BBTC: A New Update-aware Coding Scheme for Efficient Structure Join

Guoliang Li, Jianhua Feng, Na Ta, Lizhu Zhou

Department of Computer Science and Technology
Tsinghua University, Beijing 100084, China

Summary
The identification of ancestor-descendant or parent-child relationship between elements of XML documents plays a crucial role in efficient XML query processing. One of the popular methods for performing this task is to code each node in the XML document by traversing its nodes. However, the main problems of existing approaches are that they either lack the ability to support XML document update or need huge storage space. This paper proposes a novel coding scheme called Blocked Binary-Tree Coding scheme (BBTC) by taking the issues of identification, easy update and low storage cost into account. BBTC identifies the ancestor-descendant relationship in constant time. For the update, only a few simple operations for the affected document elements are needed. More importantly, for BBTC, this paper proposes a structure join algorithm BDC based on Bucket Divide and Conquer. BDC not only accelerates structure join dramatically when the input element sets are neither sorted nor indexed, but also can be applied to other coding schemes. The extensive experiments show that both the coding scheme BBTC and BDC significantly outperform the existing studies.

Key words: XML, structure join, coding scheme, partial order

Introduction

To support queries on XML databases, a number of query languages such as Lorel[1], XML-QL[9], XPath [7], and XQuery [3] have been studied. One of their core techniques is to use path expressions to express structure queries for XML documents. To evaluate such a query, for instance, “Book//Name”, a naïve tree traversal strategy could be used to scan the entire XML data tree even there are only few results. To overcome the shortcoming of traversing the entire original document of this naïve strategy, a coding scheme can be used to assign each node in the document tree a unique code so that the ancestor-descendant (or parent-child) relationship of element nodes and attribute nodes in the tree can be figured out directly. A number of such XML coding schemes have been proposed for the query, especially structure query of XML documents [11]. Based on these methods, for the “Book//Name” example, an alternative strategy can first retrieve all Book and Name elements through codes, and then find all occurrences of the ancestor-descendant relationship between these two element sets. In this way, query of XML can be converted to structure join computation by using coding schemes. However, present coding schemes either do not support XML documents update, or have high storage cost. In addition, previous algorithms about structure join require the order of input data sets or additional indices with an assumption that memory buffer can hold full input element sets.

To address these problems, this paper first proposes a new coding scheme based on binary-tree, which efficiently supports both dynamic update of XML documents and the identification of ancestor-descendant relationship. The coding scheme, Blocked Binary-Tree Coding scheme (BBTC), has average code length $O(\log(n))$, which is asymptotically minimum. In this paper, n is used to denote the number of nodes in an XML document. To skip unnecessary elements which do not participate in a join, a structure join algorithm BDC is designed based on a technique, Bucket Divide and Conquer. This algorithm partitions the two element sets for the join into different buckets and only structure join of suited buckets contribute to the results. In this way, it reduces I/Os sharply and accelerates the structure join.

In brief, the major contributions are as follows:

- First, BBTC can identify the ancestor-descendant relationship in constant time, and the inference of such relationship is based on simple equality operations (add and shift) rather than complicated operations (estimation of region range). The average code length of BBTC is $O(\log(n))$, which is asymptotically minimum.
- Second, BBTC maintains order information among sibling nodes so that the queries about sibling sequence are supported effectively, such as Book[2]//Name[3]. At the same time, BBTC supports XML update efficiently through its sibling order information.

The work was supported by the National Natural Science Foundation of China under Grant No.60573094, Tsinghua Basic Research Foundation under Grant No.JCqn2005022 and Zhejiang Natural Science Foundation under Grant No.Y105230

- Third, BDC can accelerate structure join by skipping many unnecessary elements which don't participate a join through divide and conquer strategy. BDC is significantly efficient for structure join, especially when memory size is limited.
- Last, BDC is not only efficient for our coding scheme, but also portable to other coding schemes.

This paper is organized as follows: section 2 introduces and analyzes related work on XML coding schemes and structure join. Section 3 presents our coding scheme in details. In section 4, a hierarchical storage method is introduced to BBTC. In section 5, an efficient structure join algorithm, BDC, is presented. In section 6, the experimental results of our coding scheme and structure join algorithm are presented with comparisons to other approaches. Last, a conclusion is made in section 7. difference images from adjacent frames and recognize hand gestures by improved centroidal profile.

This paper consists of 5 sections. The section 2 describes the hand region extracting method from a sequential color images with entropy analysis. The section 3 stresses on gesture recognition techniques from the extracted hand region images. The section 4 shows the experiment results by proposed method. The section 5 concludes this proposal.

2. Related Work

Various coding schemes have been proposed for query processing of XML documents. These existing coding schemes fall into two main classes: (1) the region based coding [5, 10, 12, 13, 21] and (2) the path based coding [8, 15, 20]. The region based coding scheme is more widely used.

2.1 Coding Schemes

The main idea of region-based coding is to assign a pair of numbers <*start, end*> called Region Code to each node in the XML document tree, satisfying that the Region Code of a node covers its descendants' codes. That is, node u is the ancestor of node v iff. $u.start<v.start$ and $v.end<u.end$. The coding schemes in [5, 10, 12, 13, 21] are all based on Region Code. They require level or height information to differentiate the ancestor or parent. In addition, complex operation, instead of equality estimation, is used when deciding parent-child and ancestor-descendant relationships. Moreover, these methods cannot support dynamic update of XML documents well. Some study [13] proposed a solution which preserves code space for the *size* in <*order*, *size*> or makes *order* the extended pre-order traversal number so that extra space can be preserved to support update. But it is difficult to decide how much space the preservation is to make, and when the preserved space is used up, the XML document has to be recoded again. N.Wirth proposed the Bit-vector coding scheme in [19]. Dewey code has been proposed in [17]. Dynamic labeling structure has been proposed in [4, 8, 20], however, such long labels not only incur high storage overhead, but also are less useful in query processing because they are more expensive to process than shorter labels. W-BOX and B-BOX, two novel structures for maintaining order-based labeling of XML elements were presented in [15, 16]. Wang et al. [18] also proposed PBiTree, which converted an XML document tree into a binary-tree, and numbered each node sequentially. The main difference of PBiTree from our approach is that, it does not support update and involves large storage.

2.2 Structure Join

Structural join is to find all occurrences of structural relationship between two element sets, which is a core technique to query path expression. Presently structure join is almost about region based technique, and always requires ancestor list and descendent list in order or maintains indices with assumption that memory buffer can hold full element sets. Zhang et al. [21] propose a variant of the traditional merge join algorithm, called multi-predicate merge join (MPMGJN). However, it may perform a lot of unnecessary computation and I/Os for matching structural relationship. Similarly, EE-Join and EA-Join in [13] may scan an element set multiple times. The Stack-Tree-Desc algorithm proposed in [2] improved the merge based structural join algorithms with stack mechanism. The basic idea is to take the two input lists, AList and DList, both sorted on their *start* values, and merge them. A stack is introduced to maintain ancestor nodes that will be used later in the join. As such, only one sequential scan is performed on AList and DList. Chien et al [6], proposed a stack-based structural join algorithm that could utilize the B^+-tree indices built on the *start* attribute of the participating element sets. An enhancement to the basic B^+-tree approach is to add sibling pointers based on the notion of containment. XR-tree in [14] utilizes a new index to skip ancestor nodes, which overcomes the drawback of [6]. However, there is a problem that both of them have no indices on intermediate results. As a result, existing algorithms require both of the input element sets sorted or need to build additional indices (but no indices for intermediate results).

