World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:2, No:10, 2008

Query Optimization Techniques for XML Databases

Su Cheng Haw, and G. S. V. Radha Krishna Rao

***Abstract*—**Over the past few years, XML (eXtensible Mark-up Language) has emerged as the standard for information representation and data exchange over the Internet. This paper provides a kick-start for new researches venturing in XML databases field. We survey the storage representation for XML document, review the XML query processing and optimization techniques with respect to the particular storage instance. Various optimization technologies have been developed to solve the query retrieval and updating problems. Towards the later year, most researchers proposed hybrid optimization techniques. Hybrid system opens the possibility of covering each technology's weakness by its strengths. This paper reviews the advantages and limitations of optimization techniques.

***Keywords*—**indexing, labeling scheme, query optimization, XML storage.

I. Introduction

XML (eXtensible Mark-up Language) was primarily used as a human consumable exchange format. However, the amount of exchanged XML data often grows exponentially via the Web medium. Web application such as search engine, e-commerce, e-learning portals require advanced tools for managing the data. Communities no longer use the search engine just to retrieve the full-text queries but also request for more specific data (structure queries). This drives the requirement for storing and querying large-scale XML data as efficient and reliable as possible. Several XML query languages have been proposed such as Lorel [1], Quilt [2], XML-GL [3], XPath [4], XQuery [5], XOM [6], XAL [7] and YATL [8]. These query languages utilize regular path expression, thus using the conventional approach such as tree traversal may have performance degrade especially on concurrent access.

This paper is mainly divided into three sections: XML data model, XML storage, indexing techniques and labeling techniques. Section II presents the XML structure and data model. Section III describes the possible alternatives to store XML. Being a semistructured data, there are three possible ways. Section IV presents the optimization techniques via indexing and labeling scheme techniques respectively. Section V wrap-ups and suggest hybrid optimization techniques by comparing the advantages and disadvantages of these techniques. However, the mathematical and theoretical proofs are omitted in this paper.

Manuscript received January 28, 2005.

Su Cheng Haw is with Department of Information Technology, Multimedia University, Malaysia. (phone: 603-83125410; fax: 603-83125264; e-mail: schaw@mmu.edu.my).

G. S. V. Radha Krishna Rao is with Department of Information Technology, Multimedia University, Malaysia (email : gsvradha@mmu.edu.my).

II. XML Structure and Data Model

In this paper, object exchange model (OEM) is used to represent data in XML. Data in the OEM is represented as an edge-labeled graph. XML elements are represented by node while element-subelement, and element-attribute are represented by the edge labeled with corresponding names. The values of XML data are represented by leaf node in the OEM. Fig. 1 shows an XML document and its corresponding OEM representation. We assume that no element has attributes other than the attribute *ID* and *IDREF* [9]. Thus, it may need restructuring if there exists attributes other than *ID* and *IDREF* for easier maintenance and reading. We replace the other attribute as child element of the respective element. For example, referring to Fig. 1, the attribute title will be replaced by a child element title of the element category.

III. XML Storage

There are three main alternatives to store XML document. The first approach is to develop a specialized data management system, which is also known as native XML database (NXD). The second approach is to employ object-oriented database management system (OODBMS) as the underlying database while the third approach is to map XML data into relational database management system (RDBMS).

Each of these approaches has its own advantages and limitations. The first approach may work best, especially on scalability and handling huge amount of data. These data can be stored and retrieved in their original structure, with no mapping process require. Nevertheless, it is not suitable especially when integration within heterogeneous XML documents is needed. The second approach uses class inheritance to protect database integrity and able to support more complex relationships. However, they are vulnerable especially of evaluating query of a very large database. The third approach provides maturity, scalability, portability and stability [10]. Since majority of the data on the web are currently resides in RDBMS, the third approach seemed more viable. Therefore, in this paper, we will be focusing only on NXD and RDBMS as the instance storage.

```
<library>
    <book id = "B001-05">
        <category  title= "Computer Concepts 2005">
                <author>Gary Smith </author>
                <author>Robert Wood </author>
                <publisher>Thomson Learning</publisher>
          </category>
    </book>
    <book id= "B002-02">
        <category  title= "Introduction to Calculus"/>
    </book>
    <journal id= "J001-05" >
          <author>S. C. Haw </author>
    </journal>
</librarv>
```

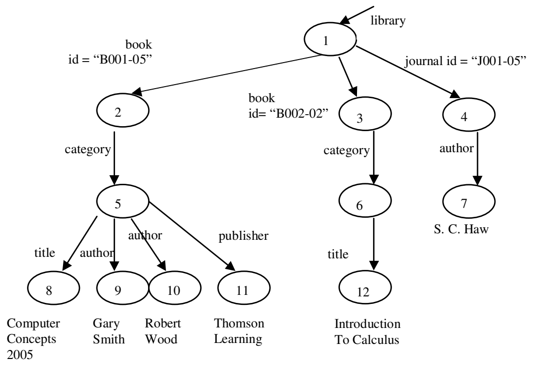


Fig. 1 A small XML document and its OEM representation

The main challenge of using RDBMS as instance storage is that, one needs to resolve the conflict between the hierarchical nature of XML data model and the two-level nature (row and column) of relational data model [11]. There are two ways of doing so : (1) using the graph-based approach and (2) by inferring schema from Document Type Definition (DTD) [12].

