Efficient Indexing and Querying of XML Data using Modified Prüfer Sequences

K. Hima Prasad   P. Sreenivasa Kumar

Dept of Computer Science and Engineering
Indian Institute of Technology Madras
Chennai - 600 036, India
{hima, psk}@cs.iitm.ernet.in

ABSTRACT

With the advent of XML as the new standard for information representation and exchange, indexing and querying of XML data is of major concern. In this paper, we propose a method for representing an XML document as a sequence based on a variation of Prüfer sequences. We incorporate new components in the node encodings such as level, number of a certain kind of descendants and develop methods for holistic processing of tree pattern queries. The query processing involves converting the query also into a sequence and performing subsequence matching on the document sequence. We establish certain interesting properties of the proposed method of sequencing that give rise to a new efficient pattern matching algorithm. The sequence data is stored in a two level B^+-trees to support query processing. We also propose an optimization for parent-child axis to speed up the query processing. Our approach does not require any post-processing and guarantees results that are free of false positives and duplicates. Experimental results show that our system performs significantly better than previous systems in a large number of cases.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: [Systems—Query processing]; H.3.1 [**Information Storage and Retrieval**]: [Content Analysis and Indexing—Indexing methods]

General Terms

Algorithms, Experimentation, Performance

Keywords

XML, Indexing, Query Processing, Modified Prüfer Sequences, Tree pattern queries

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CIKM'05, October 31–November 5, 2005, Bremen, Germany.
Copyright 2005 ACM 1-59593-140-6/05/0010 ...$5.00.

1. INTRODUCTION

XML has emerged as the new standard for representing and exchanging information on the web[13]. XML data is self describing and can be modeled as an ordered node-labeled tree. Each node in the tree represents a tag in the XML document and the values are stored at the leaf level. Indexing and querying XML data has been a major research issue in the database world. Many query languages such as XQUERY[12] and XPATH[1] are proposed for querying XML data. Path expressions are the basic building blocks of these languages. A path expression specifies a tree pattern where the nodes are separated by location steps. These location steps can be any of parent-child(/), ancestor-descendant(//), a wild card(*) etc. Using a path expression one can specify constraints over both structure and content of the XML data. Nodes in the path expression can contain predicates which can be value predicates or can be simple path expressions.

One way of solving tree pattern queries is using the structural join approach where an XML document is encoded using interval encoding scheme. In this approach structural join between two element lists corresponding to nodes given in the query is performed and it takes multiple binary structural joins to solve a tree pattern query. There have been many efforts for solving tree pattern queries using structural joins[11, 10, 2, 8]. To avoid expensive individual join operations, Pathstack and Twigstack algorithms[6] have been proposed. In this, stack based algorithms for holistic processing of path and twig queries are proposed. These have been proved to be performing better than earlier simple join based approaches.

Since XML data and Queries are tree patterns, there have been efforts to convert both into sequences and do a subsequence match of the query sequence on the data sequence to get the results. Vist[15] and PRIX[9] follow this approach. They use a virtual trie index proposed in Vist[15] for indexing subsequences. The PRIX system uses Prüfer sequences[3] for transforming XML data into sequences. In both of these approaches mere subsequence matching does not result in valid results. One has to do post processing after subsequence matching to filter the matches that are valid according to the given query structure.

In this paper we propose a new way of generating a sequence for an XML document called *modified Prüfer sequence* which is inspired by Prüfer sequences[3]. We give a new indexing mechanism for these sequences that can index

sequences of any length and supports holistic processing of twig queries efficiently. In our system we club the validation and subsequence matching process together which results in elimination of invalid results and thus resulting in reduction of disk I/O's performed while querying. Our method also ensures that results are free of false positives and duplicates and does not require any post-processing. We also propose optimization for parent-child axis.

The main contributions of this paper are summarized as follows:

- A new way of transforming XML documents into sequences based on a variation of Prüfer's method.
- We propose a suitable indexing scheme for indexing the sequences.
- An efficient pattern matching algorithm, which processes twig queries without breaking them into linear paths and also gives the results that are free of duplicates and false positives, is proposed.
- An optimization for parent-child axis is proposed.

The rest of this paper is organized as follows. Section 2 gives the background and related work. In Section 3 we describe our work. In Section 4 we give the details of optimizing parent-child axis. Section 5 gives the implementation details. In Section 6 we present our experimental results. Section 7 gives the conclusions of the paper.