3 A New Update-aware Coding Scheme

3.1 Coding Scheme and Corresponding Algorithm

Our coding scheme is a pair `<order, sibling_order>`, in which *`order`* represents the position information of a node in the XML document tree, and `sibling_order` means the sequential number of an element among its siblings, which reflects order

characteristic of XML documents and can help to deal with update operation, therefore query of sibling relationship is effectively supported. The approach to code an XML document is described in Algorithm 1, in which the order of the leftmost child is its parent's order multiplied by 2, and its `sibling_order` is 0; the orders of other children except the leftmost child are their preceding siblings' order multiplied by 2 plus 1, and `sibling_orders` are their preceding siblings' `sibling_order` plus 1(order of the root is 1, `sibling_order` is 0).

```
Algorithm 1: Coding_XML_Tree(N)
Input: N is a node of an XML document tree T
Output: <N.order, N.sibling_order>
1. if N is the root of T then {N.order=1;
N.sibling_order=0;}
2. else
3.    if N is N.parent.leftmost_child then
          {N.order=N.parent.order*2;
          N.sibling_order=0; }
4.    else {
     N.order= N.preceding_sibling.order*2+1;
        N.sibling_order=N.preceding_sibling.
                          sibling_order+1;}
5.    endif
6. endif
7. for each NC, NC is a child of N
8.    Coding_XML_Tree (NC);
{recursive calling all the children nodes of N based on depth first
search}
9. endfor
```

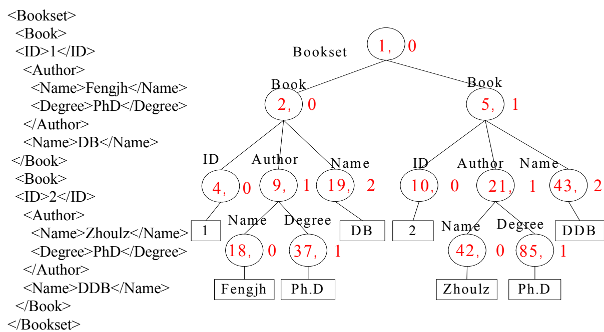


Figure 1. XML file Bookset.xml and corresponding codes

Using Algorithm 1 the codes for all nodes can be worked out during once scan of the XML document tree. A simple example is shown in Figure 1 (codes of all TEXT_NODEs are omitted).

3.2 Properties of Our Coding Scheme

We can infer the relationship between any two nodes using simple operations.

Given nodes A, D and N in an XML document tree T.

1. N.*level* = number of zeros in the binary representation of `N.order` plus 1. This value represents which level the node N lies in the XML document tree.
2. A is an ancestor of D *iff.* (3-1) is true; (3-1) is equivalent to (3-2) and (3-2) is more easy to be inferred in computer.

$$2*A.order = \lfloor D.order/2^{\lfloor \log_2 D.order \rfloor - \lfloor \log_2 A.order \rfloor - 1} \rfloor \quad \textbf{(3-1)}$$

$$(A.order<<1) = (D.order>>(\lfloor \log_2(D.order) \rfloor - \lfloor \log_2(A.order) \rfloor - 1)) \quad \textbf{(3-2)}$$

3. A is a parent of D *iff.* (3-1) or (3-2) is true and D.*level*= A.*level* + 1.

In this paper, $\lfloor X \rfloor$ denotes the integer part of number X; << denotes shift leftward; >> denotes shift rightward.

For instance in Bookset.xml (the XML document in Figure 1), suppose we want to infer the relationship between `/Bookset/Book[1]` (order is 2) and `/Bookset//Degree` (1st node's order is 37, 2nd is 85). As $\lfloor \log_2 37 \rfloor=5$, $\lfloor \log_2 85 \rfloor=6$, and $2*2=\lfloor 37/2^{5-1-1} \rfloor$, so 1st `Degree` node is a descendant of `Book[1]`, but not a child, because *level* of `Book[1]` is 2(there is one zero in binary representation of $2=(10)_2$), and *level* of 1st `Degree` is 4(there are 3 zeros in binary representation of $37=(100101)_2$). As $2*2 != \lfloor 85/2^{6-1-1} \rfloor$, 2nd `Degree` node is not a descendant of `Book[1]`.

3.3 Support of XML Documents Update

According to the basic idea, our approach does not need to code the XML document again when updated. Only insert and delete operations are concerned here, and other update operations, such as changing the value of a tag, are isolate to coding schemes. Due to the characteristic of our coding scheme, a new node is inserted as the 'last' child of its parent, and the order of the new node can be calculated very easily, only to increase its following siblings' `sibling_orders` by 1.

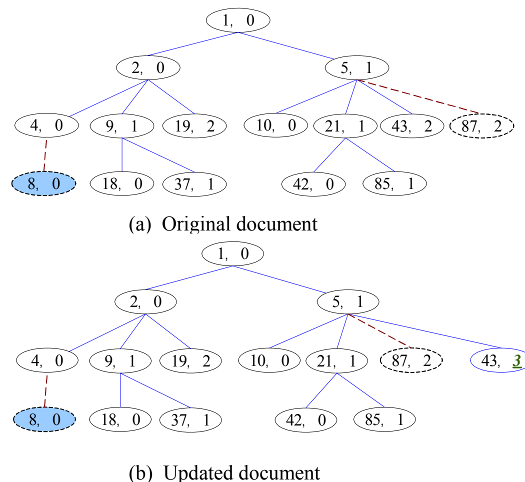


Figure 2. Update of an XML document

An example is shown in Figure 2. The dashed nodes in Figure 2(a) are inserted nodes. The solid dashed node is

appended to node 4, and its code is (4*2,0). But hollow dashed node 87 is inserted before node 43, so insert it as the 'last' child of its parent node 5, and its `order` is 43*2+1=87, but its `sibling_order` is 2 which denotes its order among its siblings, and increase all of its following siblings' `sibling_orders` by 1, and in Figure 2(b) node 43's `sibling_order` is updated to 3. Delete operation is very easy, and it is only needed to remove the deleted node (dn) and decrease `sibling_orders` of all the dn's siblings by 1 whose `sibling_orders` are greater than dn.`sibling_order`.

4 Storage of BBTC

General XML document trees are not regular, but the binary-tree has such advantages that it has regular hierarchical structure and is easily to be stored. Therefore, the general XML document trees are converted to ordered binary-trees.

4.1 Conversion from an XML Document Tree to a Binary-tree

Let T be an XML document tree and BT be the corresponding converted binary-tree, the corresponding conversion rules are as follows:

1) A∈T, if A is the root of T, then A is the root of BT.

2) ∀A, D∈T, if D is the first child of A in T, then D is the left child of A in BT.

