PBiTree Coding and Efficient Processing of Containment Joins[*]

Wei Wang Haifeng Jiang Hongjun Lu [†]
Dept. of Computer Science
The Hong Kong University of Sci. & Tech.
Hong Kong, China
{fervvac, jianghf, luhj}@ust.hk

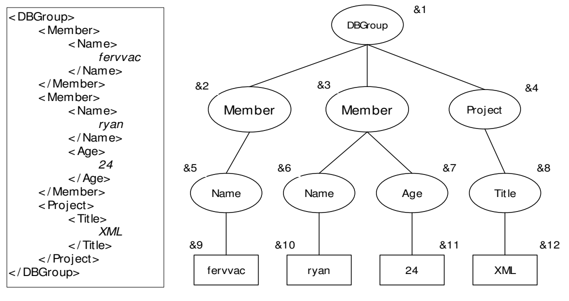
Jeffrey Xu Yu
Dept. of Systems Engineering
and Engineering Management
The Chinese University of Hong Kong
Hong Kong, China
yu@se.cuhk.edu.hk

Abstract

This paper addresses issues related to containment join processing in tree-structured data such as XML documents. A containment join takes two sets of XML node elements as input and returns pairs of elements such that the containment relationship holds between them. While there are previous algorithms for processing containment joins, they require both element sets either sorted or indexed. This paper proposes a novel and complete containment query processing framework based on a new coding scheme, PBiTree code. The PBiTree code allows us to determine the ancestor-descendant relationship between two elements from their PBiTree-based codes efficiently. We present algorithms in the framework that are optimized for various combinations of settings. In particular, the newly proposed partitioning based algorithms can process containment joins efficiently without sorting or indexes. Experimental results indicate that the containment join processing algorithms based on the proposed coding scheme outperform existing algorithms significantly.

1. Introduction

This paper addresses issues related to containment join processing in tree-structured data. By tree-structured data, we mean the data that can be modelled by trees or their variants. Examples of such data include textual documents, XML data and semi-structured data. With the rapid development of the Internet and the World-Wide-Web, and the trend of XML being a standard of information exchange and sharing over the Internet, querying such type of data has received great attention recently. Tree-based models were also proposed to model such data, such as the Document Object Model (DOM). With the tree models, data objects, e.g., elements, attributes, text data, etc, are modelled as nodes of a tree, and relationships among the data elements are represented by edges connecting them. Figure 1(a) is a simple example of XML data and Figure 1(b) is the tree that models the data, where the internal nodes are elements, the leaf nodes are text data, and the edges represent nesting between an internal node and its child nodes. Note that the containment relationship is equivalent to the ancestor-descendant relationship in the tree data model.



```
<DBGroup>
    <Member>
        <Name>
            fervvac
        </Name>
    </Member>
    <Member>
        <Name>
            ryan
        </Name>
        <Age>
            24
        </Age>
    </Member>
    <Project>
        <Title>
            XML
        </Title>
    </Project>
</DBGroup>
```

(a) An XML Document   (b) An XML Data Tree

Figure 1. An XML document and its data tree

Different from “flat” data in relational database systems, data with structures can be queried by values, structures, or their combinations. Although XML query languages proposed so far have different syntaxes, features, and constructs, it is identified that queries based on containment relationship are a core component to all such query languages [20, 12]. For example, the XQuery query

//Section[Title=“Introduction”]//Figure

finds all figures that are *contained* in a section with title “Introduction”. Such query is also termed as *containment query* and can be efficiently processed by containment join algorithms [1, 4]. Formally, we define containment join as follows.

Definition 1 (Containment Join) *Given an ancestor set* $A = \{a_1, a_2, \ldots, a_m\}$ *and a descendant set* $D = \{d_1, d_2, \ldots, d_n\}$ *(where both* A *and* D *consist of nodes*

[*]The project was partly supported by the Research Grant Council of the Hong Kong SAR, China (grants AoE/E-01/99, HKUST6060/00E).
[†]This author's work was also supported by a grant from the National 973 project of China (No. G1998030414).

from the data tree), a containment join of A and D, denoted as $A \lhd_{A,D} D$, returns all tuple pairs (a_i, d_j) where $a_i \in A$ and $d_j \in D$ such that a_i is an ancestor of d_j.[1]

The importance of the containment query lies in the fact that it is commonly believed as the core part of XML query processing engine [1]. It is identifed that all XML queries based on structural conditions can be decomposed into a series of sub containment queries [12]. Many algorithms have been proposed recently to process such joins efficiently. They are all based on certain coding schemes. A good coding scheme makes it possible to check the ancestor-descendant or parent-child relationship of two elements in $O(1)$ time without accessing data other than the elements themselves. For example, in the most popular *region coding scheme*, each element is tagged with a region in the form of $(Start, End)$, which records the (logical or physical) starting and ending positions of the element. To check the ancestor-descendant relationship (which is equivalent to containment relationship) between x and y, it suffices to check whether $(x.Start < y.Start \wedge y.End < x.End)$ holds.

However, existing containment join algorithms make one of the following assumptions: 1) both element sets have indexes or 2) both element sets are sorted or 3) both. [20] proposed index nested loop join algorithm which requires B$^+$-tree indexes on the input element sets. [16] proposed converting the containment join to spatial containment join and using R-Trees to process the join efficiently. [20, 12] proposed a sort merge based join algorithm which requires both input element sets sorted. [1] improved the previous algorithm by using a stack. Their stack-tree algorithms have optimal worst-case I/O and CPU complexities. More recently, [4] considered the case when both element sets are sorted and there are B$^+$-tree indexes on them. Their Anc_Des_B+ algorithm can make use of the additional indexes to skip part of the elements that will not participate in the join.

Note that not all elements in an XML database are sorted or indexed. Furthermore, indexes do not exist on intermediate results. To apply the above mentioned existing algorithms, such datasets have to be either indexed or sorted on the fly, which will be costly. To the best of our knowledge, there has been no known efficient method to join two datasets which are neither sorted nor indexed, except for some naïve solutions that usually do not perform well. Motivated by this observation, we propose in this paper a new coding scheme called PBiTree coding. We show that PBiTree codes are more versatile than the traditional region codes in that 1) it can support verification of ancestor-descendant relationship efficiently, which is the fundamental property required by any coding scheme and 2) it can be efficiently converted to region codes. We also propose a set of novel partitioning based algorithms that can efficiently process containment join by making full use of the good properties offered by PBiTree codes. They are demonstrated experimentally to be superior to existing algorithms when neither sorted data or indexes are available. Meanwhile, we show how existing algorithms can be adapted to work with PBiTree encoded data with little overhead.

[1] $A \lhd_{A,D} D$ can be abbreviated to $A \lhd D$ if there is no ambiguity.

The contributions of our work reported in this paper can be summarized as follows.

