

# Improving XML Query Performance

## Using Social Classes

Weining Zhang and Douglas Pollok

Department of Computer Science

University of Texas at San Antonio

{wzhang, dpollok}@cs.utsa.edu

30th October 2003

### Abstract

In a state-of-the-art XML database, an XML query is evaluated as a sequence of structural joins in which positions of data nodes are used to perform each individual structural join. In this paper, we define the notion of social classes of data nodes and present a framework of query evaluation in which both positions and social classes of data nodes are used with structural joins to further improve query performance. A social class of a data node is defined as an equivalence class induced by tags of other nodes that are associated with the given node in a given structural relation. In our framework, social classes of data nodes are obtained during data loading. Then during query compilation, queries are analyzed to determine required structural relations among query nodes and to derive required social classes for each individual query node. The positions of data nodes, the social classes of data nodes, and the required social classes of query nodes are used during query evaluation to provide an effective mechanism for filtering and indexing XML data. We present a number of algorithms that implement this framework and report on results from our experiments. Our results show that this new method could substantially improve performance of XML queries that require multiple structural joins.

## 1 Introduction

The eXtensible Markup Language (XML) [BPSMM00] has been widely accepted as the standard format of electronic data exchange. As XML data becomes ubiquitous, the management of such data in database

systems becomes an increasingly important research area. Since XML documents are semistructured, they often do not conform to or even have a schema and they are often modeled as ordered labeled trees. To query XML data, XML query languages such as XQuery [BCF<sup>+</sup>02] rely on path navigation patterns to specify portions of an XML data tree that should be retrieved and transformed. Such path navigation patterns are expressed as path queries in the proposed standard query language XPath [BBC<sup>+</sup>02].

Path queries can be viewed as pattern trees and their evaluation can be viewed as a process of finding all possible embeddings of the pattern trees in the data tree. Due to the importance of path queries, many methods have been proposed for path query evaluation. These methods take two different approaches: the graph index approach and the structural join approach. In the graph index approach [CMS02, KBNK02], data nodes are partitioned into equivalence classes based on a similarity relation. These equivalence classes define a summary graph of the data tree in which nodes represent the equivalence classes and edges represent structural relationships between equivalent classes. A path query is evaluated against the summary graph instead of the data tree. The graph nodes that satisfy the query are then used as an index to retrieve data nodes of the answer. The effectiveness of this approach relies critically on the summary graph being much smaller than the data tree. However, for some datasets, the graph index may become much larger than the data tree itself. In the structural join approach [CVZ<sup>+</sup>02, WJLY03, AKJK<sup>+</sup>02, JLWO03, BKS02, WPJ03], a path query is evaluated through a sequence of structural join operations. A structural join takes two streams of data nodes as input and produces pairs of nodes, where each pair of nodes satisfy a required structural relationship. Structural join algorithms [AKJK<sup>+</sup>02, ZND<sup>+</sup>01, JLWO03, CVZ<sup>+</sup>02, WPJ03, BKS02] use positions of data nodes within XML documents to quickly determine if any two given nodes have an ancestor-descendant or parent-child relationship. By reading input streams in ascending order of node positions and keeping ancestor nodes in an in-memory stack, algorithms in [AKJK<sup>+</sup>02] only need to scan each input stream once. Improved algorithms in [CVZ<sup>+</sup>02, JLWO03] use node position based indexes to skip data nodes that do not actually produce any join result, so that to achieve a much better performance. Structural joins of a query can be evaluated in an order different from the order of navigation steps specified in the query, according to estimated evaluation cost. Using structural joins, the answer of a query can be computed in a-set-at-a-time fashion, which often results in a good performance.

Determining the optimal join order for a path query with many structural joins is known to be a difficult task [AKJK<sup>+</sup>02]. In practice, suboptimal join orders are often chosen by query optimizers. However as explained in Section 5, even if an optimal join order and the best structural join algorithm are chosen to

evaluate a path query, there may still be many data nodes that are relevant to some intermediate result (for example, by participating in one or more structural join) but are irrelevant to the final result of the query. We call these nodes *false relevant* nodes. False relevant nodes contribute to the overall evaluation cost, yet are difficult to get rid of using techniques that know only node positions. Thus we need additional structural information to identify false relevant nodes and to discard them from input streams of structural joins. In this paper, we make the following contributions.

1. We define the notion of social classes of data nodes, which are equivalence classes of data nodes induced by tags of other nodes that are associated with the given nodes in given structural relations.
2. We present a framework of query evaluation in which both positions and social classes of data nodes are used with structural joins to further improve query performance. In this framework, social classes of data nodes are obtained during data loading. During query compilation, explicitly expressed structural relations among query nodes are used to derive implicitly required structural relations, which are then used to obtain required social classes for each individual query node. The positions of data nodes, the social classes of data nodes, and the required social classes of query nodes are used during query evaluation to provide an effective mechanism for filtering and indexing XML data.
3. We present a number of algorithms that implement the proposed framework of query evaluation.
4. We performed a number of experiments to evaluate the effectiveness of social class based methods. Our results show that these new methods can substantially improve performance of XML queries that require multiple structural joins. In addition, our method can drastically reduce the differences in performance caused by different join orders, thus making the choice of the optimal join order less critical.

The remaining part of this paper is organized as follows. In Section 2, we present basic concepts of XML data model and discuss our motivation using an example. In Section 3, we define the notion of social classes of data nodes and required social classes of query nodes. In Section 4, we present our query evaluation framework and algorithms. In Section 5, we present results from our experiments. In Section 6, we discuss related work. Finally, Section 7 concludes the paper.

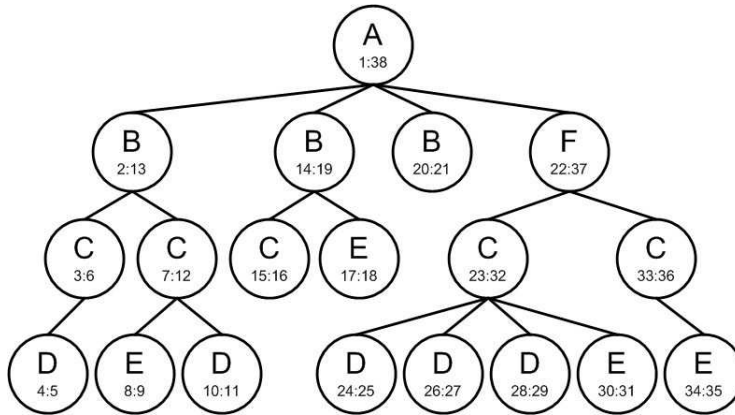


Figure 1: An XML Data Tree

## 2 Preliminaries and Motivation

In this section, we briefly introduce basic concepts of XML data model and XML query. We also discuss our motivation using an example.

