Handling Updates in Sequence Based XML Query Processing

K.Hima Prasad Ch.Rajesh P.Sreenivasa Kumar

Department of Computer Science and Engineering
Indian Institute of Technology Madras
Chennai-600036
India
email: hima@cs.iitm.ernet.in, rajesh_chalavadi@yahoo.com, psk@cs.iitm.ernet.in

Abstract

With the increasing importance of XML databases, suitable indexing and querying scheme that supports efficient handling of updates has become a major concern. Recently sequence based query processing has gained importance because of holistic processing of queries. Though the current sequence based systems process queries efficiently, they do not handle updates efficiently. In this paper we propose an update-aware sequencing and indexing mechanism that supports updates without having to change the identities of existing nodes each time an update occurs. We propose a sequencing method called *Modified Prüfer Sequence with Gaps* to support updates. The main idea is to use monotonically increasing sequence of real numbers as position numbers in the sequence so that appropriate gaps can be left for accommodating future insertions. We propose algorithms for insertion, deletion, replacing, and renaming of nodes in the new scheme and demonstrate that updates can be performed without any degradation in the query processing performance. We present experimental evaluation of the proposed scheme and show that updates can be processed efficiently.

1 Introduction

XML has emerged as the new standard for information representation and exchange on the web[14]. Indexing and querying of XML data has been a major research issue in the database world. As XML documents evolve or change over the period, supporting updates is a must for the XML storage system.

XML data can be modeled as an ordered node labeled tree. XPATH[1] and XQUERY[13] are proposed for querying XML. The need to support updates are discussed in[7] with respect to language extension to XQUERY. An update in an XML document can change both the structure and content of the document.

Recently sequence based query processing[11, 16, 15] is gaining importance because of its holistic query processing feature. Though the current sequence based indexing systems support insertion of new documents, they do not support the updates that occur within the document. Whenever an update occurs the entire sequence corresponding to the document is to be rebuilt and inserted in the database. This becomes a major overhead when the data is changing frequently.

There have been efforts to support updates in Interval Encoding[3, 12], where gap is provided within the interval for future insertions[10]. In QRS[2] the node identities are made floating point numbers to support more updates. Patrick et.al introduced the concept of ORDPATH [8] for updating nodes identified by DeweyIds. To our knowledge this paper is the first effort to introduce update mechanism in sequence based XML query processing.

In this paper, we propose an update-aware sequencing and indexing mechanism for XML documents. This is an extension to the MPS querying system built using Modified Prüfer Sequences[9]. MPS is chosen because with modification it can support updates without changing the identities of the other nodes each time an update occurs and also it performs better than the earlier sequence based systems. We give a brief description of the querying system and explain more about how the updates are handled. The main contributions of the paper are summarized as follows.

- An update-aware sequencing and indexing mechanism based on a variation of Prüfer sequence
- Algorithms for handling updates
- Experimental results showing significant performance gains on updates

International Conference on Management of Data
COMAD 2005b, Hyderabad, India, December 20–22, 2005
©*Computer Society of India, 2005*

The rest of the paper is organized as follows. Section 2 gives the background and related work. In section 3 we describe update-aware modified prüfer sequencing system with all the details about querying and handling updates. Section 4 gives the experimental results. Section 5 gives the conclusions of the paper.

2 Background and Related Work

Tree pattern queries are basic building blocks of both XPATH and XQUERY. To process tree pattern queries several indexing mechanisms have been proposed.

In interval encoding[3, 12, 10], nodes are assigned an interval such that a child node's interval is contained within the interval of parent node. In this scheme, by providing appropriate gaps one can support updates efficiently[10]. QRS[2] uses floating point numbers for assigning intervals and it has been shown that it can support more number of updates efficiently. The problem with the interval encoding is that it requires multiple structural joins to get the results of a query. ORDPATHs [8] are introduced in Microsoft SQL server which support update of nodes identified by DeweyIds.

Sequence based XML querying systems[16, 11] aim at processing tree pattern queries holistically with out breaking them into root to leaf paths. Though the current systems support insertion of new documents, updates within the document are not handled. Both Vist and Prix use virtual Trie to store the sequences which makes it rigid to support the updates as insertion at some part of the sequence changes the identities of all the elements that follow it. Prix uses post-order numbering to label the nodes. So, when some node gets inserted/deleted the identities of the elements change and thus the entire sequence has to be regenerated. Apart from these both Prix and Vist suffer from duplicate results and both perform post-processing to generate the final result.

We developed a system called MPS[9] using a variation of Prüfer sequences, called *Modified Prüfer Sequence*, which processes tree pattern queries holistically and generates results without any post-processing. In this paper, we modified the existing MPS scheme to allow updates by providing gaps in the sequence for future insertions. We concentrate more on update related issues in this paper. Query processing details of the MPS system can be found in [9].

3 Supporting Updates in MPS

This section gives the details of the XML querying system built using modified Prüfer sequences which handles updates efficiently. Before defining the sequencing mechanism, we give an insight into the type of updates that can occur in an XML document.

3.1 Updates in XML

This section gives the details of the type of updates that occur in XML databases. These are discussed in detail with respect to XQUERY[13] language extension in[7]. An update operation on an XML document can change both structure and content of the document. From the database point of view updates can be broadly classified as follows.