1. We propose to use a new scheme, PBiTree coding, to assign identifiers to data elements. The coding scheme has a nice property that, given two element identifiers, we can determine whether one element is an ancestor of the other with minimal computation cost without need of any other information. Furthermore, the coding scheme is independent from the physical data organization. It can be easily implemented along with any data storage scheme proposed in the literature, and the containment join can be efficiently processed based on the identifiers. Discussion of various issues of PBiTree code, in comparison with the popular region code, is also presented.
2. We developed a set of algorithms that process containment join where the elements in the sets are identified by their PBiTree codes. While data encoded in PBiTree codes can still take advantage of the state-of-the-art structural join algorithms, new algorithms based on horizontal and vertical partitioning schemes are also possible choices. We show that the new partitioning based structural join algorithms are *complementary* to the state-of-the-art join algorithms. In particular, they are favorable when none of the input sets is sorted or indexed.
3. We conducted extensive experiments of the proposed algorithms based on PBiTree code. The results verified our analysis that the newly proposed algorithms can outperform existing ones significantly when none of the data is sorted or indexed and provided insight into the relative performance of different algorithms in different parameter settings.

We would like to emphasize the significance of the work reported here. With the newly proposed coding scheme and related containment join algorithms, we have a complete set of efficient containment join processing algorithms for input datasets with different characteristics, as listed in Table 1. As a result, our work broadens the scope of choices for XML query optimizers. We believe that it has positive impact to the query processing and optimization of XML data. The techniques and related issues in transforming an XML query into logical processing plans including containment joins are themselves an open research area, and they are out of the scope of this paper. Interested readers can refer to [12] for a preliminary study.

The remainder of the paper is organized as follows. Section 2 describes the PBiTree coding scheme and discusses its related issues. Section 3 presents our containment join processing algorithm framework based on the PBiTree

Table 1. Selection of containment join algorithms

Index	Sorted	Existing Coding Scheme	With PBiTree Coding
$\surd$	$\times$	Index Nested Loop	Index Nested Loop[a]
$\times$	$\surd$	Stack-Tree Algorithms	Stack-Tree Algorithms[a]
$\surd$	$\surd$	Anc_Des_B+[b]	Anc_Des_B+[b] [a]
$\times$	$\times$	*Unknown*	*MHCJ+Rollup or VPJ*

[a] Adapted for PBiTree encoded data.

[b] *XR-stack* has been shown to outperform Anc_Des_B+ algorithm in [8].

code. Section 4 presents the experimental results and Section 5 presents the related works. Section 6 concludes the paper.

2. PBiTree and PBiTree coding for tree structured data

In this section, we introduce the PBiTree and its properties, followed by an efficient algorithm to assign identifiers to elements in structured data using a PBiTree. We conclude this section with a brief discussion.

2.1. PBiTree

Definition 2 *A PBiTree is a tagged perfect binary tree, where each non-leaf node has two children and all leaf nodes are at the same level. Each node is tagged (or encoded) with a number (of the* `in-order` *of traversal of the tree). The number associated with a node is called the* code *of that node.*

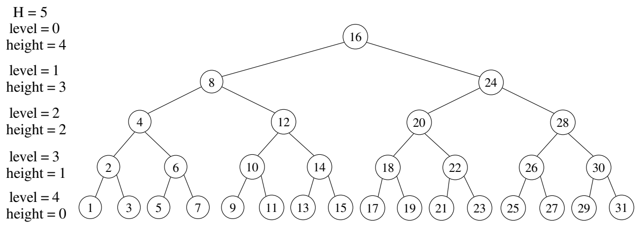


Figure 2. An example PBiTree

An example PBiTree is shown in Figure 2. The height of the tree, H, is 5. Leaf level nodes have *height* equal to 0. For ease of reference, we also marked the *level* of tree from root to leaf such that the level of root is 0. The code of each node is also shown in the figure. The code of a node n_i is referred to as $n_i.Code$. In the following discussion, we use the node (n_i) and its code ($n_i.Code$) interchangeably, because the code uniquely identifies a node. The important properties of PBiTree coding are as follows.

Property 1 *For a given node n_i of a perfect binary tree, its ancestor n_j at a given height h_j can be directly calculated by $\mathcal{F}(n_i, h_j) = 2^{h_j+1} \cdot \lfloor n_i/2^{h_j+1} \rfloor + 2^{h_j}$.*

For example, for the node with code 18 in Figure 2, its ancestor at height 2 is encoded as $2^{2+1} \cdot \lfloor 18/2^{2+1} \rfloor + 2^2 = 20$. Similarly, the codes of its ancestors at height 3 and 4 are exactly 24 and 16, respectively. It is important to note that the $\mathcal{F}$ function can be evaluated using shifting and integer operations and will be fast on modern architectures. For example, $\lfloor 18/2^{2+1} \rfloor$ is to right-shift 18 by 3. No floating point operations are needed.

Property 2 *Given the code of a node n, its height $height(n)$ is the position of the rightmost '1' bit in its binary representation. The level of a node, therefore, can be obtained by $H - height(n) - 1$.*

For example, in Figure 2, code 18 is for a node at height 1, because the rightmost '1' bit is the second rightmost bit in the binary representation 10010_2 (= 18_{10}) (position 1). Its level is $5 - 1 - 1 = 3$. In other words, the height/level information of a node is also encoded in its code.

With the above properties, we have the following lemmas.

Lemma 1 *Given two nodes n_i and n_j in a PBiTree, n_i is an ancestor of n_j if and only if $n_i = \mathcal{F}(n_j, height(n_i))$.*

Lemma 1 shows that, given two nodes n_i and n_j, we can simply check whether n_i is an ancestor of n_j. By this lemma, as we will see later, if the elements involved in a containment join are PBiTree coded, the join can be evaluated in a similar way as the equality join in relational database systems.

Lemma 2 *For any node n in the PBiTree, let l be the level of n and α be the zero-based position index of element nodes from left to right, i.e. $\alpha \in [0, 2^l - 1]$, then $n.Code = \mathcal{G}(\alpha, l)$, where $\mathcal{G}(\alpha, l) = (1 + 2\alpha) * 2^{H-l-1}$.*

We term such (l, α) code as *top-down* PBiTree code, or top-down code. Lemma 2 shows the equivalence between the (original) PBiTree code and the top-down PBiTree code. The latter is used in the PBiTree construction algorithm discussed in the following sections. For example, for node 18, it is the 5-th node on the 3rd level, therefore its top-down code is $(4, 3)$ and the PBiTree code can be obtained by $\mathcal{G}(4, 3) = (1 + 2 * 4) * 2^{5-3-1} = 18$.

2.2. Encoding tree structured data with PBiTree

Tree structured data, as shown in Figure 1(b), is usually not modelled as a perfect binary tree. In order to make use of the properties of perfect binary trees, we *embed* the original data into a corresponding PBiTree. Let u_i and u_j be nodes in the original data tree. The relationship between the PBiTree and the original data tree can be described using an injective function h, such that

1. $h(u_i) = h(u_j)$ if and only if $u_i = u_j$.
2. $h(u_i)$ is an ancestor of $h(u_j)$ in the PBiTree if and only if u_i is an ancestor of u_j in the original data tree.

Figure 3 shows a PBiTree that corresponds to the data tree in Figure 1(b). The codes of the elements in the original data are obtained from their corresponding nodes in the PBiTree. For example, the code of "`&9 (fervvac)`" is 1. From Figure 3, we can see that, in the PBiTree representation, there are some virtual nodes which do not exist in the original data tree. All the unshaded nodes in the figure are virtual nodes.