2. BACKGROUND AND RELATED WORK

Path expressions are the basic building blocks of both XPATH[1] and XQuery[12]. To support these queries several indexing methods and querying algorithms are proposed.

2.1 Querying XML using Interval Encoding:

As XML is a tree structured data, to find the twig match efficiently, structural relations among the elements in the query twig are to be satisfied. To solve the structural relationship between any two elements, Khalifa et al. proposed a numbering scheme [11] which is based on the interval encoding of the elements. In an XML document each element is assigned with a tuple ($DocID$,$start$,end,$level$) where ($start$, end) denotes an interval. If an element B is a descendant of A then it is assigned an interval which is contained in the interval of A. To find structural relationship between two elements one element's ($start$,end) has to be contained within the other element's interval. Structural join algorithm [11] takes two lists of elements and gives the pairs of elements that are structurally related. There are many efforts that are based on the positional representation of [10, 2, 8]. Using this approach to compute the result of a path or twig query needs many binary structural joins on element lists which is very expensive.

Bruno et al. proposed stack based pattern matching algorithms called Pathstack and Twigstack[6] which uses interval encoding of elements to find twig match. These operate on input stacks corresponding to each element in the query and the stacks are linked according to the ancestor descendant relationship in the query twig. A variant of Twigstack algorithm called TwigstackXB[6] that uses XB-trees to speed up the processing when input lists are long by skipping the elements within the lists that won't result in the solution. Twigstack also suffers from sub-optimality in case of solving queries with parent-child relationships.

2.2 Querying XML by Subsequence Match:

Wang et al. proposed a new method that transforms XML data tree and Twig queries into structure encoded sequences and do a subsequence match of the query sequence on the data sequence to find a valid match in the Vist system[15]. The structure encoded sequence is a sequence of (symbol,prefix) pairs (a_1, p_1), $(a_2, p_2)...(a_n, p_n)$ where a_i represents a node in the XML document tree and p_i represents the path from root to a_i. The nodes $a_1, a_2...a_n$ are in pre-order. They proposed a new technique for building virtual trie using B^+-trees, which is useful in subsequence matching. The main drawback of the above method is that indexing large sequences result in underflow and it takes many disk I/O's to find matches because of the top down transformation of the tree to sequence. Apart from this it gives false positives in case of documents having identical sibling nodes and also doesn't handle duplicates resulting from the process of matching.

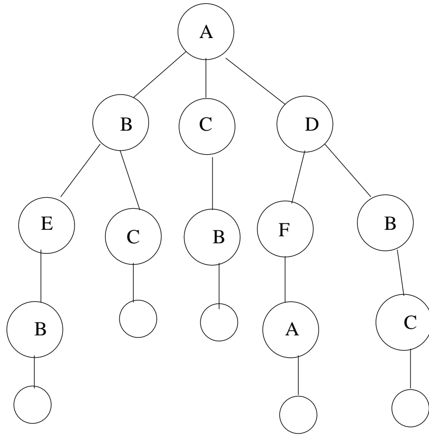
Praveen Rao et al. proposed an another method for querying XML using Prüfer sequences[9]. In this XML document is transformed into sequence of labels by Prüfer's[3] method that constructs a one to one correspondence between tree and the sequence. All the nodes in the tree are given unique numbers from 1 to n. A Prüfer sequence of a tree T_n is generated by deleting one node at a time from T_n. Delete a leaf node with smallest number to form a smaller tree T_{n-1}. Let a_1 be the label of the node that was the parent of the node deleted. Repeat this process on T_{n-1} to determine a_2 and continue until only two nodes are left. The sequence $(a_1, a_2, ..., a_{n-2})$ is called *Labeled Prüfer sequence* of tree T_n. The numbers corresponding to those nodes in sequence form a *Numbered Prüfer sequence*. The PRIX system of[9] uses *Extended-Prüfer sequence* to support value based queries. This sequence is generated by attaching dummy nodes to the leaves and constructing Prüfer sequence for it. In PRIX post-order numbering is given to the nodes and a Prüfer sequence of length $n - 1$ is generated by removing elements till one node is left. It eliminates the problem of false positives. It reduces the number of disk I/O's taken for subsequence matching substantially because of bottom-up transformation of tree into a sequence. It gets all the subsequence matches from the database in first phase and does a document-wise post processing using the numbered Prüfer sequence in the next phase to get valid results. Recently Wang et.al proposed a constraint sequencing method which aims at finding best sequencing strategy to index and query XML data[14].