3) ∀D1, D2∈T, if D2 is the following sibling of D1 in T, then D2 is the right child of D1 in BT.

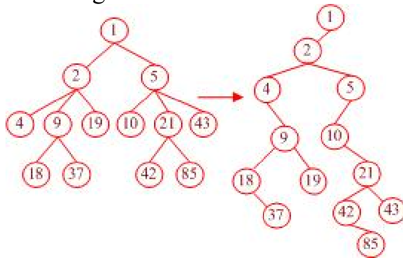


Figure 3. XML document tree and its binary-tree

BlockID(BID)	ElementNO	(IID,sibling_order)	(IID,sibling_order)	……	(IID,sibling_order)

Figure 4. Storage architecture of BBTC

Using these rules, the XML document tree in Figure 1 can be converted to a binary-tree in Figure 3.

In this way, an element can be coded in the XML document through its binary-tree directly. The detailed coding method is as follows:

1) if R is the root of BT then R.`order` =1

2) ∀D1, D2, A∈BT, if D1 is the left child of A then D1.`order` = A.`order`*2;

3) if D2 is the right child of A then D2.`order` = A.`order` *2+1.

The coding method of the binary-tree is identical with Algorithm1 in section 3.1.

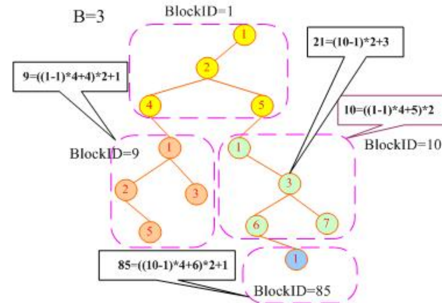


Figure 5. BBTC

4.2 BBTC

The disadvantage of the binary-tree code is that it needs relatively large storage space. To solve this issue, we propose new storage architecture. Due to properties of the binary-tree, some descendent nodes store prefix of their ancestors repeatedly. Therefore, the binary-tree can be partitioned into different sub-blocks with elements in each sub-block having the same ancestor, and then each element can be coded relative to the ancestor (root of the sub-block) in each sub-block. Thus, the common prefix of each element's code can be stored only once. We call it Blocked Binary-Tree Coding scheme (BBTC) as shown in Figure 4. The common prefix called BlockID (BID), and the other part of the code to distinguish each other in the sub-block is called InnerID (IID). In Figure 4, ElementNO denotes how many elements there are in a sub-block. That is to say in BBTC, each element has its own IID, and all elements in a same sub-block share a common BID, which is the original order code of this sub-block's root. There are some issues to be addressed in BBTC, such as how large a sub-block should be and what structural relationship should be maintained between sub-blocks. To describe quantitatively how large a sub-block is, we introduce B, which is the height of the sub-block (the height of the sub-block which has only one node is 1), and 2^B-1 represents the maximum number of elements in a sub-block.

The rules to partition a binary-tree are:

1. All nodes whose height differences to the root less than B (including the root itself) are in a same sub-block. The first sub-block's BID is 1, and the sub-block's root is the root of the binary-tree.
2. ∀N_A, N_A is a leaf node of one sub-block, if N_A has a child N_D, then a new sub-block is created with N_D as its root, and all of N_D's descendents whose height differences to N_D less than B are in this sub-block. Suppose that IID of N_A is C, and the BID of that sub-block which N_A belongs to is D, then the BID of the new sub-block rooted at N_D can be computed by

formula 4-1:

$$BID=\begin{cases}((D-1)*2^{B-1}+C)*2 & N_D \text{ is the left-child of } N_A\\((D-1)*2^{B-1}+C)*2+1 & N_D \text{ is the right-child of } N_A\end{cases} \quad \textbf{(4-1)}$$

3. The IID of each sub-block's root is always 1; other nodes' IIDs and `sibling_orders` are coded in each sub-block according to 4.1.

Table 1. The storage of BBTC

Original code				BBTC					
				BID	ElemetNO	(IID,sibling_order)			
(1,0)	(2,0)	(4,0)	(5,1)	1	4	(1,0)	(2,0)	(4,0)	(5,1)
(9,1)	(18,0)	(19,2)	(37,1)	9	4	(1,1)	(2,0)	(3,2)	(5,1)
(10,0)	(21,1)	(42,0)	(43,2)	10	4	(1,0)	(3,1)	(6,0)	(7,2)
(85,1)				85	1	(1,1)			

Let B be 3, the binary-tree of the XML document, Bookset.xml (Figure 3), can be partitioned into four sub-blocks (Figure 5), and Table 1 lists the details.

For each sub-block, we can compute each node's original `order` code through its IID and its sub-block's BID. Suppose that a node's IID is C and its sub-block's BID is D, and its order code in the original XML document tree is:

$$(D-1)*2^{\lfloor \log_2 C \rfloor}+C \quad \textbf{(4-2)}$$

For example, the original `order` of node (IID is 7, BID is 10) is 43($(10-1)*2^2+7$). Therefore, when inferring whether two nodes have the ancestor-descendant relationship or not, firstly their original codes are figured out according to (3-2), and then the relationship according to the rules in section 3.2 can be gotten. However, it is not necessary to compute the original codes to infer the relationship. In next section some simple rules will be given, through which a node's IID and its sub-block's BID will be used directly to infer their relationship.

There are two definitions to be introduced before we present the rules in detail.

Definition 1: Sibling Sub-blocks

Given two different sub-blocks, suppose that their BIDs are B1 and B2 respectively, if B1 and B2 satisfy $\lfloor \log_{2^B} B1 \rfloor = \lfloor \log_{2^B} B2 \rfloor$, then these two sub-blocks are called sibling sub-blocks.

Definition 2: Collateral Sub-blocks

Given two different sub-blocks, suppose that their BIDs are B1 and B2 respectively, if B1 and B2 do not have the ancestor-descendant relationship according to rules in section 3.2, then these two sub-blocks are called collateral sub-blocks.

Three rules are given to infer two elements' relationship by their BIDs and IIDs:

1. If two nodes have the same BID (in the same sub-block), then the rules in section 3.2 can be used to infer their relationship through their IIDs.
2. If the sub-blocks that the two elements belong to are sibling sub-blocks or collateral sub-blocks, then there is no ancestor-descendant relationship between them.
3. Otherwise, suppose the IIDs of the two nodes are C1 and C2 respectively and their BIDs are D1 and D2 (without loss of generality, we suppose D1<D2). Their relationship can be inferred by evaluating relationship between N1 and D2 through the rules in section 3.2, where $N1=(D1-1)*2^{\lfloor \log_2 C1 \rfloor}+C1$, N1.*level*=C1.*level*+D1.*level*-1.

Using the three rules, the relationship between them can be inferred. For instance, the relationship between some nodes in Table 2 can be inferred, in which A-D and P-C stand for Ancestor-Descendant and Parent-Child relationships.