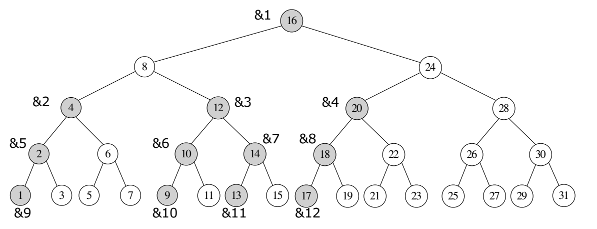


Figure 3. Embed a data tree into a PBiTree

We call the process of embedding a data tree in a PBiTree *binarization*. The PBiTree code for each data tree node can be obtained during the binarization process. Given a data tree, there are a large number of ways to binarize it. Currently one simple yet effective binarization algorithm is used (by calling Algorithm 1 with $(T, root(T), 0, 0)$). The key observation used in the algorithm is that binarization of a data tree can be done by 1) binarize the current node and its child nodes and 2) recursively binarize the subtrees rooted at each child node. For the first step we use a heuristic that places all child nodes of a node in the data tree contiguously at the *same* level in the PBiTree, which will assist processing containment and proximity queries. If the level of the mapped parent node is l, the first possible level to place all the child nodes is $l + k$, such that $2^k \geq n$. For example, suppose a node A has three child nodes in the data tree. The child nodes will be mapped to 2 levels below the mapped node of A in the resultant PBiTree, as $2^2 = 4 > 3$.

The algorithm BinarizeTree (Algorithm 1) makes use of the equivalence between PBiTree codes and their corresponding top-down PBiTree codes (See Lemma 2). Initially, the top-down code for the root node is set to $(0, 0)$ and in each recursive call, the top-down codes, i.e., l and α values for all of its child nodes are calculated before further recursive calls (line 6). The time complexity can be shown to be $O(n)$, where n is the number of nodes in the data tree.

For example, the PBiTree shown in Figure 3 is obtained from the data tree shown in Figure 1(b) after binarization. The binarization starts from the root node (`&1`), with top-down code $(0, 0)$. Thus the PBiTree code for the root node is $\mathcal{G}(0, 0) = 16$. The node in question has 3 child nodes (`&2, &3, &4`). By our heuristic, they are placed at two levels lower than their mapped parent node in the PBiTree ($k = 2$). Thus, their α and l values ($(0, 2)$, $(1, 2)$ and $(2, 2)$) are passed on to the recursive calls respectively. Note that the codes of the non-virtual nodes are determined solely by their position in the data tree and thus no virtual node in the PBiTree is physically generated.

Algorithm 1 BinarizeTree(T, e, α, l)

Input:
T is an arbitrary data tree. e is a node in T and (l, α) is the top-down code of e.

Output:
Every node n in T has the correct PBiTree code in $n.Code$.

Description:

```
1: $e.Code = \mathcal{G}(\alpha, l)$ {From Lemma 2}
2: **if** $e$ is not a leaf node **then**
3:    $n = NumOfChild(e)$
4:    $k = \lceil \log_2 n \rceil$.
5:    **for all** child node $e_i$ (the $i$-th child node from left to right) **do**
6:       BinarizeTree($T$, $e_i$, $2^k \cdot \alpha + i - 1$, $l + k$);
7:    **end for**
8: **end if**
```

2.3. Discussions

In this subsection, we compare the proposed coding scheme with other existing coding schemes, followed by a brief discussion on issues that astute readers may raise.

2.3.1 Comparison of coding schemes

Introduction to various codes for XML data is presented in Section 5. Here, we will highlight the difference between region-based coding scheme and our PBiTree based coding scheme: a) PbiTree based coding scheme encodes nodes of a special perfect binary tree constructed from the original data tree and only one code is needed instead of two required by region-based schemes, and b) PbiTree code contains richer structure information and thus provides support for efficient query processing. For example, with the help of $\mathcal{F}(n, h)$ function, given any node n_i, we can calculate the code of its ancestor at a given height h efficiently, without accessing any additional data. This property is heavily used in our newly proposed algorithms to be introduced later. Such structural information, however, cannot be obtained for region-based schemes without accessing additional data together with expensive calculations, and c) PBiTree code is more versatile than previously proposed scheme in the sense that it is easy to convert PBiTree code to other codes (in particular, region codes) efficiently using local information only while the reverse conversion is costly and requires global information. The conversion formulae are given in Lemma 3 and Lemma 4.

Lemma 3 *Given a node n, let H be the height of the PBiTree and h be the height of n, i.e., $h = height(n)$, $(n - (2^h - 1), n + (2^h - 1))$ can serve as the region code of n in the form of $(Start, End)$.*

Lemma 4 *Given a node n, let H be the height of the PBiTree and h be the height of n, i.e., $h = height(n)$, the binary representation of $n \gg h$ can serve as the prefix code of n ($\gg$ is the right shift operator).*

One important implication from the advantages PBiTree offers is that we can process queries using existing algorithms originally developed for other coding schemes, as will be illustrated in Section 3.

2.3.2 Virtual nodes

It is important to note that the virtual nodes have little impact on the performance of PBiTree. This is simply because they are never physically generated or stored. Interestingly, virtual nodes , on the other hand, may serve as placeholders and thus be advantageous to update.

2.3.3 Coding space

The coding space represented by a PBiTree of height H is $[1, 2^H - 1]$, and thus each code requires H bits. However, many real datasets, such as the DBLP and Benchmark data [18] used in our experiments, have corresponding binary trees within a constant number of levels. Therefore, although the height of the PBiTree could be $O(n)$ (where n is the number of nodes in the data tree) in the worst case, we expect such highly skewed data distribution rare in practice and our method still applicable to most real-world datasets.

3. Containment join algorithms

In this section, we present the framework of a set of containment join algorithms that work on data encoded in PBiTree code. Most of the previous algorithms are based on data encoded in region code. In the previous section, we have demonstrated the advantages of PBiTree code over region code. Consequently, we will show that 1) all the algorithms previously proposed for region code can be trivially adapted to algorithms working on PBiTree code and 2) we propose a class of new partitioning based algorithms that requires no sorting or indexes. These new algorithms are shown to outperform existing algorithms in our experiments. Therefore, we propose a novel containment query processing framework that includes above-mentioned algorithms working on PBiTree coded data.

We will first briefly introduce how to modify previous algorithms based on region code to work with data in PBiTree code. Those algorithms are classified as *non-partitioning* algorithms as they do not partition the data. In contrast, we next focus on two classes of partitioning based algorithms: horizontal partitioning algorithms and vertical partitioning algorithms, which partition the data in horizontal and vertical directions respectively.

In the following, we will use A and D to denote the ancestor and descendant sets, respectively. $|R|$ and $||R||$ denote the number of elements (tuples) in set R and the number of disk pages of R, respectively. B denotes the disk page size as well as the memory buffer page size. b denotes the number of buffer pages.

3.1. Non-partitioning algorithms

This class of algorithms is originally designed for data coded in region format, e.g., $(Start, End)$. PBiTree codes can be efficiently converted to such format on the fly and thus these algorithms are still applicable. Due to the limited space, we will only briefly introduce the algorithms and the necessary modification. Interested readers can refer to the original papers for further details.

