Available online at www.sciencedirect.com

SCIENCE DIRECT®

Information Processing Letters 91 (2004) 69–75

Information Processing Letters

www.elsevier.com/locate/ipl

Efficient structural joins with clustered extents ☆,☆☆

Jongik Kim [a,*], Sang Ho Lee [b], Hyoung-Joo Kim [a]

[a] *School of Computer Science and Engineering, Seoul National University, Seoul 151-742, Republic of Korea*
[b] *School of Computing, Soongsil University, Seoul 156-743, Republic of Korea*

Received 3 December 2003; received in revised form 19 March 2004

Available online 6 May 2004

Communicated by F.Y.L. Chin

Keywords: Databases; XML; Path query; Structural join

1. Introduction

Tree-structured models are widely used to represent XML data. Queries on a data tree consist of the two parts, selection predicates on tree nodes and structural relationships between selection predicates. For example, the path query:

street[name = 'Tehran']//restaurant

matches restaurant elements that are descendant of street elements that have a child name element whose content is the string value 'Tehran'. There are four selection predicates (street, name, 'Tehran', restaurant) and three structural relationships (street/name, name/'Tehran', street//restaurant) in this path query.

Processing such a path query goes through the following three steps:

(i) retrieval of an extent, which is a list of nodes, for each selection predicate,
(ii) finding occurrences of structural matches between tree nodes in extents, and
(iii) "stitching" together structural matches generated in step (ii).

In order to retrieve an extent in the first step, an inverted index built on selection predicates is used. For each selection predicate, an extent can be easily retrieved by looking up the inverted index.

Finding occurrences of the structural matches in the second step is a core operation in XML query processing. To solve this sub-problem, Zhang et al. [6] proposed the multi-predicate merge join (MPMGJN) algorithm, which is an extension of the traditional merge join algorithm. Al-Khalifa et al. [1] generalized the MPMGJN algorithm to the tree-merge join algorithms that consider the order of join results. Furthermore, they proposed the stack-tree join algorithms that can improve the tree-merge join algorithms. Those algorithms are dependent on the representation of positions of XML elements (and string values) to determine structural relationships between tree nodes. Recently, Chien et al. [3] proposed a structural join algo-

☆ This work was supported in part by the Brain Korea 21 project.
☆☆ This work was supported in part by the Ministry of Information and Communications, Korea, under the Information Technology Research Center (ITRC) Support Program.

* Corresponding author.
E-mail addresses: jikim@oopsla.snu.ac.kr (J. Kim), shlee@comp.ssu.ac.kr (S.H. Lee), hjk@oopsla.snu.ac.kr (H.-J. Kim).

rithm that uses B+ tree to skip unnecessary elements in the extents for descendant nodes without scanning. However, the B+ tree technique cannot effectively skip elements in the extent for ancestor nodes.

Stitching together structural matches in the final step poses the problem of selecting a good join ordering. Wu et al. [5] proposed a cost-based join order selection of structural joins. Bruno et al. [2] developed a holistic structural join algorithm that can evaluate the second and third step simultaneously.

In this paper, we discuss issues of the structural join in the second step, and propose an efficient structural join technique. Our technique partitions all the nodes in an extent into several clusters. Given two extents to be joined, the proposed technique filters out unnecessary clusters in both extents before joining. We also take advantage of the representation of positions of XML elements. Unlike other techniques, however, we use the position information only to cluster nodes in the extents.

2. Our approach

2.1. The representation of element positions

The position of each element in an XML document is represented by a 3-tuple value (DocId, StartPos:EndPos, LevelNum), where

(i) DocId is an identifier of the document,

(ii) StartPos represents the number of words from the beginning of the document to the start of the element, and EndPos represents the number of words from the beginning of the document to the end of the element, and

(iii) LevelNum denotes the nesting depth of the element.

A string value in an XML document has the same value as StartPos and EndPos.

Once each node of an XML data tree is represented in the aforementioned way, the structural relationship between tree nodes can be easily determined as follows. For the ancestor-descendant relationship, a tree node n_1 is an ancestor of a tree node n_2 if and only if n_1.DocId $=$ n_2.DocId, n_1.StartPos $<$ n_2.StartPos, and n_1.EndPos $>$ n_2.EndPos. For the parent–child relationship, a tree node n_1 is a parent of a tree node n_2 if and only if n_1.DocId $=$ n_2.DocId, n_1.StartPos $<$ n_2.StartPos, n_1.EndPos $>$ n_2.EndPos, and n_1.LevelNum $+ 1 =$ n_2.LevelNum.