3. ENCODING AND QUERYING XML USING MODIFIED PRÜFER SEQUENCES

In this section we give the details of the XML querying system built by us using modified Prüfer sequences.

3.1 Sequence Generation

A sequence for an XML document tree is constructed in a manner similar to the one used in Extended-Prüfer sequence. We call this sequence **Modified Prüfer Sequence**. We delete the nodes in post-order sequence and output the label of parent nodes with additional information. We assume that for each leaf node in the XML data tree, a dummy child node is attached. We define $elementNumber$ for each node with label L in the document as follows:



(B,2,4,1) (E,1,3,2)(B,1,2,3)(C,1,3,1)(B,1,2,2)(A,1,1,6)
(B,3,3,1)(C,2,2,2)(A,1,1,3)
(A,2,4,1)(F,1,3,2)(D,1,2,3)(C,3,4,1)(B,4,3,2)(D,1,2,3)(A,1,1,7)

Figure 1: The Sample XML Document and corresponding Modified Prüfer Sequence

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).
Let X be the node being deleted and P be its parent in the tree. The tuple generated corresponding to the deletion of X will have the following components.
(label, elementNum, level, count)

- *label:* Denotes the tag of the parent node P.
- *elementNum: elementNumber* of P
- *level:* Level at which P is present in the XML document tree.
- *count:* Number of nodes(including dummy nodes) in the subtree rooted at the node X in the original document tree.

XML documents are converted into sequences as described above. An Example XML document and the sequence corresponding to that can be seen in the Figure 1. The letters in the nodes denote the element tags in the XML document. 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 similar labels. The Count field in the sequence denotes the number of nodes in the subtree of the deleted node. This takes dummy nodes also into consideration. Take for instance, When the node B at level 2 is deleted the tuple output is (A, 1, 1, 6) at position 6. It has the count of the nodes in the subtree containing the node $(B, 1)$, which is nothing but the count of the nodes in the first subtree of the root. Given an XML document we can generate the above sequence in one pass of the document using a SAX parser.

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. 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 $p_1, p_2, ..., p_k$, let l be the count value of (A, n) at position p_1 and 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_1-l+1} to T_{p_1-1} correspond to the subtree rooted at the first child of the node (A, n).*
3. *Tuples T_{p_i+1} to $T_{p_{i+1}-1}$ $(i < k)$ correspond to the subtree rooted at the $i+1$th child of node (A, n).*

The proof of the theorem follows the definition of the sequence. Due to space constraints we do not provide the proofs of these theorems here. For details one can refer to[7].
Example 1: *Take the sequence for the Figure 1. One can observe that element $(A, 1)$ appears at three positions $6th$, $9th$ and $16th$. The tuples from position 1 to 5 is the sequence corresponding to the subtree rooted at the first child of $(A, 1)$. Similarly we can see that tuples from $7th$ to $8th$ positions correspond to the subtree rooted at the second child of (A, n). 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 $(j - i) < c_j$.*

Example 2: *Take two tuples at positions 3 and 6 in the Figure 1. The difference in their positions $6-3$ is less than count of the 6th tuple which is 6. So we can say that they are connected. Even from the tree it is clear that $(B, 1)$ and $(A, 1)$ are connected.*

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).*

For a given Modified Prüfer sequence one can always reconstruct the tree uniquely using Theorem 3. The detailed algorithm can be found in[7].

3.1.1 Query Sequence

A sequence for a query twig 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 2. 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.

