Efficient Processing of XML Containment Queries using Partition-Based Schemes

Zografoula Vagena, Mirella M. Moro, Vassilis J. Tsotras
University of California
Riverside, CA 92521
{foula, mirella, tsotras}@cs.ucr.edu

Abstract

XML query languages provide facilities to query XML data both on their value as well as their structure. The structural part usually involves the selection of elements that have a specified tree structure. A basic operation in processing and optimizing such queries is the *containment join*, which takes two sets of elements and returns pairs of elements where one is the ancestor (or descendant) of the other. Most of the techniques proposed so far assume that the two sets are already sorted (e.g. structural joins) or utilize preexisting indexing schemes (e.g. navigation based techniques). In contrast, a partition-based technique does not require indexing or sorting. Instead, the XML data is augmented with additional information and the containment join is processed by dividing the input sets into smaller partitions. Recently, a partition-based technique has been proposed that uses *perfect binary tree* (PBiTree) encoding. Nevertheless, this technique presents several shortcomings in the way it handles the document structure, as well as the available main memory, that may greatly affect performance. In this paper we present a new partition-based scheme that gracefully adapts to different document sizes, and takes a more thorough consideration of the document structure to avoid the above shortcomings. We experimentally validate the superiority of our approach, by comparing it with the PBiTree encoding. Moreover, our experiments demonstrate that the new partition-based algorithm provides a viable alternative to non-partition join algorithms when the input data is not sorted or indexed.

1 Introduction

XML is gaining considerable attention as a standard for e-commerce applications. As more and more organizations and systems employ XML within their information infrastructure, the need to devise and employ specialized techniques for efficient storage, management and retrieval of XML data arise.

An XML database can be modeled as a forest of unranked (in the sense that the number of children nodes of a particular node can be unbounded), ordered, node-labeled trees, where each tree represents a single document. Each node in the tree corresponds to an element, attribute or value, and the edges represent immediate element-subelement or element-value relationships. Figure 1a presents the tree representation of a sample XML database containing bibliographic entries. Each node is also associated with a unique id which corresponds to the node's position in the inorder traversal of the document tree.

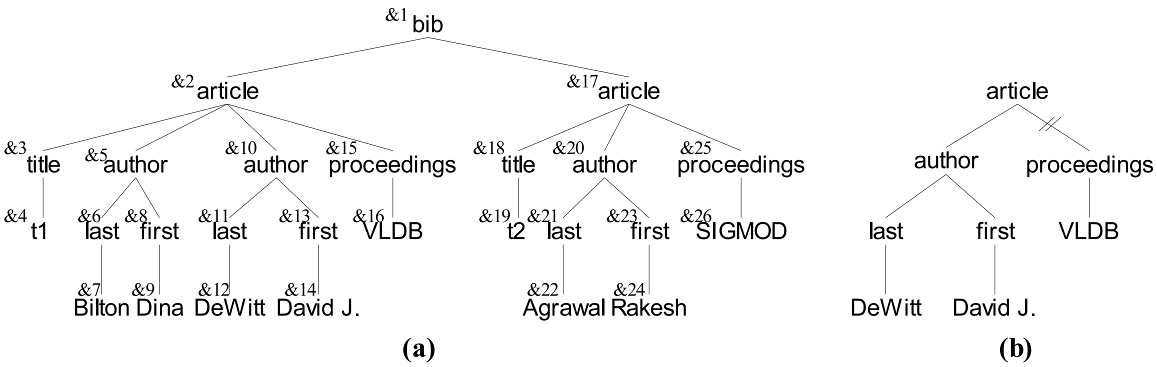

Figure 1: (a) Bibliography XML database representation, (b) Sample Query

The tree-centric model has been adopted by popular XML query languages like XQuery [4] and XPath [7, 3], which qualify elements for retrieval both by their structure and the values of their attributes. Those languages make use of twig patterns that are matched against the document model, in order the enable structure based-retrieval. Each query node may represent element tags, attribute-value comparisons and string literals. Query edges represent either ancestor-descendant or parent-child relationships between the nodes. As an example, consider the query:

//article[./author[@last="DeWitt" and @first="David J."]]//proceedings[./VLDB]

that requests all proceedings of articles that have an author with last name "DeWitt" and first name "David J." and have appeared in a VLDB conference. The graphical representation of the query is provided in Figure 1b. This query consists of two types of conditions:

- **@last="DeWitt"** and **first="David J."** which are are value-based since they select elements using their values
- **//article//proceedings**[**./VLDB**], **//article**[**./author**[**./last**]] and **//article**[**./author**[**./first**]] which are structural conditions as they impose restrictions on the structure of the retrieved elements (e.g. a `proceedings` element must exist under an `article` element, while the latter must have at least one `last` and at least one `first` element as children.

While value-based queries can be evaluated by adopting traditional indexing schemes, such as B+-trees, supporting twig queries is a harder problem. Various proposals have appeared recently on the efficient computation of twig expressions. Most techniques can be divided into two groups: (1) navigation-based algorithms [10, 9, 21, 14] which compute results by analyzing the input one node at a time, and, (2) set-based techniques [23, 16, 2, 6, 5, 12, 22, 13, 17] that take advantage of precomputed, mainly range-based, numbering schemes to identify the relationships on the input element sets. Typically, a range-based numbering scheme maps the ancestor-descendant condition, into a range containment one. Among the set-based techniques, some regard the query holistically ([5, 21, 18, 14]) while others decompose the twig expression into pairs of query nodes, processing those pairs individually and then merging the results. Interleaving of the two methods has also been proposed [11] and shown to be advantageous under certain circumstances.