Index based The index nested loop (INLJN) algorithm [20] iterates over the outer relation, probes the index built on the inner relation and outputs the result. Given the $(Start, End)$ information of each node, the containment relationship of (a, d) holds if and only if $(a.Start < d.Start) \wedge (d.End < a.End)$. The condition can be simplified as $(a.Start < d.Start) \wedge (d.Start < a.End)$ for well nested structures, such as trees. In [20], B$^+$-tree index is considered. Index-probing D with A can be efficiently supported because it is to search all $d.Start$ within a given range $(a.Start, a.End)$. However, index probing A with D is *not* efficient, because B$^+$-tree can only support lexicographical order on compound keys, and thus it results in many unnecessary node accesses. We propose to use disk based interval tree for this case [7].

Since index probing will be most likely to result in random I/O, we use a heuristic that minimizes such index probing and thus overall cost as follows: We use the smaller set as the outer relation and use appropriate index to probe the inner relation for each element in the outer relation. Therefore, the I/O cost is $\min(||A|| + |A| * O(\log |D|), ||D|| + |D| * O(\log |A|))$.

To use index nested loop algorithms for PBiTree encoded data, we only need to have 1) a custom index building module, which builds appropriate indexes based on the region code calculated from the PBiTree code for each element and 2) calculate the $(Start, End)$ code on the fly during the join process.

Sort-merge based with or without indexes Multiple Predicate Merge Join (MPMGJN), or its equivalent $\mathcal{EE}$-join, sorts both the ancestor set and the descendant set according to the $(Start, End)$ and $Start$ attribute(s), respectively, and merges them according to the ancestor-descendant relationship. The merge process will scan A once, but possibly repeatedly scan segments of D multiple times. Stack tree join algorithms were later proposed in [1]. Their algorithms improve upon MPMGJN by using a stack to hold adjacent elements (from A or D) that have ancestor/descendant relationships. Their algorithms are shown to have the optimal worst case CPU and I/O cost. Furthermore, their algorithms can output results in either A or D sorted order, which is favorable for further containment joins. The I/O cost of the algorithms is $O(||A|| + ||D||)$.

More recently, Anc_Des_B+ algorithm [4] was proposed to utilize additional indexes (B$^+$-tree or R-Tree) to accelerate the stack-tree algorithms. The basic idea is to use indexes to skip scanning/comparing those data that will not participate in the join. The I/O cost is still $O(||A|| + ||D||)$, but the cost tends to be much smaller for many real datasets.

To use the above algorithms for PBiTree encoded data, we only need 1) a custom sorting routine that can sort data encoded in PBiTree code in either $Start$ or $Start, End$ order by online conversion between the codes and 2) calculating the $(Start, End)$ code on the fly in the join process and 3) a custom index building module (as that required in INL case) if we want to use Anc_Des_B+).

Summary The efficient conversion from PBiTree codes to the corresponding region codes enables us to use the state-of-the-art structural join algorithms for our PBiTree encoded data with little additional cost. It is obvious that the adapted algorithms have the same performance with the original algorithms in terms of disk I/O. Therefore, they are still the preferred algorithms in our framework under their corresponding settings.

Next we will present newly proposed partitioning based algorithms unique to the PBiTree encoded data. They do not need sorting or index and thus complementary to the above-mentioned algorithms. They are further classified into horizontal partitioning algorithms and vertical partitioning algorithms.

3.2. Horizontal-partitioning algorithms

The horizontal partitioning algorithms take advantage of the property that the containment relationship can be determined efficiently in a PBiTree, and are based on equijoins rather than θ-joins. The rationale is that evaluation and optimization techniques for equijoin operations are already mature, compared with those for θ-join.

Single height containment join (SHCJ) We first introduce a basic algorithm to evaluate containment joins when all nodes in the ancestor set are at the same height of the tree. We call it *single height containment join*, denoted as SHCJ. SHCJ algorithm shows that in this case PBiTree encoded data enables us to process such containment joins efficiently by taking advantage of the highly optimized equijoin evaluation techniques.

The SHCJ algorithm (shown in Algorithm 2) evaluates containment join $A \lhd D$ as to evaluate an equijoin $A \bowtie_{A.Code=\mathcal{F}(D.Code,h)} D$. Here, h is the identical height of all nodes in A. $D.Code$ is the node code for a node $d \in D$.

Algorithm 2 SHCJ(A, D)

```
Input:
    $A$ is the ancestor node set of a single height and $D$ is the descendant node set.
Description:
  1: Let $h$ be the height of nodes in $A$.
  2: Evaluate $A \bowtie_{A.Code=\mathcal{F}(D.Code,h)} D$.
```

The SHCJ algorithm is both I/O and CPU efficient. First, the $\mathcal{F}$ function can be applied on the $Code$ attribute of D on the fly, therefore, no additional disk I/O is required. Hash-based join algorithms can be effectively used, with I/O cost of $3(||A|| + ||D||)$ when enough memory is available. Second, let $\gg$ ($\ll$) be right (left) shift operators. Because $\mathcal{F}(n, h) = 2^{h+1}\lfloor n/2^{h+1}\rfloor + 2^h = ((n \gg (h+1)) \ll (h+1)) + (1 \ll h)$, computation of $\mathcal{F}(D.Code, h)$ only incurs little additional CPU cost because its implementation only involves integer addition and shifting operations.

Multiple height containment join (MHCJ) We now present our algorithm for the general case, where nodes in A are distributed at multiple heights. The MHCJ algorithm (shown in Algorithm 3) *horizontally* partitions the ancestor set A into several partitions (based on height): $A_1, A_2, \cdots, A_k$. Each partition A_i can be processed using SHCJ with D. This is based on the observation that $A \lhd D = \bigcup_{1\le i\le k}(A_i \lhd D)$. Because $(A_i \lhd D) \cap (A_j \lhd D) = \emptyset$ if $A_i \neq A_j$, the union operation can be simply evaluated as appending the result of $A_i \lhd D$ into the final result set.

Algorithm 3 MHCJ(A, D)

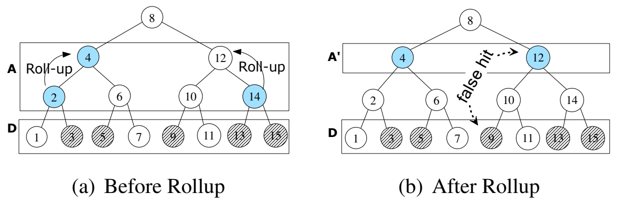
```
Input:
    $A$ is the ancestor node set of multiple heights and $D$ is the descendant node set.
Description:
  1: if $A$ is of a single height then
  2:     return SHCJ($A$,$D$) {Route to the single height containment join algorithm}
  3: end if
  4: Partition $A$ into $A_i$ ($1 \le i \le k$) according to node's height. Let $h_i$ be the height of $A_i$.
  5: for all Partition $A_i$ do
  6:     output SHCJ($A_i$, $D$);
  7: end for
```

The I/O cost of the MHCJ algorithm is simply the sum of the partitioning cost and sum of the costs of SHCJ algorithm for all partitions. Thus the total cost can be estimated as $5||A|| + 3k * ||D||$. Therefore the cost of MHCJ tends to be high if there are many (horizontal) partitions and when D cannot be accommodated in memory. In the following, we will present a technique to reduce the cost of MHCJ algorithm.