For simplicity we deal with a single document in this paper. An extension to multiple documents is trivial. We only consider the ancestor-descendant relationships. The parent–child relationships can be easily verified by comparing LevelNum values of ancestor-descendant relationships. Hence, we use only two numbers, StartPos and EndPos, to represent positions of the tree nodes.

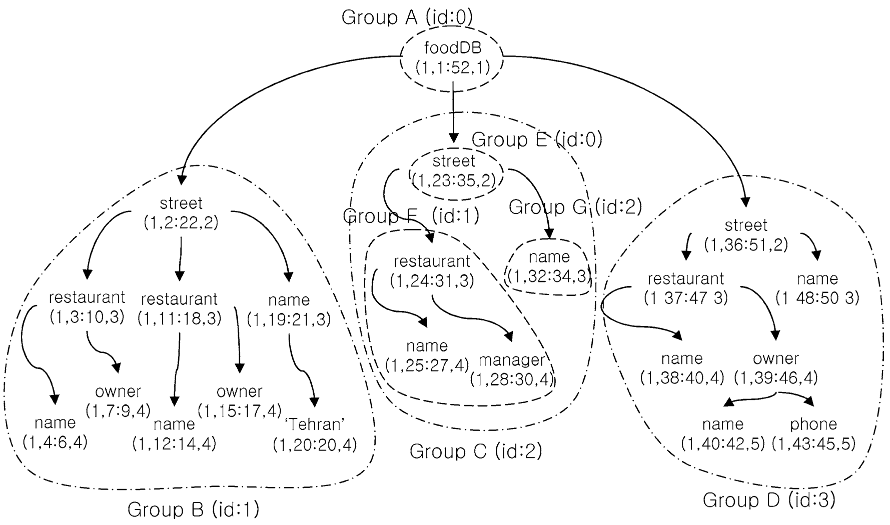


Fig. 1. An XML data tree.

2.2. Proposed technique

In this section, we present a new structural join technique. Our basic idea is that we can reduce the number of nodes to be scanned for the join by (i) partitioning the data tree into several groups and (ii) performing the join operation between tree nodes that belong to the same group. First, we describe how to group the nodes in the data tree and how to cluster the nodes in the extent. Second, we present our join algorithms that take advantage of the cluster of notes in the extent.

We partition a data tree into several groups. Each group of tree nodes has a unique identifier, GroupId. Each group is constructed as follows. (i) The *root* node is a group, whose GroupId is 0. (ii) Each subtree of the root node is a group, where a subtree of a node, n, denotes a tree that is rooted at a child of the node, n. The GroupId of the subtree that is rooted at the ith child of the root is i. It is obvious that each group is disjoint. Fig. 1 shows a data tree that is grouped accordingly. Each subtree of the root node is partitioned recursively. For example, *Group C* is recursively partitioned into the three groups $(\mathrm{E}, \mathrm{F}, \mathrm{G})$ in Fig. 1. When a subtree of a node is partitioned into groups, GroupId of each group is assigned starting at 0. For example, In Fig. 1, GroupId of *Group A* that contains a node *foodDB* is 0. GroupId of *Group E* that contains a node *street* is also 0. Note that GroupId is assigned to a group, and not to a tree node.

Utilizing the groups in the data tree, we also partition nodes in an extent into a number of clusters, which are lists of tree nodes. Essentially, tree nodes that belong to the same group in the data tree are

```
Algorithm ClusterExtent(root, extent)
{
    // root is the root node of the subtree currently processed.
    // extent is a list of tree nodes, in sorted order of StartPos

    CList = an empty list of clusters;
    desc = extent → firstNode;

    if(desc = root){
        // the groupId of the root is 0
        CList → append(new Cluster(0, desc));
        desc := desc → nextnode;
    }
    if(desc = NULL) return CList;

    for(id := 1; id < root → numberOfChildren; id++){
        newcluster := an empty list of tree nodes;
        // append descendant nodes that belong to idth subtree
        for(anc := root → child(id);
            anc is an ancestor of desc; desc := desc → nextNode)
            newcluster → append(desc);

        if(newcluster is not empty){
            Cluster c := ClusterExtent(anc, newcluster);
            // the groupId of the nodes that belong to idth subtreeroot is id
            CList → append(new Cluster(id, c));
        }
    }
    return CList;
}
```

Algorithm 1. Clustering tree nodes in an extent.

mapped into the same cluster in an extent. Tree nodes in the cluster in the extent are partitioned recursively, similarly in the way that the data tree is grouped recursively.

