Designing and Querying a Compact Redundancy Free XML Storage

Radha Senthilkumar*, Priyaa Varshinee and A. Kannan

Department of Information Technology, MIT Campus of Anna University, Chennai, India

Abstract: XML, the universal data format for data exchange has seen phenomenal increase in database size necessitating the need for its compact storage coupled with simple accessibility. In previous works, all XML databases are implemented as a tree based structure which leads to increased space complexity. The proposed structure called RFX (Redundancy Free XML storage structure), addresses this issue by using a non tree based structure. This unique blend of hierarchical and relational databases in a single structure is largely effective in saving storage space thus achieving an increased compaction. Further RFX conceptualizes the separation of the information in the entire database as the topology layer, the tag layer and the data layer. The RFX structure has been designed to accommodate multiple document types i.e. Containment, Intra and Inter types of XML documents. Moreover in this proposal, it is ensured that the relationships among documents are never disturbed and are seamless with respect to their original counterparts. The document is parsed and stored in a different form to permit simple accessibility which is largely effective while querying and maintaining documents involving one-to-many relationships. Thus RFX paves way to effectively query and maintain the XML databases along with substantial compaction. This approach shows that the proposed RFX structure is space efficient, redundant free, and time efficient to update and to query Single, Intra and Inter structured XML document. The effectiveness of the approach makes it suitable for memory limited devices such as PDA. Performance evaluation over variety of XML documents and the user queries conform to the same.

INTRODUCTION

XML is fast becoming the standard format to store, exchange and publish over the web and is getting embedded in applications. The advantages of XML include that it can be used as an instrument to share data and application models in wide networks like internet and are platform independent. However the two main challenges in handling XML are its size and the complexity of search which involves path and content searches on labeled tree structures. The XML documents are highly verbose which attributes to their huge size. The verbosity also attributes to the redundancy in XML document.

This paper concentrates in overcoming these issues by eliminating the redundancies which results in reduced storage space while improving on query and update performance. The XML files are first analyzed for their relationships in the database (i.e) Containment, Intra or Inter document relationships. Subsequently, the XML files are stored conforming to the structural norms of the proposed RFX system. Following this, the XML file stored in RFX can be navigated and queried in near constant time. Additionally, non-tree based layered structure augments the capability of our system with respect to performance. Blending of layered approach with non tree based structure and complete removal of redundancy makes RFX structure edge over the others proposed so far [1-4]. RFX has thus achieved compaction coupled with effective querying and maintenance within a single system.

*Address correspondence to this author at the Department of Information Technology, MIT Campus of Anna University, Chennai, India;
E-mail: radhasenthil@annauniv.edu

In particular we address the problems of how to compress XML data, how to provide efficient access to its contents, how to navigate and how to support efficient update.

MOTIVATION

'Compact Storage' in its traditional sense involves the technique of space reduction of XML documents with no insight to the existing relationships in the database. Many works on compact storage have been proposed [3, 4] yet with no specific citations to one-to-many relationships. Changes that are consequences of these relationships have been disregarded. However in the present real world systems of distributed and very large databases, a centralized file without any one-to-many relationship is an impossible scenario.

'Query Optimizations' in XML documents involving one-to-many relationships are being worked on. Researches on 'Storage' of the database involving such relationships have been left unexplored. Storage systems like ISX [5], XMill [1], XGrind [4] have concentrated on effective storage of XML files but never speak about the adaptability of their structure to suit existing relationships in XML documents. However, our proposed compact structure can be used for XML documents having one-to-many relationships too.

OUR CONTRIBUTIONS

This paper proposes a Redundancy Free XML Storage, to store XML documents. Also it discusses about the one-to-many relationships that exist between XML documents and the importance to preserve those relationships. Experimental results conform that the RFX structure makes efficient usage of available space and stores XML documents efficiently, while preserving the relationship between the elements in the document and also the relationship between the XML docu-

1874-1339/09

ments. This paper also details the query and update efficiency achieved in RFX Compact Storage.

The proposed structure is extensively advantageous when existent relationships in XML documents should be left undisturbed. Although Query Optimization techniques [6] have been proposed in this area, research on storage schemes for documents with one-to-many relationships is yet at a nascent stage.

PRELIMINARIES

Representing arbitrary relationships between data items is a critical part of the data model. A standard mechanism for such representations is "modeling one-to-many relationships" in XML documents. RFX structure is designed to suit such relationships.

CONTAINMENT RELATIONSHIPS

In a Containment Relationship, a structure is defined where one element is contained within another. In the strongest form of this relationship, the "contained" element ceases to exist when the "container" element is removed. As given in the Fig. (**1**) the element "EDITOR" ceases to exist if the "TITLE" element is removed.

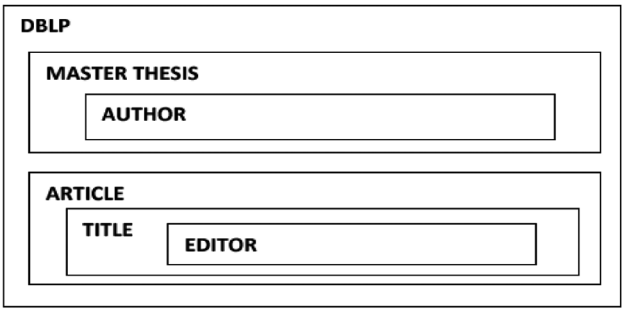

Fig. (1). Containment Relationships.

INTRA DOCUMENT RELATIONSHIPS

A dependency relationship between two entities within a single document represents an Intra document relationship. In a case where we have one element with many other related elements, rather than a first element containing second element, each second element will have a relationship to the first element. A first element "KEY" is used as a reference to the second element(KEYREF). For e.g rather than the author element being contained in the book element, book element is referenced by author element *via* "KEYREF" attribute as given in Fig. (**2**). This is very similar to a foreign key in a relational database.