- **Delete :** Delete operation deletes a subtree in the document rooted at node. This can be expressed as
  delete Node.
  Where Node can be an XPATH expression which selects the node or nodes to be deleted. This deletes the entire subtree under the node selected. Delete can also be used to delete some content(i.e PCDATA, attributes) in the XML document.

- **Insert:** Since XML is an ordered tree, the insertions are to be more specific. The type of insertions can be as follows
  insert Node1,Node2
  insertAfter Node1, Node2
  insertBefore Node1, Node2
  Where *node* represents the root of the subtrees being referred to. This *node* can be selected by giving an XPATH expression. If the expression has set of nodes as output then insertions are performed after/before all those nodes. In case of normal *insert Node1, Node2* statement second node is inserted as the last child of the first node. In case of *insertAfter Node1,Node2*, Node2 is inserted as the following sibling of Node1 that immediately follows Node1.*insertBefore Node1, Node2* inserts the Node2 as the preceding sibling Node1 immediately preceding Node1.

- **Replace:** The Replace operation replaces a subtree with some other subtree. The operation can be as follows.
  Replace Node1, Node2

- **Rename:** This operation is used to rename an existing node. This can be expressed as follows.
  Rename Node1,Name

A good sequencing and indexing mechanism should support all the above defined operations efficiently. An insertion can occur at any position in an XML document. Suppose a node has k children then insertions can occur at any of the $k+1$ positions as shown in the Figure 1. We propose a sequencing mechanism that takes care of all these points and provides scope for all possible insertions without changing the identities of other nodes to the maximum possible extent. The proposed sequencing and indexing mechanism is for single document and can be easily extended to multiple documents.

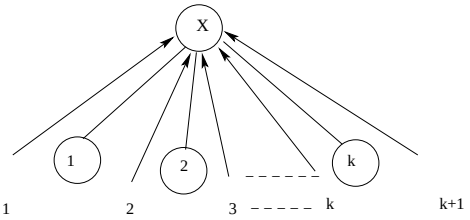


Figure 1: **A node with K children and possible places of insertion**

All the sequence based XML querying systems do a subsequence match of the query sequence on the data sequence and give the positions of the match in the data sequence as output. This positions can be then used to get the required result from the document. So to support updates a gap is introduced in position numbers given to a sequence. To support all possible insertions at each level as discussed above, appropriate gap is to be given before and after each subtree of a node. If a node has k subtrees then $k+1$ gaps are to be provided before/after the sequence corresponding to each subtree. All these details are discussed in the next subsection.

3.2 Update-Aware Sequence Generation

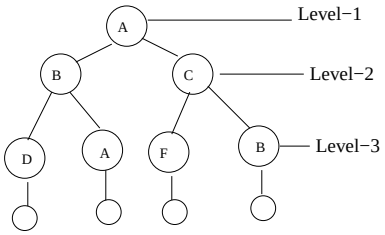
This section gives the details of the update-aware sequence generated using a variation of the Prüfer sequence[4].

A sequence called *Modified Prüfer Sequence* for an XML document is generated in a way similar to the Prüfer sequences. Instead of giving fixed post order numbers to nodes, we define an *elementNumber* for each node with label L in the XML document.
elementNumber: 1+(number of nodes with same label as L that appear before this node in the depth first traversal order(i.e, document order) of the document tree).

The $elementNumber$ distinguishes an element among the elements with same label. The above definition of *elementNumber* is used for generating the initial sequence of a document. However, when the insertions occur newly added nodes may not follow this definition. We make sure that $elementNumber$ of the newly inserted nodes is different from the ones already present in the document.

We define a parameter called G which is used to give gaps in the sequence. At each level a gap of $(maxLevel - level) * G$ is given. The position number is incremented by the gap value every time a gap is provided for insertion.

A *Modified Prüfer sequence* of a document is generated by deleting the nodes in post-order sequence and outputting the label of the parent node with additional information. We assume that for each leaf node in the XML data tree, a dummy child node is attached. Let X be the node being deleted and Y be its parent node



35 : (D,1,3,5), 40 : (B,1,2,15), 60 : (A,2,3,5), 65 : (B,1,2,15), 75 : (A,1,1,60), 110 : (F,1,3,5), 115 : (C,1,2,15), 135 : (B,2,3,5), 140 : (C,1,2,15), 150 : (A,1,1,60).

Figure 2: **The Sample XML Document and corresponding Modified Prüfer Sequence**

in the tree. The tuple generated corresponding to the deletion of X will have the following components.
(position, label, elementNum, level, count)

- *position:* Position number of the tuple in the sequence.
- *label:* Denotes the tag of the parent node Y.
- *elementNum: elementNumber* of Y
- *level:* Level at which Y is present in the XML document.
- *count:* (sum of the counts of all the k instances of X) + $(k+1)$ gaps given for insertions at that level

There is one-to-many correspondence between nodes in the tree and the tuples in the sequence. Note that if a node N with *label* P has r children, then there will be r tuples with the label P corresponding to the node N. The $elementNum$ component of these tuples will all be the same. Whereas if there are m nodes in the XML tree all with *label* P, corresponding to each such node, there will be a set of tuples. The $elementNum$ distinguishes between these sets by having a distinct number for all tuples in one set. Thus, given a *label* and $elementNum$, one can easily identify the node in the XML data tree associated with the tuple. All the tuples corresponding to a single node are said to be consistent with each other(Two tuples T_i and T_j with same *label* are said to be consistent if they have the same $elementNum$).