Table 2. Inferring relationships between some nodes

Node A			Node D			A-D	P-C	Reason
order	BID	IID	order	BID	IID	√	√	in the same sub-block (rule1)
9	9	1	**37**	9	5			
order	BID	IID	order	BID	IID	×	×	collateral sub-blocks (rule2)
18	9	2	**85**	85	1			
order	BID	IID	order	BID	IID	√	×	N1=5, 5*2=⌊85/2^(6-2-1)⌋ (rule3); but (5.level +1=2) !=(85.level+1=4)

4.3 Analysis of Code Length

To analyze the average code length of BBTC, some definitions are introduced firstly.

Definition 3: Saturation (ξ)

Saturation of a sub-block means the ratio of actual number of nodes (AN) in this sub-block to the max number of nodes (TN) that this sub-block can contain, that is: $\xi=AN/TN=AN/(2^B-1)$.

Definition 4: Fan-out (ϒ)

Fan-out can only be defined on a sub-block which has child sub-blocks. For such a sub-block, ϒ denotes the number of its leaves which have children in the binary-tree.

For example, in Figure 5, $\xi_{10}=4/7$, $\xi_9=4/7$; $\Upsilon_1=2$, $\Upsilon_{10}=1$.

Definition 5: Average Code Length (λ)

Average code length of a coding scheme is the quotient of the sum of each node's code length to the total number of nodes in an XML document. A node's code length is the number of bits of its order code in its binary representation.

Theorem 1: *if ξ=1 for all sub-blocks then λ=O(log(n))*

Proof:

$$\lambda=\frac{\overbrace{\sum_{k=1}^{\frac{\log_2 n}{B}}((k-1)B+1)2^{(k-1)B}}^{total\ length\ of\ BIDs}+\overbrace{\sum_{k=1}^{B}k2^{k-1}*\frac{n}{2^B-1}}^{total\ length\ of\ IIDs}}{n}$$

$$=\frac{\overbrace{\frac{(\frac{\log_2 n}{B}-1)*B*n-B\frac{n-2^B}{2^B-1}+n-1}{2^B-1}}^{total\ length\ of\ BIDs}+\overbrace{\frac{(B*2^B-2^B+1)*n}{2^B-1}}^{total\ length\ of\ IIDs}}{n}$$

$$\approx\frac{\log_2 n}{2^B-1}+B-1$$

Similar to Theorem 1, it can also be proved when ϒ≥2, λ=*O(log(n))*.

Theorem 2: *if ∃δ∈[ln2/2, 1), ∀sub-blocks, ξ≥δ, then*

when B=ln2/(2δ) log_2n, λ is asymptotically minimum and λ=O(log(n)).*

Proof: if ∃δ∈[ln2/2, 1), ∀sub-blocks, ξ≥δ. Suppose $\ell = n/(\delta(2^B-1))$, which is the max number of sub-blocks (this is the worst case, and each sub-block has only one child sub-block), and even if under these conditions:

$$\lambda_{Max} = (\sum_{1}^{\ell}(1+B*(i-1)))/\ell + B$$

$$= 1+\frac{B}{2}*(\ell-1)+B = \frac{B*\ell}{2}+\frac{B}{2}+1$$

$$\lambda'_{Max}|_B = \frac{n}{2\delta(2^B-1)} - \frac{B*n*2^B*\ln 2}{2\delta(2^B-1)^2}+\frac{1}{2}$$

$$= \frac{n*(2^B-1)-B*n*2^B*\ln 2+\delta(2^B-1)^2}{2\delta(2^B-1)^2}$$

if $\lambda'_{Max}|_B = 0$,then λ_{Max} reaches minimum that is O(log(n)). However, there is no analytical expression about B, whereas once n and δ are given, B can be figured out. As $\lambda \le \lambda_{Max}$, and usually B=ln2/(2δ)*$\log_2$n, then $\lambda \le \lambda_{Max} \approx$ (ln2/(2δ))*log(n), and even if δ=ln2/2, $\lambda \le \lambda_{Max} \approx$log(n).

Similar to Theorem 2, it can also be proved that when ξ≥ln2/2 for more than 50% sub-blocks, then *λ=O(log(n))*. This condition can be easily satisfied with real data. In section 6.1, it will be validated by experiments.

5 BDC: An Efficient Algorithm for Structure Join

5.1 Basic Concepts

Definition 6: Structure Join [14]

Given an ancestor set AList={a_1, a_2, …, a_n} and a descendant set DList={d_1, d_2, … ,d_m}, where both AList and DList consist of nodes from the XML document tree. A structure join of AList and DList, denoted as AList ∞DList, where the symbol ∞ denotes structure join, returns all tuple pairs (a_i, d_j), where a_i∈AList and d_j∈DList, such that a_i is an ancestor of d_j.

Structure join is a core technique in XML query processing, which directly impacts the efficiency of query processing. In substance, the mainstream method of structure join is maintaining an in-memory stack to skip some unnecessary nodes which don't participate in a join. However, when the memory buffer cannot hold all the elements of the two lists, it needs many I/O operations. Therefore, a novel algorithm BDC is proposed based on Bucket Divide and Conquer, which is not only efficient for BBTC, but also portable to other coding schemes. To determine the sequential relationship between two elements in the XML document tree, some partial order operators are introduced to accomplish the comparison between two elements. For BBTC, if node a and b are in different sub-blocks, only a.BID is compared with b.BID, or otherwise only a.IID is compared with b.IID.

Definition 7: Simple Operators $\prec', \succ', =', \le', \ge'$

$\forall a,b \in Z^+, a=(a_1a_2\cdots a_n)_2, b=(b_1b_2\cdots b_m)_2$
if $\exists\ j = \text{minimum}\{i \mid a_i \ne b_i$ *and* $i \le \text{minimum}(m,n)\}$ *then*
  if $a_j = 0$ *then* $a \prec' b$
  elseif $a_j = 1$ *then* $a \succ' b$
else
  if $m = n$ *then* $a =' b$
  elseif $n < m$ *then* $a \le' b$
  else $a \ge' b$

Definition 8: Complex Operators ' and '

a ' b *iff* $a \prec' b$ *or* $a \le' b$ *or* $a =' b$

a ' b *iff* $a \succ' b$ *or* $a \ge' b$ *or* $a =' b$

In fact, stack-based algorithms are applied to BBTC, but this cannot address the problem that it is inefficient when the memory is not enough to hold all the elements. The two element sets can be partitioned into different buckets, and only structure join of suited buckets is useful for the join results. That is, AList∞DList= $\bigcup_{i=1}^{n_b}$ (AList$_i$∞DList$_i$), where AList$_i$ and DList$_i$ denote different buckets of AList and DList respectively, n_b denotes the number of buckets, that is, only AList$_i$∞DList$_i$ are helpful to the results, and AList$_i$∞DList$_j$=∅(i≠j) or AList$_i$∞DList$_j$⊆AList$_j$∞DList$_j$. In section 5.2, this will be described in detail.