### 2.1 XML Data Model and XML Queries

An XML document contains a sequence of elements which are marked by a pair of start and end tags. Each element may contain a number of attributes, one or more sub-elements, or a data value. An XML document is represented as an ordered labeled tree in which nodes represent XML fragments (such as, elements, attributes and data values), and edges represent the containment of various XML fragments in elements. Internal tree nodes are labeled with tags of elements or attributes, and leaf nodes are labeled with data values. Figure 1 shows the tree of a sample XML document with data values omitted to keep the figure simple. In this XML document, the root element has tag A, and it contains three B elements and one F element. The first B element contains two C elements, and so on.

Each XML fragment (and its node in the XML data tree) has a pair of start and end positions in the XML document. These positions are shown in Figure 1 as a pair of integers (representing a word count from the beginning of the XML document). For example, the positions of the leftmost B-node are 2:13. For simplicity, we identify data nodes by their start positions. Structural relationships, such as parent-child, ancestor-descendant, following-preceding-siblings, etc., are defined according to the convention for trees.

To be more precise, a data tree is defined by  $D = (V, E, L)$ , where  $V = \{v_1, \dots, v_n\}$  is a set of nodes,

$E \subseteq V \times V$  is a set of edges where  $\langle v_i, v_j \rangle \in E$  means that element  $v_i$  contains element (or attribute)  $v_j$ , and  $L = \{l_1, \dots, l_m\}$  is a set of node labels. For a node  $v \in V$ , we denote its label by  $tag(v) \in L$ , start position by  $start(v)$ , end position by  $end(v)$ , and tree level by  $level(v)$ . In general, the  $start(v)$ ,  $end(v)$ , and  $level(v)$  are referred to as the position of node  $v$ . The positions of any two data nodes  $v_i$  and  $v_j$  have the following properties.

1. Intervals  $[start(v_i), end(v_i)]$  and  $[start(v_j), end(v_j)]$  are either disjoint or completely nested.
2. If  $[start(v_i), end(v_i)]$  contains  $[start(v_j), end(v_j)]$ , then  $v_i$  is an ancestor of  $v_j$ . Furthermore, if  $level(v_i) = level(v_j) + 1$ ,  $v_i$  is also the parent of  $v_j$ .

These properties are the basis for using node positions to quickly determine if two data nodes are in a containment relation (ancestor-descendant or parent-child).

A structural relation is defined as a subset of the cross-product of nodes, that is,  $R \subseteq V \times V$ . Instances of structural relations are ordered. For example, if  $\langle v_i, v_j \rangle \in Parent$ , the second node,  $v_j$ , is the parent of the first node,  $v_i$ , but the reverse is not true. In this case, we say that  $v_j$  is a *relative* of  $v_i$  in  $R$ .

A path query consists of a sequence of navigation steps (or location steps in XPath terminology). For example, “/child::A/descendant::E/following-sibling::D” is a simple path query that retrieves a set of D-nodes that are following-siblings of some E-nodes, which in turn, are descendants of top level ancestor A-nodes. In this query, “child::”, “descendant::”, and “following-sibling::” are called axes, and they specify required structural relations among data nodes of the answer. The only node in the data tree in Figure 1 that will satisfy the above query is the second D-node (i.e., node 10). In contrast, query “/child::A/descendant::E[/following-sibling::D]” will retrieve the E-node (node 8) instead. In this query, the square bracket defines an additional condition on E nodes, namely that they must have at least one D-node sibling to its right in the tree (that is, a following-sibling).

A query is represented by a *query tree* defined by  $Q = (V_Q, E_Q, L)$ , where  $V_Q$  is a set of query nodes,  $E_Q = \{\langle q_i, R, q_j \rangle \mid q_i, q_j \in V_Q, R \in SR\}$  is a set of edges labeled by the names of structural relations ( $SR$ ) that are defined on data nodes, and  $L$  is the set of node labels. Each distinct edge label  $R$  defines a required structural relation  $R' = \{\langle q, q' \rangle \mid \langle q, R, q' \rangle \in E_Q\}$ . For simplicity, we often refer to both the structural relation  $R$ , defined on data nodes, and the required structural relation  $R'$ , defined on query nodes, as the same relation  $R$ , although technically they are different. Since different query nodes may have identical labels, we identify query nodes with unique numeric identities. Figure 2 shows query

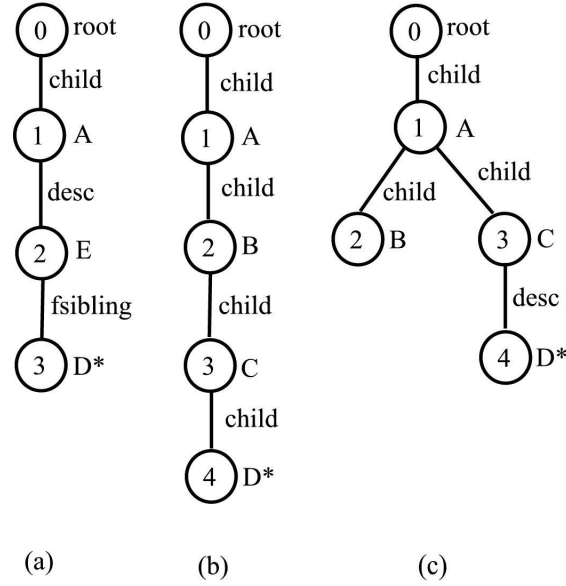


Figure 2: Query Trees

trees of three queries, where tree (a) represents “/child::A/descendant::E/following-sibling::D” and tree (c) represents “/A[/B]/C//D”. Each query tree has a unique root node representing the document root. Nodes labeled with “\*” are called output nodes.

Let  $D = (V, E, L)$  be an XML data tree,  $Q = (V_Q, E_Q, L)$  be a path query, and  $SR$  be the set of structural relations defined on  $D$ . An **embedding** of  $Q$  (in  $D$ ) is a mapping  $e : V_Q \rightarrow V$ , so that

1. for every query node  $q \in V_Q$ ,  $tag(q) = tag(e(q))$ , and
2. for every query edge  $\langle q, R, q' \rangle \in E_Q$ ,  $\langle e(q), e(q') \rangle \in R$ , and  $R \in SR$ .

If  $e$  is an embedding of  $Q$ , we say that data node  $e(q)$  satisfies query node  $q$ . The answer to query  $Q$  is the set of data nodes that satisfy the output node in possible embeddings of  $Q$ . For query tree (a) in Figure 2 and the data tree in Figure 1, there exists only a single embedding with the following mapping:

Query node $q$	Data node $e(q)$
0	document root
1	1
2	8
3	10

Thus, the answer contains a single D-node, namely node 10.

A path query can be evaluated as a sequence of structural joins, one for each navigation step. For example, the query “/child::A/descendant::E/following-sibling::D” can be evaluated as a sequence of three structural joins for “(root)/child::A”, “A/descendant::E”, and “E/following-sibling::D”, respectively. Consider the structural join for “A/descendant::E”. This structural join takes a stream of A-nodes (the ancestor stream) and a stream of E-nodes (the descendant stream) as input, with both streams sorted in ascending order on start positions, and uses the properties of node positions to identify E-nodes that are descendants of some A-nodes. Based on its intended use, the result of the join can be a set of A-nodes, a set of E-nodes, or a set of A-E node pairs. No matter what the result will be, the basic algorithm is the same. For each root-to-leaf path in the data tree, the algorithm keeps ancestor A-nodes in a stack to avoid scanning the descendant stream repeatedly. As a result, the complexity of a structural join is  $O(n + m + k)$ , where  $n$ ,  $m$  are numbers of nodes in the two input streams, and  $k$  is the number of nodes in the output. If one or both input stream has an index on node position, the index can be used to skip nodes that do not produce any join output. For example, A-nodes that are not ancestors of any E-node and E-nodes that are not descendants of any A-node could be skipped. Skipping irrelevant input nodes can substantially improve the performance of structural joins.