Algorithm 1 clusters an extent on a basis of the groups of a data tree. In the algorithm, a cluster is represented as a pair of a GroupId and a pointer. The pointer in a cluster refers either a tree node or a list of tree nodes that are clustered recursively. *new Cluster*(*id, pointer*) in line 11 and line 26 makes a cluster with id and pointer. From line 9 to line 13, the algorithm identifies whether *extent* contains the root node of a data tree or not. If *extent* contains the root node, the algorithm makes the root node a separate cluster. In the *for* loop in line 16, the algorithm clusters tree nodes in *extent* that belong to the same group in a data tree. In line 24, the tree nodes in each cluster are clustered recursively.

When nodes in an extent for a selection predicate are clustered, the extent can be represented as a tree, which we call *extent tree*. An intermediate node in an extent tree is a list of clusters (or GroupIds) in a sorted order of GroupId. Leaf nodes in an extent tree are the nodes in the data tree that satisfy the selection predicate. Interestingly, the leaf nodes of an extent tree are sorted by StartPos. In the extent tree, a leaf node of an intermediate node, n, denotes a leaf node of the subtree that is rooted at n.

Example 1. The list $L = [(3{:}10), (11{:}18), (24{:}31), (37{:}47)]$ contains the tree nodes that match the selection predicate, *restaurant*, in Fig. 1. The tree nodes in L are partitioned into the three clusters, [(3:10), (11:18)], [(24:31)], and [(37:47)], where GroupIds are 1, 2, and 3, respectively. The cluster [(3:10), (11:18)] is partitioned into the two clusters, [(3:10)] and [(11:18)], where GroupIds are 1 and 2, respectively. [(24:31)] and [(37:47)] are partitioned into [(24:31)] and [(37:47)], respectively, where GroupIds are 1 and 1. Finally, [(3:10)], [(11:18)], [(24:31)], and [(37:47)] are partitioned into (3:10), (11:18), (24:31), and (37:47), where all GroupIds are 0. Fig. 2 shows the extent tree.

Now, we describe our join algorithm that takes advantage of clustered extents. Consider an ancestor-descendant structural relationship (e_1, e_2), for example (*restaurant*, *owner*). Let $\text{AList} = [a_1, a_2, \ldots]$ and $\text{DList} = [d_1, d_2, \ldots]$ be root nodes of extent trees, where the leaf nodes of AList correspond the predicate e_1 and the leaf nodes of DList correspond the predicate e_2. Both AList and DList are sorted by GroupIds of their elements. For example, AList for the selection predicate, *restaurant*, is $[1, 2, 3]$ as depicted in Fig. 2.

Basically our algorithm performs either (i) AList $\bowtie$ DList or (ii) (AList $-$ $[a_1]$) $\bowtie$ DList, using the traditional merge join algorithm where the join condition is the equality between *GroupId*s of clusters. If GroupId of a_1 is 0, then the child node of a_1 should be the root node of the data tree (or the subtree that is being processed). Since the root node is an ancestor of every node in the data tree, our algorithm outputs $(a_1 \to \mathit{child}, n_i)$ for each leaf node n_i of DList and then performs (AList $-$ $[a_1]$) $\bowtie$ DList. Otherwise (i.e., GroupId of a_1 is not 0), our technique performs AList $\bowtie$ DList. For each (a_i, d_j) of AList $\bowtie$ DList, our algorithm repeats the same procedure recursively.

Algorithm 2 shows our join algorithm. From line 7 to line 12, the algorithm outputs partial results of the join. From line 15 to line 22, the algorithm performs the traditional join procedure with the equality condition of GroupIds. For each matched pair, the al-

