Comprehensive Exam Project
Advisor : Prof. Wesley Chu

Indexing Techniques for XML Structure Patterns

Tony Lee

klmarx@cs.ucla.edu

March 2003

Indexing Techniques for XML Structure Patterns

Tony Lee

Table of Contents.

Abstract

The eXtensible Markup Language (XML) is intended to become a universal format for structured documents and data. The main reason for relaxing XML query is due to the heterogeneous and structural nature of XML data that can make query formation tedious. Users need to know well the content as well as the structure of data to formulate queries which is not easy, especially in the presence of optional data elements. Therefore, relaxing XML queries becomes essential when no exact match is found for a particular query. Moreover, users can explicitly instruct the search engine to return, in addition to exact query matches, similar answers. This paper is intended to presents a study on XML query relaxation using similarity clustering and conceptual hierarchy, and the related indexing solutions.

1. Introduction

This Report investigates on *X-TAH representative*, the problem of indexing fragments of XML data for the purpose of XML Query Relaxation.

The study of *X-TAH representatives* is motivated by the *Type Abstraction Hierarchy* proposed and implemented by the CoBase project at UCLA. X-TAH stands for *XML type Abstraction Hierarchy*, the XML version of the relational TAH. In a conventional database system, queries are answered with absolute certainty. If the database search engine cannot find an exact match to the user query, no result would be returned. Query relaxation is applied when no exact answers to the user query can be found in the database. In query relaxation, a query scope is relaxed to enlarge the search or include additional information. Enlarging and shrinking a query scope can be accomplished by viewing the queried objects at different conceptual levels, since an object representation has wider coverage at a higher level and inversely, more narrow coverage at a lower level. TAH is created to provide multi-level knowledge representation for the data stored in the data source [3].

X-TAH representative refers particularly to the representation of the internal nodes of the indexing tree. Internal representation is of important concern because it enables fast search for closest match to the given query in the data set clustered at the leaf level in X-TAH. TAH can be constructed by clustering close data sets. Distance of two pieces of data is defined by distance function [1,2]. Multiple clustering algorithms have been proposed and implemented [1,2]. If no exact answer to a query is found in the database, the process of query relaxation involves first finding the closest base cluster, which contains actual data from the database, and then possibly further relaxing the query by a series of generalization (traversing up) and specialization (traversing down) in TAH. In other words, in order to relax query, a base cluster, one that is closest in distance to the query, should be found by traversing down the hierarchy. The most naïve way is scan over the entire set of data and find such cluster, which is very costly in the presence of large amount of indexed data objects. X-TAH representative can improve search efficiency by providing clues on which branch to explore at a particular internal node in the search.

2. Related Works

Some other XML query relaxation techniques have been proposed in the past. In [9], the authors proposes to uses weighted tree patterns and query evaluation joins to accomplish query relaxation. The weighted tree pattern methods supports KNN search and threshold. However, it is not clear how these weights to each node/edge can be assigned that would accurate corresponds to the actual domain semantics. In [4], the authors proposes to use some embedding criteria to find approximate subtree patterns to the given query. However no similarity measure was specifically proposed to solve the ranking problem; nor is that any indexing to make the embedding search in runtime within practical limit. On the other hand, tree embedding technique can be complementary to our work in that it can be used to effectively reduce the candidate data set by pruning away subtree patterns that do not make sense semantically or are unlikely to be queried. In both techniques mentioned, the XML database engine needs to be modified significantly to facilitate the relaxation process.

Our work focuses on relaxation based on similarity clustering and conceptual hierarchy, called X-TAH, motivated by the similar approach taken in the Cobase project [3]. The use of similarity clustering takes account of the semantics of the actual data; general distance measure between trees in [6,7,8] and XML specific distance measure proposed in [5] provides important insight into inter-object distance measure crucial to similarity clustering. Furthermore, conceptual hierarchy enables a flexible query relaxation process, for example, if the user is not satisfied by a particular set of answers from previous relaxation, a refined queries can be found by simply traversing the conceptual hierarchy. Moreover, X-TAH is built and runs on top of the existing XML Database engines and thus make use of the existing efficient indexing capabilities.

3. Preliminaries

3.1 Data Model

A XML data repository consists of a collection of XML documents. Each of these documents conforms to a specific schema. XML data can be viewed as a graph or tree. Using IDREF and REF attributes, the same elements can appear at (be pointed to from) many other places in the XML document. If we ignore these two attributes, XML data can be viewed as trees. We further assume a bag/ordered tree model.

XML query matches (which are trees) can lie anywhere in the XML data tree. We call the schema representation of these subtrees, "structure patterns". That is, a structure pattern can have multiple corresponding matches in the actual XML data.

In figure 1, there is a XML schema, three data instances that conform to that schema, and three structure patterns extracted from the schema. T1 matches instance1; T2 matches both instance 2 and 3; T3 does not have actual match in data, despite being

Figure 1: XML data model example

XML Schema

```
<xs:schema>
  <xs:element name="book">
       <xs:element name="title" type="xs:string"/>
       <xs:element name="author" type="xs:string" minOccurs="0" maxOccurs="2"/>
   </xs:element>
</xs:schema>
```

(a) XML Schema

Instance 1

```
<book>
     <title> The Republic
</title>
</book>
```

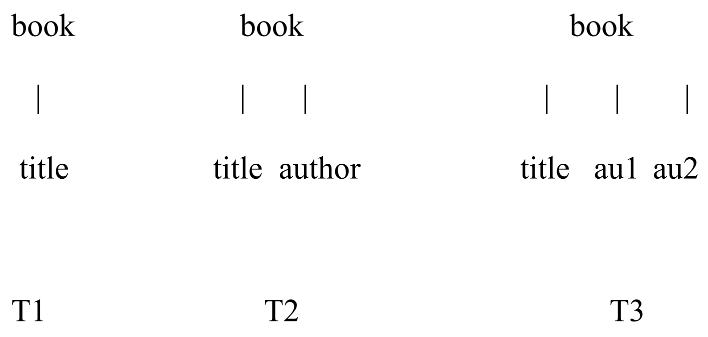
Instance 2

```
<book>
     <title> The Warrior</title>
     <author> Millman
</author>
</book>
```

Instance 3

```
<book>
     <title> Peace</title>
     <author> Jack
</author>
</book>
```

(b) XML data instances



(c) XML Structure patterns extracted from data

a potential structure pattern from the schema. In our indexing solution, we only index structure patterns like T1 and T2 that have actual matches in the data.