## 2.2 Motivation

Consider the query “/A/B/C/D” (which is the shorthand of “/child::A/child::B/child::C/child::D”) against the data tree in Figure 1, The answer to the query contains two D-nodes: nodes 4 and 10. This query needs to compute four structural joins for “(root)/A”, “A/B”, “B/C” and “C/D”, respectively. If the output of each join is pipelined into the next join, these joins can be performed in four different orders, one of which is given in the previous sentence. Notice that “/A”, “A/B”, “C/D”, “B/C” is not a feasible pipelined join order because neither the A-nodes nor the B-nodes obtained from the join for “A/B” are input to “C/D”. These different join orders result in different query costs. Let us first consider the structural join for “B/C”. To be concrete, we assume that the join result is a set of B-C node pairs and that the join is the first to perform. From Figure 1, the two input streams of this join contains 3 B-nodes and 5 C-nodes, respectively. Using the algorithm of [AKJK<sup>+</sup>02], all 8 nodes will be retrieved. If the B+-tree index based algorithm of [CVZ<sup>+</sup>02] is used, two C-nodes will be skipped and the remaining 6 nodes (the same 3 B-nodes and C-nodes 3, 7, and 15) will be retrieved. If the XR-tree index based algorithm of [JLWO03] is used, then 5 nodes (B-nodes

2, 14 and C-nodes 3, 7, 15) will be retrieved. No matter which join algorithm is used, the result of the join will always be the three node pairs  $\{ \langle 2,3 \rangle, \langle 2,7 \rangle, \langle 14,15 \rangle \}$ . However, among the three node pairs only  $\langle 2,3 \rangle$  and  $\langle 2,7 \rangle$  will contribute to the final answer. Thus, both B-node 14 and C-node 15 are false relevant. In current structural join approach, these false relevant nodes are not discarded until much later in the evaluation process, after a possibly high evaluation cost has been paid. It is obviously desirable for false relevant nodes to be discarded as early as possible in the evaluation process. Note that node positions do not provide enough information for this purpose, therefore, we need additional information.

By analyzing the query more closely, it becomes clear that the set of conditions implied by the query is much larger than just those explicitly expressed. For example, with “B/C”, the query only explicitly expresses that in order for an input C-node to be relevant to the query, it must have a parent B-node. But together with other navigation steps, the query really implies that in order for an input C-node to be relevant to the query, it needs to have an ancestor A-node, a parent B-node, and a child D-node. Any C-node that does not satisfy all three conditions will be false relevant to this query. Current structural join algorithms check only the explicitly expressed conditions, and that is why they are unable to identify those false relevant nodes. If all three conditions were checked, only C-nodes 3 and 7 would be retrieved during the join. An analysis of other structural joins in this query will lead to a similar conclusion. This motivates us to design our method.

In order to check all conditions for each query node, we need to know first, the types of parents, children, ancestors, etc. that each data node has, and second, the types of parents, children, ancestors, etc. that each query node requires their matching data nodes to have. In the next Section, we define the notion of social classes to represent such information.

### **3 Social Classes**

The main idea behind the social class is to partition data nodes into equivalence classes based on types of their parents, children, ancestors, etc., so that nodes with the same types of parents are in one class for the parent relation, nodes with the same types of children are in one class for the child relation, and so on. In the following, we present the details, first for data nodes, and then for query nodes.

### 3.1 Social Classes of Data Nodes

Although many structural relations among data nodes can be defined, in this paper we consider only six structural relations: parent, child, ancestor, descendant, following-sibling and preceding-sibling. However, our framework can be easily extended to include other structural relations.

Let  $R$  be a structural relation and  $v$  be a data node. The set of *relatives of  $v$  in  $R$*  is defined by  $\mathcal{N}_R(v) = \{v' \mid \langle v, v' \rangle \in R\}$ . The *relative tags of  $v$  in  $R$*  is the set of tags of the relatives of  $v$  in  $R$ , and is defined by  $\mathcal{T}_R(v) = \{tag(v') \mid v' \in \mathcal{N}_R(v)\}$ .

**Definition 3.1** Two data nodes  $v_1$  and  $v_2$  are *similarly associated* in a structural relation  $R$ , denoted by  $v_1 \overset{a}{\sim}_R v_2$ , if they have the same set of relative tags, that is,  $\mathcal{T}_R(v_1) = \mathcal{T}_R(v_2)$ . The relation  $\overset{a}{\sim}_R$  is called the *association similarity* induced by  $R$ .

It is straightforward to show that the association similarity is an equivalence relation on the set of data nodes. Thus, it induces a partition  $\mathcal{P}_{\mathcal{T}_R} = \{C_1, \dots, C_k\}$  of data nodes, where  $\bigcup_{i=1}^k C_i = V$ ,  $C_i \cap C_j = \emptyset$  for any  $i \neq j$ , and each  $C_i$  is an equivalence class, called a social class, as defined in the following definition.

**Definition 3.2** A *social class* induced by a structural relation  $R$  is the set  $C$  of data nodes, such that, two data nodes  $v_1 \in C$  and  $v_2 \in C$  if and only if  $v_1 \overset{a}{\sim}_R v_2$ .

Each social class  $C$  is identified by a numeric identity  $id(C)$  and has a set of relative tags  $\mathcal{T}_R(C)$ . For each structural relation  $R$ , each data node  $v$  is assigned to a social class  $C_R(v)$  which has the identity  $id(C_R(v))$  and relative tags  $\mathcal{T}_R(C_R(v))$ . Obviously,  $\mathcal{T}_R(C_R(v)) = \mathcal{T}_R(v)$ . Figure 3 shows a class mapping table and identities of social classes of some nodes of the data tree in Figure 1. The class mapping table keeps track of relative tags of social classes induced by the six structural relations. In this table, each cell represents a set of relative tags of a social class, where a blank cell represents an empty set, and a cell marked by an “-” indicates that the social class does not exist. According to Table (b) in Figure 3, node 2 is in social class  $C_5$  induced by ancestor relation, and in social class  $C_4$  induced by descendant relation. Then, from the class mapping table, node 2 has an ancestor A-node, and descendant nodes with tags C, D, and E. The following lemma directly follows from previous definitions.