3.2 Indexing

Once the sequence is ready, we need to store it along with the sequences corresponding to other documents using a suitable indexing mechanism that supports 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 3. For subsequence matching we maintain a Two level B^+-tree similar to the one used in the Vist system [15]. We take a sequence and assign a $(position, tail)$ pair to each tuple in the sequence. *position* is the position number of the tuple

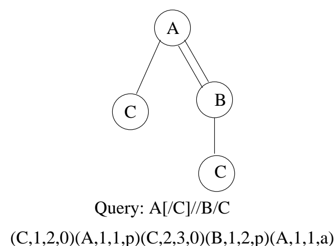


Figure 2: An Example XPATH Expression Tree and the Equivalent Sequence

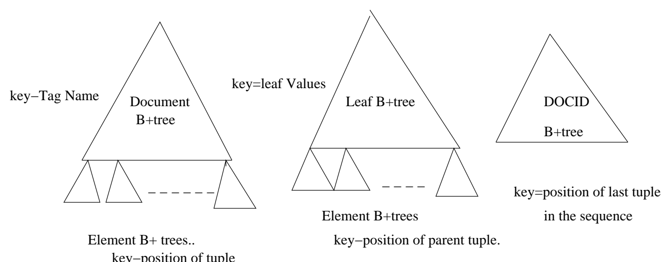


Figure 3: Different Index Structures Maintained in the System

in the sequence and *tail* denotes the number of tuples in the sequence following the tuple being considered. 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. Element B^+-tree 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. 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 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.

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.

The space complexity of the index structure is of the order of the number of tuples in the sequence. We use this single indexing scheme(containing three index structures) for processing all types of queries, whereas PRIX uses three different types of indexing schemes. First one is Prüfer index, used for simple path expressions, second one is extended Prüfer index (EPI), which is used for value based queries and the third is Reverse Prüfer index, which is used for optimizing the query processing time. The space complexity of EPI is almost equal to the number of nodes in the tree as sharing is minimal.

3.3 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 3.2, we do a non-contiguous subsequence match of the label part of the query sequence with the label part of the data sequence.

Now we give a brief description of how query processing is done by subsequence match taking a simple example. Consider an XPATH expression A/B/C which will result in a query sequence (C,1,3,0)(B,1,2,p)(A,1,1,p). To solve this first we get all C elements from the document. After getting all the C's, for each C we retrieve B's that follow it in the sequence and for each B we retrieve A tuples that follow it in the sequence. This is the process of simple subsequence matching on a single document. All the subsequences matched may not be valid results of a given query. For a subsequence to be valid it has to satisfy the tests given below. These tests are applied at each stage in the subsequence matching. All the tuples resulting in stage $i(i > 0)$ are validated using the validation checks given below and only the tuples satisfying them are retained for the next stage.

The above subsequence matching can be easily extended to multiple documents which are indexed in the way described in Section 3.2. 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 $(position, position + tail)$.

In all the examples given below we use the data sequence and query sequence of Figure 1 and Figure 2. Here x:tuple is used to denote each tuple, where x is the position number in the sequence.

Data Sequence:1:(B, 2, 4, 1), 2:(E, 1, 3, 2), 3:(B, 1, 2, 3), 4:(C, 1, 3, 1), 5:(B, 1, 2, 2), 6:(A, 1, 1, 6), 7:(B, 3, 3, 1), 8:(C, 2, 2, 2), 9:(A, 1, 1, 3), 10:(A, 2, 4, 1), 11:(F, 1, 3, 2), 12:(D, 1, 2, 3), 13:(C, 3, 4, 1), 14:(B, 4, 3, 2), 15:(D, 1, 2, 3), 16:(A, 1, 1, 7).

Query Sequence:(C, 1, 2, 0) (A, 1, 1, p) (C, 2, 3, 0) (B, 1, 2, p) (A, 1, 1, a)

3.3.1 Connectedness Check

Let us say that we are matching tuple Q_i in the query sequence in the range of data tuple D_{i-1} of the previous stage, we get positions $\{p_1, p_2, ..., p_n\}$ in the document where the matching has occurred. We retrieve data tuples $\{d_1, d_2, ..., d_n\}$ corresponding to these positions. If the $Q_i.relationship$ is one of 'p', 'a' or '*' then we do a connectedness check between tuples D_{i-1} and $d_j(0 < j \leq n)$ and retain those d_j's that are connected as per the required relationship. The connectedness check is carried out as per Theorem 2. For ancestor-descendant relationship, connectedness check is enough. Whereas in case of parent-child relationship the level difference between the two tuples should be one and in case of wild card '*' it has to be two. Tuples that fail to satisfy the structural relationship are also discarded at each stage.