In this report, we are only interested in relaxing the structure of XML data, i.e. the difference between instance 2 and 3 are not considered basis for structural relaxation since they have the same structure pattern, T2.

3.2 General Objectives

When users ask a XML query Q, the objective of query relaxation, in our case, is to find some structure pattern Q' (based on the already existing data) that closely match Q. Q' is a potential relaxed query. Q' is then submitted to the XML data repository for exact matches. Since Q' is extracted from the existing data itself, Q' is guaranteed to give the user some results.

Therefore, objective of query relaxation is to find the structure patterns from the data that approximate the original query. These structure patterns make up the set of relaxed queries.

The following questions then arise:

1) how can we extract the structure patterns from the data? How many of them are there?
2) how can we quickly find the structure patterns we want for a particular query?

In the next section, we answer the first question. And the rest of the report addresses the second one.

3.3 Space Complexity

The problem of space complexity is to the exponential number of possible structure patterns that exist in the data. A XML document can be viewed as a "big" tree, within which, there are many sub-trees. Structure patterns are actually the schema(structural) representation of these sub-trees. The number of possible sub-trees grows super-exponentially with the number of tree nodes. Consider a full binary tree,

$T(n)$ is # of sub-trees of an n node tree.

1 level: $T(1) = 1$

2 levels: $T(3) = 4$

3 levels: $T(7) = 25$

4 levels: $T(15) = 676$

The number of possible sub-trees is approximately,

$T^f(n) \approx (1 + T(n/f))^f$, $T^f(n)$ denote the number of sub-trees of n node tree with fanout f.

It is too costly to enumerate all the possible sub-trees. However, we can significantly reduce the space complexity in the following ways,

1) We only relax queries to structure patterns that have the same root node. This requirement is due to the fact that the tree distance functions we found so far in our research all take the assumptions that two trees, of which the distance is calculated, have the same root node. Therefore, if we study the query pattern, i.e. the frequently asked queries, we only consider the structure patterns that have the same root node as these queries. If a large portion of the asked queries have the same root nodes, we considered this class of queries frequent.
2) Typical size of XML queries is 2-6 levels deep and 3-4 fanout. We can limit our consideration to the structure patterns that are of approximately the same size as that of a typical query.
3) Some structural permutations can be pruned because they do not semantically make sense or are unlikely to be queried. [4] suggested an algorithm to prune away irrelevant permutations.

In short, we can always control which node to relax and to what extent in size. Moreover, since indexing can be done offline, we can assume a large amount of storage available. Therefore, with ability to control relaxation extent, and the availability of large storage space, search space complexity can be controlled within practical limit.

4. Problem Defined

Recall in section 3.2, the second question we are trying to address – How can we quickly find the closely matched structure patterns for a particular query?

We therefore define the problem in terms of finding an algorithm that meets the following requirements,

1) Optimality Requirement: find the closest match.

   Given a query q and XML document set S, ST is the set of all possible sub-trees in S, the algorithm should find a set of sub-trees $Q' \subseteq T$ such that

   $\forall t' \in ST-Q', \forall t \in Q', D(t',q) > D(t,q)$

   Relaxed set $R = \{Q' \cup \{q\}\}$

2) Performance Requirement: Find the match fast

   It is not uncommon in our case to have tens of millions of structure patterns, a total scan would not be acceptable in that case. The actual evaluation for the performance requirement would be done in implementation. The bottom line is that the search time should not take longer than what the user can tolerate.

3) Maintainability requirement:

   If a data structure is required for indexing purpose, can the data structure be efficiently maintained in a relatively dynamic database environment. Our general goal is to find an algorithm that enables fast similarity search.

5. Approaches

In a naive way, we can scan all the structure patterns and find the closest match to the query. Using brute force would guarantee that we find the closest match but definitely result in poor performance. Therefore, indexing the structure patterns is required for fast search. Indexing techniques can be generally categorized into two classes, in terms of what is given about the indexed objects (structure patterns in our case).

5.1 Feature Transformation Based Indexing

The first class of indexing techniques requires feature transformation, which transforms important properties of complex objects into high-dimensional vectors (feature vectors)[14,15]. Thus the similarity search corresponds to a search of points in the feature space which are close to a given query point and therefore correspond to a nearest neighbor search[11, 13]. The domain expert use domain knowledge to convert domain objects into feature vectors and provide similarity measures based on those feature vectors. Numerical data is an example of such transformation in which the numerical value represents the location of an object in a single dimension space.

Once objects can be pinpointed in k-dimensional space, spatial clustering techniques can be used to turn proximity search into exact match search. For example, in the one dimensional case, if we can turn objects into one-dimensional points, we can cluster them by ranges [1,2], e.g. ages can be clustered and represented by ranges such as (10-30), (30-50) ...; a search can be conducted to find the range that contain the asked age, once the range is found, it is certain that the closest matched ages must be within the range (assuming that the boundary points {10,30,50} are actual data points). Another example would be Voronoi cells [11]. Two-dimensional space can be partition into some Voronoi cells, each of which contains 1 or 0 data point. Because voronoi cell has such property that if a query point A is inside the cell that contains data point B, B is guaranteed to be the closest point to A. Again, a proximity search becomes an exact-match search. Once the search becomes exact-match, many efficient high-dimensional indexing can be used to index these clusters (R-tree, X-tree, etc)[11,15].

However, in the case of XML Query Relaxation, some important problems prohibit us from using feature transformation for indexing. Most importantly, complex object cannot always be transformed into multidimensional vectors,(represented as multi-dimensional points), because complex similarity distance functions may not be represented by a simple feature vector distance or the objects are too high in their dimensionality as they could be efficiently managed by multi-dimensional index structures. Furthermore, in our case, it is not entirely clear how a tree can be transformed into a vector, i.e. what feature should be selected. Much essential tree structural information may be lost in the transformation. For these reasons, feature transformation is considered too difficult for indexing XML structure patterns.

5.2 Distance Function Based Indexing

If object cannot be transformed into vectors, distance function can be used for indexing, since distance function measures the relative distance among objects. In this way, objects are preserved. The trade-off lies in that, in indexing, we can only have an approximate idea of where the leave objects lie, since only the pair-wise distance between objects are provided.