**Lemma 3.1** For a given data tree and a given structural relation  $R$ , the set of social classes induced by  $R$  forms a unique partition on data nodes, where each social class has a unique set of relative tags. The total

Relation	$C_0$	$C_1$	$C_2$	$C_3$	$C_4$	$C_5$	$C_6$
ancestor		A,B,C	A,B	A,C,F	A,F	A	-
descendant		D	D,E	E	C,D,E	B,C,D,E,F	-
parent		C	B	F	A	-	-
child		D	D,E	E	C	C,E	B,F
flw-sibling		D	C	E	D,E	B,F	F
pre-sibling		E	C	D	B	-	-

(a) Class Mapping Table

Node	par	chi	anc	des	psb	fsb
1	0	6	0	6	0	0
2	4	4	5	4	0	5
3	2	1	2	1	0	2
4	1	0	1	0	0	0
7	2	2	2	2	2	0
8	1	0	1	0	0	1
		...	...			

(b) Social Class Identities of Nodes

Figure 3: Social Classes of Data Nodes

number of social classes induced by  $R$  is upper bounded by  $2^{|L|}$ , where  $L$  is the set of node labels.

### 3.2 Required Social Classes of Query Nodes

As mentioned in Section 2.1, axes of navigation steps in a query explicitly specify required structural relations between pairs of adjacent query nodes. These explicit structural relations may imply additional, implicitly required structural relations. In each embedding of the query to the data, each query node is mapped to a data node and all explicitly and implicitly required structural relations between any pair of query nodes are satisfied by the corresponding data nodes in the embedding. For this reason, required structural relations among query nodes provide critical information for determining if a data node is false relevant. Required structural relations define relative tags for query nodes in the same way structural relations do for data nodes. Since data nodes are partitioned based on their relative tags in various structural relations, we can identify social classes whose sets of relative tags contain the required relative tags of a query node. These social classes are called required social classes of the query node. Every data node truly relevant to a query must be contained in some required social class of some query node. In the following, we formalize these ideas.

Let  $Q = (V_Q, E_Q, L)$  be a path query,  $q$  be a query node and  $R$  be a required structural relation. The relative nodes of  $q$  in  $R$  is a set of query nodes defined by  $\mathcal{QN}_R(q) = \{q' \mid \langle q, q' \rangle \in R\}$  and the required relative tags of  $q$  in  $R$  is the set of tags of relatives of  $q$  in  $R$  and is defined by  $\mathcal{TR}(q) = \{tag(q') \mid q' \in \mathcal{QN}_R(q)\}$ . Note that to simplify our presentation, we have overloaded some of our notations.

**Definition 3.3** A social class  $C$  induced by a structural relation  $R$  is a **required social class** of a query node  $q$ , if the set of its relative tags contain the required relative tags of  $q$  in  $R$ , that is,  $\mathcal{TR}(q) \subseteq \mathcal{TR}(C)$ . The set of identities of required social classes of  $q$  with respect to  $R$  is defined by  $\mathcal{SC}_R(q) = \{id(C) \mid \mathcal{TR}(q) \subseteq \mathcal{TR}(C)\}$ .

$$\mathcal{T}_R(C) \wedge C \in \mathcal{P}_{\mathcal{T}_R}\}.$$

**Lemma 3.2** For any query node  $q$  and required structural relation  $R$ ,  $|\mathcal{SC}_R(q)| = 0$  if and only if the query result is empty, and  $|\mathcal{SC}_R(q)| = |\mathcal{P}_{\mathcal{T}_R}|$  if and only if  $q$  has no relative in  $R$ , where  $|X|$  is the number of elements in set  $X$ .

The following theorem provides a necessary condition for data nodes to satisfy query nodes.

**Theorem 3.3** If a data node  $v$  satisfies a query node  $q$ , then for each structural relation  $R$ ,  $id(C_R(v)) \in \mathcal{SC}_R(q)$ .

Thus, if a data node is not in any required social class of any query node, it is guaranteed to be irrelevant to the query. On the other hand, although all nodes that are relevant to the query are in some required social classes, not every node in a required social class is relevant to the query. The importance of Theorem 3.3 is that during the query evaluation, we can safely ignore nodes that are not in any required social class.

Definition 3.3 is based on a given set of required structural relations over query nodes. A query explicitly specifies only a minimum set of required structural relations. However these explicit relations may imply many more implicit required structural relations. The derivation of implicit required structural relations can be formulated as a logic derivation problem in which explicitly specified structural relations form a set of ground facts, and implicit required structural relations are derived using a set of derivation rules based on a fixpoint semantics. A detailed description of techniques for logic derivation is beyond the scope of this paper. Here, we briefly address issues about the derivation rules.

A required structural relation can be represented as a predicate. For example, the predicate  $\text{parent}(X, Y)$  represents a required relation that  $X$  has a parent  $Y$ , where  $X$  and  $Y$  are variables to be bound to query nodes. A derivation rule is in the form of consequence:-antecedent. For example,  $\text{ancestor}(X, Y):-\text{parent}(X, Z), \text{ancestor}(Z, Y)$  is a derivation rule stating that if  $X$  has a parent  $Z$  and  $Z$  has an ancestor  $Y$ , then  $X$  also has  $Y$  as its ancestor. Rules such as  $\text{child}(Y, X):-\text{parent}(X, Y)$  and  $\text{descendant}(X, Y):-\text{child}(X, Y)$  are trivial and their meanings are straightforward. An interesting non-trivial derivation rule is  $\text{ancestor}(X, Y):-\text{ancestor}(Z, Y), \text{parent}(Z, X), \text{tag}(Y) \neq \text{tag}(X)$ , which says that if  $Y$  is an ancestor of  $Z$ ,  $X$  is a parent of  $Z$ , and in the query  $X$  and  $Y$  have different tags, then  $Y$  is an ancestor of  $X$ . The interesting point of this rule is the condition  $\text{tag}(Y) \neq \text{tag}(X)$ , which guarantees that  $X$  and  $Y$  will be mapped to different data nodes in any embedding of the query. This rule is correct since in a pattern tree, each query node can have at most one

parent	child	ancestor	descendant
<2,1>	<1,2>	<2,1>	<1,2>
<3,2>	<2,3>	<3,1>	<1,3>
<4,3>	<3,4>	<4,1>	<1,4>
		<3,2>	<2,3>
		<4,2>	<2,4>
		<4,3>	<3,4>

(a) Derived Structural Relations

query node $q$	required relative tags						required social classes					
	par	chi	anc	desc	psib	fsib	par	chi	anc	desc	psib	fsib
1		B		B,C,D			all	6	all	5	all	all
2	A	C	A	C,D			4	4,5	1,2,3,4,5	4,5	all	all
3	B	D	A,B	D			2	1,2	1,2	1,2,4,5	all	all
4	C		A,B,C				1	all	1	all	all	all

(b) Required Social Classes

Figure 4: Required social classes

parent. However, if  $tag(X) = tag(Y)$  in the query, we can not say that Y is required to be an ancestor of X because that an embedding of the query may map both X and Y to the same data node. As another example, given  $ancestor(Z, Y)$ ,  $ancestor(Z, X)$ , and  $tag(Y) \neq tag(X)$ , we can at best conclude that either  $ancestor(X, Y)$  or  $ancestor(Y, X)$  is a required structural relation, but not both. This illustrates an interesting point, namely, in certain situations, although one of several conflicting required structural relations must hold, we can not precisely determine which one does. In our method, we do not use such uncertain derivation rules. Although by doing so, we may miss some implicit required structural relations, the query performance can still be greatly improved in practice. Figure 4 shows a set of required structural relations derived from query “/A/B/C/D”, and the required social classes of each query node based on the class mapping table in Figure 3. The blank entries in table (b) indicate empty sets of relative tags, and “all” indicates that all social classes induced by the corresponding structural relation are required by the given query node.