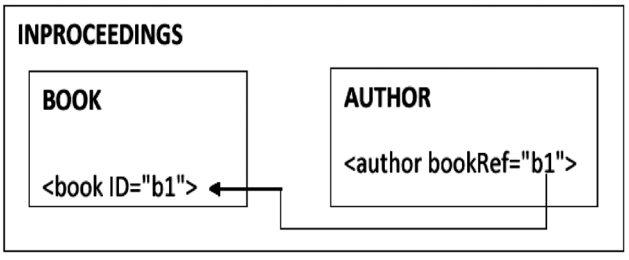

Fig. (2). Intra Document Relationship.

INTER DOCUMENT RELATIONSHIPS

The Inter-document relationship is much like the Intra-document relationship. It uses the Id and IdRef attributes to assign an attribute to a parent attribute. The difference is that in Inter-document relationship, the information spans over multiple XML documents and the documents are related by HREF or KEYREF attributes as given in the Fig. (**3**).

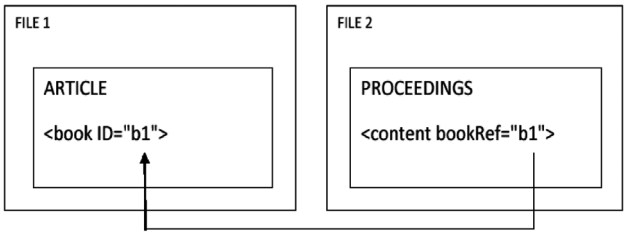

Fig. (3). Inter Document Relationship.

RFX ARCHITECTURE

The RFX storage is the compact storage which is used for storing the XML documents in a space efficient way. This is a multilayered architecture where the element and data are stored as separate layers and this facilitates the navigation and retrieval of data easily. The first layer is called as the topology layer, the second is the tag layer and the third one is the data layer. The overall design of the RFX system is given in the Fig. (**4**).

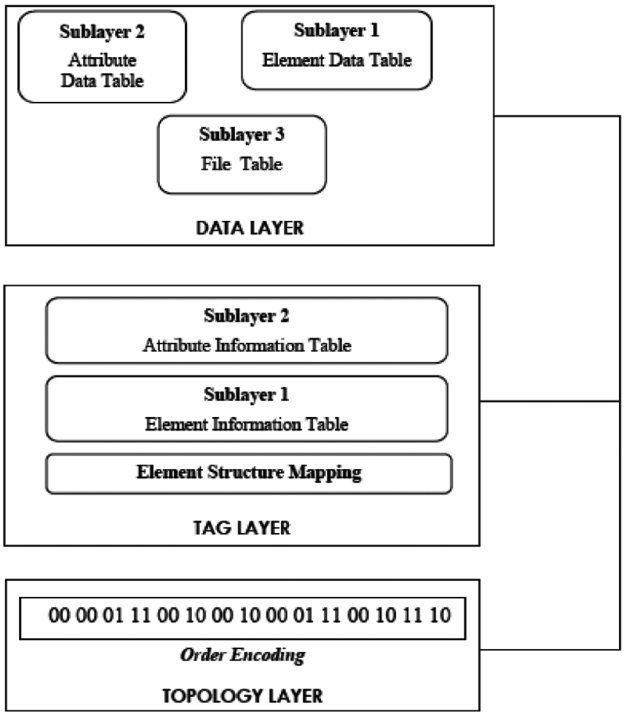

Fig. (4). RFX Architecture.

Topology Layer

This layer stores the order information of the XML Document using a novel approach. XML document since has

different levels of nesting can be modeled as a k-ary tree. We tend to store the order information of this k ary tree in theoretic minimum of 2n bits. Zhang [7] provided a succinct approach using balanced parenthesis encoding to store blocks of data. Similar to balanced parenthesis encoding, we use a novel approach to encode the order of the XML document rather than nesting information. This is called as order encoding. We discuss the advantages of doing so (encoding the order rather than nesting information) in coming sections. The nesting information of XML document is stored in layer 2 (RFX Tag Layer).

RFX topology layer uses an order array to encode the order of occurrence of elements, attributes and text data in the XML document. The order of occurrence is encoded using two bits. Hence to encode the order, we require only 2n bits, which indicates the encoding can be kept in main memory itself, thereby improving both query and update performance.The information encoded in the two bits are predicted as follows

00: Element Node
01: Attribute Node
10: Text Node of an Element
11: Text Node of an attribute.
Sample XML document:

```
<biblio>
<book id=1>
    <author>J.Austin</author>
    <title>Emma</title>
</book>
<book id=2>
    <author> C.B ronte</author>
    <title> Jane Eyre </title>
</book>
</books>
```

Order Array:
00 00 01 11 00 10 00 10 00 01 11 00 10 11 10

Hence the order of the XML document is encoded with only 2n bits which is the theoretic minimum.

TAG Layer

The TAG layer is partitioned into two sub layers, and also contains the element structure mapping, which connects the topology layer and the tag layer.

Element Structure Mapping

After the order has been encoded in the topology layer, the bit pairs in the order array have to be linked to the actual elements, attributes or text data in lower layers. Using a pointer based approach for this will increase the space usage from 2n bits to less desirable to $\Theta(n\log n)$.

To identify the actual nodes pointed by bit pairs of the order array, we make the element structure mapping an exact mirror copy of order array. Each bit of order array is represented by 4 bits in element structure mapping. Hence each node can be addressed by 2*4 = 8 bits. To find a node pointed by bit pairs in the order array, we've to search the element structure mapping in corresponding position for the node identifier. Actual labels of the node can be found from lower layers using the node identifier. Node identifier is used to map node labels of varying size into domain of fixed size.