We can build an index tree based on object distance alone, called X-TAH (XML Type Abstraction Hierarchy), analogous to the relational TAH in the CoBase project [3]. The advantage of an index tree is that based similarity clustering, an index tree can the relaxation process by a sequence of generalization and specialization operations [3]. In the remaining portion of this report, a method of building X-TAH is proposed, assuming that a distance function is given to measure the distance between any two applicable XML structure patterns.

To build X-TAH, first the structure patterns need to be clustered. There are numerous clustering algorithms based on object distance function [1,2]. These are all hierarchical clustering, which would result in a tree after the clustering. The leaves of the tree are indexed objects (or references to them). In our case, the indexed objects are structure patterns. The second step is to assign representation to internal nodes of the index tree.

X-TAH Representative, the internal node representation is a critical component of the index tree because X-TAH cannot guide the relaxation process without first locating the closest matched leaf (cluster). When the user asks a query, we need to traverse the tree top-down to find the closest matching structure pattern. The brute-force approach would conduct a linear search which is too costly. We cannot use hash table since it is not an exact match search. To improve the search efficiency, the internal nodes perform the routing function to direct the search to the leaf cluster where the (close) match can be found.

For general purpose indexing, we can treat the algorithm of initial clustering a black-box. Therefore, the task of efficiently finding closest match must lie in finding the proper representation for the internal node and thus a good routing function.

We then simplify our problem by treating clustering as a black-box of which the output is an index tree with clusters of structure patterns as leaves. And our problem would be to find the representation of internal nodes to perform the routing function.

6. Indexing Solutions

There are generally two types are routing functions, Pruning and Selection. Section 6.1 gives a study on selection routing, and 6.2 presents an indexing scheme that uses pruning routing.

6.1 Selection Routing

Selection routing picks the best choice at each decision point. Selection routing general provides better performance than its pruning counterpart, because pruning routing may result in multiple choices at each decision point. For selection routing, the difficulty mainly lie in finding a representation for internal node that have the following property, assuming that the representation is a tree:

Given a query Q, internal T1 and T2; tree t1 lies in the leaf cluster that is covered by T1, and t2 by T2.

```
for all t1 cover by T1, and all t2 cover by T2
if D(T1, Q) < D(T2, Q)

min(D(t1,Q) < min(D,t2)
```

This property can also be guaranteed if we consider t1 is one of the child node of T1, and t2 of T2. This property guarantees that if each step we pick the node T that is guaranteed to have the closest match in its subtree to Q, and prune away all others, we can be certain the closest match lies in the leaf clusters in the sub-tree rooted at T.

The singular choice constraint in selection routing requires that the internal node of the index tree must somehow provide a summary of all the nodes covered in its subtree; moreover, such summary must fully characterizes the data objects it covers, in the sense that its relation to the query can reveal whether or not the closest matching data object is in its subtree. The theoretical approach close to provide basis for selection routing is feature transformation detailed in section 5. Feature transformation enables k-dimensional "zoom-in" on the desired data object through indexing. However Feature transformation is considered too hard a problem for solving XML relaxation indexing because its rich semantic meaning encoded in its structured cannot be easily represented in vectors.

6.1.1 Representative Objects

One kind of XML summary uses tree instead vector to summarize XML data. Noticeably among various tree summary algorithms is the concept of *representative objects* introduced by Stanford DB group and implemented in the Lore DBMS as "DataGuides" [16,17,18]. The study of *representative objects* is motivated by purposes of schema discovery and path querying of semi-structured data. Despite the difference of the motivations from those that we have in hand, what interests us is the common characteristics of semi-structured data and XML data trees. In figure 2, there are five XML data trees, which can be considered as structure patterns in our case, and semistructured data in the case of RO.

```
b    b     b          b      b
|     |   /  \        |     /   \
d     e   d   e       f    e    f
|                    |          |
g                    n          m


(1)  (2)  (3)        (4)    (5)
```

Figure 2a: XML data trees

And suppose that we decide to cluster subtree 1,2,and 3 together, and 4 and 5 into another group. We can use *Representative Object*s R1 and R2 to represent the two cluster respectively as in figure 2b.

```
   R1              R2
   b                b
  /  \             /  \
d    e            e     f
     |                /   \
     g                n     m
```

Figure 2b: Representative Objects

R1 and R2 are considered the minimal summary for its clustered trees. However, in general there is a n-to-n relationship between data trees and their summaries. In other words, two different cluster of data trees can have the same representative and vice versa. This creates a problem in X-TAH, since X-TAH requires that the summary must be able to differentiate different clusters in search routing.

Another problem with representative object is that in the summary, the sibling relationships are lost. For example, a query that is exactly like R1 will not have a match in the cluster (trees 1,2,3 in figure 2) R1 covers. This deficiency can be improved by adding statistics on the occurrences of nodes and edges in the data. However, even with such modification, the summary cannot provide guarantee for the existence (in selection routing) or non-existence(in pruning routing) of a closest match in its subtree.

6.2 Pruning Routing

In top-down tree search, pruning routing provided by a internal node determines the set of child nodes on which to further conduct the search, by pruning away those that are certain to contain none of the desired answers. The internal node representation must therefore be able to provide information that gives non-existing guarantees of desired answers. Even though there are many existing index tree structures, the difficulty in applying these index structures is that trees are complex object and it is not entirely clear how a cluster of trees can be directly summarized and represented. M-Tree [8] solves this problem by avoiding creating an entirely new representation for summarizing trees. M-Tree is a generic data structure for indexing complex objects based upon object distance function alone. The object distance function must satisfy the triangular inequality property, i.e. given three objects, o1, o2 and o3, and distance function D,

1. $D(o1, o3) <= D(o1,o2) + D(o2, o3)$
2. $D(o1, o3) >= |D(o1,o2) – D(o2,o3)|$
3. $D(o1,o2) = D(o2,o1)$

Property 2 is actually equivalent to property 1, i.e., if property 1 holds, property 2 would hold as well. If structure pattern distance function satisfies property 1 and 3, we can apply the idea of M-tree to index structure patterns.

Structure patterns are trees. tree distance function, as that we studied [5,6,7,8], define tree distance by the cost incurred from a series of edit operations it takes to transform one tree to the other. There might be potentially many different series of operations that can transform one tree to the other, and tree distance is defined to be the shortest "path" in terms of cost. It is not difficult to see that tree distance function satisfies property 1, because, given that o1, o2 and o3 are trees in the previous example, in the worse case, the distance between o1 and o3 can be obtained by transforming o1 to o2 and then o2 to o3. Tree distance function also satisfies property 3 since the edit operations are reversible.