## 4 Evaluating Path Queries

We now present our query evaluation framework and several algorithms that implement this framework.

Algorithm: Compute Social Classes

Input: An XML data tree  $T$

Output: A list  $L$  of  $\langle n, C_{R_1}, \dots, C_{R_\delta} \rangle$ , where  $n$  is a data node and  $C_{R_i}$  is the social class of  $n$  in structural relation  $R_i$ ; A class mapping table  $M$

Method:

Stack  $S = \emptyset$ ;  $L = \emptyset$ ;  $M = \emptyset$ ;

ProcessNode(root( $T$ ),  $L$ ,  $M$ );

$r = \text{pop}(S)$ ;

append to  $L$  tuples created from nodes in childList of  $r$

append to  $L$  the tuple created from  $r$

return  $L$  and  $M$

ProcessNode( $n$ ,  $L$ ,  $M$ )

obtain tag sets for parent, ancestor, preceding-sibling of  $n$  from top( $S$ )

push( $n$ ,  $S$ )

For each child  $c$  of  $n$  do begin

ProcessNode( $c$ ,  $L$ ,  $M$ )

$n = \text{pop}(S)$

for each node  $c$  in childList of top( $S$ ) do

add tag( $n$ ) to tag set for following-sibling of  $c$

add tag set for descendant of  $n$  to that of top( $S$ )

add tag( $n$ ) to tag set for child of top( $S$ )

add to  $L$  output tuples generated from children of  $n$  using  $M$

append  $n$  to childList of top( $S$ )

end;

Figure 5: Compute Social Classes

## 4.1 The Evaluation Framework

In our framework, social classes of data nodes are obtained during data loading when XML documents are shredded into data nodes; the required social classes of query nodes are obtained during query compilation; the social classes and required social classes are used during query evaluation to filter data nodes or to access data nodes through indexes.

During data loading, the XML document is parsed to identify data nodes and their social classes induced by structural relations. The class mapping table is created while data nodes are assigned the identities of their social classes. We do not assume any specific storage scheme. All we require is that the identities of social classes of data nodes can be directly obtained from the nodes or from an index structure during query evaluation.

During query compilation, a set of explicitly required structural relations are obtained from a given path query and used to derive implicit required structural relations. These required structural relations are used to obtain required relative tags for each individual query node, which are then used to obtain the identities of

Algorithm : Compute Required Social Classes  
Input: A path query  $Q$ , the class mapping table  $M$   
Output: A list of  $\langle q, R, L \rangle$ , where  $q$  is a query node,  $R$  is a structural relation, and  $L$  is a list of identities of required social classes for  $q$   
Method:  
Result-list= $\emptyset$   
Work-list= required structural relations explicitly stated in  $Q$ .  
for each  $w$  in Work-list do  
  for each  $r$  in Result-list do  
    if  $w$  and  $r$  derive a new instance  $t$  based on derivation rules  
      append  $t$  to Work-list  
  remove  $w$  from Work-list and append it to Result-list  
for each query node  $q$  and each structural relation  $R$   
  find the set  $s$  of relative tags of  $q$  in  $R$  from Result-list  
  find the set  $L$  of IDs of social classes in  $R$  from  $M$  using  $s$   
  output  $\langle q, R, L \rangle$

Figure 6: Compute Required Social Classes

required social classes for query nodes using the class mapping table. The class mapping table is extremely small in practice (less than 100KB for each dataset tested) and is kept in memory for better performance.

During query evaluation, the identities of required social classes are used to filter data based on the identities of social classes of data nodes or to skip irrelevant data using an index that uses both node positions and identities of social classes. The filtering method works by comparing the identities of social classes of input data nodes with those of required social classes for the corresponding input stream, and preventing false relevant data nodes from generating any join result, thus reducing evaluation cost for subsequent joins. The indexing method works by using the identities of required social classes to never retrieve input data nodes that do not belong to any of the required social classes, resulting in reduced input as well as output for the current join.

## 4.2 Compute Social Classes for Data Nodes

Figure 5 shows an algorithm that assigns social classes to data nodes. This algorithm takes the data tree as the input and generates a list of tuples as well as the class mapping table. Each tuple contains the identity of a data node and the identities of social classes of the node (one for each structural relation).

The algorithm performs a depth-first traversal of the data tree. For each node encountered, the algorithm builds relative tags of the nodes for each structural relation and keeps an ordered list of child nodes for the node. Initially, the sets of relative tags and the children list are empty. As the data tree is traversed, the tag

sets will be filled with relative tags of the data node, and the children list will have child nodes appended to it. Once the subtree rooted at a node is completely traversed, the sets of relative tags of each child of the node will be used to find the identities of social classes of that child node.

When a data node  $n$  is visited, the top of the stack contains the parent  $p$  of  $n$  and the children list of  $p$  contains preceding-siblings of  $n$ . Thus relative tags of  $n$  in ancestor, parent and preceding-sibling relations can be directly obtained from  $p$ . Relative tags of  $n$  in child and in descendant relations will stay incomplete until all descendants of  $n$  are visited. So,  $n$  is pushed onto the stack at this point and the algorithm visits the children of  $n$  recursively. When the recursive calls finish, relative tags of  $n$  in child and descendant relations are complete. Node  $n$  is popped out of the stack and its children are used to generate output tuples. However, relative tags of  $n$  in the following-sibling relation will stay incomplete until all of siblings of  $n$  are visited. Thus,  $n$  is placed in the children list of its parent which is once again on the top of the stack. At the same time, relative tags of  $n$  in child and in descendant are used to update relative tags of the parent of  $n$  and preceding-siblings of  $n$ .

When generating an output tuple from a node, the relative tags of the node in each structural relation are used to obtain the identity of the social class in that structural relation from the class mapping table. New identities are created for previously unseen sets of relative tags and the class mapping table is updated as well.

### 4.3 Derive Required Social Classes From Queries

During query compilation, the set of explicitly specified required structural relations is obtained from the set of edges of the query tree. As mentioned in Section 3.2, general logic derivation methods can be used to derive implicit required structural relations. However, a simple method is often sufficient for discovering enough implicit structural relations to ensure substantial performance gains in query evaluation. Figure 6 shows an algorithm of such a simple method.