We note that in our approach a single check is enough to determine the connectedness , whereas PRIX[9] takes as many checks as the level difference between the nodes in the data tree in case of ancestor-descendant axis check. It also generates duplicate results in the same case as it uses post-order numbers which allow the check to pass on all the instances of the tuples present in the sequence.

Example 3: *Consider the query given in Figure2.Take the subsequence matches R_1(8, 9, 13, 14, 16) and R_2 (8, 10, 13, 14, 16), the two vectors representing the positions of the matchings in the data sequence. Now we apply the connectedness check for the tuples at positions (1, 2), (3, 4) and (4, 5) in R_1 and R_2. R_1 satisfies the check at all the three places whereas R_2 fails at (1,2). So R_2 can be eliminated as an invalid match at that stage itself without going further. We can also see that R_1 satisfies all the structural relations given in the query sequence.*

At any point we require the positions of the two tuples along with their contents to do the above validations. So we can perform connectedness check at each phase of the subsequence matching and can discard invalid matches without considering them for further processing.

3.3.2 Consistency Check

We know that two tuples T_i and T_j with the same *label* are said to be consistent if they have the same $elementNum$. Suppose two elements at positions i and j are consistent in the query sequence then the elements matched in the data sequence at those positions should also be consistent with each other. For this, we maintain an array C which is of length equal to query sequence. If two elements T_i and T_j are consistent tuples in the query sequence and $i < j$ then $C[j]$ is assigned i. If T_j has no element T_i such that $i < j$ then $C[j]$ is assigned zero. This array can be generated in a single pass through the query sequence. Using this we can check the consistency of the subsequence while matching. one can see that tuples resulted from outputting same node will be consistent. For the example query taken above the C array will be (0,0,0,0,2). This indicates that element at position 5 should be consistent with element at position 2.

4. OPTIMIZING PARENT-CHILD AXIS

Occurrence of parent-child axis is more common in tree pattern queries when compared with other XPATH axes. We introduce a mechanism to optimize parent-child axis which results in reduced query processing time. To incorporate this optimization we introduce an extra field *parentPointer* in the Modified Prüfer Sequence, which is an offset from the current tuple to its Immediate parent tuple. The remaining fields in the tuple have their original meaning. The structure of the tuple in this case is given below.

($label$, $elementNum$, $level$, $count$, $parentPointer$)

parentPointer: parentPointer of tuple T_i is the difference in positions of parent(T_i) and T_i, where parent(T_i) is the Immediate parent tuple of T_i.

The sequence for the tree in Figure 1 with introduction of $parentPointer$ will be as follows.

(B, 2, 4, 1, 1) (E, 1, 3, 2, 1) (B, 1, 2, 3, 3) (C, 1, 3, 1, 1) (B, 1, 2, 2, 1) (A, 1, 1, 6, 0) (B, 3, 3, 1, 1) (C, 2, 2, 2, 1) (A, 1, 1, 3, 0) (A, 2, 4, 1, 1) (F, 1, 3, 2, 1) (D, 1, 2, 3, 4) (C, 3, 4, 1, 1) (B, 4, 3, 2, 1) (D, 1, 2, 3, 1) (A, 1, 1, 7, 0)

In the above sequence the fifth field in each tuple denotes the $parentPointer$ of that tuple. We assign zero to the $parentPointer$ of root.

By generating the sequence with this extra field, we index the sequence in a similar way described in the Section 3.2. The only change comes in the query processing, where in case of parent-child we don't retrieve tuples in a range but

try to check for the tuple in exact position by using the *parentPointer*. This could increase the index size to some extent, but overall the space complexity of the index will still be linear with the number of tuples in the sequence.

By introducing this extra field we can get the position of the parent of a tuple in the sequence. To solve a parent-child axis between two tuples Q_i and Q_{i+1} in the query sequence, we have to check if the key T_i+$T_i.parentPointer$ exists in the $Q_{i+1}.label$'s Element B^+-tree, where T_i is the tuple considered at stage i. If it exists we can continue the matching with that tuple otherwise we can stop the matching for that partially matched subsequence.

5. IMPLEMENTATION DETAILS

In this section we present the implementation details of the Query Processing system built using modified Prüfer sequences. The system has the following three basic blocks. Each of them is explained below.

Parsing: Sequence is generated using the SAX parser in one pass of the document as described in section 3.1.