All the above techniques, with the exception of [22, 17], assume that the input is accessed in a particular order (e.g. in document order, reverse document order etc.) which implies that the input is either sorted or indexed. This however may not be always available, as for example when data is produced at intermediate steps of a complex query execution. In contrast, [22, 17] propose partition-based algorithms for the containment join and thus do not impose any particular access pattern on the input data. [17] utilizes a range-based numbering scheme which however leads to large CPU overhead, during the main-memory processing of the partitions. The technique in [22] augments the input data with a numbering scheme based on PBiTree encoding. This scheme makes it possible to use equality conditions and as a result, devise hash-based schemes to answer containment joins. Unfortunately, the PBiTree numbering does not make full use of the structural characteristics of the document, and may produce false positives which need to be filtered out by additional data accesses, thus creating considerable overhead. Moreover, the join algorithms are not scalable with respect to large document sizes as they incur element replication and do not take into consideration restrictions imposed by the available main memory.

In this paper we propose an improved partition-based algorithm that uses an extension to the PBiTree numbering while addressing the previous deficiencies. We also provide efficient ways to incorporate this scheme into the input data. The main contributions of the paper are summarized below:

- An improved, partition-based algorithm is provided that adapts gracefully to large document sizes and mitigates the effects of partition overlaps.
- The utilization of a more appropriate numbering scheme for data representation is described. Furthermore, a new, scalable SAX-based mapping algorithm to incorporate this data representation is devised;
- An extensive experimental section is presented, comparing the improved partition-based technique with the original PBiTree approach, as well as other non partition-based approaches. A useful byproduct of this experimental analysis is that partition-based algorithms should be preferred for processing containment joins when sorting or indexing are not present.

The rest of the paper is organized as follows. Section 2 describes the PBiTree approach and introduces a better document mapping algorithm. Section 3 presents the new containment join algorithm, while Section 4 introduces the improved numbering scheme. Section 5 presents performance results and Section 6 concludes the paper.

2 Structural Joins using the PBiTree

The *PBiTree* is a perfect binary tree with each node tagged by two numbers: (i) its level within the tree, and, (ii) a number representing the position of the node within the sequence that corresponds to the *in-order* traversal of the tree. Figure 2 depicts an example of a PBiTree. Level 0 starts at the root, while the numbering of each node appears inside the node. Using this numbering scheme the following lemma holds [22]:

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

Nodes of the input XML document tree are then mapped to appropriate PBiTree nodes. The mapping algorithm [22] is DOM-based and appears in Figure 3. With this mapping, each document

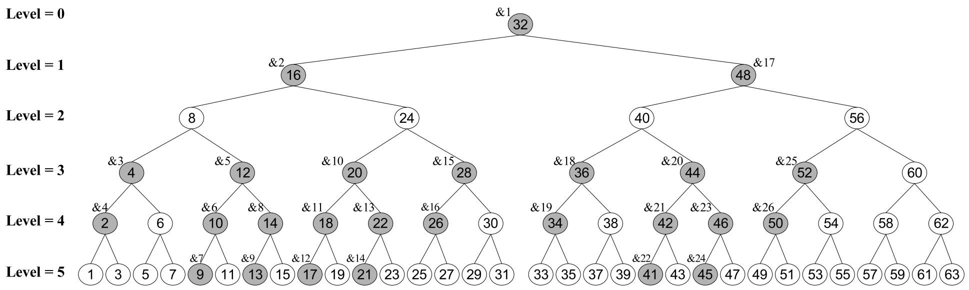

Figure 2: Embedding an XML tree within a PBiTree

element, is associated with the level and number of the PBiTree node that it is mapped to. Thus the PBiTree does not need to be materialized in practice, since only the node coding is needed. In Figure 2 the nodes in gray indicate the mapping of the document nodes from Figure 1a to those of the PBiTree. The nodes in white are not considered. For visualization purposes, the number at the top of each gray node indicates which document node was mapped there.

Input: T is an arbitrary data tree.
e is a node in T.
(l, α) is the top-down code of e.
Output: Every node n in T has the correct PBiTree code in $n.Code$.

Function BinarizeTree (T, e, α, l)
$e.Code = G(\alpha, l)$
if e is not a leaf node **then**
  $n = NumOfChild(e)$
  $k = \lceil \log_2 n \rceil$
  for all child node e_i //i-th child node left to right
  do
    BinarizeTree$(T, e_i, 2^k \cdot \alpha + i - 1, l + k)$
  end for
end if

Figure 3: DOM-Based Tree Binarization

Using the PBiTree encoding, checking whether a node is an ancestor of another is very efficient (it involves only shifting and integer operators). Two partition-based algorithms for answering containment queries were proposed. The *horizontal* partitioning algorithm partitions the elements according to the level they reside within the PBiTree. With the *vertical* algorithm elements are partitioned according to which path in the PBiTree they reside. The horizontal algorithm is efficient for specific (but limited) scenarios, as for example, when document elements do not have recursion and the number of elements in each node is equal. In general however, the horizontal algorithm has various shortcomings that make it inefficient for disk-based data (i.e., it provides false positives, it requires multiple scans of descendant sequences, etc.)