In all the places, we use $(label, elementNum)$ to talk about a particular tuple in the sequence and we use this interchangeably for node and tuple in the rest of this paper. In the rest of the paper the term *sequence* refers to modified Prüfer sequence generated using above procedure.

An Example XML document and the sequence corresponding to that can be seen in the Figure 2. The sequence is represented in the figure as *position:(label, elementNum, level, count).* Here the gaps are given with the gap parameter G equal to 5.The letters in the nodes denote the element tags in the XML document.

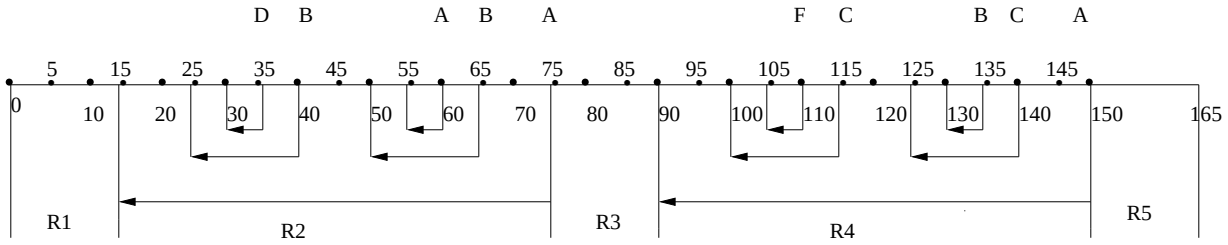


Figure 3: **Figure showing sequence numbers with gaps and region covered by each tuple**

We can see that number of tuples in the sequence is one less than the number of nodes(including dummy nodes) in the tree. The nodes without any label denote the dummy nodes that are added to include the leaf nodes into the sequence. In an actual XML document the values at leaf level would correspond to the dummy nodes. The $elementNum$ distinguishes elements with same labels. The Count field in the sequence denotes the number of nodes in the subtree of the deleted node and that include gaps also into consideration. Take for instance, When the node B at level 2 is deleted the tuple output is (A, 1, 1, 60) at position 75. Its count is equal to the sum of the counts of two instances of (B,1) at positions 40 and 65 and three gaps of 10 each at level two, i.e (15+15)+(3*10), which is equal to 60. Given an XML document we can generate the above sequence in one pass of the document using a SAX parser.

The way gaps are given and the way count and position numbers are assigned for document in Figure 2 can be seen the Figure 3. Each tuple has a count value that shows the range of positions in the subtree rooted at its previous tuple's node. So, at level 3 a gap of 5 is given, at level two a gap of 10 is given and at top level a gap of 15 is given. Here, we have non-contiguous position numbers assigned to the tuples. The position numbers that are not assigned are reserved for the future insertions. The region R2 denotes the range of positions assigned for the first subtree of the document in Figure 2. The region R4 denotes the second subtree of the document and the regions R1, R3 and R5 are the three gaps provided at the top level for the future insertions. This property holds for all the tuples in the sequence, all the regions covered by them are shown with an arrow in the Figure. Suppose a sequence corresponding to a subtree is getting inserted before the node (C,1) in the document then that sequence is assigned position numbers from the range 76 to 89, i.e positions in the region R3. One has to change the gap parameter G to control the number of insertions. Once the gap is less than the number of tuples to be inserted, one has to delete the existing sequence from the index and new sequence is to be regenerated and inserted into the database.

To provide more insertions at a given place, both position and count values can be made real. In this case, whenever a new sequence is to be inserted, a normal sequence is generated with the given gap value. This gives rise to a tuples spread in a range of positions like the one shown in Figure 3. If the range of positions obtained is (0, R) and the Interval where the sequence to be inserted is (a, b), then we get a fraction $y = (b - a)/R$ and each tuple T_i of the sequence is assigned position and count values as follows.
$T_i.position = a + T_i.position * y$
$T_i.count = T_i.count * y$

Once the fraction y reaches some lower bound, we stop dividing the interval further and rebuilt the sequence completely. By providing gaps this way we can avoid rebuilding the index every time an insertion occurs. Deletions of some part of the document results in a gap that can be used for future insertions.

For a given sequence we can prove the following properties.

Theorem 1. *Let a node (A, n) appear k times in the sequence at positions $p_1, p_2, ..., p_k$ and $l_1, l_2, ..., l_k$ be the count values of the tuples respectively, let T_{p_i} denote the tuple at position p_i, then*

1. *There are exactly k children for the node (A, n).*
2. *Tuples from $T_{p_i - l_i}$ to T_{p_i} $(1 \leq i \leq k)$ correspond to the subtree rooted at the ith child of node (A, n).*

Proof. 1. From the way sequence is constructed, it is clear that a node appears whenever one of its children is removed. Since (A, n) appears k times in the sequence it should have k children.
2. The nodes are deleted in post-order to construct the sequence, so all the nodes below a certain node get deleted before the node itself gets deleted. From the definition of count it is clear that a tuple's count includes counts of all the instances of the node being deleted and the gaps that are given for subtrees rooted at that node. We know that ith tuple instance of an element occurs on the deletion of ith child of the node. So the tuples from $T_{p_i - l_i}$ to T_{p_i} will correspond to the ith subtree of the node (A, n). □