Indexing: Indexing of the sequences is done in a way given in Section 3.2. We use the BerkeleyDB [5] library for implementing B^+-tree's.

Query Processing:
Algorithm 1 **findMatch** is used for solving tree pattern queries. The algorithm presented here is for evaluating tree pattern having only equality predicates on values. This procedure is invoked by calling **findMatch**(Q,1,0,MAXRANGE). Q_i denotes *ith* tuple in the query sequence Q. The function *retrieve(Q_i, database, start, end)* retrieves the tuples from the Element B^+-tree of $Q_i.label$ from the specified database in the range (*start, end*) whereas function *get(Q_i, k)* returns the tuple at the specified position k in the Element B^+-tree if it exists otherwise it returns null. Since we index the leaf values in the XML document separately to speed up the processing, query containing value based predicates should be handled separately. The lines from 1 to 7 in the algorithm are used for handling value nodes. The algorithm presented here handles only equality predicates and can be easily extended to allow other predicates also. Let the tuple preceding the current tuple being matched correspond to a leaf value in the XML tree, one can get the parent of the leaf node directly using the *get()* function, if it exists in that Element B^+-tree. The lines from 8 to 10 perform this task.

The remaining algorithm is for handling regular cases. In line 11 of the algorithm we get all the tuples matching ith tuple in the query sequence within the range (*start*, *end*) of the Element B^+-tree of $Q_i.label$. Line 14 to 16 check for connectedness and structural relationship as explained in Section 3.3.1. The function *checkRelationship()* takes two tuples as input and uses the level information encoded in them and returns true if the relationship specified in the query is satisfied. Lines 17 to 19 perform consistency check described in the Section 3.3.2. Throughout the Algorithm k denotes the position number of a sequence tuple and t denotes the structure *(tail, ElementNum, level, count)* of the tuple which is stored as the data field in the Element B^+-tree. We remove the duplicate tuples from the result of a range query at line 22. This task can be accomplished using the *elementNum* field in the tuples of a sequence. This along with the connectedness check ensures duplicate free results.

Algorithm 1 Algorithm for Pattern matching

Input:$\{(Q, i, start, end)\}$:Q is the query sequence; index i; $(start, end)$ is a range;
Output:$\{(P, D, id)\}$ P denotes Positions of matching; D is the data tuples matched; id is the document number;

Procedure:findMatch($Q, i, start, end$)
1: **if** $Q_i.relationship$ = 'v' **then**
2: $M \leftarrow$ retrieve($Q_i, leaf, start, end$);{for leaf nodes}
3: **for all** k in M **do**
4: findMatch($Q, i+1, k, k$);
5: **end for**
6: return;
7: **end if**
8: **if** $Q_{i-1}.relationship$ = 'v' **then**
9: $N \leftarrow$ get($Q_i, start$); {handling parent of leaf nodes}
10: **else**
11: $N \leftarrow$ retrieve($Q_i, document, start, end$);
12: **if** $Q_i.level < Q_{i-1}.level$ **then**
13: **for all** (k, t) in N **do**
14: **if** $not(k - P[i-1] < t.count$ and checkRelationship()=True) **then**
15: Remove (k, t) from N; {Connectedness check}
16: **end if**
17: **if** $(C[i] \neq 0$ and $t.elementNum \neq D[C[i]].elementNum)$ **then**
18: Remove (k, t) from N; {Consistency check}
19: **end if**
20: **end for**
21: **else**
22: Remove duplicates from N;
23: **end if**
24: **end if**
25: **for all** (k, t) in N **do**
26: $D[i] = t$; $P[i] = k$;
27: **if** $i = |Q|$ **then**
28: id = getDocId($k + t.tail$);
29: output(D, P, id);
30: **else**
31: findMatch($Q, i+1, k, k + t.tail$);
32: **end if**
33: **end for**
34: return;

The advantage of the new algorithm over the existing subsequence matching algorithm presented in [9] is that we do validation along with subsequence matching. This reduces the number of range queries as the invalid tuples are eliminated at each stage. This actually results in a cascading effect. If 10 tuples are eliminated at stage 2, then it will result in reduction of 10 range queries in stage 3(as they are removed from N) and number of range queries resulting from those tuple's in the next stages. Even if each of the 10 elements generate 5 invalid tuples in 3rd stage, that will come around to 50 range queries till that stage. When querying on large databases with large query sequences the gains are significant. As the range queries are expensive operations in the above algorithm, reducing them results in more efficient query processing. The effect is higher in case of data having deep recursive structure and large sequence size.