Input: b is the number of buffer pages.
A is the ancestor node set.
D is the descendant node set.

Function V-Partition-Join (b, A, D)
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
5: **do**
6:    **if** $||A_i|| > b \wedge ||D_i|| > b$ **then**
7:       V-Partition-Join(b, A_i, D_i) //Recursive partition
8:    **else**
9:       Memory-Containment-Join(b, A_i, D_i)
10:    **end if**
11: **end for**

Figure 4: Vertical Partitioning

Thus [22] suggested using the vertical partitioning algorithm shown in Figure 4. The algorithm creates corresponding partitions for the ancestor and descendant element sequences, in such a way that elements of one (ancestor) partition, say A_i, can be joined only with elements that belong to the corresponding (descendant) partition, say D_i. In the first phase, a number of partitions are created so as to maximize the possibility that their size can fit into main memory. (This is performed in step 1, by choosing the k_0 with respect to b, the available buffer space.) For each element sequence, the partitions are then formed by choosing an appropriate level (called the *partition level*) in the *PBiTree*. This is the smallest level in the binary tree which has enough nodes to cover the number of partitions that need to be created. Each of these nodes identifies a partition. Subsequently, the algorithm scans each element sequence individually and distributes its elements along the partitions. The node distribution proceeds as follows: if an element, say e, resides below the partition level, the code of e's ancestor element in the partition level (say element f) is identified and e is placed within the partition identified by f. Otherwise, i.e., if e is above the partition level, all the elements in the partition level that are descendants of e are identified and e is placed in their corresponding partitions. As a result an element may be replicated among those partitions.

After nodes are partitioned, the corresponding partition pairs are joined. Given a partition pair, if none of its partitions can fit in main memory (step 7), the partitioning is recursively applied. Otherwise, main-memory containment join is performed (step 9). In particular, if the descendant partition can fit in memory, it is loaded and sorted in main memory (according to the node code). Then for each ancestor element we perform a binary search to find the join pair. If the ancestor partition fits in main memory the join is performed by horizontal partitioning. Nevertheless, the vertical partitioning algorithm has two shortcomings: (i) it does not make full use of the available memory, and, (ii) it incurs the cost of replication. As explained in detail in Section 3 and shown in the experimental section, both facts can greatly affect performance.

2.1 A more efficient mapping for the XML document

There are three main assumptions that the original mapping algorithm [22] makes:

- The height of the *PBiTree* is known in advance. This is not possible in the general case as different XML documents will result in different *PBiTrees*.
- The children of a node are placed in the binary tree to the first level under this node that can hold all of them. For example if a document tree node is at level i and has three children then all of them will be placed in level $i+2$ in the PBiTree.
- The number of children of a node is known before visiting any of them. A detailed discussion on this issue follows.

To be able to satisfy the first and second assumptions one has to access the XML document through a DOM interface. As of today, the XML processors that provide this interface achieve it by first loading the XML document into main memory and adding additional structure to enable navigation within the document, making its size even larger (up to 5 times the size of the document). As a result, the size of the document to be encoded is limited by the size of main memory.

In order to be able to embed the desired encoding, an algorithm utilizing the SAX interface and perhaps some limited additional main memory is more appropriate and in some cases the only viable solution. In Figure 5 we devise a new algorithm that does exactly that.

The algorithm operates by scanning the input documents three times in document order. The sequential scan is performed by the XML parser that implements that SAX interface. Our technique utilizes the callback functions of the parser, specifically (a) *startElement()* which is triggered when the opening tag of an element is visited, and, (b) *endElement()* which is triggered when the closing tag of an element is encountered. During the first scan, the number of the children for each node is

```
Input: T is an arbitrary data tree.
Output: Every node n in T has the correct PBiTree
        code in n.Code.

Function SAXBinarizeTree ( T )
        Parse Document to identify number of
            children of each node
        Parse Document to identify level height h of tree
        Parse Document to identify level of each node
            and its code
```

Figure 5: SAX-Based Tree Encoding

computed, while on the second scan the height of the PBiTree is found. Then a third scan computes the actual codes.

The number of children for each node is found by utilizing a stack that keeps, for each node that belongs to the current path, the number of its children that have already been visited. When considering document order, the children of a node reside within the opening and closing tags of the node. As a result when the closing tag of a node is encountered, it is guaranteed that all its children have been found and the node can be discarded and popped from the stack. Taking into account the fact that the SAX parser accesses the nodes in document order, the length of the stack is bounded by the length of the maximum path from the root to the leaves in the XML document.

The height H of the tree is the largest level plus 1. The second scan determines the level of each input node in the PBiTree by using the level of the corresponding parent node, as well as the number of the node's siblings. The level of the parent node is again kept within a stack whose size is similarly bounded by the length of the maximum path in the XML document.

We note that the improved mapping algorithm, can also be used with minor changes with the new numbering scheme we introduce below.

3 The new partition-based Containment Join

We start by elaborating on the shortcomings of the vertical partitioning algorithm (section 2) and then provide a viable alternative to eliminate them.

The first shortcoming stems from the way the number of partitions is computed. In particular, [22] attempts to make the size of each partition fit into main memory by dividing the size of the element sequence with the size of the main memory. This technique does not take into account the paginated nature of I/O neither does it consider the number of available buffers. If the number of partitions becomes larger than the number of main memory buffers and given that the input is not sorted, the overhead of this partition phase can become as large as 2 I/Os per individual element accessed (one read and one write).