There are several aspects to the M-Tree index scheme: firstly, the algorithms for range and KNN queries; secondly, the algorithms for maintaining the M-Tree data structure, including insertion/deletion, and split policies.

6.2.1 Basic Data Structure of M-tree

M-Tree uses pruning routing, which disqualifies nodes that do not meet certain criteria and thus must not contain results sought. One disadvantage of pruning, as opposed to Selection, is that performance is not guaranteed, at each choice point, there might be multiple choices that qualify; in the worst case, the whole tree is scanned. The performance of M-Tree indexing is largely determined by degree of cohesion of the clusters, that is how close the elements are to each other in the clusters. If at each level, each cluster only contains elements very close to each other, for any query, maximum number of nodes can then be pruned.

M-tree internal node, T, consists of the following: (see figure 3)

a. a tree: selected from one of T's child nodes.

b. $D(T,T_p)$, distance between T and T's parent T_p

c. Covering radius of T,

$D_c = \max(T,T_j)$, for all Tj in the sub-tree rooted at T (covered by T)

if T is just above the leaves, $D_c = \max(T,T_j)$, all T_j in the clusters covered by T

If T is a internal node beyond the second lowest level,

Dc = max ($D(T_j, T) + Dc(T_j)$), all child nodes of T

Both $D(T,T_p)$ and D_c are pre-computed in building/maintaining the index tree.

6.2.2 Using M-tree to Answer Range Queries

Figure 3 illustrates the pruning process to answer range query using M-tree, i.e. finding answers that are within certain distance to the query , M-tree performs the following pruning searching:

Given an internal node T, its parent Tp and any node Tj covered by T. User asks a query Q and wants to find structure patterns within distance of D_q from Q.

$D(Q,T_p)$	: distance between Q and T_p
$D(Q, T)$	: distance between Q and T
$D(T,T_p)$	: distance between T and T_p (pre-computed and stored with T)
$D_c(T)$	: covering radius of T
D_q	: max error range for Q

We can prune away the sub-tree of T, if we can prove that $D(Q,T_j) > D_q$, for all T_j covered by T.

We can disqualify T by examining the following inequality:

$$| D(Q,T_p) - D(T,T_p)| > D_q + D_c(T) \qquad (1)$$

Only $D(Q,T_p)$ needs to be calculated, other values are already known. From triangular inequality we have

$$\mathbf{D(Q,T) >= | D(Q,T_p) – D(T,T_p)|} \qquad \mathbf{(2)}$$

From (1) and (2),

$$\mathbf{D(Q,T) > D_q + D_c(T)} \qquad \mathbf{(3)}$$

From Covering Radius

$$\mathbf{D(Q,T_j) >= D(Q,_T) – D_c(T), \text{ for all } T_j \text{ covered by } T} \qquad \mathbf{(4)}$$

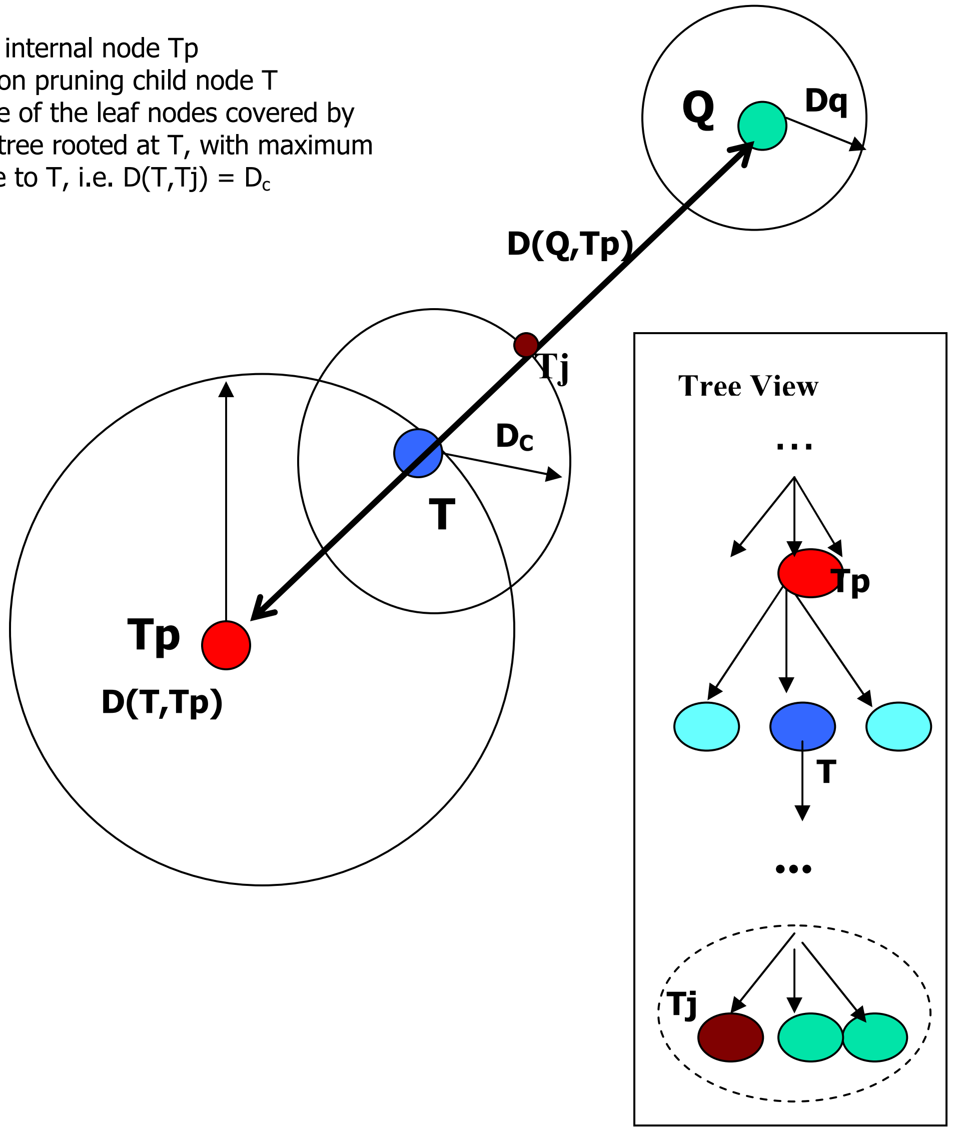
From (3) and (4),

$$\mathbf{D(Q,T_j) > D_q} \qquad \mathbf{(5)}$$

Therefore, just by calculating the distance between Q and T_p, we will be able to prune away the children of Tp whose sub-tree certainly does not contain the answers we seek. The complete algorithm in pseudo-code can be found in *Appendix A*.