Example 1: *Take the sequence for the Figure 2. One can observe that element $(A, 1)$ appears at two positions $75th$ and $150th$. Tuples from position 15 to 75 in the sequence correspond to the subtree rooted at*

the first child of $(A, 1)$. *Similarly we can see that tuples from* 90 *to* 150 *positions correspond to the subtree rooted at the second child of* $(A, 1)$. *This property is maintained for all the tuples in the sequence.*

Finding the structural relationship between two elements in the XML document is the basic operation for solving the XPATH expressions. A structural relationship could be parent-child (/) or one of the XPATH's wild card relationships ancestor-descendant (//) or '*'. These structural relationships between two elements denote that there is a linear path connecting the two nodes in the tree with some constraint on the level of the nodes. We define the connectedness of two tuples as follows.

Connectedness: We say that two tuples in the sequence are connected, if the nodes corresponding to the tuples are connected by a linear path. A linear path here means that it should not form a twig structure.

We define *immediate ancestor tuple* of a node as follows.

Immediate ancestor tuple: Let a node A be an ancestor of a node B in the XML tree. We call the tuple corresponding to A that occurs immediately after all the tuples corresponding to B as the *immediate A-ancestor* tuple of B.

If A is the parent of B in the XML tree, then the immediate A-ancestor tuple of B is called *immediate parent tuple* of the node B. In the following Theorem we specify how the connectedness check between a pair of nodes can be done given the modified Prüfer sequence. We restrict our connectedness check of a node to the immediate ancestor tuple in the sequence.

Theorem 2. *Let* $T_i = (A, x, l_i, c_i)$ *and* $T_j = (B, y, l_j, c_j)$ *be two tuples at positions* i *and* j $(i < j)$ *in the modified Prüfer sequence corresponding to an XML data tree such that there is no tuple* $T_k = (B, y, l_k, c_k)$ *at position* k, $i < k < j$ *then,* (A, x) *is a descendant of* (B, y) *iff* $(j - i) < c_j$.

Proof. Assume that (A, x) is descendant of (B, y).
It is given that there is no (B, y) present in between the positions i and j. Using the Theorem 1 we can say that (A, x) at position i will be within the range of $j - c_j$ to j. So we can say that $i > j - c_j$. Thus $j - i < c_j$.
Take the sufficient condition $(j - i) < c_j$
Suppose that (B, y) belongs to the kth instance of the node (B, y) and given that $(j - i) < c_j$. Using the Theorem 1 we can say that (A, x) belongs to the kth subtree rooted at (B, y) since position i is in the range of $j - c_j$ to j. So we can say that (A, x) is the descendant of (B, y) and hence the proof of the theorem. □

Example 2: *Take two tuples at positions* 60 *and* 75 *in the Figure 2. The difference in their positions* $75 - 60$ *is less than the count of the latter tuple which is* 60. *So we can say that they are connected. Even from the tree it is clear that* $(B, 1)$ *and* $(A, 1)$ *are connected.*

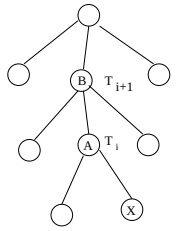


Figure 4: **An XML Tree**

We say there is *drop in level* between two successive tuples in the sequence if the level of the former is higher than the level of the latter. Similarly we say there is a *rise in level* between two tuples if the level of the former is less than the level of the latter.

Theorem 3. *Let* T_i *and* T_{i+1} *be two consecutive tuples in the sequence corresponding to the nodes* (A, x) *and* (B, y), *then*

1. *If there is drop in the level from* T_i *to* T_{i+1} *then we can say that* (B, y) *is the parent of* (A, x).
2. *If there is rise in the level from* T_i *to* T_{i+1} *then we can say that* (B, y) *belongs to the subtree rooted at the next available child of* (A, x).

Proof. 1. Consider the scenario as depicted in the Figure 4. Here, X is the last child of A and B is the parent of A and tuples T_i and T_{i+1} correspond to nodes A and B respectively. We claim that this must be the scenario if there is a drop in level from T_i to T_{i+1}. For, Otherwise if X is not the last child of A then in the post-order method of removing nodes, A can not be removed and the next tuple generated corresponds to some other node in the subtree rooted at the following sibling of the X and hence has level greater than or equal to that of node A. Hence we can conclude that B is a parent of A.
2. From the above discussion it is clear that if there is a rise in the level from T_i to T_{i+1} then we can say that (B, y) belongs to the subtree rooted at the next available child of (A, x). □

For a given Modified Prüfer sequence one can always reconstruct the tree uniquely using Theorem 3. The detailed algorithm can be found in[9].

3.2.1 Query Sequence

A sequence for a query pattern is also generated in a same way except that the tuple will have a relationship field instead of count. Relationship describes the relation of the tuple with the preceding tuple in the sequence. It could be one of parent-child or ancestor-descendant or the wild card '*' present in the XPATH expression. For leaf nodes we maintain a special value

as there won't be any relationship between preceding node to itself. An example XPATH expression, its equivalent tree representation and the sequence corresponding to that are given in Figure 5. The last field denotes the relationship. The relationships '0', 'v', 'p' and 'a' denotes a leaf, value, parent and ancestor respectively. A '*' in the relationship denote the wild card '*' in the query expression. Encoding a query sequence in this way helps validation of subsequence which is described in later sections.

