a *twig* of size 'm' nodes.

From the technical point of view, the bitmapped XML structure and the introduction of the previous/next axes [2] has played a major role in such cost reduction. Unlike node-labeling based techniques, e.g., [11][18][21], the use of the next matrix -to encode the document order- has narrowed the spread of label changes to the adjacent nodes (only) of the targeted node. Also encoding the basic XML structures (the child/parent and descendant/ancestor relationships) using the bitmapped node couplings (the child and desc matrices) has reduced the high cost and complexity that result from using path-summaries [5][9] [7][20] and sequences [10][15] to encode such structures. The analysis has shown that the number of changes in the child structure is bounded by a small constant 'c' (where 'c' is an extremely small number comparing to the number of nodes 'n' in the database) in most cases except the 'insertNonLeaf' primitive which requires 'α' number of hits depending on the node's breadth degree.

Another source of cost reduction in PACD's update transactions is the separation between the textual contents representation and the XML hierarchal structure representation. Because of that, the content-based primitives only affect the NodeSet container while the structure-based update primitives affect the bitmapped matrices. This is not the case of path-summary and sequence-based techniques where the underlying path-summary or sequence has to be changed for either type of updates (content-based or structure-based). 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.

VI. A COMPARATIVE STUDY

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 IX while the characteristics of the 3 XML databases are given Table X. Also Table XI shows the XML/RDBMS mapping schema of the three compared techniques, PACD, XParent and Edge, while other specifications of these techniques can be found at the references [2], [9] and [24] respectively.

The experiment counts the number of changes (hits) done over the technique's underlying representation (2nd column of Table XII) of the XML database, and lists them per query ID in separate columns (see Table XII) over each XML database. The number of hits, over all components, is summed up in the last 3 rows of Table XII which summarizes the overall experimental results.

TABLE IX. THE EXPERIMENTAL 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 X. FEATURES OF THE EXPERIMENTAL XML DATABASES

	DBLP [25]	XMark [27]	Treebank [26]
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

† Figures exclude leaf nodes

†† Dataset also contains two versions of each database at 50% and 25% of the size of the base database. Both the depth and the average breadth of the base databases are maintained in the smaller databases

 ISBN: 978-1-61208-280-6

TABLE XII. 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

B. *Experimental Results Summary*

Comparing to other techniques, 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 which requires at most two amendments for any node update operation.

TABLE XI. THE EXPERIMENTAL COMPARABLE XML TECHNIQUES

Technique	Components (XML/RDBMS Mapping Schema)
PACD	XMLNodes(nodeID, type, tagID) XMLSym(tagID, desc) XMLValues(nodeID, value) childOf(childID, parentID) OIMatrix(Source, Target, relType) descOf(descID, anceID) nextOf(nextID, prevID)
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)

VII. 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.

The analysis and the experimental results in this paper have shown that the computation cost of XML update queries can be improved using the update primitives, which specifically act on the PACD data representation. The cost analysis of all update primitives is provided in Tables IV, VI and VIII.

REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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, I. 1, 2007, pp. 126-149.

[7] 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, I. 6, 2009, pp. 1025-1035.

[8] 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.

 ISBN: 978-1-61208-280-6

[9] 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.

[10] 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, I. 4, Article 13, 2009, pp. 13.1-13.51.

[11] 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.

[12] 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.

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

[14] 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.

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

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

[17] 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.

[18] 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.

[19] 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.

[20] 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, I. 1, NY, USA, 2001, pp. 110-141.

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

[22] C. Zhang, J. Nsughton, D. DeWitt, Q. Luo, and G. Lohman, "On supporting containment 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.

[23] 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.

[24] 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.

[25] DBLP. The DBLP Website, Available at http://dblp.uni-trier.de/, [Last accessed on 28/04/2013].

[26] PennProj. The Penn Treebank Project Website, Available online at http://www.cis.upenn.edu/~treebank/, [Last accessed on: 28/04/2013].

[27] 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.

 ISBN: 978-1-61208-280-6