To illustrate the point consider the simple case, shown in Figure 6, where the main memory contains three buffer pages, i.e. one input buffer ($B0$) and two output buffers, each of which can hold up to two records. Assume that the *PBiTree* under consideration is the one shown in Figure 2, and let the *partition level* be at the second level of the binary tree (i.e. we will create four partitions, identified by the nodes with codes: 8, 24, 40, 56). Suppose that we want to partition the document elements that appear at level 3 of the PBiTree (i.e, nodes with codes 4, 12, 20, ..., 52). However, assume that these elements are stored without any order. The right part of Figure 6 depicts the disk pages storing the sequence elements; for example, page P0 holds two records, for elements with codes 20 and 36, etc. The partition will proceed as follows: element with code 20 will be placed in buffer $B1$, then element 36 in buffer $B2$, as it belongs to a different partition. Element with code 52 belongs to yet another partition and thus it will cause buffer $B1$ to be written to disk. Element 4 also belongs to a new partition and the LRU policy will cause buffer $B2$ to be written to disk. Subsequently, element with code 28 will flash buffer $B1$ which will then be loaded with the very first partition, that contains element 20. Similarly, element 44 will flash buffer $B2$ and will load the second partition that contains element 36. Clearly, the above partition example causes two I/Os per element placement which becomes prohibitive for large element sequences.

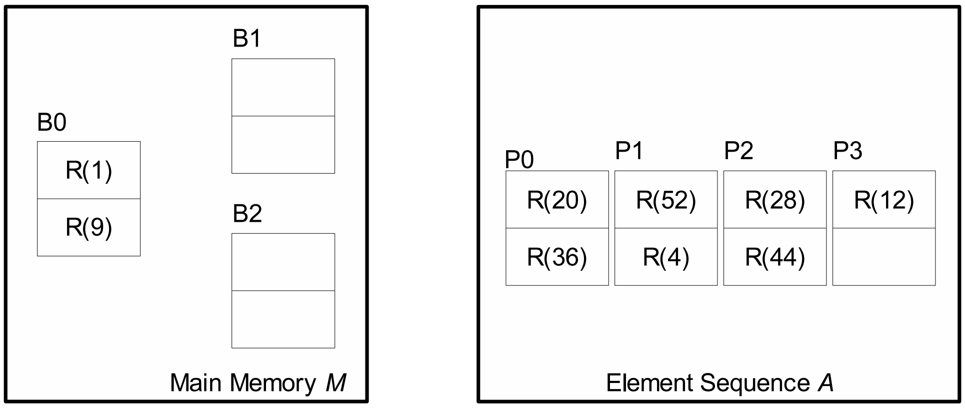

Figure 6: Example of poor partitioning

Taking into account that the size of the XML database can be arbitrary, while the available main memory is fixed and bounded, it is not difficult to have such a scenario materialize.

A solution would be to always bound the maximum number of partitions with the number of available buffers. Then, during the partitioning phase a buffer is flushed out only when it is full. This tactic might lead to partitions with larger sizes, and as a result to a larger number of repartitioning. Nevertheless, the number of repartitioning is logarithmic with base equal to the number of available buffers, so the overhead cannot overpass the gains. Our experiments agree with the previous observation.

The second shortcoming arises from the way elements in [22] are replicated. The size of the replication is linear to the size of the elements that belong to more than one partitions, creating a space overhead. This overhead then results to time overhead as the partitions become larger, and increases with the size of elements that span several partitions. Moreover, elements that belong to the descendant sequence and reside above the partition level are handled the same way as the elements below that level, and are thus considered candidates for join with all the other ancestor elements from corresponding partitions. However, there is room for optimization since the descendant elements that reside above the partition level can only be joined only with ancestors residing above this level. As a result they do not need to be replicated to any of the descendant partitions.

In [20, 17] the authors utilized a technique that eliminates data replication by using a small size of additional memory. Mapping it into our environment, each element that spans multiple partitions is only saved within the partition that is identified with the smallest code. In the partition join phase the partitions are used in ascending order of the identifying codes. When a partition is evaluated the ancestor elements that need to be used in the next partition, are cached and maintained for the subsequent partitions. For each partition, the number of elements to be cached from each partition is bounded by the path from the document root to the node identifying the current partition. We use the same technique to handle ancestors that belong to more than one partitions.

For each descendant element d, if there exists and ancestor of it, then it will belong to at least the same partitions as d (and possibly more). That enables us to save the descendant element within the partitions whose identifying element has the smallest code. The ancestor element will either be saved in the same partition, or will be cached from the previous partitions.