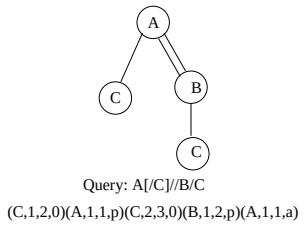


Figure 5: **An Example XPATH Expression Tree and the Equivalent Sequence**

3.3 Indexing

This section gives the details about the way the sequences are indexed for efficient subsequence matching. In our system we maintain three different kinds of indices that are useful in query processing. All the three index structures can be seen in the Figure 6.

For subsequence matching we maintain a Two level B^+-tree proposed in Vist[16]. We take a sequence and assign a *tail* to each tuple in the sequence. The *tail* denotes the number of tuples in the sequence that follow the tuple being considered (This takes gaps also into account). We build a B^+-tree for the element tags in the document and we call it the Document B^+-tree. Each leaf node in this points to another B^+-tree called Element B^+-tree. All the tuples with the similar labels go into the same Element B^+-tree. It is indexed on the *position* of the tuple as key and it stores the values of *tail* along with $(elementNum, level, count)$ values of the tuples in the sequence.

Indexing the data values at the leaf nodes of the XML tree is done in a different way which actually helps in speed up the processing of the value based queries. This again is a two level B^+-tree with the top level B^+-tree being built on distinct leaf values which in turn points to another Element B^+-tree. The key of this Element B^+-tree is the *position* of the tuple corresponding to the node to which this leaf node is attached. By indexing in this way the parent of a leaf can be reached directly.

For supporting indexing of multiple documents we maintain a DocID index. Tuples in each document sequence are given a distinct range of position numbers. For example, if we have two sequences of size 20 and 15, we give position numbers in the range of (1,20) to

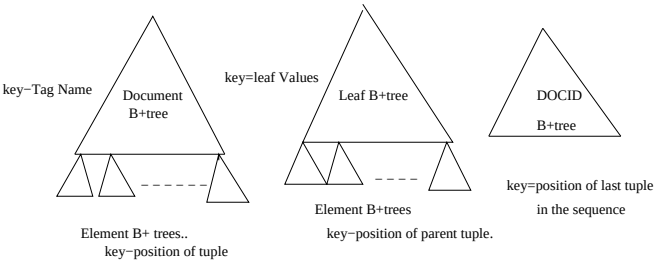


Figure 6: **Different Index Structures Maintained in the System**

the first sequence and (21,35) to the second sequence. We assign a new range whenever a new sequence is inserted into the database. A DocId B^+-tree is maintained separately which indexes the last tuple's position number in each sequence. It stores the $(DocId$, *size of the sequence*) as data value.

3.4 Query Processing

This section gives the details about query processing using the subsequence matching. Once the data sequence is indexed according to the method given in Section 3.3, we do a non-contiguous subsequence match of the label part of the query sequence with the label part of the data sequence. We perform following two checks at each phase of the subsequence matching to ensure that the subsequence being matched is a valid query result.

- **Connectedness check:** If two tuples T_i and T_j at positions P_i and P_j matching query tuples Q_i and Q_j, if $Q_j.level < Q_i.level$ (Which means they are connected), then T_j is retained if it satisfies the test $P_j - P_i < T_j.count$ given in Theorem 2. Along with this a structural relationship check is also carried to check whether the two tuples satisfy the relationship given in the query tuple Q_j. This can be done using the level information encoded in the tuples.

- **Consistency check :** If two tuples Q_i and Q_j have the same $elementNum$ in the query twig then the matching tuples T_i and T_j should also have the same $elementNum$ i.e, $T_i.elementNum = T_j.elementNum$.

Now we give a brief description of how query processing is done using subsequence match taking a simple example. Consider an XPATH expression A[B]/C which results in a query sequence (B,1,2,0) (A,1,1,p) (C,1,2,0)(A,1,1,p). To solve this, first all the B tuples are retrieved from the document. After getting all the B's, for each B_i we get all the A tuples that follow it in the sequence. Now The connectedness check is applied for each A_j with B_i and the tuples that satisfy are retained. For each of the retained A_j, following C tuples are fetched and no check is performed here as

it has no relationship with the previous tuple in the query sequence. For each C_k all the A tuples that follow it are fetched. Connectedness check is performed for each A_l with C_k and those satisfy this are checked for consistency with tuple A_j and the resulting tuples are retained and are output with the corresponding C, A and B tuples. This is the process of simple pattern matching.

The above subsequence matching can be easily extended to multiple documents which are indexed in the way described in Section 3.3. The only difference is that in the first stage we get all the tuples from Element B^+-tree of the first tuple. This will get the tuples from all the documents. After getting all the elements in the first stage we get the next elements within the range of each of these element. Here, range means $(position, position + tail)$.

Detailed algorithm explaining the process of pattern matching can be found in[9]. An optimization mechanism for parent-child axis is also presented in[9]. Where an extra *parentPointer*, an offset from the tuple to the immediate parent tuple, is maintained to speed up the query processing. With this one can directly get the parent tuple of a tuple by adding its *position* with *parentPointer* instead of retrieving all the tuples in its range and then filtering them using the connectedness.

3.5 Handling Updates

This subsection gives details of how the updates are handled on an XML document. This mainly deals with how the index is updated on each operation.