Table 1: Queries

	Query	Dataset
Q1	//article[./month="August"][./year="1994"]	DBLP
Q2	//mastersthesis[./author][./year]	DBLP
Q3	//inproceedings[./author="Jim Gray"][./year="1990"]	DBLP
Q4	//proceedings[./isbn][./url]	DBLP
Q5	//Entry[./Org="Piroplasmida"][.//Author]//from	SWISSPROT
Q6	//Entry[./Org="Theileria"][.//DB="MEDLINE"]	SWISSPROT
Q7	//Entry[.//Author="Smith A.G"]	SWISSPROT
Q8	//NP[./RBR_OR_JJR]/PP	TREEBANK
Q9	//NP/P/NP[./NNS_OR_NN][./NN]	TREEBANK
Q10	//S//NP/SYM	TREEBANK

6. EXPERIMENTAL RESULTS

We implemented our system in C++ for indexing and querying of XML data. We also implemented PRIX and obtained TwigstackXB from the authors for comparison purpose. We use the B^+-tree API provided by BerkeleyDB [5] for implementing the B^+-trees. All these experiments were carried out on a Linux machine running Red Hat 7.2 with a 2.4 GHz processor and 256 MB main memory. We used 4-byte number to index the Element B^+-tree. We fixed the page size of 8K in all our experiments.

Data Sets :
We carried out our experiments on three datasets DBLP, SWISSPROT and TREEBANK. All the Datasets are downloaded from the University of Washington XML repository[4]. The information about the datasets and the details associated with them are given in Table 2.

Table 2: Description of Data Sets

Name	Size	#Sequences	MaxDepth
DBLP	127MB	32858	6
SWISSPROT	109MB	50000	5
TREEBANK	82MB	56385	36

Queries :
We compared the performance of our system with PRIX and TwigstackXB for the queries given in Table 1. Some of the queries in the Table 2 are taken from [9]. Apart from these queries, we tested the system for several other queries and in all the cases MPS and MPS-P outperform other two systems.

6.1 Analysis of Results

6.1.1 Comparison With PRIX

In Table 3 time taken for processing different queries is given. Here MPS is the proposed Modified Prüfer Sequence based system and MPS-P is the MPS system with parent-child optimization. We observe that query processing for value based queries in all the cases takes lesser time in our system($Q_1, Q_3, Q_5, Q_6, Q_6, Q_7$). This gain is because of the reduction of excessive range queries in the findMatch procedure. Since PRIX has two phases where in one phase it gets all the subsequences matches from the database and then performs a document-wise post-processing to validate matched subsequences, its performance is poor. Even the validation process takes additional disk I/O's as the numbered Prüfer sequences are to be fetched from flat files that

Table 3: Comparison Of Query Processing Times in secs

Query	MPS	MPS-P	PRIX	TwigstackXB
Q1	0.030	0.020	0.098	0.38
Q2	0.018	0.016	0.014	0.68
Q3	0.035	0.019	0.11	1.01
Q4	0.20	0.10	0.15	0.41
Q5	0.043	0.043	0.70	2.5
Q6	0.019	0.019	0.050	0.54
Q7	0.014	0.014	0.055	0.43
Q8	0.016	0.013	0.11	0.68
Q9	0.023	0.017	0.26	0.33
Q10	0.016	0.012	0.044	0.20

are stored separately. Our observation shows that number of invalid results generated are more compared to the actual ones in the first phase of PRIX. In case of DBLP dataset similarity in the sequences is very high. PRIX gets the advantage of sharing of sequences to large extent in case of simple path expression which doesn't contain any value based predicates. These are processed using regular Prüfer index, whereas when it comes to value based queries PRIX uses *Extended Prf̈er index* and the amount of sharing is almost negligible because of the bottom-up approach used. Query Q2 and Q4 which are processed with regular Prüfer index got the advantage of sharing compared to our system. Query Q1 and Q3 are value based queries on DBLP dataset and we can see that MPS performs better than PRIX.