(1) Using the graph-based approach

This is the simplest approach. Instead of generating relational tables for each XML element, several XML elements are combined into a single table named *edge* to reduce the number of join operations incurred while querying the data. The *edge* table stores the object identifiers (oids) of the source and target objects of each edge, the label of the edge, the ordering of the edge and a flag to shows whether the edge is a leaf or non-leaf node. If the node is a leaf, then a corresponding record will be stored in the *value* table. This table has the field of vid (storing oids of values) and value of the string. Fig. 2 shows some fragment of respective edge and value table based on Fig. 1.

Edge Table

Source	Target	Label	Order	Flag
Root	1	library	1	Ref
1	2	book	1	Ref
1	3	book	2	Ref
5	v8	title	1	String
5	v9	author	2	String

Value Table

vid	Value
v8	Computer Concepts 2005
v9	Gary Smith

Fig. 2 Fragment of Edge and Value Table

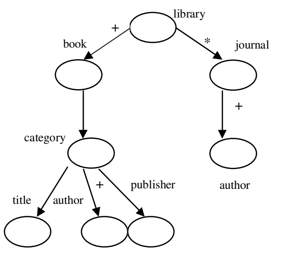
(2) Storing XML from a schema

Another way to map XML into RDBMS is to derive a relational schema from either a XML schema or DTD. This technique is not applicable for XML without a schema. This limitation, however, has been overcome [13].

According to Garafalakis M. et al, a DTD graph $G_D = (V,E)$ is a graph where V is a set of nodes, i.e. elements, attributes and operators. $E \subseteq V \times V$ is a set of edges representing relationships between nodes.

Below we extract some of their keys proposition:-

(1) Relations are created for element nodes that have an in-degree of zero. Otherwise, the element cannot be reached.
(2) Elements below a '*' or a '+' node are made into separate relations. This is necessary for creating a new relation for a set-valued child.
(3) Nodes with an in-degree of one are inlined.
(4) Among mutually recursive elements all having in-degree one, one of them is made into a separate relation.
(5) Element nodes having an in-degree greater than one are made into separate relations.



library (libraryID : integer, library.ParentID : integer, library.ParentCode : integer)
book (bookID : integer, book.ParentID : integer, book.ParentCode : integer, book.category : string, book.category.title : string, book.category.publisher : string)
book_author (book_authorID : integer, book_author.ParentID : integer, book_author.ParentCode : integer)
journal (journalID : integer, journal.ParentID : integer, journal.ParentCode : integer)
journal_author (journal_authorID : integer, journal_author.ParentID : integer, journal_author.ParentCode : integer)

(a) (b)

Fig. 3(a) DTD graph derived from Fig. 1 (b) Relational schema

A DTD graph is created in Fig. 3(a) based on XML document in Fig. 1. Based on Fig. 3(a) and Garofalakis M. et al's proposition, below are the extractions:

(1) library
(2) book, journal, book_author, journal_author
(3) book, category, title, book_author, publisher, journal, journal_author
(4) & (5) Not applicable as in Fig. 3(a)

Traversing down from library on the left-side, we have '+' edge follows by book element. Traversing further down, we reach category element before reaching to title, author and publisher element. All elements and attributes nested within category occurs at most once, except the author elements. Hence, we can store category, title, publisher in the same relation as book. The resulting relational schema is shown in Fig. 3(b).

In each relation, the *ID* field serves as the primary key, while the *parentID* field serves as the foreign key to match with the values in the primary key. For example, in the relation journal, it has *journal.parentID* that joins journal with library.

A NXD defines a logical model of an XML document. It does not require any particular underlying physical storage model. Thus, it can be built on a RDBMS, OODBMS, hierarchical DBMS, proprietary model, or file storage [14]. Basically, a NXD fall into two main categories: (1) text-based storage and (2) model-based storage.

(1) text-based storage
This will store the entire XML document in text form and provide some database transaction support (indexing, materialized view, etc) in accessing the document.

(2) model-based storage
This method model the XML document as the Document Object Model (DOM) and map the DOM to relational tables such as Entities, Elements, Values, Attributes, Levels, etc.

IV. Query Optimization Techniques

There are many query optimization techniques discovered and implemented by researchers. Among them are path traversal, use of indexes, use of materialized views, pipeline evaluation, structured join order selection, schema-based optimization, reformulation of XML constraints, duplicate removal, tree pattern matching and labeling scheme. In this paper, we will review the path traversal, uses of indexes and labeling scheme techniques.