The new algorithm is presented in Figure 7. In the partition phase, the algorithm uses all the available memory to compute the number of partitions having as upper bound the number of available main memory buffers. It then partitions each element, handling the ones that span multiple partitions as explained earlier. Each pair of corresponding partitions is then processed as follows: if at least one of the partition pairs can reside in main memory, a hash-based scheme is utilized; the smallest element sequence is scanned and each element is put into a heap structure according to the value of its code. Subsequently, the larger relationship is probed and for each element, we utilize the heap to identify its ancestors or descendants. In the case where none of the two partitions can reside in main memory, partitioning is recursively applied for those partitions and so on until the partitions have been processed.

```
Input: b is the number of buffer pages.
       A is the ancestor node set.
       D is the descendant node set.

Function Partitioned-Based-Join (b, A, D)
k0 = ⌈min(||A||,||D||) / b⌉
Let l = ⌈log2 k0⌉. Let k = 2^l.
if k > b − 1 then k = b - 1.
Partition A and D into k partitions based on the
  nodes at level l. Save nodes above element l only in the
  corresponding partition identified by smallest code in l.
for all partition A_i and its corresponding partition D_i
do
    if ||A_i|| > b ∧ ||D_i|| > b then
        Partitioned-Based-Join(b, A_i, D_i) //Recursive partition
    else
        Memory-Containment-Heap-Based-Join(b, A_i, D_i)
    end if
end for
```

Figure 7: Partition Based Containment Join

The algorithm avoids duplicates since for each element, the pairs that it considers are produced during the processing of only two corresponding partitions. As a result the computation can stop as soon as at least one ancestor has been identified and resume with the next element to be processed. In summary, the advantages of the improved algorithm are:

- It adapts gracefully to limited memory or large document sizes
- It incurs no replication of elements
- It produces no duplicates.

4 An Improved Numbering Scheme

We present an improved numbering scheme created using the PBiTree, that enables better performance for the original join algorithm (Figure 4) as well the new algorithm (Figure 7). In the PBiTree, virtual nodes (e.g. the white nodes in Figure 2 which do not map to a node in the document tree) are necessary to allow a node with more than two children to be represented in the binary tree. Depending on the number of children of the document nodes, there is possibility that whole levels of the tree are not going to be utilized. For example, if a node in the original document has three children a, b, c, it will have two virtual nodes as children in the binary tree, and a, b and c as grandchildren. Although virtual nodes are not physically stored, they affect considerably the height of the tree. Moreover, they utilize space from the numbering scheme that is not used.

Furthermore, the possibility that elements with the same type exist at different heights increases as the level of each node is dependent not only on their level within the input XML document but also on the number of their children. This causes more horizontal partitions (during the horizontal or vertical partition algorithms of [22]) that need to be taken into consideration. As a result the number of false positives during the evaluation of the containment join increases. That leads to a larger overhead: to evaluate a result pair we need to access the actual data and perform the evaluation, i.e., one I/O per result pair (assuming no particular node clustering).

Instead we propose to utilize a k-nary tree [8, 15] where k is the largest number of node children. Using the same tree traversal (i.e. in-order) node codes can be found with similar functions to compare the ancestor-descendant relationships between two nodes. Moreover, note that with a k-nary tree, the same number or more ancestors will end up being in the same level. Therefore, a k-nary tree presents better results than the *PBiTree* in the general case where a considerable part of the nodes in the document has more than two children. For example, consider the document tree within the gray box presented in Figure 8. The document has chapters(C) and sections(S), where one section has three subsections(s). Figure 8 also presents the respective *3*-nary tree(a) and *PBiTree*(b). For clarity, virtual nodes are illustrated as gray circles, and only one virtual leaf of the *PBiTree* has been drawn. Finally, note that all sections (*S1*, *S2*, and *S3*) are at the same level in the original document and in the *3*-ary tree, while they are binarized to different levels in the *PBiTree*.

The reason for this more beneficial node clustering is that the k-nary tree takes better consideration of the structural characteristics of the document, namely the number of the children of each

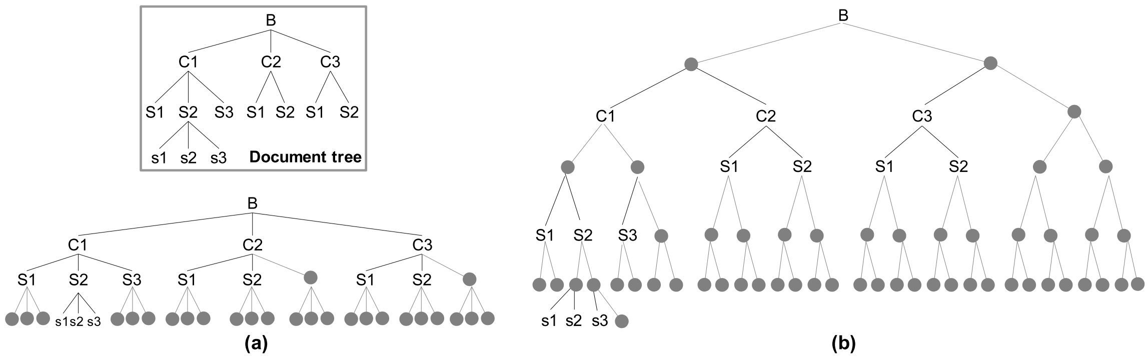

Figure 8: Document tree, and respective *tertiary tree and PBiTree.*

node as well as the node's level within the original document.

However, such an approach still suffers from several disadvantages, namely:

- It might require a large numbering space when the arity and the height of the complete k-nary tree becomes large.
- It limits the number of children to k.

One solution to the problem would be to combine the two techniques and create a *PBiTree* whore arity will be determined by the largest number of children. That would work if the number of children among nodes is similar, otherwise a large numbering space would be wasted. For this reason we can assign as k the most frequent number of children.