Since RFX is intended to store schema less documents too, there is no way to calculate total number of unique elements and attributes prior to parsing of the document. Hence there is no way to pre identify the minimum cell size of Element Structure mapping which is log E, where E is the number of unique elements in the document. Hence we fix the cell size of element structure mapping to optimal 4 bits, which we found enough for existing XML documents as number of unique internal nodes in a XML document is very less. However, the cell size can be increased to satisfy future needs.

Fixing the cell size of element structure mapping as 4 bits limits the number of identifiers for text data to 256 (Two consecutive cells address a single node corresponding to a bit pair address a single node), in case of [5] n log E which we found as a severe drawback. To overcome this drawback, we use the approach used in [8]. We use multiple levels of indirection to increase the number of available identifiers to 1183907.We describe this approach in the following paragraph.

As in [8] we use blocks to provide various multiple levels of indirection however the number of blocks is 8. Note that these 8 blocks can be addressed by 3 bits. Each block is of size 32 bytes. Every cell of the element structure mapping that which corresponds to text data, points to one of the 8 blocks and also specifies the offset inside the block. The three higher order bits of the cell specifies the block address while the lower order five bits specifies the byte offset within the block.

Among the eight blocks, blocks 0,1 and 2 are direct blocks, 3 and 4 are single indirect blocks, 5 and 6 are double indirect blocks and 7 being triple indirect block. A single direct block contains 32 direct text label identifiers. Each single indirect block contains 32 direct blocks. Each double indirect block contains 32 single indirect block and so on. This approach can be extended to generate more ids by varying the block size and the number of indirect blocks.

Hence, the number of identifiers in each block is as follows

3 direct blocks = 2*32 = 96 identifiers (0 – 95)

2 single indirect blocks = 2*32*32= 2048 identifiers (96 - 2143)

2 double indirect blocks = 2 * 32 *32 *32 = 65536 identifiers (2144 – 67679)

1 triple indirect block = 1 * 32 * 32 * 32 * 32 = 1048576 identifiers (67680 - 1183906).

Note that all these blocks (672 bytes in total) too can be stored in main memory which results in fast resolution of the referenced text node.

Sub Layers

The first sub layer contain the element table to store information about the elements and the second part contains

the attribute table to store information about the attributes.Each row in the element table corresponds to an element of the XML document. Every row entry consists of five parts. The serial ID of the element and the name of the element are first and fifth parts respectively. The nesting information of every node in the XML document is stored as Level ID in the second part. If the nesting information is stored in topology layer then it results in wastage of O(log E) bits in element structure mapping for element or attribute present. This is evident in [5]. Hence we move the nesting information to layer 2.

For Inter relational documents, information is stored across the XML documents i.e., in more than one XML file. In such cases, one XML document refers to one or more XML documents. To keep track of such references, we introduce a column called file ID in element table.

For Containment and Intra relational documents, different elements with same name may be present at the same level. For e.g. In a university database, the name of the department as well as the name of the employees may occur at the same level, and they need not necessarily have the same parent element. In such cases, an XPath query to retrieve the name of the departments will also retrieve the name of the employees and vice versa. To resolve this ambiguity, we add a column named 'parent ID' in the element table. This resolves the 'same name-same level' problem as we use ID of the parent element to identify the correct child element needed.

Element ID	Level ID	File ID	parent ID	Element name

Fig. (5). Element Information Table.

Information about an attribute consists of three parts which are stored as a single row in the attribute table. The third part is the name of the attribute. The second part is the ID of the attribute and the first part specifies to which element the attribute belongs to, using the element ID.

Element ID	Attribute ID	Attribute name

Fig. (6). Attribute Information Table.

Data Layer

The Data layer is formed with three data tables. The data tables store text data. Leaf nodes of XML tree are stored in the data layer.

Fig. (**7**) shows the element as well as the attribute data tables containing three attributes. The first is the serial ID of the element or attribute to which the data belongs to and the second is the serial ID of the data value and the third value is the text data itself.

In the storage of Inter relational document, each XML document which is referenced from another XML document, is given a unique file ID. The file ID and the name of the

Element/ Attribute ID	Element /Attribute Data	Element/ Attribute Data

Fig. (7). Element/Attribute data table.

referenced XML document are stored in the file table (as given in Fig. **8**) of the referencing document. The file ID column acts as a map to the file table. Use of the file table to store the file name, reduces the complexity of querying the Inter relational document, else there would be no direct way to find the referenced document in the database, as the name of the XML document itself serves as the key to find the document in the database.

File number	File name

Fig. (8). File Table.

Thus the internal nodes of the XML tree are stored non-redundantly in the Tag Layer.

RFX IMPLEMENTATION

The implementation of RFX structure for maintenance of XML Database is described in Fig. (**9**). The major modules designed for the system implementation are parsing the XML document, separating the one to many relationships, separating the given document as tag, attribute and data, separating the data further into tag data and the attribute data and searching the tags, attributes and the data to eliminate redundancy and finally writing into the respective layers.