A. Path Traversal

McHugh and Widom proposed three approaches that process the path traversal: top-down, bottom-up and hybrid traversal [15]. Top-down approach starts the traversal from root to each node along each path, and pick out the target nodes matching the path expression. It requires search each children of a node to find a match each time. Conversely, bottom-up approach begins traversing from the bottom of the tree, which contains leaves and traverse up to match the parents. The hybrid evaluation combines both the top-down and bottom-up traversal, and stops when a convergence is found. In the worst cases, all of these approaches need to search the whole data graph for regular path expressions with wildcard '//' comprehensively, hence are inefficient.

B. Use of Indexes

Index structures have been introduced to address the problem of performance degradation due to excessive traversal. These techniques certainly reduce the portion of the XML tree to be scanned during query processing. Among them are DataGuide [16], 1-index, 2-index, T-index [17], A(k)-index [18], Index Fabric [19], APEX Index [20] and extension of inverted list [21].

B.1. DataGuide

DataGuide were first introduced by Goldman and Widom by modeling XML data as OEM [16]. DataGuides are general path indexes that summarize all paths in the OEM (source) that start from the root.

Similar to automata theory, single non-deterministic finite automation (NFA) may have several equivalences deterministic finite automation (DFAs). A single source may be converted into multiple dataGuides. Thus, it is important to decide what kind of dataGuide should be built and maintained. Intuitively, a minimal dataGuide might seem desirable, as it is most compact in size, thus less traversal path. Yet, a minimal dataGuide (as shown in Fig. 4(a)) is hard to maintain especially during insertion of new node. A strong dataGuide as shown in Fig. 4(b) is created based on the intention that label path that reach the same set of objects in the source is the same as the label path that reach the same object in dataGuide.

Each node in a dataGuide has an extent for the corresponding nodes. For example, the extent in the node p4 is the set of {5,6}, where each element can be reach by the path *book.category.title* in Fig. 4(b).

However, dataGuide is not feasible in the multiple path expressions. For example, consider the following query Q1:

(Q1) : //book//title = "Introduction to Calculus"

This query wants to retrieve all books where the title is "Introduction to Calculus". If we traverse the dataGuide, the returned result are nodes p2 and p6 and the corresponding extent are {2,3} and {8,12} respectively. Since there is no path information between nodes p2 and p6, the result of the query cannot be returned by only traversing the dataGuide alone. We need to traverse the original data source and dataGuide simultaneously. In the worst case, a strong dataGuide grows linearly for a tree-structured and exponentially for a graph.

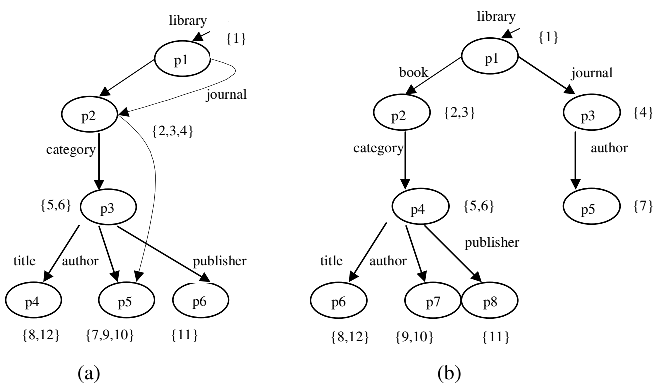


Fig. 4(a) Mimimal dataGuide (b) Strong dataGuide

B.2. 1-index, 2-index and T-index

Milo and Suciu introduced on index family that consists of 1-index, 2-index and T-index [17]. The 1-index technique is somehow similar to dataGuide. 1-index is restricted to simple path queries only. To support multiple path expression processing, the 2-index and T-index are proposed. The 2-index can be useful for a query, which has branching (multiple) path expressions. In the worst case, 2-index grows quadratic to the size of a data graph [22]. However, T-index can only be used in the case where appropriate templates to be built in advance.

B.3. A(k)-index

Kaushik et al introduced A(k)-index which reduces the size of index graph [18]. The key observation exploited by A(k)-index is that not all structure is interesting. This technique only indexes paths that have a length that is less than k. As a result, only approximation is used for answering longer queries [23]. This index graph is never larger than the source in size; however, it may not be precise. Increasing the parameter k, results in more accurate representation. For example A(1)-index guarantee that there is no false path of length 1 while A(2)-index guarantee that there is no false path of length 2 in any of the edges of a particular node.

B.4. Index Fabric

Index Fabric is a structure that scales gracefully to large numbers of keys and is insensitive to the length of inserted strings. These features are necessary to treat XML semistructured data paths as strings. Index Fabric's data structures are based on tries [19]. A trie is a tree that uses parts of the key to navigate the search. Each key is a sequence of characters, and a trie is organized around these characters rather than entire keys [24]. Patricia trie (PT) is a simple form of compressed trie which merges single child nodes with their parents.

Each data path are encoded using unique designators (special characters) starting from the root element. For example, for the XML in Fig. 1, library is represented by **L**, **B** for book, **C** for category and so on. Thus, the path library/book/cateogry is encoded by **LBC**.

PT can become large and unbalanced structures thus resulting in significant performance degradation. To balance the tries and support large XML, Cooper et al, proposed building multiple layers of tries, and conducting searches by proceeding from layer to layer. Index Fabric do not contain parent-child relationship among elements since it does not keep the information of XML elements which do not have data values. Thus, it is inefficient for processing partial matching queries [20].