Finally, we node that the arity of the tree can be recomputed when a full renumbering of the tree is necessary (i.e. after a number of node additions) so that the latter takes better consideration of the current document structure.

The algorithm to create the document encoding based on the new numbering scheme is similar to the SAX-based mapping algorithm we presented in Figure 5 (after we substitute 2 with k).

5 Experimental Evaluation

We first present experimental results comparing the performance of our proposed algorithm (which we refer to as `XPJ`, for XML-data Partition-based Join), with the vertical partition of [22] (also referred as `VPJ`). We then compare `XPJ` with a representative from the set-based approaches. In particular, we utilize the state-of-the-art structural join algorithm [2], in combination with the XB-Tree indexed structure, proposed in [13]. This index enables discarding both ancestor and descendant elements that are not participating in the join result, without processing them. We utilized techniques to skip descendants [6] and ancestors [12]. The combination of these techniques with the use of the XB-Tree index (in the rest referred as `XBJ` for XB-Tree Join), results to a superior representative of the set-based techniques.

Note that set-based structural joins typically utilize the range-based numbering scheme and assume that the input is either sorted or indexed on the left position of the range. To make the experiments more comprehensive we include the query times when the input is sorted as well when it is not (so as to incur the overhead of the sorting).

5.1 Experimental Setup

We implemented all algorithms in C++ using a native storage manager implementing the LRU replacement policy. All the experiments were contacted on a 2.6GH Pentium IIII with 512MG of main memory running RedHat Linux 9, using the GNU compiler version 3.2.2. In all experiments we fixed the page size to 8K and the number of main memory buffers to 100.

The comparisons use both synthetic and benchmark data. We used synthetic datasets so as to achieve various selectivities, structure and size. To generate the synthetic datasets we created a *PBiTree* with a large height (100) and from there we created sequences of ancestor and descendant elements suitable for our experiments, by incorporating both the PBiTree coding as well as range-based coding. We also used the XMark benchmark [1], specifically the 1G XML database.

We parsed each XML file in order to add the positional representation using the algorithm described in 5. We implemented the algorithm utilizing the callback functions of an event-based XML parser conforming to the SAX interface [19]. A similar parser was used when the range-based code had to be incorporated in the results. In this case the parser performs two passes over the XML document. One pass allocates the positional presentation, while a second pass allocates the parent information (when needed). Subsequently, we created the element sequences, with an additional pass through the documents. We wrote the elements in random order on disk.

To evaluate join performance we measure total execution time for each algorithm. Each set of experiments is repeated 10 times with hot cache and the average time is reported.

5.2 Comparing XPJ and VPJ

In the first group of experiments we compare the behavior of the `XPJ` and `VPJ` algorithms with regard to the data size, as well as the degree of replication.

Effect of input size. We varied the size of the smallest relation, starting with element sequences that can reside in main memory and then we gradually increased the data, so that main memory is only 1% of the sequence size. In these experiments we had 100% query selectivity and the degree of replication was 0, i.e. no elements spanned multiple partitions. The results are presented in Figure 9.

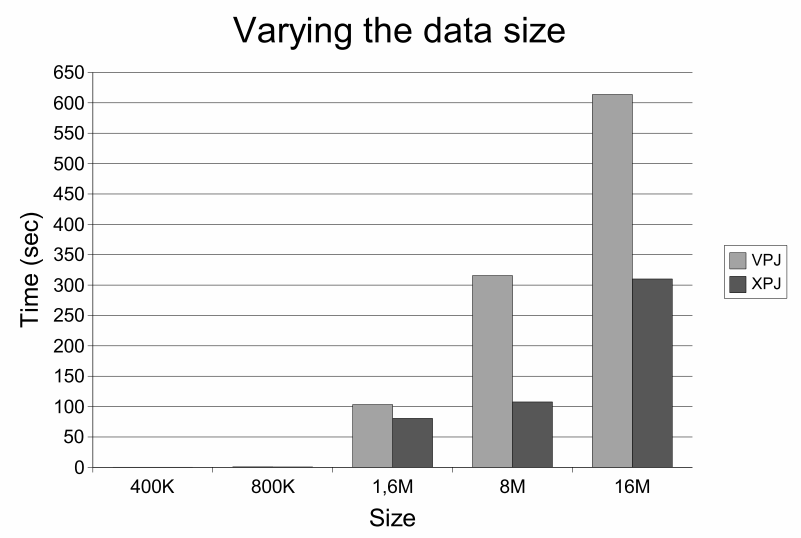

Figure 9: Performance of the partition-based algorithms with varying data sizes

For small input sizes the performance of the two partition-based algorithms is almost the same (not depicted in the figure; `VPJ` took $0,21$ sec for the 40K input and 0,82sec for the 80K input, while the corresponding values for *XPJ* were 0,17 and 0,64sec). This is to be expected since for small data sizes, the two algorithms behave similarly. However, when the input size is increasing, the performance of `VPJ` becomes prohibitively large. During the partitioning phase the `VPJ` performs an excessive amount of I/O operations. On the other hand, `XPJ` avoids this problem by making more efficient use of main memory. `XPJ` adapts gracefully to the size of the input and it is more scalable (i.e. it can still perform satisfactorily even with limited memory).