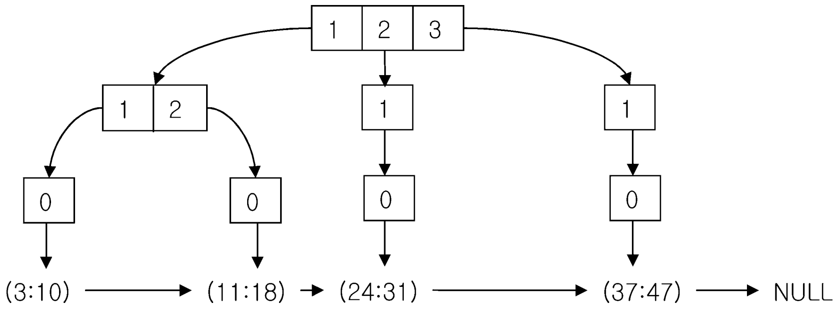


Fig. 2. An extent tree for the selection predicate, restaurant.

```
 1: Algorithm ExtentJoinAnc(AList, DList)
 2: {
 3:   // AList is a list of node clusters, in sorted order of GroupId
 4:   // DList is a list of node clusters, in sorted order of GroupId
 5:
 6:    a := AList → 1stCluster;
 7:    if(a → groupid = 0){
 8:       subtreeroot := a → child;
 9:       foreach leaf node, desc, of DList
10:          append (subtreeroot, desc) to OutputList;
11:       a := AList → 2ndCluster;
12:    }
13:    d := DList → 1stCluster;
14:
15:    while(both of the input lists are not empty){
16:       while(a → groupid > d → groupid) d := d → nextCluster;
17:       if(a → groupid = d → groupid){
18:          ExtentJoinAnc(a → child, d → child);
19:          d := d → nextCluster;
20:       }
21:       a := a → nextCluster;
22:    }
23: }
```

Algorithm 2. ExtentJoinAnc with output in an ancestor order.

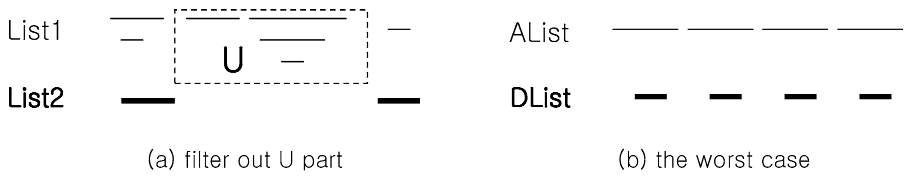


Fig. 3. Example extents.

gorithm calls the *ExtentJoinAnc* procedure recursively in line 18.

Algorithm 2 outputs results by traversing an extent tree of Alist in a *top-down* and *left-to-right* fashion. Consequently join results are generated in an ancestor order. To get join results in a descendant order, we need to modify Algorithm 2 slightly as follows:

Instead of outputting a *subtreeroot* in line 10 of Algorithm 2, we push the *subtreeroot* into a stack. Note that the stack contains all the ancestor nodes of a subtree currently being processed, and that the top of the stack could be the root node of the current subtree. Therefore, all possible combinations of every leaf node of DList and every node in the stack should be outputted. For tree nodes in DList that have no ancestor in AList, we can output the tree nodes in DList with the nodes in the stack. For tree nodes in DList that have at least an ancestor in AList, we do not output them at this point, but resort to the next recursion to get join results. Using a stack, Algorithm 3 outputs join results in a descendant order. From line 5 to line 8 in Algorithm 3, the algorithm pushes the root node of the current subtree to the stack, if it exists in AList. In line 14 and line 25, the algorithm outputs the tree nodes in DList that have no ancestor in AList.

Our algorithms efficiently filter out unnecessary clusters (or subtrees) in both AList and DList in each recursion, so that they can reduce the number of nodes that participate in the join. For example, the *U* part of Fig. 3(a) is filtered out by comparing a few GroupIds in our algorithms, while the existing algorithms should

```
Algorithm ExtentJoinDesc(AList, DList)
{
   // node-stack is a global stack
   a := AList → 1stCluster;
   if(a → groupid = 0){
      node-stack → push(a → child);
      a := AList → 2ndCluster;
   }
   d := DList → 1stCluster;

   while(both of the input lists are not empty){
      while(a → groupid > d → groupid){
         if(node-stack is not empty)
            doOutput(d → 1stLeaf, d → lastLeaf);
         d := d → nextCluster;
      }
      if(a → groupid = d → groupid){
         ExtentJoinDesc(a → child, d → child);
         d := d → nextCluster;
      }
      a := a → nextCluster;
   }

   if(node-stack is not empty and d is not NULL)
      doOutput(d → 1stLeaf, DList → lastLeaf)

   if((AList → 1stCluster) → groupid = 0)  node-stack → pop()
}

Procedure doOutput(firstLeaf, lastLeaf)
{
   for(desc = firstLeaf; desc != lastLeaf; desc := desc → nextLeaf)
      for each element, anc, in node-stack, output (anc,desc)
}
```

Algorithm 3. ExtentJoinDesc with output in a descendant order.