B.5. APEX Index

Adaptive Path indEX (APEX) is a method to manage adaptive path indexes for XML data [20]. While the strong dataGuide, T-index, A(k)-index and Index Fabric maintains all paths from the root, APEX does not keep all paths starting from the root. However, it utilizes frequently used paths by applying sequential pattern mining technique [25].

APEX consists of two structures, a graph structure (G_{APEX}) to represent the structural summary of frequently used paths of XML data and a hash tree structure (H_{APEX}) that consists of required paths to nodes of the graph structure. The hash tree is used to find nodes of the structure graph for a given label path. For example, using the following query Q3,

(Q2) : //book/category

The query processor looks up the hash tree with *book.category* in reverse order. That is, the hash tree enables efficient finding of nodes for partial matching path queries.

B.6. Extension of Inverted List

Zhang et al proposed using inverted list to process containment query of XML data stored in RDBMS. Containment queries are a class of queries based on relationships among elements, attributes and their contents [21]. To support processing semistructured XML document, the inverted index is extended to text index (T-index) and element index (E-index). T-index is similar to the traditional index in information retrieval system while E-index maps element to inverted lists. Fig. 5 illustrates the structure of the two indexes based on the sample XML in Fig. 1.

Each inverted list records the occurrences of a word or an element known as term. Each occurrence is indexed by its document number, position and depth within the document, which is denoted as (*docno, begin : end, level*) for E-index and (*docno, wordno, level*) for T-index. The position (*begin, end, wordno*) are generated by counting word numbers in the XML document. This position can be generated by doing a depth-first traversal. The limitation of this approach is it assumes that once a position is assigned, it is never changed, which is not suitable in handling updating of XML data.

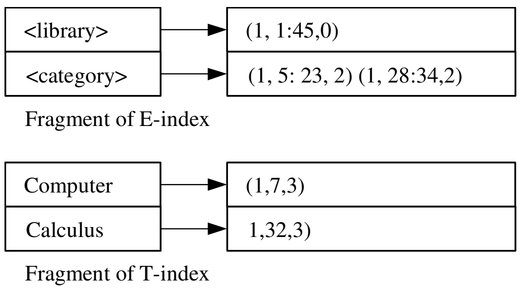


Fig. 5 Extension of Inverted List

C. Numbering Scheme

Several numbering scheme have been proposed to replace the structural indexing. They can be categorized into range-labeling scheme and prefix-labeling scheme. In range-labeling scheme, the label of a node is interpreted as a pair of numbers (start position, end position). When a new node is inserted, the label usually needs to be regenerated. Thus, range-labeling scheme is also known as non-persistent labeling scheme. However, in the prefix-labeling scheme, the label of a node is single number. Under heavy update, prefix-labeling scheme may not need to be recomputed. It is therefore also known as persistent labeling scheme [26]. Some of the methods in these categories are discussed below.

C.1. Tree traversal order

Dietz introduced the first numbering scheme based on tree traversal order [27]. This is in the category of range-labeling scheme. Each node is labeled with a pair of unique integer consist of preorder and postorder traversal sequence as shown in Fig. 6(a). His proposition was : *for two given nodes x and y of a tree T, x is an ancestor of y if and only if x occurs before y in the preorder traversal of T and after y in the postorder traversal.*

As in Fig. 6(a), we can tell that node (2,6) is an ancestor of node (3,5) because node (2,6) comes before node (3,5) in the preorder traversal (i.e, 2 < 3) and after node (3,5) in the postorder traversal (i.e, 6 > 5). By using this approach, we can determine the ancestor-descendant relationship easily. But, this method is inefficient for a dynamic XML document. Whenever a new node is inserted or deleted, the preorder and postorder value may needs to be recomputed.

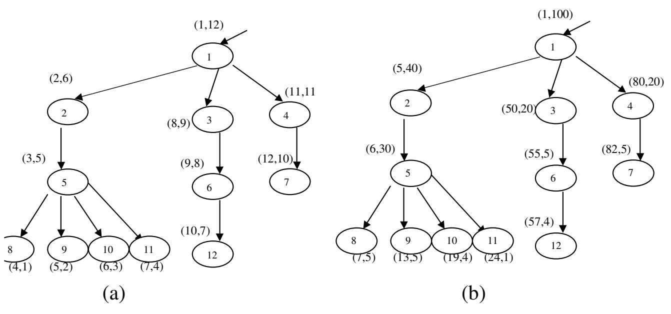


Fig. 6(a) Tree traversal order numbering (b) Extended Preorder numbering

C.2. Extended Preorder Traversal

To overcome the limitation of Dietz's numbering scheme, Li and Moon proposed a numbering scheme integrated with indexing mechanism for XML documents, elements and attributes, which enables efficient search by value and structure [28]. This numbering scheme also enables quick determination of the ancestor-descendant relationship between elements and/or attributes in the hierarchy of XML data. It is designed based on the notion of extended preorder traversal to accommodate future insertion gracefully. Each node in the XML tree is labeled with a pair of numbers <order, size> as shown in Fig. 6(b).