The algorithm keeps two lists of instances of required structural relations, where each instance of a structural relation  $R$  is of the form  $R(q_i, q_j)$ , where  $q_i$  and  $q_j$  are query nodes. One list, the Work-list, contains instances to be processed, and the other list, the Result-list, contains instances that have been processed. We use a set of derivation rules of the form  $R_1(X, Y) : -R_2(U, V)$  or  $R_1(X, Y) : -R_2(U, V), R_3(W, Z), \Theta$ , where  $R_i$  is a required structural relation,  $U, V, W, X, Y, Z$  are variables to be instantiated by query nodes, and  $\Theta$  is a comparison such as  $tag(W) \neq tag(U)$ . Initially, the Work-list contains all explicit required

structural relations, and the Result-list is empty. The algorithm iteratively goes through the Work-list. For each instance  $w$  in Work-list, every instance  $r$  in Result-list is considered to see if the two instances can derive any new instance using any single derivation rule. If a new instance can be derived, it is appended to the Work-list. After every instance in Result-list is considered for  $w$ ,  $w$  is removed from the Work-list and appended to the Result-list. The algorithm then continues with the next instance in Work-list. The algorithm terminates when Work-list becomes empty.

Once the derivation process is completed, the instances in Result-list are grouped by query node and structural relations. Each group results in a set of relative tags of the query node in the structural relation. Using the class mapping table, we can then find identities of all required social classes for the query node.

#### **4.4 Filtering vs. Indexing**

There are several ways to extend node position based techniques to use social classes. In this section, we consider filtering and indexing.

The filtering method is a simple extension to the structural join algorithms (with or without using indexes). After a data node is retrieved from an input stream (maybe through an index access) and before it is used to produce a join result, the identities of its social classes are compared to those of required social classes of the query node corresponding to the input stream. If for every structural relation, the identity of the social class of the node is among the identities of required social classes in the given structural relation, then the node is used as usual in the join. Otherwise, the join proceeds without using the data node. Since the node is retrieved before filtering is performed, the filtering step does not improve the performance of the current join. However, since the output of the current join could be reduced by the filtering step, the performance of subsequent joins can benefit from the filtering. The effect of applying filtering early in a sequence of structural joins can quickly multiply to result in a substantial performance gain.

The indexing method is more complex. The idea is to index data nodes using a combination of node positions and identities of social classes. One possible way is to encode both the position and social classes of a node into a single key, and use this key to build an index. Another way is to have separate keys for node position and social classes and to build a multiple key index. Either way, data nodes must be accessed sequentially according to their positions in order for the structural join algorithms to work appropriately.

One complication is that when using such an index, the search key will contain one start position (with or without the matching end position) and a set of identities of required social classes. Take a B+-tree index

Data Set	Raw Size	Nodes Table	Classes Table	Values Table
Shakespeare	7.28MB	3.46MB	3KB	7.58MB
XMark	111.1MB	46.88MB	97KB	111.59MB
DBLP	164.35MB	32.50MB	32.4KB	52.18MB

Table 1: Sizes of Data Sets

Query Id	Query Expression
Q1	/Play/Act/Prologue
Q2	/Play/Act/Scene/Speech/StageDir
Q3	/Play/Act/Scene/Speech/Subhead
Q4	/Play/Act/Scene/Speech[/Subhead]/Line
Q5	/Play[Act/Scene/preceding-sibling::Prologue]//Line
Q6	/Site/Regions/Europe/Item/Description/Text/Keyword
Q7	/Site/Regions/Europe/Item[/Featured]/Description/Text/Keyword
Q8	//Item[/Description[/Keyword][//Emph][//Bold]]
Q9	//Person[/Phone][//CreditCard][//Address]/Name
Q10	//People//Person[/Homepage]/EmailAddress

Table 2: A Set of Queries

as an example. A subtree can be skipped, if either none of the nodes in the subtree has the right position (that is being contained in the range of the search key) or none of the nodes in the subtree belongs to any required social classes. Both the structure and algorithms of the index will have to be more complex. The design of such indexes is still an open problem and will be addressed in our future research. In the next Section, we report results from our simulations which illustrate potential performance gains of using our evaluation framework.

## 5 Experiment Results

The purpose of our experiments is to study the effectiveness of combining social classes with node positions to reduce the cost of path query evaluation.

We performed a number of experiments using three XML datasets: the XMark XML benchmark [SWK<sup>+</sup>02], the Shakespeare’s plays dataset [Bos], and the DBLP bibliography dataset [Ley]. Due to space limitations, we only present results from the Shakespeare plays and XMark datasets. For each dataset, we obtain node positions and social classes for data nodes during data loading. The datasets are shredded and stored in a commercial RDBMS in several tables including a Nodes table and a Values table. Node positions and iden-

ties of social classes are stored in the Nodes table. We implemented a variant of the algorithm presented in Section 4.2 in Java. This algorithm uses an SAX parser to parse XML documents. During the parsing, it assigns social class identities to data nodes as well as obtains node positions. It also obtains the class mapping table. Table 1 shows the sizes of various tables for the datasets. The Nodes table contains one record for each internal node of the data tree. Each node contains both the node position and identities of six social classes, one per structural relation. The storage size of the identities of social classes amounts to approximately one half of the node size. The Values table contains string values of the leaf nodes. The class mapping table contains the relative tags of each social class for each structural relation.

For each dataset, we choose five types of queries. In order to focus on performance of structural joins, we exclude queries with value-based conditions. Table 2 shows the set of queries used in our experiments. Among these queries, Q1 to Q5 are for the Shakespeare plays dataset and Q6 to Q10 are for the XMark dataset. Q1, Q2, Q3, and Q6 are simple path queries with different lengths and involve only the parent-child relationship. Q4 and Q7 are simple branching path queries that involve both parent-child relationships and a single step secondary branch. Q8, Q9 and Q10 are non-trivial branching path queries involving both parent-child and ancestor-descendant relationships and multiple secondary branches. Q5 is also non-trivial and it involves parent-child, ancestor-descendant, and sibling relationships and a secondary branch with multiple steps.

We implemented the algorithm presented in Section 4.3 to compute the required social classes of each query node. The set of derivation rules used by the algorithm was manually created. The derivation rules have the form described in Section 4.3. Since the class mapping table is small enough to be kept in the memory, the required social classes can be obtained with a negligible cost as compared to the cost of actual query evaluation.