Figure 3: Internal pruning in answering range queries

- Explore internal node Tp
- Decide on pruning child node T
- Tj is one of the leaf nodes covered by the subtree rooted at T, with maximum distance to T, i.e. D(T,Tj) = D_c



6.2.3 Using M-tree to answer KNN Queries

To answer K-Nearest-Neighbor queries, The KNN algorithm uses a branch-and-bound technique, with the use of two data structures, PR (a priority queue) and NN (a K-element array) , in addition to the internal node data structure.

Firstly, two types of distances are defined, D_{max} (T) and D_{min} (T),

Let

D_{max} (T) = D(T,Q) + D_c(T), assuming T is an internal node.

D_{max} (T) is the upper-bound distance between query and any node in the subtree covered by T.

Let

D_{min} (T) = D(T,Q) - D_c(T), assuming T is an internal node.

D_{min} (T) is the lower-bound distance between the query and any node in the subtree covered by T.

PR holds the root nodes of the currently active subtrees that might contain the results; PR initially contains the root node. The algorithm recursively pop elements one at a time from PR and perform a search on its subtree.

NN contains the current k-nearest neighbors and will contain the results at the end of the execution. NN is updated at each examination of tree nodes. NN contains two types of values, D(T, Q), if T is the leave, D_{max}(T) if T is an internal node.

If NN is sorted in ascending order, let D_k be the last element of NN and also the largest distance among the current k nearest neighbors, while examining node T', if

D_{min} (T') > D_k ,

Which means that any node covered by T' would not be or contain the KKN answer. So T' and the subtree it covers can be safely pruned away. Considering the following simple example, assuming k = 1 (single closest match),

Radius of O3 and O4: R(O3) = 2. R(O4) = 3.

Distance from O3, O4 to Q: $D(O_3,Q) = 3$, $D(O_4,Q) = 10$

$d_{max}(O_3) = 2 + 3 = 5$

$d_{min}(O_4) = 10 – 3 = 7$

=> $d_{min}(O_4) > d_{max}(O_3)$

* O_4 does not contain the closest match, thus its subtree can be prune away in the search.

The complete algorithm in pseudo-code can be found in Appendix A.

6.2.4 Maintaining M-Tree

One of the advantages of M-Tree is that it can be maintained dynamically, which enables the indexing scheme useful in not only static but dynamic database environment. Similar to many balanced tree data structure, M-Tree maintains itself by splitting and merging internal nodes. The split policy in particular is a major factor in affecting the indexing performance. The split policy includes promotion algorithm and node distribution algorithm. Promotion algorithm determines two new routing objects (internal nodes in place of the old one) when a split occurs, and the node distribution algorithm determines how to distribute the objects in the original cluster to the two new clusters. The ideal split policy should promote two routing objects such that the two new clusters would have minimum covering radius and minimum "overlap" (maximum intra-cluster distance). This goal is consistent with the objective of a good clustering algorithm to produce most coherent clusters – minimize inter-object distance and maximize intra-cluster distance.

7. Implementation

In figure 4 is the general Cobase relaxation architecture. The current XTAH implementation focuses on the XTAH mediator. The X-TAH mediator has two components as shown in figure 5:

- XTAH Manager (online)
- XTAH Builder (offline)

XML structural patterns are first extracted as the basis for initial clustering. These patterns are then run through a clustering algorithm (e.g. ICE) and written to a "pre-XTAH" file. This process is under on-going research; the representation format of these XML patterns are not yet finalized. To improve storage and file access efficiency, we use object mapping, relating each structural pattern, as an object, to an object ID. This mapping is recorded in a "object mapping" file, created along with the clustering.

XTAH Manager and Builder are implemented with JAVA in a complete object oriented fashion. Specifically, we allow programmer to develop various object mapping schemes that work with the two XTAH modules without recompilation. This is done by defining an object specification interface.

XTAH Builder first parses the "pre-XTAH" file and "object mapping file"; it then assigned internal objects by promoting objects that minimizes the covered clusters (refer to the section 6). After internal assignments, a complete XTAH tree is built, which is then written to a "XTAH" file. Both the "pre-XTAH" and "XTAH" file are written in XML format.

XTAH manager implements the JAVA RMI interface, thus allowing remote client to directly reference the XTAH manager object. XTAH manager loads the "XTAH" file as requested by the client. XTAH manager supports operations such as specialization and generalization. XTAH manager does not keep the internal state of querying (i.e. it would not know which query is at which generalization/specialization level). This is so designed such that the XTAH manager loaded with a particular XTAH can answer many different applicable queries without loading the same XTAH multiple times. The querying state is kept by the Relaxation Manger which lies between the application client and XTAH manager. Relaxation Manager keeps the querying state by obtaining the reference to a "relaxation state" object after the first time it asks the XTAH manager to locate the closest matched cluster.

Complete source code can be found in Appendix B.

8. Summary

This report presents the study on using M-Tree algorithm to assign internal representation of X-TAH. M-Tree is a promising indexing solution to X-TAH, because it preserves complex objects like trees, and reduce our problems into finding a good distance function and clustering algorithm. X-TAH differs from the original M-tree in that the leaf clusters are initially constructed by similarity clustering; assuming good clustering will group similar objects in a single cluster, the pruning should be more effective than the general M-tree. Our first stage implementation shows promising result of large percentage of data objects pruned in the searches conducted. Our object oriented implementation facilitate further research on object representation and object distance measure algorithms.