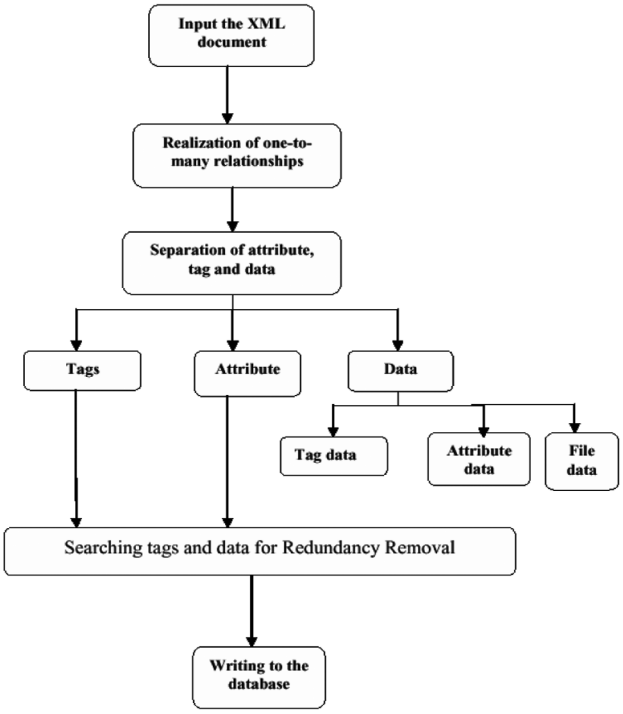

Fig. (9). RFX Implementation Flow Chart.

QUERYING METHODS

Querying is basically performed by searching the structure and searching the value. Searching the structure is done by using the order encoding. This finds the ancestor-descendant relationship in O(1) as the information is stored directly as level ID in the Element Information table. Similarly the parent child relationship can be found in O(1). Also the other frequently used step axes for an XML document such as next-sibling, previous-sibling, next-following and next-preceding can be determined easily from the order encoding and element structure mapping.

Basic Node Navigation Operations

This section gives algorithms for basic node navigation operations. Given an arbitrary node x in a large XML document, the basic node navigation operators described below in Figs. (**10-13**) enables the user to traverse back and forth the XML document efficiently.

PARENT(node)
1. if node exists
   a. return parent_id(node)
2. return error

Fig. (10). Algorithm to find parent node.

FIRSTCHILD(node)
1. if node exists
   a. Find position(next_element_node) in order array
   b. Find serial_id(node) at same position in Element structure mapping
   c. Find child node from Element information table using serial_id
   d. If (level_difference(node, child)==1)
      i. Return child
   e. Return error
2. return error

Fig. (11). Algorithm to find first child of a node.

NEXTSIBLING (node)
1. if node exists
   a. while(1)
      i. find position(next_element_node) in order array
      ii. find serial_id(node) at the same position in element structure mapping
      iii. find sibling node from element information table using serial_id
      iv. if(level_difference(node, sibling_found)==0)
         1. return sibling
2. return error

Fig. (12). Algorithm to find next sibling of a node.

Queries Identified

Any XPATH query can be categorized according to tree structure given in Fig. (**14**).

PREVIOUSIBLING(node)
1. if node exists
   a. while(1)
      i. find position (previous_element_node) in order array
      ii. find serial_id(node) at the same position in Element structure mapping
      iii. find sibling node from Element Information table using the serial_id
      iv. if(level_difference(node,sibling_found)==0)
         1. return sibling
2. return error

Fig. (13). Algorithm to find previous sibling of a node.

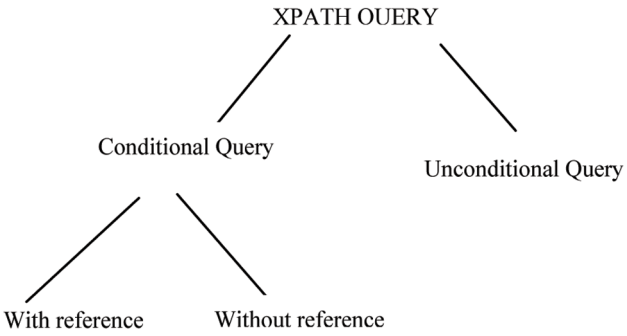

Fig. (14). Querying types.

Conditional queries are the queries that traverse the tree structure of XML documents checking for the condition given by the user and returning the values of those nodes that satisfy the specified conditions. These types of queries are executed by the method of search by value. The query execution path is guided by certain values retrieved according to the condition.

In unconditional queries, the user does not specify any condition(s). These types of queries follow the mechanism of search by reference.

eg. /employees/employee/department/Name

Conditional queries with reference are queries that refer to nodes of other scope (intra relational documents) or those which refer to nodes in another document (inter relational documents).

eg.//Department[@id = current()/@DepartmentRef]

Conditional Queries without reference does not navigate to the nodes of other documents. The XPATH query returns results from the same XML document. Such type of queries are the conditional queries of containment documents

eg./departments/department[Name="Enterprise Development"]
eg. /employees/employee/department

Search by Reference (Unconditional Query):

For unconditional queries, for each node in XPATH query, the ancestor-descendant relationship is checked with

previous node. And the values needed are retrieved at the end of the query. This is known as search by reference.

SEARCH BY REFERENCE

1. While (there are more nodes in XPath Query)
   a. Get Current node
   b. Retrieve the Current node's row from Element /Attribute table (tag layer)
   c. Validate ancestor descendant relationship
   d. if (ancestor descendant relationship not preserve)
      i. Invalid query, exit
   e. else
      i. if(!last node of the query)
         1. continue
      ii. Find the position of retrieved ID in the element Structure mapping
2. Filter the selected positions using order encoding
3. Print nodes that are descendent of filtered IDs
4. Exit

Search by Value (Conditional Query)

For conditional queries, values are retrieved according to the given condition. After a condition is encountered in the query, further values are retrieved only according to the conditions. This is known as search by value.