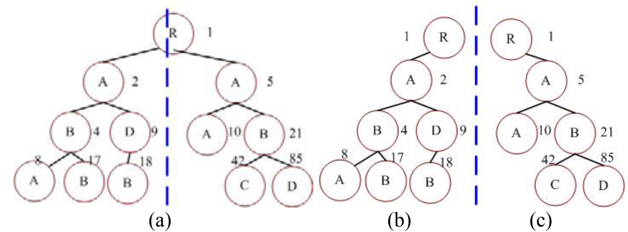


Figure 6. Partition an XML document tree into different buckets

For example, in Figure 6, 6(a) is partitioned into 6(b) and 6(c). A∞B in 6(a) is equivalent to A∞B in 6(b) union A∞B in 6(c); but $A_{6(b)}$∞$B_{6(c)}$ and $A_{6(c)}$∞$B_{6(b)}$ are unnecessary. Enlightened on the idea of this, AList and DList are partitioned into different buckets until memory can hold AList$_i$ or DList$_i$. The next section will explain how BDC fulfills this.

5.2 BDC

In BDC, AList and DList are partitioned into different buckets satisfying the following three conditions:

1) DList = $\bigcup_{i=1}^{n_b}$ DList$_i$ and DList$_i$ ∩ DList$_j$=∅ (i≠j)

2) $\bigcup_{i=1}^{n_b}$ AList$_i$ ⊆ AList

3) $\text{AList} \infty \text{DList} = \bigcup_{i=1}^{n_b} \text{AList}_i \infty \text{DList}_i$

Condition 1) means $\forall n_D \in$ DList, $\exists i$, $n_D \in \text{DList}_i$, and $\forall i, j$, if $i \neq j$, $n_D \in \text{DList}_i$ then $n_D \notin \text{DList}_j$. Condition 2) means there may be one element $n_A \in$ AList, but $\forall j$, $n_A \notin \text{AList}_j$, that is n_A does not have any children in DList. On the contrary, there may be one element n_A which is in different buckets, that is, this element has children in all of these buckets. The equation in condition 3) guarantees the correctness of the method. BDC can be applied to other coding schemes only assuring that they satisfy the three conditions. To apply those three conditions to the coding scheme, we may partition AList and DList into many different buckets, and make the size of each bucket under the capacity of memory buffer. However, in order to satisfy those three conditions, the number of buckets (n_b) and how to load nodes to different buckets must be determined. Suppose b_s is the number of nodes that memory can hold, $n_b = \lceil \text{minimum}(|\text{AList}|, |\text{DList}|) / b_s \rceil$. The range of the first bucket is[s,s+r), and the range of the i-th bucket is[s+(i-1)*r, s+i*r)($1 \leq i < n_b$), the range of the last bucket is[s+(n_b-1)*r, $\text{order}_{\max}$]. r and s are determined by the following equations[1].

$$d = |\log_2 order_{\max} - \log_2 order_{\min}|$$

$$r' = \begin{cases} order_{\max} - order_{\min} & \log_2 order_{\max} = \log_2 order_{\min} \\ (order_{\max} << d) - order_{\min} & \log_2 order_{\max} < \log_2 order_{\min} \\ order_{\max} - (order_{\min} << d) & \log_2 order_{\max} > \log_2 order_{\min} \end{cases}$$

$$r = \begin{cases} \left\lfloor r' / n_b \right\rfloor & r' \geq n_b \\ \left\lfloor (r' << \lceil \log_2 n_b \rceil) / n_b \right\rfloor & r' < n_b \end{cases}$$

$$s' = \begin{cases} order_{\min} & \log_2 order_{\max} \leq \log_2 order_{\min} \\ (order_{\min} << d) & \log_2 order_{\max} > \log_2 order_{\min} \end{cases}$$

$$s = \begin{cases} s' & r' \geq n_b \\ (s' << \lceil \log_2 n_b \rceil) & r' < n_b \end{cases}$$

There are two rules about how to assign nodes into different buckets.

Rule 1: Partitioning DList in BDC

$\forall n_D \in DList,$

$\quad if\ n_D.order = DList.\max\ or\ n_D.order \leq' DList.\max$

$\quad\quad then\ n_D \in DList_{n_b}$

$\quad else\ \ n_D \in DList_i$

$(i = \text{minimum}\{k \mid n_D.order \prec' DList_k.\max\ or\ n_D.order \leq' DList_k.\min\})$

Rule 2: Partitioning AList in BDC

$\forall n_A \in AList$

$if\ n_A <' DList.\min\ or\ n_A >' DList.\max\ or\ n_A \geq' DList.\max$

$\quad discard\ n_A$

$else\ n_A \in AList_i \qquad m \leq i \leq n$

$\quad m = \text{minimum}\{k \mid AList_k.\max \nless '2 * n_A.order\}$

$\quad n = \text{maximum}\{k \mid AList_k.\min \prec '2 * n_A.order + 1\}$

$\quad that\ is: i \in \{k \mid AList_k.\max \nless '2 * n_A.order\ and\ AList_k.\min \prec '2 * n_A.order + 1\}$

AList_k.min=DList_k.min=s+(k-1)*r; AList_k.max=DList_k.max= s+k*r.

Theorem 3: According to Rule 1 and Rule 2, AList and DList can be partitioned into different buckets and the partition satisfies the three conditions.

For example, suppose b_s=1, the XML document in Figure 6(a) can be partitioned into four buckets as shown in Table3.

Table 3. Result of partitioning AList and DList

<table>
<tr><td>order_{max}</td><td>order_{min}</td><td>d</td><td>n_b</td><td>r′</td><td>r</td><td>s′</td><td>s</td><td>.max</td></tr>
<tr><td>21</td><td>4</td><td>2</td><td>4</td><td>5</td><td>1</td><td>16</td><td>16</td><td>21</td></tr>
<tr><td colspan="4">AList</td><td colspan="4" rowspan="2">DList</td><td>.min</td></tr>
<tr><td colspan="4" rowspan="2">A2,A5,A8,A10</td><td>4</td></tr>
<tr><td colspan="5">B4, B17, B18, B21</td></tr>
<tr><td>AList1</td><td>AList2</td><td>AList3</td><td>AList4</td><td rowspan="2"></td><td>DList1</td><td>DList2</td><td>DList3</td><td>DList4</td></tr>
<tr><td rowspan="3">A2
A8</td><td rowspan="3">A2</td><td rowspan="3">A2</td><td rowspan="3">A5
A10</td><td>B4</td><td>B17</td><td>B18</td><td>B21</td></tr>
<tr><td>.min</td><td>16</td><td>17</td><td>18</td><td>19</td></tr>
<tr><td>.max</td><td>17</td><td>18</td><td>19</td><td>21</td></tr>
</table>

We can design our algorithm BDC according to the two rules. BDC, first partitions two data lists into different buckets in line 3, then joins suited buckets in memory in line 8. If both AList_i and DList_i can not be loaded into the memory buffer, it will continue to partition AList_i and DList_i in line 6.

AList and DList are partitioned until one of AList_i and DList_i can be loaded into memory. If $|\text{DList}_i| \leq b_s$, DList_i is loaded into memory, and AList_i is in disk. For each element n_A in AList_i, obviously its descendent n_D satisfies $2n_A.\text{order} \leq' n_D.\text{order} \prec' (2n_A.\text{order} + 1)$ (represented as $n_D \in [2n_A, 2n_A + 1)$), that is, all of n_A's descendents in DList_i must be in the range of $[2n_A, 2n_A + 1)$. When DList_i is in memory buffer, SJIM first sorts all elements in DList_i in line 21, then for each element n_A in AList_i, SJIM finds n_A's first descendent n_s and last descendent n_e in DList_i using the binary search in line 23. Thus, all elements in the range of $[n_s, n_e]$ are n_A's descendants. In this case, SJIM only needs |AList|+|DList|(read)+|SJR|(write) I/Os.