Figure 4. Cobase Relaxation Architecture

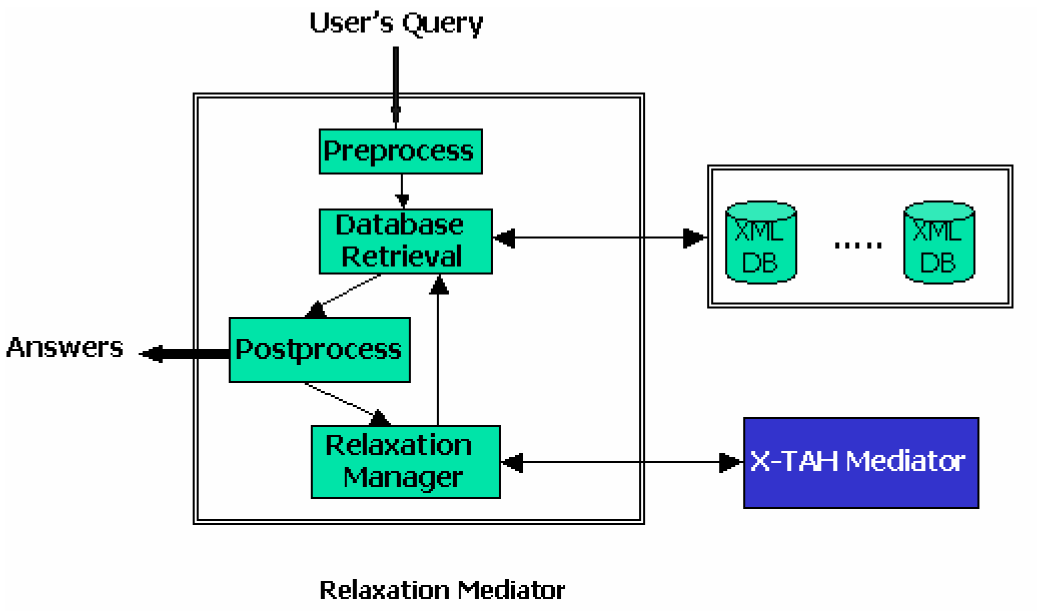
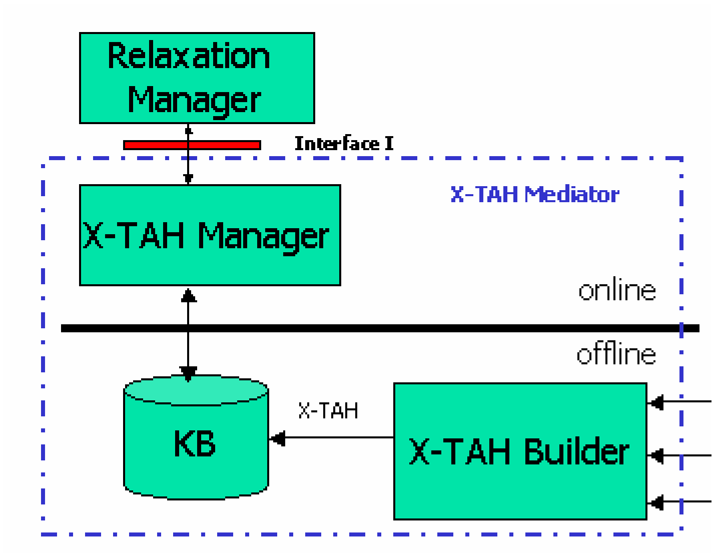


Figure 5. X-TAH Mediator



References

Clustering

[1] M. A. Merzbacher and W. W. Chu. Pattern-Based Clustering for Database Attribute Values in Proceedings AAAI Workshop on Knowledge Discovery in Databases, Washington D.C., 1993. (8 Pages)

[2] Wesley W. Chu, Kuorong Chiang, Chih-Cheng Hsu, Henrick Yau, "An Error-based Conceptual Clustering Method for Providing Approximate Query Answers", 1996

[3] Wesley W. Chu, Hua Yang, Kuorong Chiang, Michael Minock, Gladys Chow, and Chris Larson. "CoBase: A Scalable and Extensible Cooperative Information System" *Journal of Intelligence Information Systems*. Vol 6, 1996, Kluwer Academic Publishers, Boston, Mass.

Distance Function and Relevance Ranking

[4] T. Schlieder and F. Naumann. Approximate Tree Embedding for Querying XML Dasta. In Proc. ACM SIGIR Workshop on XML and Information Retrieval, Athens, Greece, July 2000.

[5] P. Ciaccia and W. Penzo. Relevance Ranking Tuning for Similarity Queries on XML Data. First VLDB Workshop on Efficiency and Effectiveness of XML Tools and Techniques (EEXTT 2002), Hong Kong, China, 2002. A. Nierman and H. V. Jagad

[6] K. Zhang and D. Shasha, "Simple Fast Algorithms for the Editing Distance between Trees and Related Problems", SIAM Journal of Computing, 18(6): 1245-1262, 1989

[7] K. Zhang and D. Shasha. Fast Algorithms for the unit cost editing distance between trees. Journal of Algorithms, 11:581-621, 1990

[8] K. Zhang, R. Statman, D. Shasha, "On the Editing Distance between Unordered Labeled Trees", Information Processing Letters, 42: 133-139, 1992

[9] S. Amer-Yahia, S. Cho, D. Srivastava (AT&T Labs), "Tree Pattern Relaxation", EDBT 2002

KNN Indexing

[10] Thomas Seidl, hans-Peter Kriegel, "Optimal Multi-step k-Nearest Neighbor Search", SIGMOD '98

[11] Stefan Berchtold, Bernhard Ertl, Daniel A. Keim, Hans-Peter Krigel, Thomas Seidl, "Fast Nearest Neighbor Search in High-dimensional Space", International Conference on Data Engineering (ICDE '98), Orlando, Florida.

[12] Paolo Ciaccia, Marco Patella, Pavel Zezula, "M-tree: An Efficient Access Method for Similarity Search in Metric Spaces"

[13] David A. White Ramesh Jain, "Similarity Indexing : Algorithms and Performance", 1999

[14]Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, Angela Y. Wu, "An Optimal Algorithm for Approximate Nearest Neighbor Searching in Fixed Dimensions", Fifth annual ACM_SIAM Symp. 1998

[15] Stefan Berchtold, Daniel A. Keim Hans-Peter Kriegel, "The X-tree: In index Structure for High-Dimensional Data", 22nd VLDB conference, India, 1996

DataGuide
[16] Representative Objects: Concise Representations of Semi-structured, Hierarchical Data
Vetlozar Nestorov, Jeffrey Ullman, Janet Wiener, Sudarshan Chawathe
Proceedings of 13th International Conference on Data Engineering (ICDE'97), Birmingham, England, April 1997.

[17]Inferring Structure in Semi-structured Data
Svetlozar Nestorov, Serge Abiteboul, Rajeev Motwani
Proceedings of Workshop on Management of Semistructured Data held in conjunction with SIGMOD'97, Tucson, Arizona, May 1997.