Their propositions were:

(1) *For a tree node y and its parent x, order(x) < order(y) and order(y) + size(y) <= order(x) + size(x). In other words, interval [order(y), order(y) + size(y)] is contained in interval [order(x), order(x) + size(x)].*
For example, consider a tree in Fig. 6(b), assume that node y is (5,40) and node x is (1,100). In this case, 1 < 5 and 45 <= 101, thus interval [5, 45] is contained in interval [1, 101].

(2) *For two sibling nodes x and y, if x is the predecessor of y in preorder traversal, order(x) + size(x) < order(y)*
For example, consider a tree in Fig. 6(b), assume that node y is (50,20) and node x is (5,40). Using the preorder traversal navigation, node x is predecessor to node y (node x is reached before node y), thus, 45 < 50.

(3) *For a tree node x, size(x) >= $\sum y$ size(y) for all y's that are direct child of x.*
For example, assume that node x is (6,30), thus, 30 >= 5+5+4+1

To enable future insertions gracefully, size(x) can be an arbitrary integer larger than the total number of current descendant of x. However, a global reordering is necessary when all the reserved spaces have been consumed. Moreover, it is not clear how one assigns a large enough value for "size", based on the three propositions.

Three major index structures are proposed, namely, element index, attribute index and structure index; and two other components are name index for storing name strings and value table for storing values (i.e. attribute value and text value). Besides that, they also proposed three path-join algorithms based on the idea of decomposing complex path expression into several simple path expressions. These are (1) EE-Join for searching paths from an element to another element, (2) EA-Join for scanning sorted elements and attributes to find element-attribute pairs and (3) KC-Join for finding Kleene-Closure on repeated paths or elements. The results of the simple path expression are then investigated to determine the type of join. These are then combined and processed by the query processor.

For example, decomposition steps of a complicated query (Q3) are illustrated in Fig. 7.

$$(Q3) : (E_1/E_2) * / E_3 [@A = v]$$

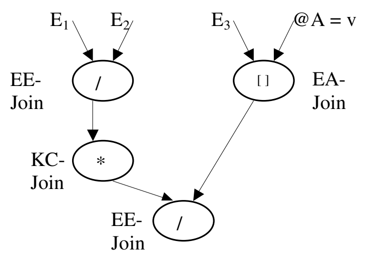


Fig. 7 Decomposition of a path expression Q4

C.3. Tree Location Address

Kimber proposed an approach of using "tree location address" to locate a node in a tree by selecting an ancestor node at each level of the tree [29]. Each identifier of an ancestor node is a prefix of its descendant. A node id (nid) is the concatenation of the nid's through the path from the root to the respective node. For example, from Fig. 8(a), node 1112 means the second child of the first child of the first child of the root. With this, a parent-child relationship can be easily detected. However, using this methodology, it requires variable space to store the identifiers. Thus, the time to determine the ancestor-descendant relationship is not constant as it depends on the length of identifiers. As a result, this method may not be practical for large databases.

C.4. Simple Prefix

Cohen E. et al proposed a simple prefix-labeling scheme. Each label inherits its parent's label as prefix [30]. The root is labeled with an empty string (""). The first child of the root is labeled with "0", the second child with "10" and followed by the third and fourth with "110" and "1110" respectively. For any node L(*v*) denotes the label of *v* the first child of *v* is labeled with L(*v*)"0", the second child of *v* is labeled with L(*v*)"10" and the i^{th} child with L(*v*)"$(1..1)^{i-1}$0". Referring to Fig. 8(b), for example, node 2 is an ancestor of node 7, for "0" is a prefix of "000". The limitation of this technique is, when the number of fan-out is large, the total length of labels scales up quadratically with increasing depth of the tree.

C.5. Compressed prefix scheme

To reduce the size required for the labeling scheme, Kaplan H. et al introduced a compressed prefix scheme. Using this approach, the tree is first transformed into a balance tree. The tree is partitioned into several paths. Then a collection of prefix free binary strings is assigned to the edges of the compressed tree outgoing of each virtual node. The nodes of the original tree are then labeled using the assignment as in the compressed tree [31]. The drawback of this is a complicated ancestor test, which incorporates another comparison in addition to the prefix decision. This scheme is practical for tree with many levels, but it is still impractical in the case of tree with large fan-out.