For each query, we generate evaluation plans in the form of left-deep join trees with all possible join orders that allow for the output of each join to be pipelined into the next join. We consider five evaluation strategies of join trees: Naive, B+-tree, XR-tree, Class-filter, and Class-index. All evaluation strategies use some variation of the StackTree structural join algorithm to ensure that each node accessed in a join need only be accessed once. The naive evaluation uses the original structural join algorithm for all joins. This algorithm reads every node in each input stream. The B+-tree evaluation uses the position-based B+-tree index for each join to skip over descendant (or child) nodes that are irrelevant to the join. The XR-tree evaluation uses the position-based XR-tree index to skip irrelevant ancestor and descendant nodes. As done

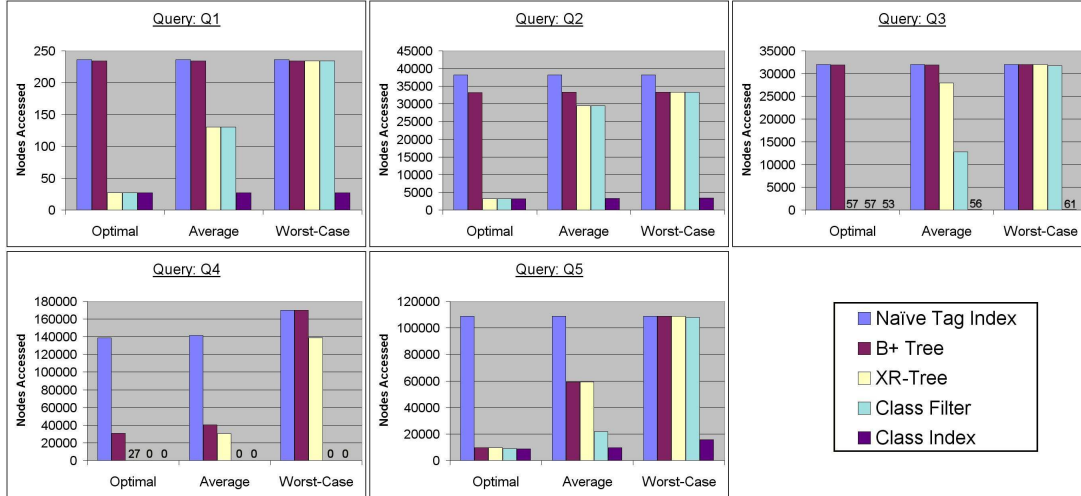


Figure 7: Results on the Shakespeare Plays Data Set

in the XR-tree evaluation, the Class-filter evaluation accesses input nodes using an XR-tree index, and adds a filtering step as explained in Section 4.4. The Class-index evaluation incorporates social classes into the position-based XR-tree index in order to skip over more irrelevant nodes than possible using XR-tree evaluation alone. In our experiments, these evaluation strategies are simulated using SQL queries in the RDBMS.

Our cost metric for comparing evaluation strategies is the number of nodes retrieved during the evaluation of the join trees. Measuring the performance using the number of nodes retrieved is reasonable because the complexity of the original structural join algorithm is  $O(inputSize + outputSize)$ , where the input size is the number of nodes in the input streams and the output size is the number of pairs of nodes in the output. In our case, since the output of each join (except the last one) is pipelined into the next join and the output of the last join is the same for all evaluation strategies and all join trees, the relative costs of various evaluation strategies on various join trees can be clearly indicated by the number of input nodes retrieved. Using this cost measure for each query, we identify the optimal and worst-case join orders and their costs, as well as the average cost of all join orders. For each query, the optimal and worst-case join orders are likely to be different for different evaluation strategies.

Figures 7 and 8 show the results of our experiments. For each query, the chart shows the number of input nodes retrieved for the optimal join tree as well as for the worst-case join tree, and the average number of nodes retrieved taking all possible join trees into consideration.

Let us look more closely at the results. Consider Q1 in Figure 7. B+-tree evaluation is not much different than the naive evaluation for all join orders. This is because in this dataset, every play contains acts, but there are only 12 prologues in the entire dataset. For B+-tree evaluation, no matter what join order is chosen, every node in the child input stream participates in the join. Thus it is not effective in skipping over input nodes. For XR-tree evaluation, since the optimal join order is to do “Act/Prologue” before “Play/Act”, many irrelevant parent input nodes can be skipped, resulting in a much better performance. However, with the worst-case join order which is to do “Play/Act” before “Act/Prologue”, XR-tree evaluation cannot skip over any input node in the first join because every node is relevant to the join. Since the result of the first join is pipelined into the second join and no index is available for this join result, XR-tree evaluation also has very poor performance in this worst case. The reason for Class-filter evaluation performing so poorly in the worst-case join order is because this query has only two joins and the filtering technique is not capable of improving performance for queries that consist of only one or two containment joins. The filtering technique prevents false relevant nodes from ever generating join results in order to prevent irrelevant nodes from being processed in subsequent joins. But if only one join remains to be performed after the filtering, the XR-tree index is already capable of skipping these same irrelevant nodes even without the initial filtering. Thus no improvement is gained by the filtering. However, the Class-index technique is capable of improving performance in this case since it can skip false relevant nodes from the inputs of the first join, in preparation for the second join of this query.

Q2 and Q3 show similar properties as Q1. Q4 is particularly interesting since it has an empty answer. Evaluating this query without the use of social classes involves performing several joins only to find that no answer nodes satisfy this query. Both Class-filter and Class-index evaluations can detect if any query node has an empty set of required social classes, according to Lemma 3.2, the existence of such a query node clearly indicates that no data node can ever satisfy this query and the answer should be empty. Therefore, the query evaluation can terminate without accessing a single node. B+-tree and XR-tree evaluations will obtain empty intermediate results and terminate early if the optimal join order is chosen, but will perform several unnecessary joins if the worst-case join order is chosen.

For Q5, all methods are able to do very well in the optimal case because starting with the side branch will result in a small number of play nodes, and the position-based index can be used to skip over many irrelevant Line nodes. However, in the worst-case, the join for “Play//Line” will be the first to perform and except in the case of the Class-index, all evaluation strategies have to retrieve all nodes in both input streams



Figure 8: Results on XMark Data Set

of this very expensive join. This explains the tremendous cost differences in this query.

Q6 to Q10 show similar results. However, Class-filter and Class-index evaluations are much more effective in improving query performance since these queries require more joins. In general, the longer the query path is, the more opportunity there is for social classes to be used to reduce the evaluation cost.

From examining the results of these queries, we can see that there is a considerable amount of room for improvement over using the Naive evaluation. B+-tree and XR-tree indexes significantly reduce the number of nodes accessed. But the addition of social class based filtering and indexing techniques to the already efficient XR-tree index, provides for even fewer irrelevant nodes to be accessed. One other interesting observation is that for the Class-index, the performance difference between the optimal and worst-case join orders is often negligible. This is important because determining an optimal join order is a computationally difficult problem and with this evaluation strategy, poorly chosen join orders are much less detrimental to the performance of query evaluation.

To summarize, for the five evaluation strategies, Naive, B+-tree, XR-tree, Class-filter, and Class-index, each strategy is at least as effective in eliminating irrelevant input nodes as the previous strategy, respectively. As we can see from these results, the use of the social classes can often make a substantial improvement.

## 6 Related Work

The containment query was first considered in [ZND<sup>+</sup>01], where the position is used to define containment queries as SQL queries which perform joins on node tables. A new merge-join algorithm was proposed to replace standard join algorithms for efficiently processing containment queries. The StackTree family of algorithms for structural joins was proposed in [AKJK<sup>+</sup>02], as well as the term “structural join.” The complexity of these algorithms was shown to be linear to the sizes of the input and output. Improvements over the StackTree algorithms were proposed with B+-tree indexing on descendant input streams, and with XR-tree indexing on both input streams [AKJK<sup>+</sup>02, BKS02, ZND<sup>+</sup>01]. A method for join order selection of query plans involving a tree of structural joins is presented in [WPJ03], where all joins are evaluated with some StackTree algorithm, and the sorting operator is also included in the query plan. Twig queries are considered in [BKS02], where ideas similar to that of StackTree algorithms were proposed to evaluate multiple structural joins simultaneously. The algorithm utilizes multiple linked stacks to compactly represent multiple sequences of parent-child and ancestor-descendant relationships. However, the method is most effective for simple twig queries involving only parent-child relationships.