Effect of replication degree. In the second group of experiments, we kept the size of the input fixed and we varied the number of elements (ancestor and descendants) that can span several partitions. For these experiments, data was sorted in order to avoid the slow performance of the *VPJ* algorithm with large inputs. (This is to avoid the problem indicated in Figure 6. If the input is sorted then the buffers are written to disk when they become full and no partitioned data needs to be reread from disk, during the partition phase.) The results are shown in Figure 10.

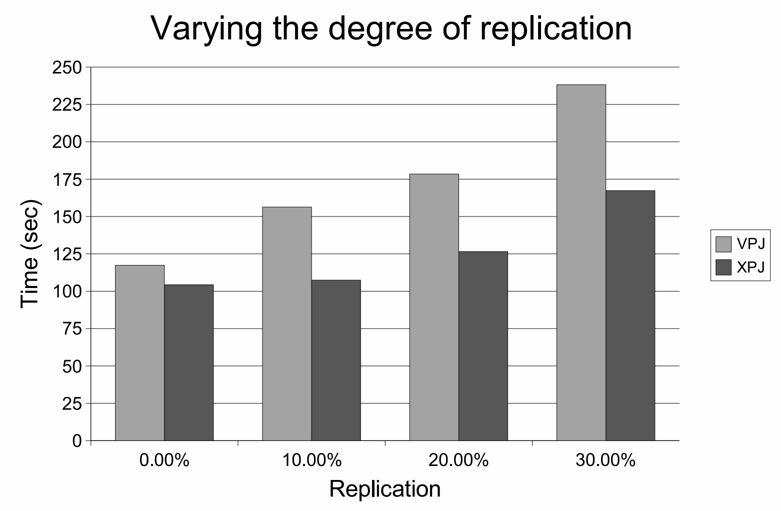

Figure 10: Performance of of the partition-based algorithms with varying degree of replication

Again, `XPJ` outperforms `VPJ` in all cases; the gap increases as the degree of replication increases. The reason is twofold: (a) the partitions are smaller and (b) descendant data do not "pollute" partitions where their corresponding ancestor elements do not exist. As a result, fewer element comparisons that will not lead to results are performed, saving processing time.

5.3 Comparison of partition- and set-based approaches

As the previous experiments showed, `XPJ` outperforms `VPJ`. An interesting open question is to consider how a partition-based join compares against a strong set-based competitor. We thus compare the `XPJ` algorithm against the `XBJ` approach. Note that replication is not an issue for these techniques and hence we consider only the effect of data size.

Effect of input size. We vary the size of the input sequences, exactly as in Section 5.2. The join selectivity is again 100%. The results are shown in Figure 11.

When the input is already sorted, the `XBJ` algorithm performs better. However, when the

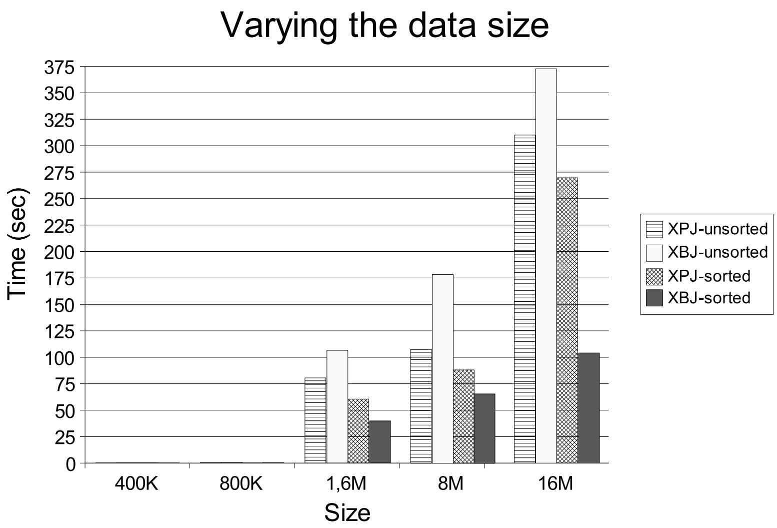

Figure 11: Partition- against set-based algorithms with varying data sizes (synthetic data)

input is not sorted, `XPJ` has a great advantage. Unsorted input can appear in many situations, for example, it can be produced as intermediate result of a more complex query. Interestingly, sorting also improves the performance of *XPJ*. The reason is that with sorted input the partition phase is very efficient, as it occurs mostly sequential I/O. In this case double buffering can further improve the performance by masking the effect of writes that are needed for the partitions to be created.

Experiments with XMark data. We also run experiments using benchmark data so as to check the validity of our results with data that come from actual XML documents. To achieve varying size for the input files, we used two element sequences and for each run we used different portions of the element sequences. The results are shown in Figure 12 and show similar trends as with custom data.

In conclusion, the experiments reveal the suitability of `XPJ` as an alternative technique for processing containment joins when data is not sorted or indexed in any particular way.