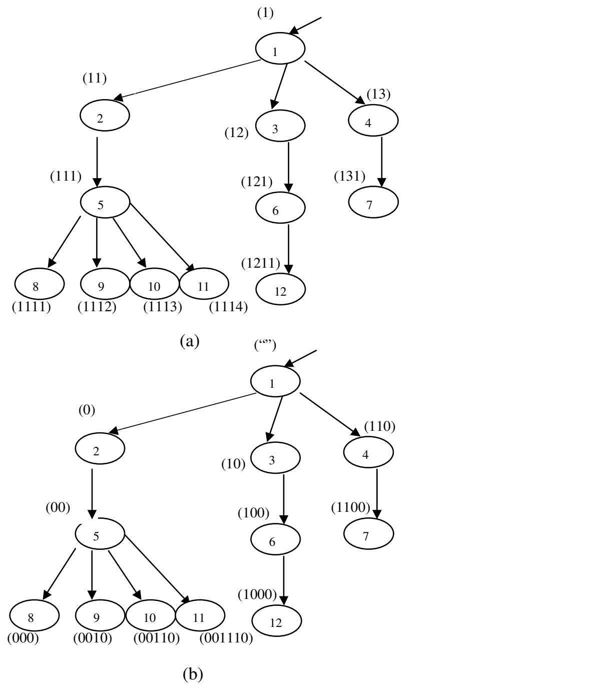


Fig. 8(a) Tree location address (b) Simple prefix labeling

C.6. Group Based Prefix Labeling Scheme (GRP)

Lu. J & Ling T.W. introduced grouping the XML tree into subtree to enhance the labeling scheme. Each node contains the label with groupID and a group prefix label, where groupID is an integer and group prefix label (similar to convention proposed by Cohen E. et al) is a binary string. All nodes within the same subtree will have the same groupID, but with different prefix label. For example, a root node has a fixed label: (1, "0"). They defined the maximum number of nodes in group i to be i, i.e, group 1 will contains at most 1 node and similarly group 3 will contains at most 3 nodes. If the number of nodes in one group reaches the maximal value, the group is defined as *full*. Using this approach, any new inserted node will firstly check whether their parent's node is *full.* If it is not, it will be group with their parent, else the node further checks whether the group of node youngest sibling is *full*. If it is not, it will be labeled with the group of its youngest sibling, else it will be assigned into a new group.

For example, referring to Fig. 1, supposed the inserting sequence is 1,2,3,4, …, 12, a GRP is derived as in Fig. 9. The root itself (node 1) is label as (1, "0"). Since node 2 is the child of the root and group 1 (root) is full, node 2 is grouped into group 2 and labeled as (2, "0"). The next node is node 3, since its youngest sibling (node 2) is not full, node 3 is grouped into group 2 and labeled as (2, "10"). However, when node 4 is inserted, group 2 is full at this point. Thus, node 4 forms a new group and labeled as (3, "0"). This process continues for the rest of the nodes.

This labeling scheme has much smaller size as compared to simple prefix scheme. It is believed to be practical for XML tree with many fan-outs and levels.

C.7. Using Prime Number

Wu X. et al proposed using prime numbers to label the XML tree via top-down and bottom-up approach [32]. Below are some of the prime number properties:

(1) In an integer A has a prime factor, which is not a prime factor of another integer B, then B is not divisible by A.
(2) In a bottom-up prime number labeling scheme, for any nodes x and y in an XML tree, x is an ancestor of y if and only if label(x) mod label(y)=0.

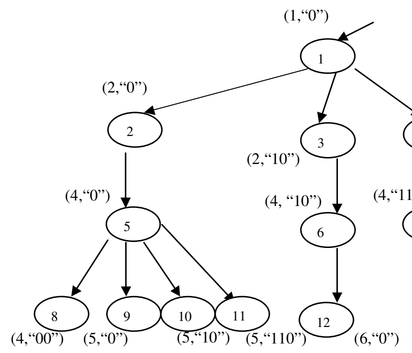


Fig. 9 GRP labeling scheme

Fig. 10(a) illustrates the basic bottom-up approach. Prime number is assigned to each leaf nodes. Then for each subsequent level, the parent's label is a product of their children's labels. Based on property (2), the ancestor-descendant relationship can be determined quickly. This approach has two drawbacks. Firstly, it will cause a relatively large numbers being assigned to the node at the tops. Secondly, it is not possible for nodes with a single child only.

To overcome the weakness of bottom-up approach, top-down approach was proposed. With this, the root will be labeled with the first prime number 1. Each non-leaf node will be given a unique prime number. The label of each node is the product of its parent nodes' label (parent-label) and its own assigned number (self-label). This approach is good for dynamic updates. When a new node is inserted, an unassigned self-label prime number is allocated. Thus, no re-labeling is required.

However, one disadvantage of the prime number labeling scheme is that each prime number can only be used once. Thus, the size of label is increases as it reaches the bottom of the tree. The size storage of the label has direct impact on the performance of XML query processing. It is therefore, not suitable for a tree with many levels. They also suggested three optimization techniques:

(1) Assign small prime number to the root and level right after the root so that the value inherited will be smallest possible.
(2) Use number 2 and its sequence ($2^1, 2^2, \dots, 2^n$), the only even prime number to label the self-labels of the leaf nodes and the odd numbers for the non-leaf nodes.

Combine those paths which occur multiple times (similar to dataGuide concept) can reduce redundancy and further decrease the size of the labels.

V. Conclusion

The optimization of queries on small documents seems not very useful. But, as the usage of XML shifts towards the data-oriented paradigm, more efforts need to be done to allow the efficient retrieval and processing of query.