SEARCH BY VALUE

1. While (there are more nodes in XPath Query)
   a. Get Current node
   b. Retrieve the Current node's row from Element/Attribute table (Tag layer)
   c. Validate ancestor descendant relationship
   d. if(ancestor descendant relationship not preserved)
      i. Invalid query, exit
   e. if (condition encountered)
      i. Retrieve data ID satisfying the condition (Data Layer)
   f. Retrieve the positions of the selected ID in the element structure Mapping
   g. Filter the positions using order encoding
   h. if (condition encountered previously)
      i. Filter the selected positions with position retrieved by previous condition
   i. if (last node of the query)
      i. Print the nodes that are descendants of nodes in selected positions (Element structure mapping)
2. Exit

/* General Querying method for Intra documents in RFX Compact Storage */

For intra relational documents, we retrieve parent ID of each element. In addition of checking ancestor descendant relationships between consecutive nodes of the query, we check if elements are in same scope by comparing the parent IDs.

/* General Querying method for Inter documents in RFX Compact Storage */

For inter relational documents, to navigate to nodes of another XML document, we use file ID of that element. When an element present in current document refers to an other document, then file ID of referenced document is got from Element table and the corresponding file name is got from file table.

INTRA DOCUMENT QUERYING

1. For each node in the XPATH query
   a. Retrieve parent ID, level ID and element/attribute ID(s)(Tag layer)
   b. Check for parent-child/ancestor-descendant relationships
   c. if (ancestor-descendant relationship not preserved)
      i. Invalid query , exit
   d. else
      i. Select the correct element ID by using parent ID(Tag layer)
      ii. if(attribute)
         1. Check if element ID of previous element and element ID of the attribute in Attribute table are same (Data layer)
2. Use search by value/search by reference to return results.
3. Exit

INTER DOCUMENT QUERYING

1. for each node in XPATH query
   a. Retrieve file ID, level ID and element ID(s), parent ID(Tag layer)
   b. if(file ID not null)
      i. Get file name of referenced file from file table using file ID(Tag layer)
2. Use search by value/search by reference technique to retrieve data values from the referenced file (Data layer)
3. Exit

UPDATE MAINTENANCE

Insertion and deletion of nodes in XML document requires the modification of topology layer as well as the Tag Layer or the text data layer as necessary. In static representation of a tree , if we insert or delete a node, we must build the sequence from the scratch. Instead to allow efficient modification of the XML tree in the topology layer, we incrementally divide the XML tree into disjoint blocks as in previous approaches [5, 9-12].

Lemma 1: There exists a dynamic representation for a dynamic sequence of 2n balanced parenthesis using 2n+o(n) bits of space and supporting operations select, insert , and delete in O(log n) worst-case time.

The order array is divided into blocks with each block representing N nodes of the XML tree, where $N_{min} \leq N \leq N_{max}$. N_{min} and N_{max} are minimum and maximum block sizes respectively. The blocks are connected by inter block pointers. Since inter block pointers should require only o(n) bits overall, the minimum block size is $\Theta(\log^2 n)$ [12]. N_{max} is fixed as $|\beta|$ bits.β is selected such that when we insert a node to a block of maximal size , we can split the block into two blocks , each of size atleast N_{min}. Hence $|\beta|$ is multiple of $\Theta(\log^2 n)$, hence $\Theta(k\log^2 n)$.

When we insert a new node x in block p, the bits (note that each node is represented by two bits in the encoding) at the position of insertion and the bits at the right of the position are shifted two places towards right uniformly. The node is inserted in the resulting empty space. Element Structure mapping and the tables in lower layers are modified accordingly. A new id is generated and the data is inserted into the Element Table if the node is an element or into the attribute table if the node is an attribute or into the appropriate text data tables if its leaf node.

PERFORMANCE ANALYSIS AND RESULTS

The performance analysis for this paper can be done in two ways:

One based on the **storage space** required for storing an XML document. Another based on the **query performance**. Yet the space requirement will have the upper hand since this paper focuses mainly on the compact storage of the given XML document in RFX structure.

STORAGE SPACE

The graph in Fig. (**15**) shows the impact of lesser storage space of the documents with respect to increase in document size as given Table **1**.

Table 1. Storage Size of RFX, ISX, XMill and Xgrind

Benchmark Database	Source Data (MB)	RFX (MB)	ISX (MB)	XMill (MB)	XGrind (MB)
Mondial	1.7	1	1	0.3	0.6
Orders	5.1	3	3	0.5	1.3
Shakespeare	7.5	5.1	5.3	0.9	2.1
EXI- Telecom	10.2	7.8	8	1.2	3.8
Lineitem	30.8	15.8	21	3.7	8.6
DBLP	127.2	73.4	87	14.9	35.8

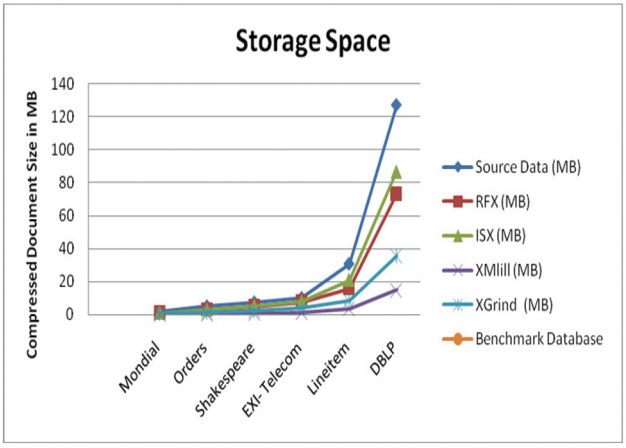

Fig. (15). Storage Space Comparisons.

Thus, we observe that the Space Complexity of XML document has dramatically reduced in RFX Compact Storage.

Execution Time

The query has been executed on system with 1.6 GHz Intel Pentium® processor, 512 MB RAM and 80 GB hard disk.