However, it is not the same for $|\text{AList}_i| \leq b_s$, that is, given node n_D, finding n_D's ancestors in AList_i is not very easy, because it is difficult to find a range except $[1, n_D]$ satisfying that all of n_D's ancestors are in that range. In this way, SJIM will classify AList_i (similar to partition) until (5-1) is satisfied.

$$|AList_{i_j}| = 1\ or\ AList_{i_j} = \{n_{A_1}, \ldots, n_{A_k}\}.\ \forall n_{A_t}, n_{A_{t+1}}\ in\ AList_{i_j},\ n_{A_t}\ is\ the\ ancestor\ of\ n_{A_{t+1}}\ (1 \leq t < k),\ that\ is,\ n_{A_1}, \ldots, n_{A_k}\ are\ in\ the\ same\ path\ from\ root\ to\ a\ leaf. \qquad \textbf{(5-1)}$$

[1] In this paper, $order_{max}$=$DList.max$ and $order_{min}$=$DList.min$, they denote the maximum order and minimum order in DList respectively according to the partial order operators $\prec', \succ'$.

For example, for classifying $AList_l$={the elements whose tag/lable is A in Figure 6(a)}, the result is that $AList_{l_1}$={A2,A8} and $AList_{l_2}$={A5,A10}.

Therefore, if $|AList_i| \le b_s$, then $AList_i$ is in memory. BDC classifies $AList_i$ in line 13 and sorts each $AList_{i_j}$ in memory, then given an element n_D in $DList_i$, IT finds n_D's nearest ancestor n_A in each $AList_{i_j}$ using binary search in line 15; in this way all of such n_A's ancestors are also ancestors of n_D. In addition, classifying and sorting $AList_i$ can be fulfilled in memory. In this case, SJIM also needs only |AList|+|DList|+|SJR| I/Os.

```
Algorithm 2 BDC(AList,DList,b_s)
Input:  AList, DList, b_s
Output: SJR={(a,d)|a∈AList,d∈DList and a is
an ancestor of d}
1. n_b=⌈minimum(|AList|,|DList|)/b_s ⌉;
2. SJR=∅;
3. partition AList, DList into AList_i, DList_i;
4. for i=1 to n_b
5.    if( minimum(|AList_i|,|DList_i|)>b_s ) then
6.       BDC(AList_i, DList_i, b_s);
7.    else
8.       SJR= SJR ∪ SJIM(AList_i, DList_i, b_s);
9.    endif
10.endfor.

SJIM (AList_k, DList_k, b_s)
Input:  AList_k, DList_k, b_s
Output: SJR_k={(a,d)|a∈AList_k,d∈DList_k and a is
an ancestor of d}
11. SJR_k=∅;
12. if(|AList_k| ≤ b_s)
13.    classify AList_k into AList_{k_i} until
           satisfying(5-1) and sort AList_{k_i}
14.    for  n_D ∈ DList_k
15.      find its nearest ancestor n_A' in each AList_{k_i} ;
16.      for  n_A ∈ AList_{k_i} and  n_A is an ancestor of n_A'
                (contain  n_A')
17.         SJR_k = SJR_k ∪ {( n_A , n_D )};
18.      end for
19.    end for
20. else  {|DList_k|≤  b_s }
21.    sort DList_k in memory;
22.    for each  n_A ∈ AList_k
23.       find the range of  n_A 's descendent in
          DList_k:[ n_s,n_e ];
24.       SJR_k= SJR_k ∪ {( n_A , n_D )};( n_D ∈[ n_s,n_e ])
25.    endfor
26. endif.
```

5.3 Analysis of BDC Algorithm

The following expression computes the I/O complexity of partitioning in BDC:

$$T(n)=\begin{cases} O(1) & n \le b_s \\ n_b * T(n/n_b)+n & n > b_s \end{cases}$$

`T(n)` denotes comparing times to partition AList and DList; n denotes number of nodes in AList or DList, when partitioning AList it is initialized as |AList |; while partitioning DList it is initialized as | DList |. n_b denotes number of buckets.

If the size of each bucket is nearly equal, BDC only scans datasets once for partitioning. In this case, `T(n)=|AList|+|DList|`. Even if in real-world data sets, BDC usually costs (`|AList|+|DList|`) for partitioning. In addition, when considering read and write, it costs 2(`|AList|+|DList|`). SJIM only costs `|AList|+|DList|+|SJR|`, which is used to read datasets to memory and write the result SJR to disk. As a result BDC costs (3(`|AList |+|DList|`)+`|SJR|`) I/Os. BDC is superior to stack-based algorithms, because previous algorithms first need to sort AList and DList, then read them to memory, the I/O cost is at least 2(`log|DList|*|DList|+log|AList|*|AList|`) +(`|AList|+|DList|+|results|`), in which sorting costs 2(`log|DList|*|DList|+log|AList|* |AList|`) I/Os and memory computing costs (`|AList|+|DList|+|results|`) I/Os. If consider maintaining indices, the I/O cost is more expensive.

BDC not only accelerates structure join by skipping more unnecessary elements, but also works for other coding schemes, such as Region Code. In that case, BDC just requires changing the general operators such as <, >, etc. to the corresponding simple operators in Definition 7.

6 Experimental Analysis of BBTC&BDC

Comprehensive experiments are conducted to study the effectiveness of BBTC and BDC. In experiment 1, B is determined, which is the key to reduce storage. Second, the time and space performances of BBTC are compared with those of Region Code: Zhang [21] and Dietz [10] in experiment 2. Then the update cost of BBTC is compared with Region Code in experiment 3. Last, since the best and most common structure join algorithm based on Region Code is XR-tree, BDC is compared with XR-tree in experiment 4. The experiments use the standard XMark [22], Shakespeare [23] and DBLP [24] datasets to test the algorithms. The experimental environment is a Windows 2000 machine with AMD2600 CPU and 1GB RAM. The programming language is standard C++.

6.1 Experiment 1: Determination of B in BBTC

Figure 7 (a) and (b) illustrate the choice of B value for Shakespeare and XMark documents respectively. They

present the space of the BBTC consumed with different B values for various XML documents. Each curve denotes an XML document. Since the lowest point of each curve represents the minimum space cost, and the B value at this point is its best choice. On the one hand, the curves with block-partition have observable advantages in storage size, but the stability of space cost with different values of B is another advantage revealed. Figure 7 (a) and (b) also illustrate the effect of different values of B on space performance for Shakespeare and XMark respectively. Compared with Non-block, BBTC reduces 30% storage space in Shakespeare data and 73% in XMark data. In addition, it can be concluded from the analysis that the best B value is related to the number of nodes in the XML document. Figure 8 indicates the relation between the best `B` value and `n` (the number of nodes in the XML document). Since the relation between the best `B` value and `n` meets logarithmic normal distribution, the best `B` value can be represented by `log(n)` approximately, which exactly complies with the analysis in Theorem 2 of section 4.3.