This survey shown, there are three methods to store XML data, i.e via RDBMS, OODBMS and native database. The type of storage will determine the possible indexing mechanism.

Various indexing technologies have been developed to solve the query retrieval and updating problems. A problem with path traversal methods is that traversing is only possible in the constrained set of path. However, for the structure summary indexing, most of it has the problem of large index size growth in the worst case and not supporting partial queries path matching. Labeling scheme allow quick determination of the relationships among the element nodes and reduce the index size, but fails to support dynamic XML data. Towards the later year, most researchers proposed hybrid-indexing techniques. Hybrid system opens the possibility of covering each technology's weaknesses by its strengths.

We believe there are still many opportunities for query optimization area. We are planning to implement a hybrid optimization technique comprising both indexing and labeling scheme in future. The advantages and disadvantages of indexing and labeling technologies is summarized as in Table I and II.

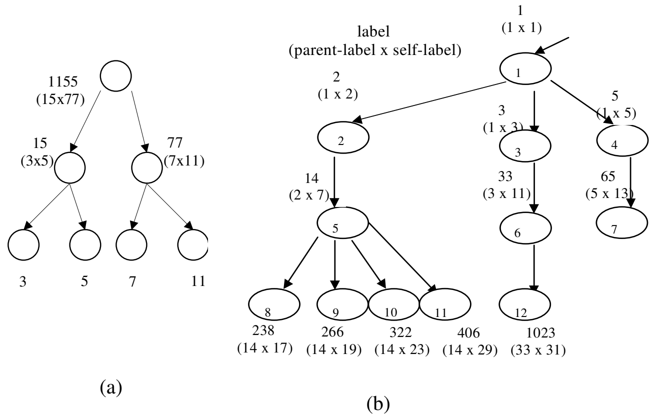


Fig. 10 (a) Bottom-up prime (b) Top-down prime

TABLE I
Advantages and Disadvantages of Indexing Technology

Type of Index	Advantages	Disadvantages
1. Path traversal (NXD)	Direct traversal. No insertion/update problem.	Whole graph may need to be traverse to search a node. Not supporting partial queries path matching.
2. Strong dataGuide (NXD)	Simplify traversal path. No insertion/update problem (set of extent can be extended).	Index graph size grows exponentially. The source and dataGuide may need to be traverse simultaneously in some cases. Not supporting partial queries path matching.
3. Index Family (NXD)	Simplify traversal path. No insertion/update problem (set of extent can be extended or created to support new node).	Index graph size may grow quadratically (2-index). Many templates must be build in advance (T-index). Not supporting partial queries path matching.
4. A(k)-index (NXD)	Size never grow larger than the source. No insertion/update problem (set of extent can be extended).	Not accurate as only approximation is used to answer long queries.
5. Index Fabric (RDBMS)	Index size controllable as it can be partitioned horizontally and vertically.	Not supporting partial queries path matching. Encoding keys may need to be regenerated.
6. APEX index (NXD)	No insertion/update problem . Supporting partial queries path matching.	Workload info needs to be collected to determine the frequently used queries.
7. Extension of Inverted List RDBMS)	Supporting partial queries path matching Size never grow larger than the source	Not supporting insertion/update (label need to be regenerated)

TABLE II
ADVANTAGES AND DISADVANTAGES OF LABELING SCHEME

Type of Labeling Scheme	**Advantages**	**Disadvantages**
1. Tree traversal order	Easy traversal (pre and post) labeling Supporting partial queries path matching Maximum size determine by the number of nodes	Not supporting insertion/update (label needs to be regenerated)
2. Extended preorder	Efficient to determine structural join and value join Support insertion/update as long as the reserved spaces have yet been consumed	Hard to determine the correct "size" for each node
3. Tree location address	Efficient to determine ancestor/descendant relationship	Length of the identifier/label grows as the level of tree increases Not supporting dynamic XML, label might need to be regenerated if new node is inserted into the left leaf.
4. Simple prefix	Efficient to determine ancestor/descendant relationship	Label's length scales up quadratically as number of fan-out and level increases.
5. Compressed prefix	Shorter label length (reduce storage size) Suitable for skewed tree and tree with many levels as balancing will be carried out	Extra cost needed to test for ancestor/descendant relationship in the original tree. Not practical for tree with lots of fan-out
6. GRP	Small size Suitable for tree with many fan-out and levels	Extra cost needed to check for grouping criteria. Not fully supporting insertion/update as label may needs to be regenerated
7. Prime number	Supporting dynamic update; no re-labeling required Size unaffected by number of fan-out and levels	Each prime number can be used only once.