Modeling XML queries as query tree patterns was proposed in [AYCLS01, CFF<sup>+</sup>02, Ram02]. Techniques were also proposed to rewrite query trees based on known structural constraints in the data.

Using node partitions to evaluate path queries is not new. Graph index approaches all use some type of similarity relation to partition data nodes and build graph indexes based on relationships among equivalence classes. In [MS99], 1-indexes are built based on notions of simulation and bisimulation both of which induce a partition on data nodes. Nodes are in the same equivalence class if and only if they have the same incoming path from root and/or the same outgoing path to leaf nodes. Due to the strong condition for nodes to be equivalent, the sizes of these indexes can be very big, sometimes even bigger than the data graph. Several techniques were proposed to reduce the index size. A(k)-indexes [KSBG02] partition data nodes based on the notion of k-bisimilarity which places nodes in the same equivalence class if and only if they have the same incoming and/or outgoing paths of a given length (k). In [KBNK02], the process of building the 1-index is carried out as a sequence of interleaving steps following the direction of the outgoing paths (F) and incoming paths (B), and is controlled by the length of this sequence. Unlike these methods, association similarity represents a much more relaxed condition. Nodes having the same set of relative tags need not to have same incoming or outgoing paths. However, since we do not have path information encoded in the

node partition, structural joins are necessary for discovering paths.

## 7 Conclusions

In this paper, we define a notion of social classes of XML data nodes and present a query evaluation framework that uses both node positions and social classes to improve performance of path query evaluation. A social class of a node is defined as an equivalence class induced by tags of other nodes that are associated with the given node in the given structural relation. In our framework, social classes of data nodes are obtained during data loading. During query compilation, queries are analyzed. Implicit required structural relations are derived from those explicitly specified in the query, and identities of required social classes of individual query nodes are obtained from the class mapping table. The positions of data nodes, the social classes of data nodes, and the required social classes of query nodes are used during query evaluation to provide an effective mechanism for filtering and indexing XML data. We present a number of algorithms that implement this framework and report on results from our experiments. Our results show that this new method can substantially improve performance of XML queries that require multiple structural joins.

A number of issues remain to be further studied in order to successfully incorporate our query evaluation framework with existing XML query processing techniques. First, structures and algorithms for indexes that combine node position and social classes need to be designed. Second, efficient methods for maintaining social classes in the event of data updates need to be developed. It is also possible to collect statistics about social classes and use these statistics in query optimization to aid in deriving the most efficient query evaluation plans. Another issue is to use both node position and social classes to evaluate more complex queries that may contain multiple path queries, value-base joins, and aggregate functions.

## References

- [AKJK<sup>+</sup>02] Shurug Al-Khalifa, H. V. Jagadish, Nick Koudas, Jignesh M. Patel, Divesh Srivastava, and Yuqing Wu. Structural joins: A primitive for efficient XML query pattern matching. In *Proceedings of the 18th Proceedings of IEEE International Conference on Data Engineering*, 2002.

- [AYCLS01] S. Amer-Yahia, SR. Cho, L. V. S. Lakshmanan, and D. Srivastava. Minimization of tree pattern queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 497–508, 2001.
- [BBC<sup>+</sup>02] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernandez, Michael Kay, Jonathan Robie, and Jérôme Siméon. XML path language (XPath) 2.0. W3C Working Draft, <http://www.w3.org/TR/xpath20/>, Nov. 2002.
- [BCF<sup>+</sup>02] S. Boag, D. Chamberlin, M. Fernandez, D. Floarescu, J. Robie, J. Simeon, and M. Stefanescu. XQuery 1.0: An XML query language. In W3C Working Draft. Available from <http://www.w3.org/TR/xquery>, Nov. 2002.
- [BKS02] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: Optimal XML pattern matching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2002.
- [Bos] J. Bosak. Sample XML documents, shakespeare-1.10.xml.zip. <ftp://sunsite.unc.edu/pub/sun-info/standards/xml/eg>.
- [BPSMM00] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. Extensible markup language (XML) 1.0 (second edition). W3C Recommendation, <http://www.w3.org/TR/REC-xml>, Oct. 2000.
- [CFF<sup>+</sup>02] Chee-Yong Chan, Wenfei Fan, Pascal Felber, Minos Garofalakis, and Rajeev Rastogi. Tree pattern aggregation for scalable XML data dissemination. In *Proceedings of the 28th Proceedings of International Conference on Very Large Data Bases*, 2002.
- [CMS02] Chin-Wan Chung, Jun-Ki Min, and Kyuseok Shim. APEX: An adaptive path index for XML data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2002.
- [CVZ<sup>+</sup>02] Shu-Yao Chien, Zografoula Vagena, Donghui Zhang, Vassilis J. Tsotras, and Carlo Zaniolo. Efficient structural joins on indexed XML documents. In *Proceedings of the 28th Proceedings of International Conference on Very Large Data Bases*, 2002.

- [JLWO03] Haifeng Jiang, Hongjun Lu, Wei Wang, and Beng Chin Ooi. XR-tree: Indexing XML data for efficient structural joins. In *Proceedings of IEEE International Conference on Data Engineering*, 2003.
- [KBNK02] Raghav Kaushik, Philip Bohannon, Jeffrey Naughton, and Henry Korth. Covering indexes for branching path queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2002.
- [KSBG02] Raghav Kaushik, Pradeep Shenoy, Philip Bohannon, and Ehud Gudes. Exploring local similarity for indexing paths in graph-structured data. In *Proceedings of IEEE International Conference on Data Engineering*, 2002.
- [Ley] M. Ley. Dblp bibliography. <http://dblp.uni-trier.de/xml>.
- [MS99] Tova Milo and Dan Suciu. Index structures for path expressions. In *Proceedings of the 7th Proceedings of the International Conference on Database Theory*, 1999.
- [Ram02] Prakash Ramanan. Efficient algorithms for minimizing tree pattern queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2002.
- [SWK<sup>+</sup>02] A. R. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for XML data management. In *Proceedings of the 28th Proceedings of International Conference on Very Large Data Bases*, pages 974–985, 2002. Available from <http://www.xml-benchmark.org>.
- [WJLY03] Wei Wang, Haifeng Jiang, Hongjun Lu, and Jeffrey Xu Yu. Containment join size estimation: Models and methods. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2003.
- [WPJ03] Yuqing Wu, Jignesh M. Patel, and H. V. Jagadish. Structural join order selection for XML query optimization. In *Proceedings of IEEE International Conference on Data Engineering*, 2003.
- [ZND<sup>+</sup>01] Chun Zhang, Jefferey Naughton, David DeWitt, Qiong Luo, and Guy Lohman. On supporting containment queries in relational database management systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2001.