Multiple height containment join with rollup (MHCJ+Rollup) In order to reduce the cost of MHCJ algorithm, we propose a roll-up technique by adaptively reducing the number of horizontal partitions of the ancestor set. Consider two partitions of A, namely, A_i at height h_i and A_j at height h_j. The A_i partition can be rolled-up into partition A_j if $h_j > h_i$. The new A_j, denoted A'_j, consists of nodes originally in A_j and ancestors of nodes in A_i. In other words, here, we reduce the number of partitions by one. It can be done efficiently with the help of $\mathcal{F}(n, h)$ function — a node in A_i can quickly determine its ancestor at height h_j. The new merged A'_j set is passed to the algorithm MHCJ followed by a post-processing to remove *false hits* that are introduced by roll-up. The MHCJ+Rollup algorithm is shown in Algorithm 4 and an example is shown in Figure 4. In this example, a false hit (12, 9) is introduced by rollup and can be filtered by $\mathcal{F}(n, h)$ function.

Algorithm 4 MHCJ+Rollup(A, D)

```
Input:
   $A$ is the ancestor node set and $D$ is the descendant node set.
Description:
1: Choose $h$ such that it is within the height range of nodes in $A$.
2: Let $A'$ be the result of rolling up all the nodes in $A$ below $h$ to
   their ancestor node at level $h$.
3: Call MHCJ($A'$, $D$) and check if every result tuple is really
   a result tuple for the original containment join in a pipeline
   fashion.
```



(a) Before Rollup (b) After Rollup

Figure 4. A rollup technique example

The I/O cost for MHCJ+Rollup algorithm will be the MHCJ cost of a height-reduced ancestor set and the original descendant set. Specifically, if we roll-up all the ancestor to the highest level, we can call the I/O efficient SHCJ algorithm with the cost of $3(||A|| + ||D||)$. In practice, we found that even this simple strategy works reasonably well for many datasets.

3.3. Vertical-partitioning Algorithms

The vertical partitioning algorithms employ the divide-and-conquer strategy by partitioning the tree vertically into subtrees (with node replication) and processing the containment joins within each subtree.

Vertical-partitioning join (VPJ) The vertical partitioning join algorithm takes the input data size and memory size into consideration and actively optimizes for disk I/O by a heuristic partitioning strategy. We identified the following two cases as optimal in terms of I/O cost for containment join between A and D: a) when A can be accommodated in memory, and b) when D can be accommodated in memory. The I/O cost of both cases is $||A|| + ||D||$ and efficient algorithms exist so that they are also efficient in terms of CPU cost. The VPJ algorithm always tries to reduce the original problem into smaller sub-problems belonging to one of the above cases. A heuristic is used which chooses the number of partitions as $\frac{min(||A||,||D||)}{b}$, so that it is highly likely that all the partitions of the smaller set can be accommodated in memory.

The partitioning starts at level l, such that 2^l is greater than or equal to the number of desired partitions. Each node n_i at level l defines a partition p_i. A node e belongs to a partition p_i if and only if e is an ancestor of n_i or e is a descendant of n_i. Note that e will be *replicated* to multiple partitions, if e is an ancestor of n_i. This replication is necessary to ensure the correctness of the algorithm. The number of replicated nodes to each partition is at most l and thus the replication is unlikely to affect the execution of the algorithm.

It can be easily proved that $A \lhd D = \bigcup_i (A_i \lhd D_i)$ by the above partitioning method (with node replication), and the *union* operation can be reduced to *union all* operation as there will be no overlapping results among sub containment joins.

Algorithm 5 V-Partition-Join(b, A, D)

```
Input:
   $b$ is the number of buffer pages. $A$ is the ancestor node set. $d$
   is the descendant node set.
Description:
 1: $k_0 = \left\lceil \frac{min(||A||,||D||)}{b} \right\rceil$;
 2: Let $l = \lceil \log_2 k_0 \rceil$. Let $k = 2^l$.
 3: Partition $A$ and $D$ into $k$ partitions based on the nodes at level
    $l$. Refine the partitioning via merging and purging.
 4: for all partition $A_i$ and its corresponding partition $D_i$ do
 5:    if $||A_i|| > b \wedge ||D_i|| > b$ then
 6:       V-Partition-Join($b$, $A_i$, $D_i$) {Recursive partition.}
 7:    else
 8:       Memory-Containment-Join($b$, $A_i$, $D_i$)
 9:    end if
10: end for
```

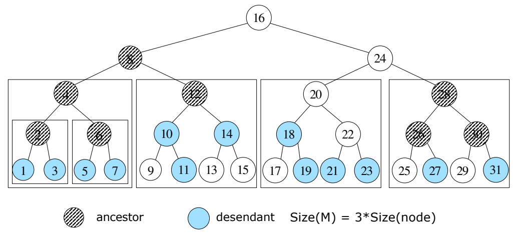
Algorithm 6 Memory-Containment-Join(b, A, D)

```
Input:
   $b$ is the number of buffer pages. $A$ is the ancestor node set. $D$
   is the descendant node set. One of the input relation is smaller
   than the memory.
Description:
1: if $||D|| < b$ then
2:    Sort $D$ and probe $D$ using binary search for each $a \in A$
      scanned.
3: else
4:    Use MHCJ+Rollup.
5: end if
```

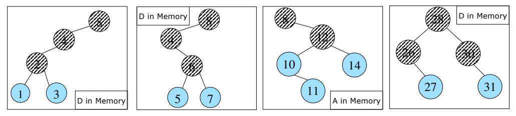
The algorithm is shown in Algorithm 5. After deciding the number of partitions, A and D are partitioned into $\{A_i\}$, $\{D_i\}$, where $1 \leq i \leq k$. For each pair of A_i and D_i, if either A_i or D_i can be accommodated in memory, an I/O optimal algorithm Memory-Containment-Join is called; otherwise, we recursively partition them by calling V-Partition-Join() on A_i and D_i.

The VPJ algorithm can adapt to data skew via the following two methods: merging/purging partitions and recursive partitioning. Merging of partitions can be done if both partitions are small and can reduce the number of total partitions.

Purging of a partition is based on the observation that if either A_i or D_i is empty, both partitions can be discarded. Recursive partitioning by invoking V-Partition-Join will again choose a good partitioning method for those "dense" partitions.



(a) Recursive partitioning



(b) Contents of each partition

Figure 5. An example of the VPJ algorithm

Figure 5 illustrates an example for the algorithm. In this example, the memory can hold three nodes. There are 8 ancestor nodes (shaded with lines) and 13 descendant nodes (grayed). The partitioning is done on level 2, resulting in four partitions. Note that the third partition is purged because there is no ancestor node in it and the first partition has to be further partitioned due to the effect of node replication (node 8). The final 4 partitions are shown in Figure 5(b), and each of them can be processed without further I/O.

The Memory-Containment-Join algorithm is shown in Algorithm 6. Depending on whether D or A can be accommodated in memory, an in-memory version of Index Nested Loop Join or a Multiple Height Containment Join with Rollup (MHCJ+Rollup) will be invoked.

The upper bound of the number of nodes replicated into each partition can be shown to be l and estimation of the number of replicated nodes can be obtained under certain assumptions. The formula to choose partitioning level l is also obtained by taking node replication into account. However, both analytical and experimental results suggest that the effect of node replication is usually negligible.

In the absence of recursive partitioning, the I/O cost with the VPJ algorithm can be estimated as $3(||A|| + ||D||)$.