Query Performance

We have carefully chosen conditional queries and unconditional queries of various complexities. The table below gives specifies the queries selected in comparison with increasing storage space. The objective of gradual increase in execution time with increase in storage space is achieved here.

Table 2. Queries

Query	XPATH EXPRESSION
Q1	Mondial/country/name
Q2	Mondial/country[@id="f0_149"]/name
Q3	/student[id=/exam[grade<'B'/id or semester>5]/name
Q4	/student[id=/exam[grade<'B'/id and semester>5]/name

Q1 and Q2 are Containment relationship queries, Q3 and Q4 are Intra and Inter document relationships queries respectively.

Tables **3**,**4**,**5** and **6** give the execution times of queries 1,2,3 and 4 in ISX , NoK and RFX compact storage.

Table 3. Performance Results for Query 1

	1MB	16MB	64MB	128MB
ISX	0.001	0.021	0.13	0.85
NOK	0.005	0.015	0.76	1.25
RFX	0.001	0.013	0.087	0.21

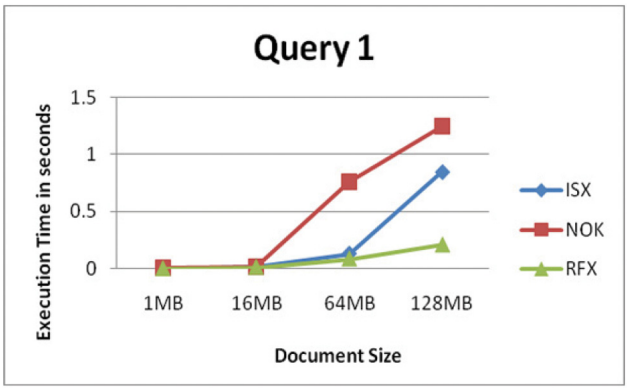

Fig. (16). Execution time comparison for query 1.

Table 4. Performance Results for Query 2

	1MB	16MB	64MB	128MB
ISX	0.02	0.5	3.52	7.54
NOK	0.015	1.03	5.02	10.36
RFX	0.01	0.235	2.003	5.47

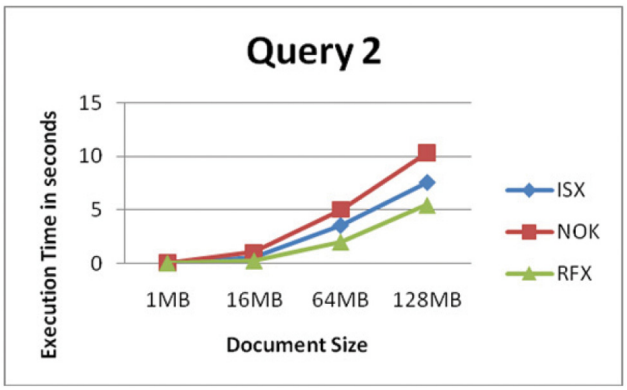

Fig. (17). Execution time comparison for query 2.

Table 5. Performance Results for Query 3

	1MB	16MB	64MB	128MB
ISX	NA	NA	NA	NA
NOK	NA	NA	NA	NA
RFX	0.023	0.35	2.74	6.63

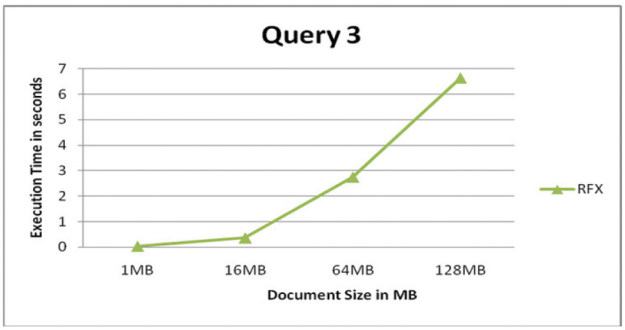

Fig. (18). Execution time comparison for query 3.

As ISX and NoK does not support intra and inter document relationships, such queries do not apply to both the storage schemes. Hence, the graphs for both query 3 and query 4 show query execution line of RFX compact storage only.

Table 6. Performance Results for Query 4

	1MB	16MB	64MB	128MB
ISX	NA	NA	NA	NA
NOK	NA	NA	NA	NA
RFX	0.026	0.41	3.38	6.86

The proposed storage technique is proved to be efficient. This is corroborated well by experimental results.

RELATED WORK

In this section, we briefly review some related work and compare our approach with those as reported in the literature.

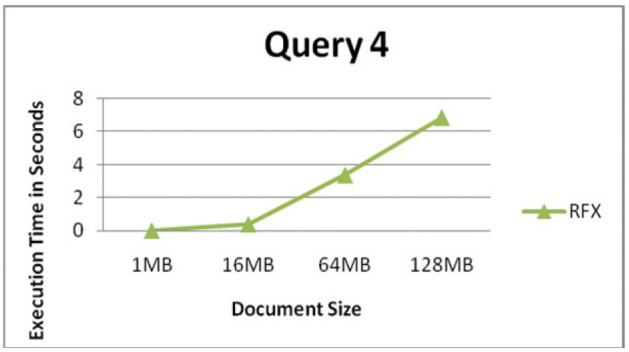

Fig. (19). Execution time comparison for query 4.

In recent years, many storage schemas for XML data have been proposed i.e. mapping XML data to relational data [13] or object relational models [14]. A key issue that faces every XML data management system is the efficient use of storage space and optimization of XML queries.