6 Conclusion

In this paper, we proposed an improved partition-based algorithm (`XPJ`) for the processing of containment joins in XML data, when the latter is not clustered, ordered or indexed in any way. Performance results prove the superiority of our technique when compared to a previously proposed

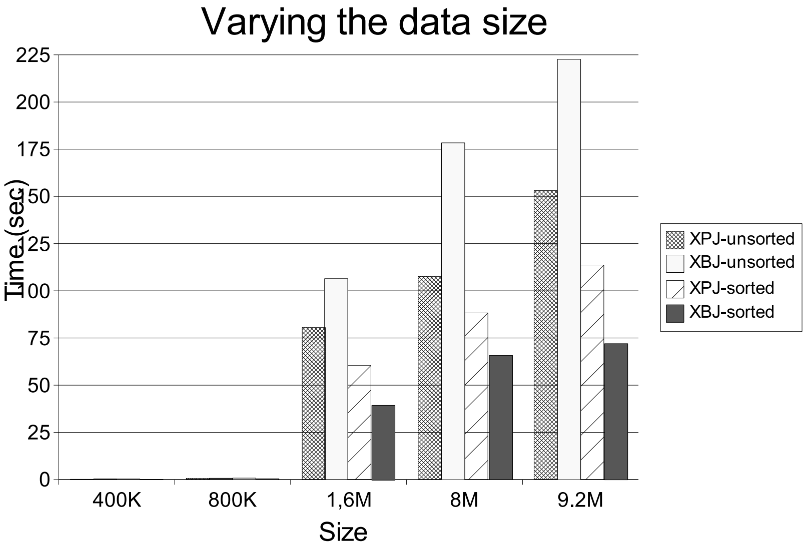

Figure 12: Partition- against set-based algorithms with varying data sizes (XMark data)

partitioned-based approach. Moreover, the new algorithm outperforms set-based approaches when no data preprocessing (in terms of indexing or sorting) has been performed to the data.

We believe that partition-based techniques are an important addition to the existing techniques, especially when the latter are not applicable, e.g. when the data are produced at intermediate steps of a query execution strategy.

As future work, we plan to devise query processing techniques utilizing different algorithms to solve containment joins and the more general query of twig join computation.

References

[1] The XML benchmark project. *Available from http://www.xml-benchmark.org.*

[2] S. Al-Khalifa, H. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. *Proceedings of ICDE*, 2002.

[3] A. Berglund, S. Boag, D. Chamberlin, M. Fernandez, M. Key, J. Robie, and J. Simeon. Xml path language (xpath) 2.0. *W3C Recommendation. Available from http://www.w3.org/TR/xpath20*, Nov. 2003.

[4] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon. Xquery 1.0: An xml query language. *W3C Working Draft. Available from http://www.w3.org/TR/xquery*, Nov. 2003.

[5] N. Bruno, N. Koudas, and D. Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. *Proceedings of ACM SIGMOD*, 2002.

[6] S.-Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo. Efficient Structural Joins on Indexed XML Documents. *Proceendings of VLDB*, 2002.

[7] J. Clark and S. DeRose. Xml path language xpath 1.0. *W3C Recommendation. Available from http://www.w3.org/TR/xpath*, Nov. 1999.

[8] T. H. Cormer, C. E. Leiserson, and R. L. Rivest. Introduction to algorithms. *The MIT Press*, 1990.

[9] Y. Diao, P. M. Ficher, M. J. Franklin, and R. To. YFilter: Efficient and Scalable Filtering of XML Documents. *Proceendings of ICDE*, 2002.

[10] G. Gottlob, C. Koch, and R. Pichler. Efficient Algorithms for Processing XPath Queries. *Proceendings of VLDB*, 2002.

[11] A. Halverson, J. Burger, L. Galanis, A. Kini, R. Krishnamurthy, A. N. Rao, F. Tian, S. Viglas, Y. Wang, J. F. Naughton, and D. J. DeWitt. Mixed Mode XML Query Processing. *Proceendings of VLDB*, 2003.

[12] H. Jiang, H. Lu, W. Wang, and B. C. Ooi. XR-Tree: Indexing XML Data for Efficient Structural Joins. *Proceendings of ICDE*, 2003.

[13] H. Jiang, W. Wang, H. Lu, and J. X. Yu. Holistic Twig Joins on Indexed XML Documents. *Proceendings of VLDB*, 2003.

[14] C. Koch. Efficient Processing of Expressive Node-Selecting Queries on XML Data in Secondary Storage: A Tree Automata-Based Approach. *Proceedings of VLDB*, 2003.

[15] Y. K. Lee, S.-J. Yoo, K. Yoon, and P. B. Berra. Index Structures for Structured Documents. *Proceendings of VLDB*, 1996.

[16] Q. Li and B. Moon. Indexing and Querying XML Data for Regular Path Expressions. *Proceendings of VLDB*, 2001.

[17] Q. Li and B. Moon. Partition Based Path Join Algorithms for XML Data. *Proceendings of DEXA*, 2003.

[18] P. R. Rao and B. Moon. PRIX: Indexing and Querying XML Using Prufer Sequences. *Proceendings of ICDE*, 2004.

[19] SAX. Simple API for XML. *http://sax.sourceforge.net.*

[20] M. D. Soo, R. Snodgrass, and C. S. Jensen. Efficient Evaluation of the Valid-Time Natural Join. *Proceendings of ICDE*, 1994.

[21] H. Wang, S. Park, W. Fan, and P. S. Yu. ViST: A Dynamic Index Method for Querying XML Data by Tree Structures. *Proceedings of ACM SIGMOD*, 2003.

[22] W. Wang, H. Jiang, H. Lu, and J. X. Yu. PBiTree Coding and Efficient Processing og Containment Joins. *Proceedings of ICDE*, 2003.

[23] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman. On Supporting Containment Queries in Relational Database Management Systems . *Proceendings of SIGMOD*, 2001.