To perform all the updates entire sequence is maintained separately in a B^+-tree. We call this B^+-tree a sequence B^+-tree. This is indexed on the position number. This is useful in reconstructing the results of any query and also while performing updates.

In all the examples discussed in this section we use the sequence in the Figure 2. i.e
35:(D,1,3,5), 40:(B,1,2,15), 60:(A,2,3,5), 65:(B,1,2,15), 75:(A,1,1,60), 110:(F,1,3,5), 115:(C,1,2,15), 135:(B,2,3,5), 140:(C,1,2,15), 150:(A,1,1,60).

The following functions are used in the algorithms given in this section.

- **parentTuple(N):** This gives the immediate parent tuple of a given tuple N if it exists, otherwise NULL is returned. For example take the tuple T={40: (B,1,2,15)}, then parentTuple(T) returns the tuple {75: (A,1,1,60)}.
- **nextTuple(N):** This outputs the tuple next to N in the sequence if it exists, otherwise NULL is returned. For example take the tuple T={60: (A,2,3,5)}, then the nextTuple(T) returns {65: (B,1,2,15)}.
- **nextTupleInstance(N):** This gives the next tuple with the same *label* and *elementNum* of N and immediately following N in the sequence if exists, Otherwise NULL is returned. For example take the tuple T={115: (C,1,2,15)}, then nextInstanceTuple(T) returns {140: (C,1,2,15)}.
- **prevTupleInstance(N):** This returns the tuple with the same *label* and *elementNum* and immediately preceding N in the sequence if exists, otherwise NULL is returned. For example take the tuple T={150: (A,1,1,60)}, then prevTupleInstance(T) returns {75: (A,1,1,60)}.
- **lastTupleInstance(N):** This returns the last tuple in the sequence with the same *label* and *elementNum* as N. For example take the tuple T={40: (B,1,2,15)}, then lastTupleInstance(T) returns {65 : (B,1,2,15)}.

3.5.1 Insert

This section deals with the details about how the insertions are handled. The basic algorithm for updating the index structure on the insert operation is given. We initially get the position of the node where insertion has to be performed and from that the gap where new node/subtree is to be inserted. We generate the sequence for the subtree to be inserted in the way described earlier. In addition, a tuple corresponding to the parent node of the subtree being inserted will also be inserted just after the sequence of the subtree. Now we get the fraction y using R obtained from sequence generation and the gap. If the fraction doesn't cross the lower bound then the new sequence is assigned position numbers within the range of the gap and the count values are modified accordingly. All the new tuples are inserted into appropriate Element B^+-trees and also into sequence B^+-tree.

We give the basic algorithms that get the gap in case of insertions. Once we get the gap we can assign the position numbers and count values to the sequence to be inserted. If there is not enough gap then we have to go for the normal process of deleting the entire sequence corresponding to a document and generating a new sequence for the document after the insertion and then reinserting the whole sequence in the database.

The Algorithm 1 gives the gap after a given tuple of a node. The tuple could be any one of the instances of the tuple corresponding to the node given. Here, parentTuple(N) gives the immediate parent tuple of the tuple N and nextTupleInstance(Y) gives the next tuple instance of node corresponding to Y. Finally gap is stored in g as start and end. Take the example document in Figure 2, the gap after node $(B, 1)$ can be found by giving any of $(B, 1)$'s instances as input to the Algorithm 1. The gap g will be assigned to $(76, 89)$. In case of tuple at 140 the output will be $(150, 165)$.

The Algorithm 2 gives the gap for insertion before a specified tuple of a node. Here, prevTupleInstance(N)

Algorithm 1 Algorithm for Getting gap after a node

Procedure :gapAfter(tuple N)

```
1: X = parentTuple(N)
2: Y = nextTuple(X)
3: g.start = X.position
4: if Y = NULL then
5:    g.end = X.position + X.tail
6: else
7:    if Y.level < X.level then
8:       g.end = Y.position
9:    else
10:      Y = nextTupleInstance(X)
11:      g.end = Y.position−Y.count
12:   end if
13: end if
14: return(g)
```

return the previous tuple if it exists otherwise NULL is returned. R is the total number of positions that are assigned for the document. The other terms have the usual meanings as defined previously. R in case of example document is 165 and it can be seen Figure 3. Gap before element $(B, 2)$ can be found by giving tuple $(B, 2)$ at position 135 and the output will be $(115, 125)$. In case $(B, 1)$ at position 40 it will be $(15, 25)$.

Algorithm 2 Algorithm for Getting gap before a node

Procedure :gapBefore(tuple N)

```
1: X = parentTuple(N)
2: g.end = X.position−X.count
3: X= prevTupleInstace(X)
4: if X ≠ NULL then
5:    g.start = X.position
6: else
7:    X = parentTuple(X)
8:    if X≠NULL then
9:       g.start = X.position−X.count
10:   else
11:      g.start = N.position + N.tail− R
12:   end if
13: end if
14: return (g)
```

The Algorithm 3 gives the gap for normal insertions where the node is inserted as the last child of the given node. Here, the function lastTupleInstance(N) returns the last instance of the node corresponding to a tuple N. For inserting a node as a child of node $(C, 1)$, we get the gap as $(140, 150)$.