[18] DataGuides: Enabling Query formulation and Optimization in Semi-structured Databases, Roy Goldman and Jennifer Widom, Proceedings of the 23rd VLDB Conference, Athens, Greece, 1997

Appendix A.

Pseudo-code for Range Queries:

```
RS(N:node, Q:query_object, r(Q): search_radius){
        let $O_p$ be the parent object of node N;
        if N is not a leaf
        then {
            for all Or in N do:
                if | $d(O_p, Q)$  - $d(O_r, Op)$ | <= $r(Q) + r(O_r)$
                then {
                    Compute $d(O_r, Q)$;
                    if $d(O_r, Q) <= r(Q) + r(O_r)$
                    then RS(* ptr($T(O_r)$), Q, r(Q));}}
                else{
                    for all $O_j$ in N do:
                    if | $d(O_p,Q) – d(O_j,O_p)$ | <= r(Q)
                    then {
                        compute $d(O_j,Q)$;
                        if $d(O_j,Q) <= r(Q)$
                        then add oid($O_j$) to the result;}}
```

Pseudo-code for KNN Queries:

```
K-NN_NodeSearch(N: node, Q: query_object, k:integer)
{
    let $O_p$ be the parent object of node N;
    if | $d(O_p,Q) – d(O_r, Op|$ <= $d_k + r(O_r)$ then.
    {
        compute $d(O_r, Q)$;
        if $d_{min}(T(O_r)) <= d_k$ then
        {
            add [ptr($T(O_r)$), dmin($T(O_r)$)] to PR;
            if $d_{max}(T(O_r)) < d_k$ then
            {
                $d_k$ = NN_Update([_, $d_{max}(T(O_r))$]);
                remove from PR all entries for which $d_{min}(T(O_r)) < d_k$; }}}}
    else /* N is a leaf */
    {
        for all $O_j$ in N do:
        if {$d(O_p, Q)-d(O_j,O_p)$| <= $d_k$ then
        {
            compute $d(O_j,Q)$;
            if $d(O_j,Q) <= d_k$ then
            {
                $d_k$ = NN_update([oid($O_j$), $d(O_j,Q)$]);
                remove from PR All entries for which dmin($T(O_r)$) > $d_k$; }}}}
```