6.2 Experiment 2: Cost of Update

BBTC supports dynamic updating of XML documents, and when XML Documents updated, the cost of updating codes in BBTC is less than Region Code dramatically. In Figure 9, when only inserting or deleting a node, Region Code nearly needs to recode the whole document, but BBTC only changes a few codes, that is why the columns of BBTC are invisible in Figure 9.

6.3 Experiment 3: Comparison of Time and Space Performance

Region Code is asymptotically minimal in space performance, however BBTC has superior space performance to the Region Code, because the Region Code maintains two large numbers for one code, but `sibling_order` in BBTC is very small and needs little storage.

6.4 Experiment 4: Comparison of Structure Join Algorithms

In this section, different data sets are used to compare BDC with XR-tree. They are real-world XML data, i.e. XMark [22] and DBLP [24]. DBLP is a set of bibliography files, the size of the raw text files is around 53.3MB. The benchmark (XMark) data is generated with SF(scale factor) = 1, and the raw text file is 113MB. Six structure joins are selected for the DBLP data, namely DSD1, DSD2, ... , DSD6. Similarly, for XMark, they are DSX1,DSX2, ... , DSX6. Six structure joins are also selected for synthetic data Bookset.xml which is similar to Figure1, namely DSB1, DSB2, ... , DSB6. In addition, in order to compare the two algorithms, structure joins are selected with different ratios, namely DS-A/D1, DS-A/D2, ... , DS-A/D6(|AList|<|DList|) and DS-D/A1, DS-D/A2, ... , DS-D/A6(|DList|< |AList|). The statistics of the data sets are shown in Table 4.

Table 4. Statistics of data sets

Name	|AList|	|DList|	Name	|AList|	|DList|	Name	|AList|	|DList|
DSX1	9750	35	DSD1	105754	294470	DSB1	1	3571
DSX2	21750	43500	DSD2	105754	171071	DSB2	1	3621
DSX3	21750	48250	DSD3	801	184465	DSB3	50	3621
DSX4	25500	12823	DSD4	2326	4969	DSB4	50	3571
DSX5	10830	59486	DSD5	84095	13660	DSB5	3571	3621
DSX6	25500	48250	DSD6	84095	82980	DSB6	3571	26728
DS-A/D1	801	184465	DS-A/D5	25500	48250	DS-D/A3	8765	1256
DS-A/D2	3571	267287	DS-A/D6	105754	171071	DS-D/A4	9750	2685
DS-A/D3	2326	49969	DS-D/A1	9750	35	DS-D/A5	13500	12823
DS-A/D4	10830	59486	DS-D/A2	8765	454	DS-D/A6	15071	13660

To compare BDC with XR-tree, Improved Ratio (IR) is defined:

$$IR=(T_{XR\text{-}tree}-T_{BDC})/T_{XR\text{-}tree}$$

where $T_{XR\text{-}tree}$ and T_{BDC} are the elapsed times for XR-tree and BDC algorithm respectively.

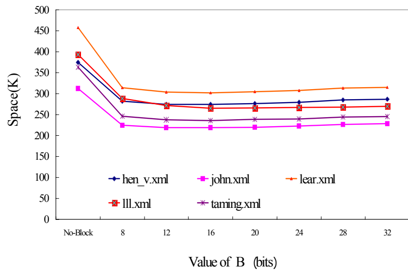
First, BDC is better than XR-tree in different data sets:

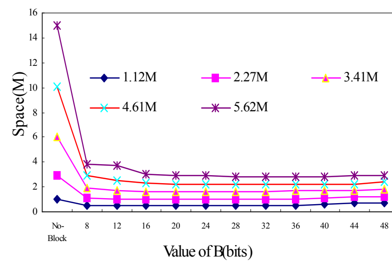
(1) Real-world data sets: for XMark, its IR is nearly 80% in DSX6 (Figure 12(a)); for DBLP, its IR exceeds 90% in DSD1 (Figure 12(b)).

(2) Synthetic data set: for Bookset.xml, its IR is nearly 1 in DSB6 (Figure 12(c)).

Second, BDC is compared with XR-tree on different buffer sizes. Because XR-tree needs large memory to sort data sets and store indices, when memory size is limited, its performance declines dramatically. As shown in Figure 13, when buffer size is 0.5% of the data sets, the performance of XR-tree is far worse than BDC; and even if the buffer size is 10% or more, it is not as good as BDC, where the datasets are from XMark with |AList|=21750 and |DList|=69969.

Last, in real queries, the sizes of AList and DList are usually not equal, even quite different, that is, |AList| is far less than |DList| or vice versa. In this way, sorting data sets is not efficient. As a result, through simply partitioning the smaller data set and loading it to memory, we can only scan the other sets once to finish the structure join. Figure 14(a) and (b) reflect the efficiency of BDC, and their IRs reach 1 in most of DS-A/Ds and DS-D/As.





(a) Shakespeare (b) XMark

Figure 7. The effect of different B values on space performance

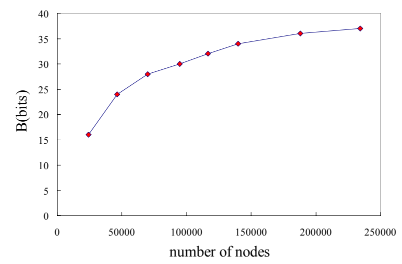


Figure 8. The best B values vs. n.

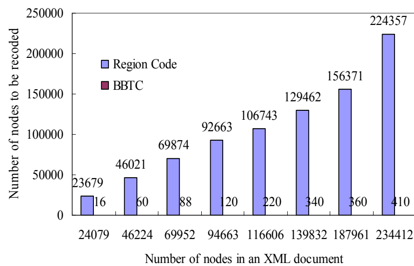
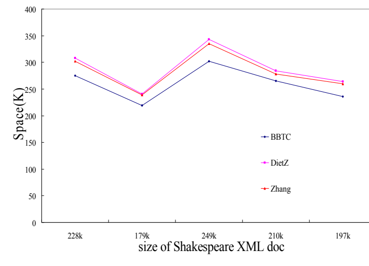
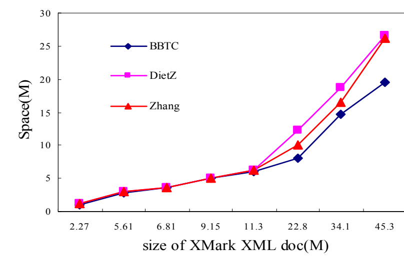