3.4. Discussion of Containment Join Algorithms

3.4.1 Comparison of Algorithms

All the existing containment join algorithms require either indexed or sorted data. There is no previous known result of the best algorithm for the case when neither condition is satisfied. The naïve solution would be to build the index or sort the data on the fly before applying appropriate algorithms. Notice that sorting two datasets will cause at least $||A|| \cdot 2\log_b ||A|| + ||D|| \cdot 2\log_b ||D||$ I/O even before the join is performed (which we estimate as $||A|| + ||D||$). In many cases, our MHCJ+Rollup or VPJ algorithm can perform the join at the cost of $3(||A|| + ||D||)$. Analytically, if $b < \min(||A||, ||D||)$, that is, neither relation can be held in memory, our new algorithms have lower cost. Similar conclusion can be obtained for naïve algorithms that build indexes on the fly, as even the bulk loading procedure needs sorting the data. The efficiency of our new algorithms are further verified by our experimental results.

3.5. PBiTree Based Containment Query Processing Framework

Our containment query processing framework processes data encoded in PBiTree code by choosing an appropriate algorithm. The choice of algorithms is shown in Table 1. For example, if the data is sorted, we will use PBiTree based stack tree algorithms; if the data is neither sorted or indexed, we will use MHCJ+Rollup or VPJ algorithm.

It is obvious that when the newly proposed algorithms are taken into consideration by a query system, the space of possible plans will be greatly expanded and it is likely that we can find a better plan than that we can find in a system without those new algorithms. Therefore we believe our new algorithms and the framework are beneficial and stimulating for a highly efficient query processing and optimization system.

4. Performance and analysis

In this section, we present some results of our comprehensive experiments conducted to study the effectiveness of PBiTree encoding scheme for processing containment joins.

We implemented the following seven algorithms on Minibase[7], namely, the single height containment join (SHCJ), the multiple height containment join (MHCJ), MHCJ with rollup (MHCJ+Rollup), vertical-partitioning join (VPJ), the improved index nested loop join algorithm (INLJN), the stack-tree join algorithm (STACKTREE) and the Anc_Des_B+ join algorithm (ADB+). The last three algorithms were proposed in [20, 1, 4], respectively. Results with the MHCJ algorithm are not reported here, since MHCJ+Rollup algorithm outperforms MHCJ in all experiments. All algorithms take as input an ancestor set A and a descendant set D encoded in PBiTree code. We focus on the case when neither A nor D is sorted or indexed and compare our partitioning based algorithms with the naïve algorithms that sort data or build index on the fly. Therefore, the sorting time or index building time is included for INLJN, STACKTREE and ADB+ algorithms. The comparison of region based algorithm and their adapted version working for PBiTree encoded data was also performed. However,

[7]http://www.cs.wisc.edu/coral/minibase/minibase.html

as can be expected from the analysis, the two classes of algorithms have almost the same performance and thus their results are not reported here.

Minibase is a C++ implementation of a DBMS developed at University of Wisconsin-Madison. We built our system using its storage manager, buffer manager and B$^+$-tree module. The storage manager was modified to operate on the raw disk directly, and thus there is no buffering effect from the operating system. All the experiments were performed on a Pentium III 450MHz PC with 256M RAM, 30G hard disk, running Windows 2000. In order to examine our algorithms for large scale applications, we used a relatively small buffer pool (500 pages for all experiments, except for the experiments on varying buffer sizes).

4.1. Experiments using synthetic datasets

In this subsection, we report our experiment results conducted on synthetic datasets. First, we will describe how to generate synthetic datasets. Then, we will discuss several experimental results.

4.1.1 Synthetic data generation

Synthetic datasets were generated to investigate three factors that have major impacts on containment join processing: the dataset size, the node distribution in a dataset (either an ancestor set or a descendant set), and the selectivity between an ancestor set and a descendant set. Here, the selectivity is the average number of descendants matched per ancestor node. We use a four-character shorthand to identify a dataset: a dataset could be single or multiple-height (S, M), size of A or D could be large or small (L, S) and the selectivity of the dataset could be high or low (H, L).

We generated 16 datasets that cover all possible combinations. The statistics of the datasets are shown in Table 2(a) and Table 2(b). Each large (ancestor or descendant) set (L) contains one million elements, and each small (ancestor or descendant) set contains 10 thousand elements. The H_A and H_D columns list the number of heights in A and D respectively.

4.1.2 Overall performance

The first experiment was conducted to study the performance of the algorithms when elements in the ancestor set and descendant set are located in the same levels, respectively. The statistics of the eight datasets are listed in Table 2(a).

The total elapsed times are listed in Table 2(e). The column "MIN_RGN" shows the minimum elapsed time among INLJN, STACKTREE and ADB+ algorithms. In other words, MIN_RGN reports the best performance of region-based algorithms. Figure 6(a) shows the improvement ratio of the SHCJ, VPJ algorithms over MIN_RGN. It is defined as $(T_{MIN\_RGN} - T_{SHCJ})/T_{MIN\_RGN}$, where T_{MIN_RGN} and T_{SHCJ} are the elapsed times for MIN_RGN and SHCJ algorithms respectively.

Some observations can be made from Figure 6(a):

- SHCJ and VPJ algorithms perform similarly.
- SHCJ and VPJ outperform MIN_RGN over 20% in general. Particularly, when one set is large and the other set is small such as SLSH, SSLH, SLSL and SSLL, SHCJ and VPJ are significantly better than MIN_RGN, by an improvement ratio over 95% (in other words, up to 30 times faster).

We also conducted the same experiment with multiple-height datasets as shown in Table 2(b). In this experiment, MHCJ+Rollup algorithm is used instead of SHCJ. In the interest of space, we only show the improvement ratio in Figure 6(b). The results suggest that the performance of MHCJ+Rollup and VPJ algorithms for multiple-height datasets is still much better than MIN_RGN, although for MHCJ+Rollup algorithm, there are false matches introduced by the rollup technique. For example, the improvement ratio is up to 96% and speedup ratio up to 30. Table 2(f) lists the number of false hits for the MHCJ+Rollup algorithm. This shows that for large datasets, all algorithms are disk I/O bound and the additional CPU cost for false hits is negligible.

4.1.3 Impact of varying buffer sizes

Experiments were also carried out to investigate how the execution times are affected by buffer sizes for different algorithms. The parameter of relative buffer size (denoted by P) is introduced. It is defined as the number of buffer pages divided by the size of the smaller set in a dataset, or, formally, $P = \textit{NumBufferPages}/\min(|A|, |D|) * 100\%$.

Two large datasets were chosen, specifically, SLLL and MLLL. Figure 6(e) and 6(f) show the elapsed times for SLLL and MLLL with varying buffer sizes.

When the buffer sizes are very small, such as 0.5% of the smaller data set, the performance of all algorithms will degrade greatly as shown in Figure 6. When the buffer sizes are around 1% of the size of the smaller set (still quite small), the MHCJ+Rollup and VPJ algorithms start to perform reasonably well. Beyond the point where $P = 2\%$, MIN_RGN remains almost constant regardless of the number of buffer pages. In contrast, MHCJ+Rollup and VPJ algorithms can gracefully utilize additional memory to speedup join processing.