Although Artem Chebotko *et al*. [15] aimed in constructing trees for XML documents for native XML databases, they did not address the reconstruction problem

Previous work on XML publishing [4, 16-18] focuses on publishing existing relational data as XML documents. We are interested in efficient storage and compression of XML documents combining the best practices of both hierarchical and relational storage systems.

XML compressors can be classified into main groups – Queriable and Non-Queriable. Notable among the non-queriable compressors are Xmill [1], XMLPPM [19], Millau [20] and SCA [21]. Xmill is the first proposed XML conscious compression architecture which was proposed by Liefke and Suciu [1]. XMill achieves good compression ratio and does not require DTD to compress the XML document. But its main drawback of requiring decompression of the whole document before querying hinders its wide usage. XMLPPM [19] achieves better compression ratio than XMill(default mode) but takes longer time to compress the document (since PPM is relatively slow compression technology). Millau [20] is a system that uses a set of compression and encoding techniques that are dedicated for XML compression. But it's a DTD aware compressor and also consumes large memory when compressing large XML documents.SCA also suffers from the same problem as Millau.

Xgrind [4], Xpress [22], XMLZip [23], XML Skeleton Compression [24], XQuec [25], XQZip [26], XCQ [27] are some of the queriable XML Compressors. Xgrind [4], the first of its type, though has lower compression ratio than that of XMill and longer compression time, it supports certain types of queries and retains the document structure. Both Xgrind [4] and Xpress [22] requires two scans over the original XML document. This attributes to low compression speed. Also both the techniques does not support set based query evaluation such as join queries. XMLZip [23] compresses XML documents that are represented as DOM trees. However, the node level grouping strategy in XMLZip does not improve compression efficiency. The advantage of XMLZip is that it allows limited random access to partially

decompressed XML documents. XML Skeleton Compression [24] extracts and compresses the document structure using a technique based on the idea of sharing common subtrees. It supports efficient querying as most of the query operations take place in main memory. But, node navigation time for XML Skeleton Compression is linear.

Like Xpress and XGrind, XQuec compresses individual data items. However in contrast to these two, it seperates XML structure from XML data itmes. Although XQuec supports large subsets of queries, it makes insufficient use of commonality of XML data and also the auxiliary data structures used incur huge space overhead. Though XQZip overcomes the drawbacks of XQuec, it does not support evaluation of complex queries such as joins and order based predicates. XCQ is a schema aware XML compressor which can process only valid XML documents and requires longer compression and decompression times.

[30] and [31]Shows efficient labeling schemes using preorder post-order labeling and sector based labeling respectively for efficient XPath axes access. However they do not address any issues related to compression and updating of XML documents

Succinct representation of dynamic binary trees has been studied by Munro *et al.* [10] and Raman [11]. [11] shows that such dynamic binary trees can be represented in 2n+ o(n) bits . Transforming dynamic k-ary trees to binary form yielded poor results. Hence Munro [10] posed this as an open problem to represent dynamic k ary trees succinctly. Chan *et al.* [28] and Raman [11] provided succinct representations for dynamic k-ary trees. Raymond *et al.* proposed [5] a new compact XML storage engine, called ISX, to store XML in a more concise structure. Theoretically, ISX uses an amount of space near the information theoretic minimum on random trees. But to the best of our knowledge ISX stores only the valid XML document, which means that it is a schema aware storage system. The succinct approach proposed by Zhang *et al.* [7] used balanced parenthesis encoding for each block of data. The major difference between our proposal and the above works is that we try to minimize space usage by eliminating redundancy while allowing efficient access, query and update of the database.

Ferragina *et al.* [29] first shreds the XML tree into a table of two columns, then sort and compress the columns individually. It does not offer immediate capability of navigating or searching XML data unless an extra index is built. However, the extra index will degrade the overall storage size (i.e., the compression ratio). RFX Compact Storage facilitates the capability of navigating and searching the data without the aid of any extra index.

Comparison of Features with other Storage Systems

The following table illustrates a thorough comparison between various storage systems for XML Databases with RFX Structure.

This paper mainly concentrates on the design of the storage scheme for XML Databases that involve one-to-many relationships. However this work can be extended and enhanced on the grounds of security and access rights for the concerned database. Further, compression techniques can also be analyzed to be incorporated in RFX. These enhancements can play a vital role in molding RFX as a fully-fledged storage scheme for XML databases.

Features	XMILL	XGRIND	NOK	ISX	RFX
Compression	YES	YES	NO	YES	NO
Document traversal	YES	YES	YES	YES	YES
Node navigation	NO	NO	UNCERTAIN	YES	YES
Update operation	NO	NO	NO	YES	YES
Support for XPath queries	NO	NO	NO	YES	YES
Inbuilt indexing	NO	NO	NO	NO	YES

CONCLUSIONS

A compact and efficient XML repository is critical for a wide range of applications including applications in memory limited mobile devices. Even in heavily loaded system, the topology layer and element structure mapping of tag layer could be stored entirely in main memory and hence substantially improve the overall performance. Besides having a compact storage, the need of the hour is the redundancy removal for which we have provided two layers the tag layer and data layer that removes the redundancy of tags and data from the original XML document. The database is scalable storing multiple documents having one to many relationships. The proposal designed here is effective and overshadows the advantages of all the previously proposed works.

REFERENCES

[1] H. Liefke and D. Suciu, "XMill: an efficient compressor for XML data," in *ACM SIGMOD international conference on management of data pages*, 2000, pp. 153-24.

[2] J. Shanmugasundaram, E. Shekita, R. Barr, M. Carey, B. Lindsay, H. Pirahesh and B. Reinwald, "Efficiently publishing relational data as XML Documents," in *Proc. of the VLDB Conference*, 2000.