REFERENCES

[1] S. Abiteboul *et al*, "The Lorel Query Language for Semistructed Data, Journal of Digital Libraries", Vol 1, No 1, 1997, pp. 68-88.
[2] J. Robie, J. Lapp, D. Schach, XML Query Language (XQL). Available http://www.w3.org/TandS/QL/QL98/pp/xql.html
[3] S. Ceri *et al*, XML-GL : A Graphical Language for Querying and Reshaping XML Documents. Available http://www.w3.org/TandS/QL/QL98/pp/xml-gl.html
[4] W3C, XML Path Language (XPath). Available http://www.w3.org/TR/xpath-datamodel/
[5] W3C, XML Query (XQuery). Available http://www.w3.org/XML/XQuery
[6] D. Zhang, Y. Dong, "A Data Model and Algebra for the Web", Proceeding 10th International Workshop on Database and Expert System Application, IEEE Computer Society, 1999, pp. 711-714.
[7] F. Frasincar, G. Houben, C. Pau, "XAL : An Algebra for XML Query Optimization", 13th Australasian Database Conference, 2002, pp. 49-56.
[8] V. Christophides, S. Cluet, J. Simeon, "On Wrapping Query Languages and Efficient XML Integration", ACM SIGMOD International Conference on Management of Data, ACM Press, 2000, pp. 141-152.
[9] T. Bray, J. Paoli, C. Sperberg-McQueen, Extensible markup language (XML) 1.0, Technical report, W3C Recommendation, 1998
[10] L. Shan, Y. Rao, "A Performance Evaluation of Storing XML Data in Relational Database Management Systems", WIDM 2001, pp. 31-38.
[11] M. Atay, Y. Sun, D. Liu, S. Lu, F. Fotouhi, "Mapping XML Data To Relational Data : A DOM-Based Approach", Proc. of the 8th IASTED International Conference on Internet and Multimedia Systems and Applications, 2004, pp. 59-64.
[12] T.S. Chung, H-J Kim, "Techniques for the evaluation of XML queries : a survey", ACM Data And Knowledge Engineering 46, 2003, pp. 225-246.
[13] M. Garafalakis, A. Gionis, R. Rastpgo, S. Seshadri, K. Shim, "XTRACT : a system for extracting document type descriptors from XML documents", Proceeding of the ACM SIGMOD Int. Conference on the Management of Data, 2000, pp. 165-176.
[14] R. Bourret, XML Database Products. Available http://www.rpbourret.com/xml/XMLDatabaseProds.htm
[15] J. McHugh & J. Widom, "Query Optimization for XML", Proceeding 25th International Conference on Very Large Databases, 1999, pp. 315-326.
[16] R. Goldman & J. Widom, "Data Guides : enabling query formulation and optimization in semistructured database", Proceeding of VLDB, 1997,pp. 436-445.
[17] T. Milo & D. Suciu, "Index structures for path expression", Proceeding of 7th Int. Conference on Database Theory, 1999, pp. 277-295.
[18] R. Kaushik, D. Shenoy, P. Bohannon, E. Gudes, "Exploiting Local Similarity to Efficiently Index Paths in Graph-Structured Data", Proceeding of Int. Conference on Data Engineering, 2002, pp. 129-140.
[19] B. F. Cooper, N. Sample, M.J. Franklin, G.R. Hjaltason, M. Shadmon, "A Fast Index for Semistructured Data", Proceeding of 27th VLDB Conference, 2001, pp. 341-350.
[20] C.W. Chung, J.K. Min, K. Shim, "APEX : An Adaptive Path Index for XML data", ACM SIGMOD, 2002, pp. 121-132.
[21] C. Zhang, J. Naughton, D. DeWitty, Q. Luo, G. Lohman, "On Supporting Containment Queries in Relational Database Management Systems", ACM SIGMOD, 2001, pp. 425-436.
[22] J. Kim & H-J Kim, "Efficient processing of regular path joins using PID", Information and Software Technology 45, 2003, pp. 241-251.
[23] K. Michal, P. Jaroslav, S. Vaclav, "Indexing XML Data with UB trees", ADBIS, 2002, pp. 155-164.
[24] D. Adam, Data Structures and Algorithms in C++, 2001, Thomson Learning
[25] R. Agrawal, R. Srikant, "Mining sequential patterns", Proceeding of the 11th Int. Conference on Data Engineering, 1995, pp. 3-14.
[26] J. Lu & T.W. Ling, "Labeling and Querying Dynamic XML Trees", APWeb, LNCS 3007, 2004, pp. 180-189.
[27] P.F. Dietz, "Maintaining order in a linked list", Proceeding of the 14th Annual ACM Symposium on Theory of Computing, 1982, pp. 122-127.
[28] Q. Li & B. Moon, "Indexing and Querying XML Data for Regular Path Expressions", Proceeding of 27th VLDB Conference, 2001, pp. 361-370.
[29] W.E Kimber, "HyTime and SGML : Understanding the HyTime HYQ Query Language", Technical Report Version 1.1, IBM Corporation, 1993.
[30] E. Cohen, H. Kaplan, T. Milo, "Labeling Dynamic XML Trees", Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, 1992, pp. 272-281.
[31] H. Kaplan, T. Milo, R. Shabo, "A Comparison of Labeling Schemes for Ancestor Queries", Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms, 2002, pp. 954-963.
[32] X. Wu, M.L. Lee, W. Hsu, "A Prime Number Labeling Scheme for Dynamic Ordered XML Trees", Proceedings of the 20th Int Conference on Data Engineering, 2004.