A general procedure for updating the index when a node/subtree is inserted can be seen in Algorithm 4. This gives the outline of all types of insertions except that one has to get the appropriate gap by appropriately calling one of the procedures defined depending on the type of insertion.

Algorithm 3 Algorithm for getting gap at the last child of a node

Procedure :gap(tuple N)

```
1: X = lastTupleInstance(N)
2: g.start = X.position
3: Y = parentTuple(N)
4: if Y==NULL then
5:    g.end = X.position + X.tail
6: else
7:    g.end = Y.position
8: end if
9: return (g)
```

Algorithm 4 General Algorithm for Insertion

Procedure :insert()

1: Get the gap where new sequence of the node/subtree is to be inserted.
2: Assign the positions and count values for the sequence
3: Insert each tuple in the appropriate Element B^+-tree with position as key.
4: Insert the new sequence in the Sequence B^+-tree with position as key.

3.5.2 Delete

Compared to the process of insert, delete process is simple. One has to delete the sequence corresponding to the subtree rooted at the node being deleted. The Algorithm 5 gives the details of how the index is updated when a delete operation is carried out.

Algorithm 5 General Algorithm for Deletion

Procedure : delete(N)

1: X=parentTuple(N)
2: Delete tuples in the interval [X.position−X.count, X.position] from Element and Sequence B^+-trees .

3.5.3 Replace

The process Replace Node1,Node2 can be done by calling insertBefore(Node1,Node2) and then calling delete(Node1). This can also be done by deleting Node1 and inserting the Node2 in the gap created by deletion of Node1.

3.5.4 Rename

This can be done by deleting the particular node instance from the corresponding ElementB^+-trees and then inserting them in the ElementB^+-tree corresponding to the label with which it has to be renamed. The label part of all instance tuples of the Node are to changed in the Sequence B^+-tree.

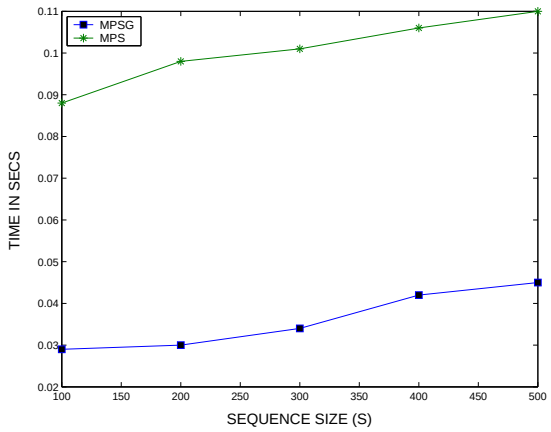


Figure 7: **Performance of the systems With N=1000 and varying S**

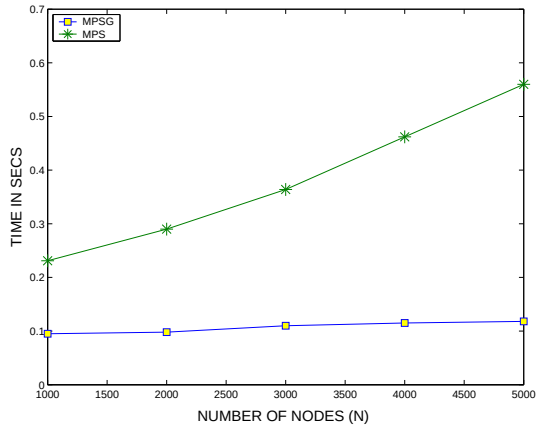


Figure 8: **Performance of the system with S=1000 and varying N**

3.6 Analysis of the Operations

This section gives the analysis of the update operations in case system that supports gaps(MPSG), and the one without gaps(MPS). The performance of the system mainly depends on the time taken to execute insert and delete operations. Let N be the original document sequence size and let S be the size of the sequence to be inserted or deleted.

We can see that an operation in the normal case depends on both N and S because entire sequence is to be deleted and reinserted. Suppose each insertion and deletion of a tuple takes t time then we can say that using earlier system the insert operation will take $Nt + Nt + St$, i.e $2Nt + St$, where Nt for deleting the original sequence and $Nt + St$ for reinserting the new sequence. In case of deletion it will be $Nt + (N - S)t$, i.e $2N - St$, where Nt is for deleting original sequence and $Nt - St$ for reinserting the new sequence.

In case of the proposed *Modified Prüfer Sequence with Gaps*(MPSG) system as long as the gap is present, the time taken is proportional to the size S of the new sequence to be inserted or deleted. In the case if the gap gets over it will be the same as earlier MPS system which doesn't provide gaps. But one should note that such cases are rare when random insertions are considered. In most of the cases we can say that it takes St time to process an update request.

Without gaps the performance of the system degrades with the increase in sequence size, i.e document size. Whereas with gaps it is independent of the size of the document.

4 Experimental Results

We implemented our system in C++ for indexing, querying and updating of XML data. SAX parser is used to parse the documents and create sequences. We used the B^+-tree API provided by BerkeleyDB[6] for building B^+-trees. All the experiments were conducted on a machine with Intel 2.4GHz processor with 256MB RAM running Fedora. We used the IEEE 64-bit floating point numbers for representing *position number, tail* and *count* of the tuples. For all the experiments we indexed a part of DBLP dataset, downloaded from[5], and inserted other parts of it into the existing database. Here we considered whole DBLP document as a single sequence and carried our experiments.