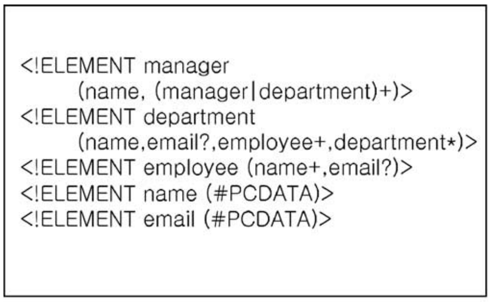
scan every node in the U part. Fig. 3(b) shows the worst case of our technique. Because there is no cluster to be filtered out, our algorithms should scan all the lists sequentially. However, such regular and rigid structured data are hardly found in XML.

3. Preliminary results

In this section, we provide preliminary results of our experiments. We used an XML database system, which includes an object storage manager, a buffer manager and a B+ tree index manager. A page of our system is 4 Kbyte, and the system has 200 memory buffers for pages. We ran our experiments on a machine with a 600 MHz Intel Pentium III processor, 192 MB of memory. We used the IBM XML data generator [4] to generate XML data in our experiments. Fig. 4(a) shows our document type definition (DTD), which is actually identical with the one [1] except some attributes that are not related with the join procedure. We generated XML data of 50 M size.

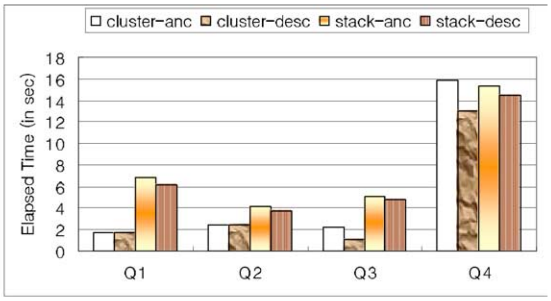
The queries used for the experiments are:

(Q1) employee//email,
(Q2) manager//department,



```
<!ELEMENT manager
      (name, (manager|department)+)>
<!ELEMENT department
      (name,email?,employee+,department*)>
<!ELEMENT employee (name+,email?)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT email (#PCDATA)>
```

(a) DTD used in our experiments



(b) Response time for structural join

Fig. 4. Preliminary results.

(Q3) department//email, and
(Q4) department//employee.

We compared our algorithms with the stack-tree algorithm family [1].

Fig. 4(b) shows the experimental results. For Q1, Q2 and Q3, our algorithms outperform the stack-tree algorithms, because our algorithms can filter out unnecessary clusters of nodes. For Q4, our algorithms perform slightly slower than the stack-tree algorithms. As for Q4, every employee element has more than one department element as its ancestor, and every department element has at least one employee element as its descendant, as is shown in the DTD. Hence, there is no cluster to be filtered out. However, the difference of performance is ignorable.

4. Conclusions

In this paper, we propose an efficient structural join technique. We partition nodes in a data tree into several groups, and perform the join operation between nodes that belong to the same group. Grouping the tree nodes allows us to filter out unnecessary groups efficiently without scanning the tree nodes, whereas other techniques cannot.

Performance results show that our technique outperforms the existing techniques in most cases. We believe that the cases in which our technique is not as good as the existing algorithms are rare in real applications.

We are currently exploring a number of optimization issues in our algorithms. Those include utilization of level information of a predicate to prune an extent tree, use of various context information on queries issued by users, and so on. Those issues remain as future work.

References

[1] S. Al-Khalifa, H.V. Jagadish, N. Koudas, J.M. Patel, D. Srivastava, Y. Wu, Structural joins: a primitive for efficient XML query pattern matching, in: Proc. IEEE International Conference on Data Engineering, 2002.

[2] N. Bruno, N. Koudas, D. Srivastava, Holistic twig joins: Optimal XML pattern matching, in: Proceedings of the ACM SIGMOD International Conference on the Management of Data, 2002.

[3] S.-Y. Chien, Z. Vagena, D. Zhang, V.J. Tsotras, C. Zaniolo, Efficient structural joins on indexed XML documents, in: Proceedings of the Conference on Very Large Data Bases, 2002.

[4] IBM XML Generator, http://www.alphaworks.ibm.com/tech/xmlgenerator.

[5] Y. Wu, J.M. Patel, H.V. Jagadish, Structural join order selection for XML query optimization, in: IEEE International Conference on Data Engineering, 2003.

[6] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, G. Lohman, On supporting containment queries in relational database management systems, in: Proceedings of the ACM SIGMOD International Conference on the Management of Data, 2001.