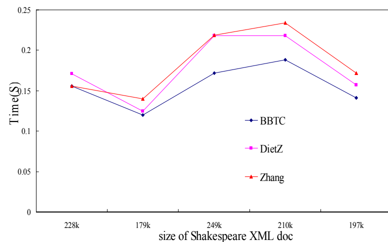
Figure 9. Cost of update

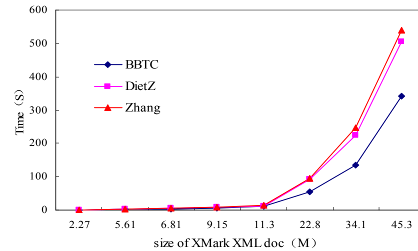




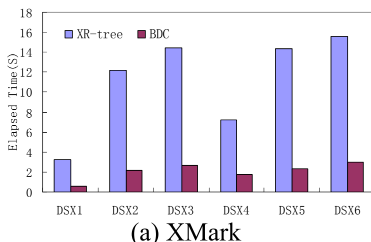
(a) Shakespeare (b) XMark

Figure 10. Comparison of space performance

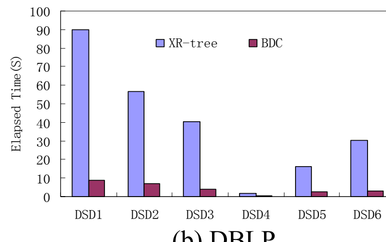




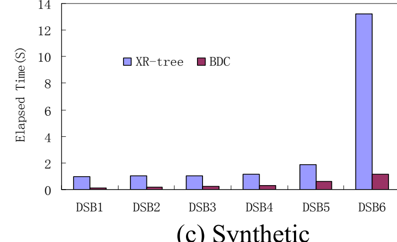
(a) Shakespeare **Figure 11.** Comparison of time performance (b) XMark



(a) XMark



(b) DBLP



(c) Synthetic

Figure 12. Elapsed time with different data sets

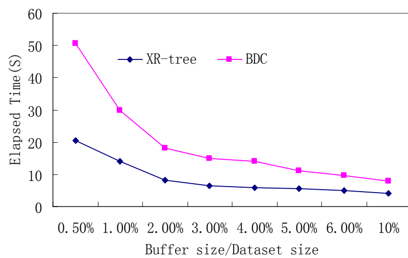
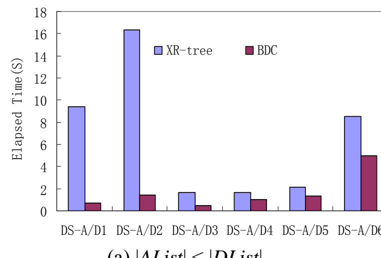
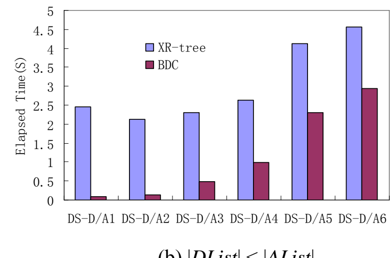


Figure 13. Elapsed time with different buffer



(a) $|AList| < |DList|$



(b) $|DList| < |AList|$

Figure 14. Elapsed time with different ratios between $|AList|$ and $|DList|$

7. Conclusion

In this paper, firstly a new update-aware coding scheme is proposed based on the binary-tree, which not only codes the XML documents easily and infers relationship between nodes rapidly, but also supports XML documents update effectively.

Second, to save storage space, BBTC is presented, which partitions an XML document into sub-blocks and reduces the average code length to O(log(n)).

At BDC is proposed, which is more efficient than previous algorithms when memory buffer cannot hold the unsorted input element sets. BDC partitions an XML document tree into different buckets and only the structure joins of suited buckets are helpful to the result, that is, BDC accelerates structure join when input element sets are out-of-order. BDC, not only accelerates structure join based on BBTC without any indices, but also has good portability, that is, it can be applied to other coding schemes.

Our experiments have proved that both the coding scheme BBTC and the structure join algorithm BDC significantly outperform the existing studies.

References

[1] S Abiteboul, D. Quass, J. McHugh et al. The Lorel query language for semi-structured data Int'l Journal on Digital Libraries, 1997.

[2] S. AI-Khalifa, H.V. Jagadish, N. Koudas, J.M. Patel, D. Srivastava, Y.Q. Wu. Structural joins: A primitive for efficient XML query pattern matching. ICDE, 2002.

[3] D. Chamberlin et al. XQuery : A query language for XML. WWW, 2001

[4] Y. Chen et al. L-Tree: a dynamic labeling structure for ordered XML data. In Proc. of the 2004 Int'l Workshop on Database Technologies for Handling XML Information on the Web.

[5] S.Y. Chien et al. Efficient complex query support for multi-version XML documents. EDBT, 2002.

[6] S.Y. Chien et al. Efficient structural joins on indexed XML documents. VLDB, 2002.

[7] J. Clark, Steve DeRose. XML path language (XPath) . W3C Recommendation World Wide Web Consortium, 1999.

[8] E. Cohen, H. Kaplan, and T. Milo. Labeling dynamic XML trees. In Proc. of the 2002 ACM Symp. on Principles of Database Systems, 2002.

[9] Alin Deutsch, Mary Fernandez, Daniela Florescu et al. A query language for XML. WWW, 1999.

[10] Paul F Dietz. Maintaining order in a linked list. The 14th Annual ACM Symp. on Theory of Computing, 1982.

[11] D. Florescu, D. Kossman. Storing and Querying XML Data using an RDBMS, IEEE Data Engineering Bulletin, 1999.

[12] D. D. Kha, Masatoshi Yoshikawa, and Shansake aemara. An XML indexing structure with relative region coordinate. ICDE, 2001.

[13] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. VLDB, 2001.

[14] H.F. Jiang, H.J. Lu, W. Wang, B.C. Ooi. XR-Tree: Indexing XML data for efficient structural joins. ICDE 2003.

[15] A. Silberstein et al. BOXes: Efficient maintenance of order-based labeling for dynamic XML data. Technical report, Duke University, 2004.

[16] A. Silberstein, H. He, Ke Yi, Jun Yang. BOXes: Efficient Maintenance of Order-Based Labeling for Dynamic XML Data. ICDE, 2005.

[17] Igor Tatarinod, Stratis D. diglas, Kedin Beyer, and Chan Zhang. Storing and querying ordered XML using a relational database system. SIGMOD, 2002.

[18] W.Wang, H.f. Jiang, Hongjun Lu, and Jeffrey Xu Yu. PBiTree coding and efficient processing of containment joins. ICDE, 2003.

[19] N. Wirth. Type Extentions. ACM Transaction on Programming Languages and systems, 1988.

[20] X. Wu et al. A prime number labeling scheme for dynamic ordered XML trees. ICDE, 2004.

[21] C. Zhang et al. On Supporting Containment Queries in Relational Database Management Systems. SIGMOD, 2001.

[22] http://www.xml-benchmark.org/

[23] http://www.xml.com/pub/r/396

[24] http://dblp.uni-trier.de/xml/

Guoliang Li, male, Ph.D candidate in Tsinghua University. His main research interests are in the areas of database, XML, semantic cache, web service, and data mining.

Jianhua Feng, male, professor in Tsinghua University. His main research interests are in the areas of database, XML, data warehouse, and semantic cache.

Na Ta, female**,** master in Tsinghua University. Her main research interests are in the areas of XML, Semantic cache.

Lizhu Zhou, male, professor in Tsinghua University. His main research interests are in the areas of database, distributed database, digital library, data mining and massive storage.