4.1.4 Scalability

We carried out scalability tests of the proposed algorithms with two classes of datasets, i.e. single-height and multiple-height datasets. Each group contains datasets with size $k \cdot B$ nodes, where $1 \leq k \leq 8$ and $B = 5 \times 10^4$. Figure 6(g) and 6(h) show the total elapsed times of different algorithms on the two classes of datasets respectively.

Table 2. Statistics of datasets and some experiment results

(a) Statistics of synthetic datasets: single-height

Dataset	#results
SLLH	906192
SLSH	8842
SSLH	18596
SSSH	9088
SLLL	94426
SLSL	363
SSLL	385
SSSL	801

(b) Statistics of synthetic datasets: multiple-height

Dataset	H_A	H_D	#results
MLLH	2	6	941056
MLSH	9	9	18758
MSLH	2	7	12263
MSSH	7	9	8692
MLLL	3	7	45315
MLSL	7	5	338
MSLL	7	4	326
MSSL	3	2	784

(c) Statistics of BENCHMARK datasets

	$\|A\|$	H_A	$\|D\|$	H_D	#results
B1	25500	1	1	1	1
B2	10830	4	59486	4	10830
B3	1	1	21750	5	21750
B4	25500	1	12823	7	12823
B5	2200	1	2200	2	2200
B6	9750	1	35	2	35
B7	9750	1	9750	1	9750
B8	21750	5	21750	6	21750
B9	21750	5	21750	6	21750
B10	12823	3	120391	8	120391

(d) Statistics of DBLP datasets

	$\|A\|$	H_A	$\|D\|$	H_D	#results
D1	116176	1	9951	5	9951
D2	116176	1	208	4	208
D3	116176	1	100	4	100
D4	116176	1	116176	6	116176
D5	200271	1	49141	8	49029
D6	200271	1	434	6	416
D7	84095	1	13660	6	13660
D8	84095	1	3	2	3
D9	84095	1	82980	7	82980
D10	120176	5	69177	6	55517

(e) Elapsed time (second) for single-height synthetic datasets

	MIN_RGN	SHCJ	VPJ
SLLH	402.7	286.5	268.3
SLSH	142.8	7.07	5.62
SSLH	223.1	7.27	7.35
SSSH	0.88	0.16	0.16
SLLL	404.1	285.2	264.4
SLSL	143.2	7.16	5.70
SSLL	223.8	7.27	7.33
SSSL	0.86	0.15	0.17

(f) False hits for MHCJ+Rollup

	#false hits
MLLH	318542
MLSH	67428
MSLH	6202
MSSH	5475
MLLL	137615
MLSL	3904
MSLL	3810
MSSL	1

As can be observed from Figure 6(g) and 6(h), all the proposed algorithms scale linearly with the data sizes and our newly proposed algorithms perform consistently better.

4.2. Experiments with real-world datasets

In this subsection, we study the performance of the proposed algorithms for real world XML data, i.e. BENCHMARK and DBLP.

DBLP is a set of bibliography files.[8] The size of the raw text files is around 50MB. BENCHMARK data is from the XML benchmark project [18]. We generated the benchmark data with SF(scale factor) = 1. The raw text file is 113MB.

We selected 10 containment joins for the DBLP data, namely D1, D2, ..., D10. These containment joins are parts of real-world XML queries that can be found in [19]. Similarly, 10 containment joins, namely B1, B2, ..., B10, are selected from benchmark queries presented in [18]. We follow the query decomposition framework reported in [12] and treat $\mathcal{EE}$-joins as containment joins. The statistics of the real-world XML datasets are shown in Table 2(c) and 2(d).

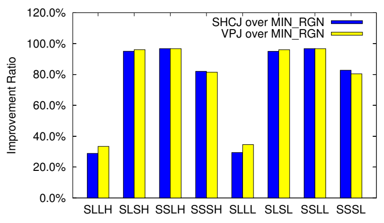
The results are shown in Figure 6(c) and 6(d). The experiments on real-world datasets show that the MHCJ+Rollup and VPJ algorithms are consistently better than MIN_RGN. The improvement ratio is up to 96% and the speedup up to 25. This conclusion is nearly identical to what we have observed from the experiments on synthetic datasets.

To conclude, our proposed algorithms (based on PBiTree coding) perform consistently better than the best of existing algorithms (based on region-based coding) on both synthetic and real-world datasets when neither datasets is sorted or indexed. Our algorithms also have salient features, such as small memory requirement, good scalability with buffer size and data size, etc.

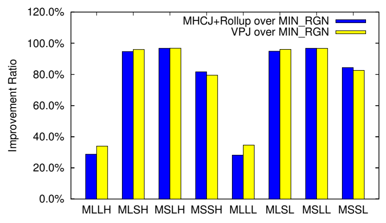
[8]It is available at `ftp://ftp.informatik.uni-trier.de/pub/users/Ley/bib/records.tar.gz`

5. Related Work

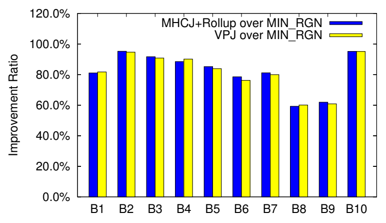
Numbering schemes Numbering schemes are the enabling technique for "set-at-a-time" processing of containment joins. A basic requirement of the number schemes is to be able to verify the ancestor-descendant relationship locally and efficiently. The most popular numbering scheme is the class named region code. In essence, each XML element is tagged with a range and ancestor-descendant relationship is equivalent to the inclusion relationship of regions. [20] proposed $(Start, End)$, which is the absolute offset of the start and the end of the element in the document. [12] proposed a variant in the form of $\langle order, size\rangle$. It is also referred to as a *durable* numbering scheme because it reserves additional code space for elements, which helps to reduce the high update cost due to overflow compared to [20]'s scheme. A relative region coding scheme was proposed in [9]. Each element is coded in using the



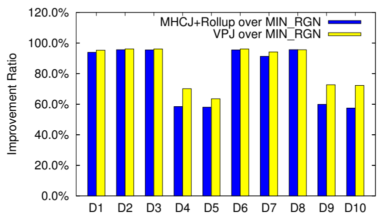
(a) Ratio of improvements for single-height synthetic datasets



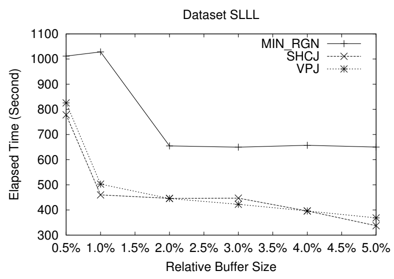
(b) Ratio of improvements for multiple-height synthetic datasets



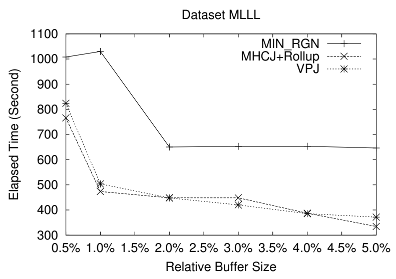
(c) Ratio of improvements for BENCHMARK datasets