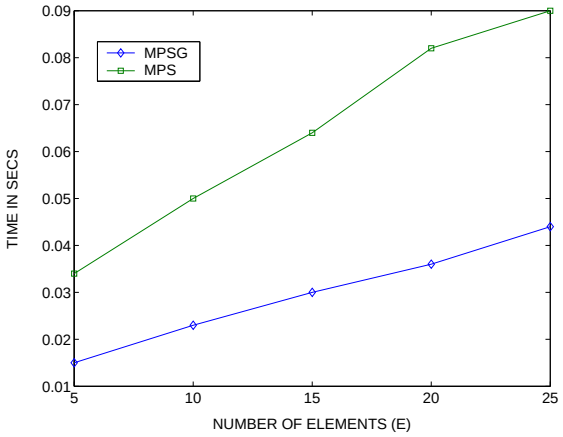


Figure 9: **Performance of the system with S=100 N=1000 and varying no of Elements**

We have compared our results of MPS with gaps(MPSG) with MPS without gaps. In case of MPS whenever an update occurs entire sequence has to be deleted and the new sequence is to be inserted. Though it requires to rebuild the entire sequence once the gap touches its lower bound in MPSG, the number of random updates that can be handled with the current system is very significant. It is observed that in all the cases MPSG outperforms MPS.

Three main factors that affect the performance of the system. One is document size N, second is sequence size S and the third is the number of different elements in the document E. The way the three fac-

tors affect the system performance is studied. In the Figure 7, E is 36 and document size N is 1000 and the performance of MPSG and MPS are tested for the insertions of various sizes by varying S. In Figure 8, E is 36 and sequence size S is 1000 and the document sequence size is varied and the performance of both the systems are plotted. One can see that performance of the system without gaps degrades with the increase in document size. In Figure 9 the performance of the systems is tested by the varying number of elements E. Here, N is 1000 and S is 100. This experiment is done by generating random data with different number of element tags. It can be seen that as the number of elements increases performance degrades because of the increase in the number of B^+-trees to be loaded. While processing all the update operations, the costly step is the loading time of B^+-trees and its effect can be seen in the Figure 9.

5 Conclusions

In this paper, we proposed a sequencing mechanism called *Modified Prüfer Sequences with Gaps* for converting an XML document into sequence. We give appropriate gaps in the sequence to support future insertions. We presented methods for handling insertions, deletions, replacing and renaming of the nodes. We have demonstrated that updates can be handled without changing the identities of the existing nodes as long as gap is present in the sequence. Our experimental results show that the proposed scheme handles updates efficiently.

References

[1] A.Berglund, S.Bong, D.Chemberlin, M.F.Fernandez, M.Kay, J.Robie, and J.Simon. XML path language(XPATH)2.0 W3C working draft. Technical report, World Wide Web Consortium, August 2002.

[2] T. Amagasa, M. Yoshikawa, and S. Uemera. QRS: A robust numbering scheme for xml documents. In *ICDE*, pages 705–707, March 2004.

[3] T. Fiebig and G. Moerkotte. Evaluating queries on structure with extended access support relations. In *WebDB*, pages 41–46, May 2000.

[4] H.Prüfer. Neuer beweis eines satzes über permutationen. *Archive für Mathematik and Physik, 27*, pages 142–144, 1918.

[5] http://www.cs.washington.edu/research/xmldatasets.

[6] http://www.sleepycat.com.

[7] I.Tatarinov, Z.G.Ives, A.Y.Halevy, and D.S.Weld. Updating XML. In *ACM-SIGMOD*, pages 413–424, May 2001.

[8] P. O'Neil, E. O'Neil, S.Pal, I.Cseri, and G.Schaller. ORDPATHs: Insert-friendly XML node labels. In *ACM-SIGMOD*, pages 903–908, June 2004.

[9] K. H. Prasad and P. S. Kumar. Indexing and querying of XML data using modified prüfer sequences. Technical report, IIT MADRAS, January 2005. http://aidb.cs.iitm.ernet.in/students/hima/mps.pdf.

[10] Q.Li and B.Moon. Indexing and querying XML data for regular path expressions. In *VLDB*, pages 361–370, September 2001.

[11] P. Rao and B. Moon. PRIX: Indexing and querying XML using Prüfer sequence. In *ICDE*, pages 288–300, March 2004.

[12] S.Al-Khalifa, H.V.Jagadish, N.Koudas, J.M.Patel, D.Srivastava, and Y.Wu. Structural joins: A primitive for efficient XML query processing. In *ICDE*, pages 141–152, Feb 2002.

[13] S.Bong, D.Chemberlin, M.F.Fernandez, D.Florescu, J.Robie, and J.Simon. XQuery1.0: An XML query laguage W3C working draft. Technical report, World Wide Web Consortium, August 2002.

[14] T.Bray, J.Paoli, C.M.Sperberg-McQueen, and E.Maler. Extensible markup language(XML)1.0. Technical report, World Wide Web Consortium, October 2000.

[15] H. Wang and X. Meng. On the sequencing of tree structures for XML indexing. In *ICDE*, pages 373–383, April 2005.

[16] H. Wang, S. Park, W. Fan, and P. S. Yu. Vist: A dynamic index method for querying XML data by tree structures. In *ACM-SIGMOD*, pages 110–121, June 2003.