Appendix B.

```
package cobase.xtah;
import org.dom4j.*;
import java.util.Vector;
import java.util.Iterator;
/**
 * Title: XTBuilder.java
 * Description: X-TAH builder
 * Copyright:    Copyright (c) 2003
 * Company: Cobase, UCLA
 * @author: Tony Lee
 * @version 1.0
 */

public class XTBuilder {
  private ObjSpec obs;
  private Document doc;
  public XTBuilder(Document doc, ObjSpec obs) {
    this.obs = obs;
    this.doc = doc;
  }
  public boolean Process(){
    buildXT_A(doc.getRootElement());
    buildXT_B(doc.getRootElement());
    return true;
  }

  private void buildXT_B(Element elm){
     float dtop;
     if(elm.isRootElement()) dtop = 0;
     else dtop = obs.distMeasure(elm.attributeValue("id"),elm.getParent().attributeValue("id"));
     elm.addAttribute("dtop", String.valueOf(dtop));
     for(int i = 0, size = elm.nodeCount(); i < size; i++){
      Node node = elm.node(i);
      if(node instanceof Element) buildXT_B((Element) node);
     }
  }

  private Vector buildXT_A(Element elm){
    Vector coverage = new Vector();
    Attribute temp;
    float rad=0;
    String id;

    if(elm.nodeCount()==0){
      rad = 0;
      assignAttrib(elm,rad,null,coverage);
      coverage.add(elm.attributeValue("id"));
      return coverage;
    }
    for(int i = 0, size = elm.nodeCount(); i < size; i++) {
```

```
    Node node = elm.node(i);
    if(node instanceof Element)
      coverage.addAll(buildXT_A((Element)node));
  }
  Candidate cd;
  cd = promoteFrom(coverage);
  assignAttrib(elm,cd.rad,cd.id,coverage);
  return coverage;
}

private void TestLoop(Element elm){
  if(elm.nodeCount()==0) return;
  else{
    for(int i = 0, size = elm.nodeCount(); i < size; i++)
      TestLoop((Element)elm.node(i));
  }
}
private Candidate promoteFrom(Vector coverage){
  if(coverage.isEmpty()) return null;
  Candidate cd;
  float min_rad = -1;
  String cur_cand = "-1";
  float max_dist = 0;
  for(int i=0; i<coverage.size(); i++){
    for(int j=0; j<coverage.size(); j++){
      float dist = obs.distMeasure((String)coverage.elementAt(i),(String)coverage.elementAt(j));
      if (dist > max_dist) max_dist = dist;
    }
    if(min_rad == -1){
      min_rad = max_dist;
      cur_cand = (String) coverage.elementAt(i);
    }
    if(max_dist < min_rad){
      min_rad = max_dist;
      cur_cand = (String) coverage.elementAt(i);
    }
    max_dist = 0;
  }
  cd = new Candidate(min_rad, cur_cand);
  return cd;
}
public void assignAttrib(Element elm, float rad, String id, Vector cover){
   if(id!=null) elm.addAttribute("id", id);
  elm.addAttribute("rad", String.valueOf(rad));
  String coverage = new String("[ ");
  /*
  for(Iterator i=cover.iterator(); i.hasNext(); ){
    coverage+=i.next();
    coverage+=",";
  }
  coverage+=" ]";
  elm.addAttribute("coverage",coverage);
*/
```

```
  }
  public class Candidate{
    public Candidate(float rad, String id){
      this.id = id;
      this.rad = rad;
    }
    String id;
    float rad;
  }
}

/* ================== */
package cobase.xtah;

/**
 * Title: ObjSpec.java
 * Description: Object specification Inteface
 * Copyright:    Copyright (c) 2003
 * Company: Cobase, UCLA
 * @author: Tony Lee
 * @version 1.0
 */

public interface ObjSpec {
  public float distMeasure(String o1, String o2);
  public String toQuery(String id);
  public String mapQuery(String query);
  public void demapQuery(String id);
}

/* ================== */
package cobase.xtmag;
import java.rmi.*;
import java.rmi.server.*;
import java.util.*;
import cobase.xtah.*;
import org.dom4j.*;
import java.io.*;
import java.net.*;
/**
 * Title: XTmanager.java
 * Description: X-TAH manager
 * Copyright:    Copyright (c) 2003
 * Company: Cobase, UCLA
 * @author: Tony Lee
 * @version 1.0
 */

public class XTManager extends UnicastRemoteObject implements XTRel{

  private Document xtdoc;
  private ObjSpec tos;
private TreeSet PR;
private float dk;
private Element target;
private String qid;

public XTManager(Document doc, ObjSpec obs) throws RemoteException{
  tos = obs;
  xtdoc=doc;

  //initialize
  target = null;
  dk = 0;
  PR = new TreeSet();
}
public RelState findTarget(String query) throws RemoteException{
  if(xtdoc == null) return null;
  //initalize state variables
  target = null;dk = 0;PR.clear();

  qid=tos.mapQuery(query);
  System.out.println(qid);

  Element root = xtdoc.getRootElement();
  dk = Float.POSITIVE_INFINITY ;
  searchTarget(root);
  RelState rs = new RelState();
  setState(target, rs);
  return rs;
}

private void setState(Element node, RelState rs) throws RemoteException{
  if(node == null) return;
   rs.addState(node);
   setState(node.getParent(),rs);
}

private void searchTarget(Element elm){
  boolean pr_prune = false;

  System.out.println(PR.toString());
  for(int i=0; i<elm.nodeCount(); i++){
    Node node = elm.node(i);
    if(!(node instanceof Element))
      continue;
    float rad = Float.parseFloat(((Element)node).attributeValue("rad"));
    float dist = tos.distMeasure(qid,((Element)node).attributeValue("id"));
    float n_dmax = dist + rad;
    float n_dmin = Math.max(dist-rad, 0);
    if(n_dmin < dk){    // qualify so far
      System.out.println("qualified element "+ ((Element)node).attributeValue("id"));
      PRPair pp = new PRPair((Element) node,n_dmin, n_dmax);
      PR.add(pp);
      }
      if(n_dmax < dk) { dk = n_dmax; target = (Element) node; pr_prune = true; }  //need to
prune PR
    }
    if(pr_prune == true){
      System.out.println("pruning "+PR.toString());
      while(!PR.isEmpty()){   //prune PR
        PRPair cur = (PRPair)PR.last();
        if( cur.dmin > dk ) PR.remove(cur);
        else break;
      }
      System.out.println("pruning result: "+PR.toString());
    }
    while(!PR.isEmpty()){ //choose node to expand
      PRPair pp = (PRPair) PR.first();
      if(!PR.remove(pp)) System.out.println("error in removing element from PR queue!");
      searchTarget(pp.node);
    }
    return;
  }

  public Vector Generalize(RelState rs) throws RemoteException{
    if(!rs.canGeneralize()) return null;
    Vector cv = new Vector();
    findCoverage(cv, rs.walkUp());
    return cv;
  }
  public Vector Specialize(RelState rs) throws RemoteException{
    if(!rs.canSpecialize()) return null;
    Vector cv = new Vector();
    findCoverage(cv, rs.walkDown());
    return cv;
  }

  private void findCoverage(Vector coverage, Element elm){
    boolean is_leaf = true;
    if(elm == null) return;
    for (int i=0; i<elm.nodeCount(); i++){
      if(elm.node(i) instanceof Element){
        is_leaf = false;
        findCoverage(coverage, (Element)elm.node(i));
      }
    }
    if(is_leaf) coverage.add(tos.toQuery(elm.attributeValue("id")));
  }

  public String getXTProperties() throws RemoteException{ return null;}

  public void bindToNamingService() throws Exception{
      InetAddress addr = InetAddress.getLocalHost();
      String localHost = addr.getHostName();
      String nameURL = "//"+localHost + "/xtmanager";
      try {
        Naming.bind(nameURL,this);
        System.out.println("XTManager bound");
      }catch (Exception e) {
        System.err.println("XTManager exception: " + e.getMessage());
        e.printStackTrace();
      }
    }
  public class PRPair implements Comparable{
    public Element node;
    public float dmax;
    public float dmin;
    public PRPair(Element node, float dmin,float dmax){
      this.node=node;
      this.dmin = dmin;
      this.dmax = dmax;
    }
    public int compareTo(Object pp){
      if(node.attributeValue("id").compareTo( ((PRPair)pp).node.attributeValue("id") ) == 0)
        return 0;
      else if(dmin <= ((PRPair)pp).dmin) return -1;
      else if(dmin > ((PRPair)pp).dmin) return 1;
      else return 0;
    }
    public String toString(){
      return node.attributeValue("id");
    }
  }
}

/* ================== */
package cobase.xtmag;
import java.util.*;
import java.io.Serializable;
import org.dom4j.*;
/**
 * Title: RelState.java
 * Description: Relaxation State
 * Copyright:    Copyright (c) 2003
 * Company: Cobase.UCLA
 * @author: Tony Lee
 * @version 1.0
 */

public class RelState implements Serializable {
  int i;
  private Vector states;
  public RelState() {
    states = new Vector();
    i = 0;
  }
  public void addState(Element n){
    states.addElement(n);
  }
  public boolean canGeneralize(){
    if(i>=states.size())  return false;
    else  return true;
  }
  public boolean canSpecialize(){
    if(i>=0) return true;
    else return false;
  }
  public Element walkUp(){
    Element elm;
    if(i >= states.size()) return null;
    elm = (Element) states.elementAt(i);
    i++;
    return elm;
  }
  public Element walkDown(){
    if(i >= states.size()) i = states.size()-1;
    if(i < 0) return null;
    Element elm = (Element)states.elementAt(i);
    i--;
    return elm;
  }
  public void display(){
    if(states == null)
      return;
    String output = "States: ";
    for (Iterator i = states.iterator(); i.hasNext(); ) {
      Element elm = (Element) i.next();
      output+=elm.attributeValue("id")+" - ";
    }
    System.out.println(output);
}

/* ================= */
package cobase.xtmag;
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.*;
import cobase.xtah.*;
/**
 * Title: XTRel.java
 * Description: X-TAH Relaxation Interface
 * Copyright:    Copyright (c) 2003
 * Company: Cobase, UCLA
 * @author: Tony Lee
 * @version 1.0
 */
public interface XTRel extends Remote{
  RelState findTarget(String query) throws RemoteException;
  Vector Generalize(RelState rs) throws RemoteException;
  Vector Specialize(RelState rs) throws RemoteException;
  String getXTProperties() throws RemoteException;
}
```