[3] J. Katajainen and E. Makinen. Tree compression and optimization with applications. *International Journal of Foundations of Computer Science (FOCS)* IEEE Computer Society, 1990, Vol. 1, pp. 425-447.

[4] P. Tolani and J. Haritsa, "XGRIND: A query-friendly XML compressor," in *18th International Conference on Data Engineering (ICDE)* IEEE Computer Society, 2002, pp. 225-234.

[5] R. Wong, F. Lam and W. Shui, "Querying and maintaining a compact XML storage," in *16thinternational conference on World Wide Web*, Banff, Alberta, Canada, 2007.

[6] R. Senthilkumar, A. Kannan, V. Prasanna and P. Hindumathi, "Query Optimization for Intra Document Relationships in XML Structured Document," presented at the *1st international conference on Advances in Computing*, Maharashtra, India, 2008.

[7] N. Zhang, V. Kacholia, and M. Ozsu, "A Succinct Physical Storage Scheme for Efficient Evaluation of Path Queries in XML," in *20th International Conference on Data Engineering (ICDE)* IEEE Computer Society, 2004, pp. 54-65.

[8] M.J. Bach, *The Design of Unix Operating System*, India, Prentice Hall of India: Private Limited, 15th printing, 2007, pp. 67-72.

[9] D. Arroyuelo and G. Navarro, "Space-efficient construction of LZ-index," in *Proceedings ISAAC, LNCS 3827*, 2005, pp. 1143-1152.

[10] J. Munro, V. Raman and A. Storm, "Representing dynamic binary trees succinctly," in *Proceedings of SODA*, 2001, pp. 529-536.

[11] R. Raman and S. Rao, "Succinct dynamic dictionaries and trees," in *Proceedings of ICALP. LNCS 2719*, 2003, pp. 357-368.

[12] D. Arroyuelo, "An Improved Succinct Representation for Dynamic K-ary Trees," CPM, Pisa Italy. pp. 277-289, 2008.

[13] D. Florescu, D. Kossmann, "Storing and querying XML data using an RDBMS," *IEEE Data Engineering Bulletin*, Vol. 22, No. 3, pp. 27-34, 1999.

[14] T. Shimura, M. Yoshikawa and S. Uemura, "Storage and Retrieval of XML Documents using Object-Relational Databases," in *Proc. Of the 10th International Conference on Database and Expert Systems Applications (DEXA'99)*, Springer-Verlag, *Lecture Notes in Computer Science*, 1999, Vol. 1677, pp.206-217,

[15] A. Chebotko, M. Atay, S. Lu and F. Fotouhi, "XML sub tree reconstruction from relational storage of XML documents," Data & Knowledge Enginreeing, Vol. 62, No. 2, pp. 199-218, 2007.

[16] M. Fernandez, A. Morishima and D. Suciu, "Efficient Evaluation of XML Middle-ware Queries," presented at the *ACM SIGMOD Conference on the Management of Data*, Santa Barbara, California, 2001.

[17] J. Shanmugasundaram, J. Kiernan, E. Shekita, C. Fan and J. Funderburk, "Querying XML Views of relational data," in *Proc. of the VLDB Conference*, 2001, pp. 204-215.

[18] M. Fernandez, A. Morishima and W. Tan, "SilkRoute: Trading Between relations and XML," presented at the *9th World Wide Web Conference*, Amsterdam, 2000.

[19] J. Cheney, "Compressing XML with multiplexed hierarchical PPM models," in *Proceedings of the IEEE Data Compression Conference*, 2000, pp. 163-172.

[20] M. Girardot and N. Sundaresan, "Millau: An encoding format for efficient representation and exchange of XML over the Web," in *Proceedings of the 9th International WWW Conference*, 2000, pp. 747– 765.

[21] M. Levene and P. Wood, "XML structure compression," in *Proceedings of the Second International Workshop on Web Dynamics*, 2002.

[22] J. Min, M. Park and C. Chung, "XPRESS: A queriable compression for XML data," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, San Diego, California, 2003.

[23] XMLSolutions Releases XMLZip - Leading XML File Compression Tool. http://www.w3.org/2003/08/binary-interchange-workshop/31-oracle-BinaryXML_pos.htm [Accessed: 29th Jan. 2009].

[24] B. Grohe and C. Koch, "Path queries on compressed XML," in *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB'03)*, Berlin, Germany, 2003.

[25] A. Arion, A. Bonifati, G. Costa, S. D'Aguanno, I. Manolescu, and A. Pugliese, "XQueC: Pushing queries to compressed XML data," in *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB'03)*, 2003.

[26] J. Cheng and W. Ng, "XQzip: Querying compressed XML using structural indexing," *EDBT*, 2004, pp. 219-236.

[27] W. Y. Lam, W. Ng, P. T. Wood, and M. Levene, "XCQ: XML Compression and querying system," in *Poster Proceedings, 12th International World-Wide Web Conference*, 2003.

[28] H.L. Chan, W.K. Hon, T.W. Lam and K. Sadakane, "Compressed indexes for dynamic text collections." *ACM TALG*, vol. 3(2), article 21, 2007.

[29] P. Ferragina, F. Luccio, G. Manzini and S. Muthukrishnan, "Compressing and searching XML data via two zips," In *Proceedings of the 15th international conference on World Wide Web*, 2006, pp. 751–760.

[30] I. Tatarinov, "Storing and Querying Ordered XML Using a Relational Database System," presented at *ACM SIGMOD,'2002*, Madison, Wisconsin, USA, 2002.

[31] R. Thoangi, "A Concise Labeling Scheme for XML Data," presented at the *International Conference on Management of Data, COMAD*, Delhi, India, 2006.

Received: February 15, 2009 Revised: May 25, 2009 Accepted: June 03, 2009