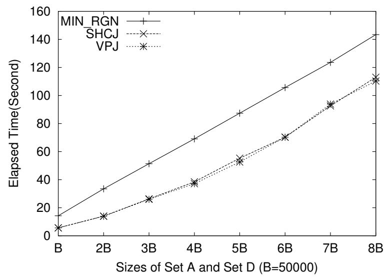
(d) Ratio of improvements for DBLP datasets



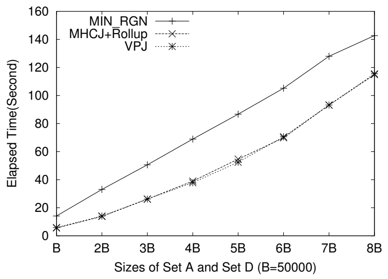
(e) Elapsed times with different buffer sizes: SLLL



(f) Elapsed times with different buffer sizes: MLLL



(g) Scalability: single-height datasets



(h) Scalability: multiple-height datasets

Figure 6. Experiment results

relative offset to its parent element. The main drawback of this approach is that the query cost is increased because all the ancestors along the path to the node in question have to be read to determine the absolute code of the node. Another coding scheme is the prefix code (conceptually in a string format), in which each element is coded by its path from the root [10]. The ancestor-descendant relationship is equivalent to the substring relationship of the prefix codes.

Containment join algorithms Most proposed containment join algorithms are based on the region codes. Therefore, their join criteria are the containment relationships of the region codes, which usually result in inequality joins. In addition, all the algorithms proposed so far have some requirement (either appropriate indexes on the element set or the element sets are sorted). [20] studied index nested loop and MPMGJN, a variant of sort-merge join. Both analytical and experimental results showed that MPMGJN usually outperforms index nested loop join unless one of the element set is of extremely low cardinality. [5] proposed to use R-Trees by viewing $(Start, End)$ as the coordinate of a point in two dimensional space and use spatial join algorithms to process the query. [1] studied the class of tree merge algorithms (MPMGJN) and the class of newly proposed stack tree algorithms. Their stack tree algorithms use an in-memory stack to avoid rescanning the inner element set and are also capable of outputting the result either by ancestor sorted order or by descendant sorted order. Their algorithms have been shown to have optimal worst case CPU and I/O complexities and outperform MPMGJN algorithms. [4] proposed to utilize available indexes to skip part of the elements that will not participate in the stack tree join algorithms.

Spatial join algorithms It is natural to interpret the containment join from spatial point of view. In [5], each region code is modelled as a point in two dimensional space and containment join is modelled as spatial join with *contain* predicate. Specifically, region r_1 contains region r_2 if and only if r_1 is in the II quadrant with r_2 as the origin (and this join can be easily converted to spatial join with *intersect* predicate)[5]. There have been many studies in the area of spatial join. Existing algorithms can be classified into three categories.

1. *Both relations have indexes.* [3, 6, 15] used the *synchronous traversal* technique to process (multiway) spatial join efficiently. The Anc_Des_B+ algorithm is analogous to those algorithms.
2. *Only one relation has index.* [13] proposed seeded tree join which actually converts the problem into the first case by creating an R-tree like index on the fly on the dataset that does not have an index.
3. *None of the relations has index.* [14] used a hash join based on sampling technique. [17] used a partition based method (with replication) to process spatial joins. Our VPJ algorithm differs from their methods in that we do not need to merge the final result as there is no *multiple counting* problem. [11] proposed another efficient join algorithm named *size separation spatial join* (S^3J), which introduced the concept of layer to avoid replication of objects during the partitioning phase. [2] proposed an external memory based spatial join algorithm. It is based on the technique of *distributed line sweeping* and achieves asymptotically optimal I/O efficiency.

6. Conclusions

There has been increasing interest in containment query processing for tree structured data such as XML and text documents recently. In this paper, we focused on efficient processing of a primitive operation, called containment join, as the building block for complex queries. To facilitate the efficient processing of such joins, we propose a PBiTree coding scheme, which has been demonstrated to be more versatile than the commonly used region coding scheme. Based on the new coding scheme, a unified containment join processing framework is proposed. Within this new framework, previous state-of-the-art algorithms are seamlessly integrated with necessary modification and new algorithms that address containment join without index or sorting conditions are proposed. The new algorithms are based on the notion of horizontal and vertical partitioning. A unique feature is that the newly proposed algorithms use equijoins instead of θ-joins and are highly optimized for disk I/O. Comprehensive experiments were conducted to evaluate the effectiveness of partitioning-based algorithms for containment join processing. Our experimental studies show that the proposed algorithms outperform the naïve algorithms based on the region coding scheme.

The PBiTree coding introduces several interesting issues. First, the regular structure of the PBiTree brings about new possibilities to maintain the statistics of the corresponding data tree, which can be in turn exploited in query processing. Second, we are working on a cost-based query optimizer that is aware of all the above-mentioned algorithms. An issue is to analyze the cost of all algorithms using a more precise disk access model.

References

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

[2] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. Scalable sweeping-based spatial join. In *VLDB*, 1998.

[3] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient processing of spatial joins using R-trees. In *SIGMOD*, 1993.

[4] S.-Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo. Efficient structural joins on indexed XML documents. In *VLDB*, 2002.

[5] T. Grust. Accelerating XPath location steps. In *SIGMOD*, 2002.

[6] Y.-W. Huang, N. Jing, and E. A. Rundensteiner. Spatial joins using R-trees: Breadth first travesral with global optimizations. In *VLDB*, 1997.

[7] C. Icking, R. Klein, and T. Ottmann. Priority search trees in secondary memory (extended abstract). volume 314 of *Lecture Notes in Computer Science*, pages 84–93. Springer, 1987.

[8] H. Jiang, H. Lu, W. Wang, and B. C. Ooi. XR-Tree: Indexing XML data for efficient structural join. In *ICDE*, 2003.

[9] D. D. Kha, M. Yoshikawa, and S. Uemura. An XML indexing structure with relative region coordinate. In *ICDE*, 2001.

[10] W. E. Kimber. HyTime and SGML: Understanding the HyTime HYQ query language 1.1. Technical report, IBM Corporation, 1993.

[11] N. Koudas and K. C. Sevcik. Size separation spatial join. In *SIGMOD*, 1997.

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

[13] M.-L. Lo and C. V. Ravishankar. Spatial joins using seeded trees. In *SIGMOD*, 1994.

[14] M.-L. Lo and C. V. Ravishankar. Spatial hash-joins. In *Proceedings ACM SIGMOD International Conference on Management of Data*, 1996.

[15] N. Mamoulis and D. Papadias. Integration of spatial join algorithms for processing multiple inputs. In *SIGMOD*, 1999.

[16] J. McHugh and J. Widom. Query optimization for XML. In *VLDB*, 1999.

[17] J. M. Patel and D. J. DeWitt. Partition based spatial-merge join. In *SIGMOD*, 1996.

[18] A. Schmidt, F. Waas, M. Kersten, D. Florescu, I. Manolescu, M. J. Carey, and R. Busse. The XML benchmark project. Technical report, CWI, Amsterdam, The Netherlands, 2001.

[19] F. Tian, D. J. DeWitt, J. Chen, and C. Zhang. The design and performance evaluation of altervative XML storage strategies. Technical report, Computer Science Department, University of Wisconsin-Madison, 2000.

[20] C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman. On supporting containment queries in relational database management systems. In *SIGMOD*, 2001.