Performance of the PRIX degrades when we have large sequences with good recursive structure. Take the queries Q8, Q9 and Q10 on TREEBANK dataset. Since TREEBANK has deep recursion and large sequences compared to the DBLP, sharing of the sequence is less in this case. All these queries are not having any value predicate. The reduction of range queries in this case is very high as the number of intermediate tuples generated while querying are high because of the large sequence size and deep recursion. One can see that MPS has out performed PRIX in all these queries.

Queries Q5, Q6 and Q7 are value based queries on SWISSPROT database. The sequence sizes are large in this case. The number of intermediate nodes that are filtered in this is very high so we get the advantage from the reduced range queries at each stage of subsequence matching. Since the reduction of range queries has a cascading effect on the latter stages, the effect gets multiplied at each stage.

So we can say that for datasets having high similarity in structure, PRIX gets a slight advantage for processing sim-

ple path queries. This effect was countered in our system by the reduction of range queries. In all the other cases our system performs better than PRIX.

Problem of Duplicates with PRIX

When a query containing ancestor-descendant relationship is given to the PRIX system, it will generate duplicate results depending on the bushiness of the tree. The check given in the PRIX system for ancestor-descendant case will be valid for all the tuples corresponding to an element in the sequence resulting in duplicate results. The queries having ancestor-descendant axis and the number of matches generated in each case is tabulated in Table 4.

Table 4: Number of results generated in PRIX

Query	MPS	PRIX
Q5	158	2234
Q6	6	86
Q7	21	260
Q10	9	15

6.1.2 Comparison With TwigstackXB

For all the queries, our system performed better than TwigstackXB. TwigstackXB uses XB trees to store the element lists. Using XB trees one can skip the elements that do not participate in join operation. The amount of skipping in Twigstack depends on the distribution of the solutions over the database. The skipping is effective only in cases where the solutions are clustered at some place. TwigstckXB also suffers from sub-optimality in case of queries having parent-child queries. In which case solutions retrieved are checked for this relationship and are filtered in the next stage. This effect can be seen in Table 3 in case of such kind of queries. We observe that Our system MPS performs almost ten times better than TwigstackXB in almost all the cases.

6.1.3 Comparison of MPS and MPS-P

MPS-P has advantage over MPS in case of queries having more of parent-child relations. This can be observed in case of queries Q1, Q2, Q3, Q4, Q7 and Q8. In case of queries with ancestor-descendant axes we can see that there is no difference in the query processing times of both the system. Queries Q4, Q5 and Q6 demonstrate that. The difference in performance is less here because of the smaller sequence size of the databases. In case of databases having large sequence size and bushy structure, MPS-P will get the advantage of skipping the range queries. One can observe this in case of DBLP, which is bushy in structure. The difference is more though the sequence size is less in this case.

7. CONCLUSIONS

In this paper, we have proposed a new way of representing an XML document as a sequence that has one-to-one correspondence with the original document based on a variation of Prüfer sequences. We have also proposed indexing mechanism for these sequences and given the algorithm that processes twig queries holistically without requiring any post-processing. The proposed optimization mechanism for parent-child axis speeds up query processing. Experimental results show that our system performs better in most of the cases independent of characteristic of the data being queried.

8. REFERENCES

[1] A.Berglund, S.Bong, D.Chamberlin, 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] H.Jiang, H.Lu, W.Wang, and B.C.Ooi. XR-tree: Indexing XML data for efficient structural join. In *ICDE*, pages 253–264, March 2003.

[3] H.Prüfer. Neuer Beweis eines Satzes über Permutationen. *Archive für Mathematik and Physik, 27*, pages 142–144, 1918.

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

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

[6] N.Bruno, N.Koudas, and D.Srivastava. Holistic twig joins: Optimal XML pattern matching. In *ACM-SIGMOD*, pages 310–321, June 2002.

[7] 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/papers/mps.pdf.

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

[9] P. Rao and B. Moon. PRIX: Indexing And Querying XML Using Prüfer Sequence. In *ICDE*, pages 288–300, March 2004.

[10] S-Y.chein, Z.Vagena, D.Zhang, V.T.Tsotras, and C.Zaniolo. Efficient structural joins on indexed XML document. In *VLDB*, pages 263–274, Aug 2002.

[11] 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.

[12] S.Bong, D.Chamberlin, 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.

[13] 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.

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

[15] 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.