Xp-tree: Xpath 로케이션 스텝의 효율화를 위한 새로운 공간기반의 인덱싱 기법

Nguyen Van Trang^{0}, 황 정 희, 류 근 호
충북대학교 데이터베이스 연구실
{nvtrang^{0}, jhhwang, khryu}@dblab.chungbuk.ac.kr

Xp-tree: A new spatial-based indexing method to accelerate Xpath location steps

Nguyen Van Trang^{0}, Jeong Hee Hwang, Keun Ho Ryu
Database laboratory, Chungbuk National University

Abstract

Nowadays, with the rapid emergence of XML as a standard for data exchange over the Internet had led to considerable interest in the problem of data management requirements such as the need to store and query XML documents in which the location path languages Xpath is of particular important for XML application since it is a core component of many XML processing standards such as XSLT or XQuery. This paper gives a brief overview about method and design by applying a new spatial-based indexing method namely Xp-tree that used for supporting Xpath. Spatial indexing technique has been proved its capacity on searching in large databases. Based on accelerating a node using planar as combined with the numbering schema, we devise efficiently derivative algorithms, which are simple, but useful. Besides that, it also allows to trace all its relative nodes of context node in a manner supporting queries natural to the types especially Xpath queries with predicates.

1 Introduction

Query languages for XML and semistructured data rely on location paths for selecting nodes in data items. In particular, XQuery and XSLT are based on Xpath.

To speed up the processing of regular path expression query, it is important to be able to quickly determine all the relationships between any pair of nodes in the hierarchy of XML data such as ancestor-descendant, parent-child, etc...[1][2]. Current index structures that support an efficient evaluation of such query, however, there is no method exists today for eliminating redudant work in the predicate evaluation part of Xpath queries. Unfortunately, the computation time is dorminated by the latter when queries have multiple predicates that are typical in XML data [4].

This work is a proposal for a databases index structure, which is applied index technology in spatial data structures, and has been specifically designed to support the evaluation of Xpath queries. By applying the spatial-indexing technology, it is easily capable to support all Xpath axes (ancestor, following, preceding-sibling, descendant...) and extended some axes (first-following sibling, second-following sibling, first-preceding sibling...). Assume that, readers are familiar with Xpath and axes.

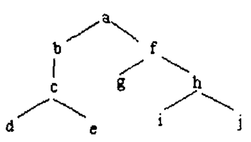
Our main goal for this method is to optimize search performance using a spatial search tree, especially for searching the sibling relations from the given node, which is proven that concentrate most of data.

2 Background and Previous work

To efficiently address complex queries, it is important to quickly determine the structural relationship among any pair of tree nodes. In this section, we first provide background information on the importance of numbering schemes in determining structural relationships, and then discuss previous works on the concept of regular path expressions

2.1 Overview of Xpath Axes and XML document region

The basic building block of Xpath is the expression, directly reflects the recursive nature of tree-shaped data. Xpath expressions operate on tree of element or attribute nodes [5].



```
<a>
    <b>
        <c>
            <d> </d><e> </e>
        </c>
    </b>
    <f>
        <g> </g>
        <h>
            <i> </i><j> </j>
        </h>
    </f>
</a>
```

Figure 1: Tree representation of small XML document instance.

In this tree, the inner nodes a, b, c, f, g, h represent XML element nodes, the leaf nodes d, e, g, i and j represent either element or attribute nodes.

XPath expressions specify a tree traversal via two parameters: (1) a context node (not necessarily the root) which is the starting point of the traversal, (2) and a sequence of location steps syntactically separated by /, evaluated from left to right. Given a context node, a step's axis establishes a subset of document nodes (a document region). This set of nodes, provides the context nodes for the next step, which is evaluated for each node of the forest in turn. The results are joined together and sorted in document order.

2.2 XML order encoding method

We are now left with the challenge to find an encoding of the tree-shaped node hierarchy in an XML document that can be efficiently supported by existing database technology. Here, efficiency means that the encoding has to map the input tree-shape into a domain in which a node's region membership may be tested by a simple relational query. Here, with the limitation of paper, we do not concerned about numbering scheme but for the simplicity we applied the Dietz's numbering schema to encode the document. To the best of our knowledge it was an efficient method and easy to understand [1].

Figure 2(a) illustrates the node distribution of the example document (shown in Figure1) after its nodes have been mapped into the pre/post plane and figure 2(b) is an overview of a real XML document when mapped into plane with Dietz's numbering schema.

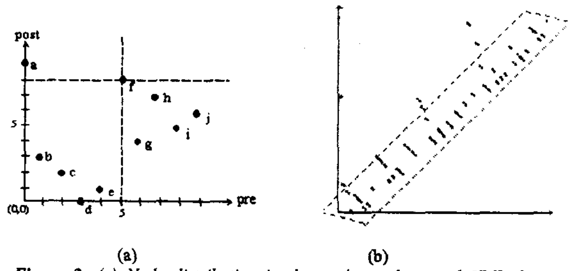
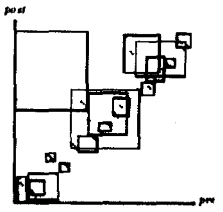


***Figure** 2: (a) Node distribution in the pre/post plane and XML document regions as seen from context nodes f, (b) Example of a pre/post rank distribution for an XML document instance of 100 nodes.*

2.3 Indexing with R-tree

In the concept of accelerating Xpath location steps [5], the authors proposed a databases index structure that can completely live inside a relational database system. The proposal index structure can benefit from advanced index technology R-tree. The data driven R-tree remains balanced even in the present of skewed distributions.



Axis α	Query window window(α, v)				
	pre	post	par	att	tag
child	⟨(pre(v), ∞)	, [0, post(v))	, pre(v)	, false	, * ⟩
descendant	⟨(pre(v), ∞)	, [0, post(v))	, *	, false	, * ⟩
descendant-or-self	⟨[pre(v), ∞)	, [0, post(v)]	, *	, false	, * ⟩
parent	⟨[par(v), par(v)]	, (post(v), ∞)	, *	, false	, * ⟩
ancestor	⟨[0, pre(v))	, (post(v), ∞)	, *	, false	, * ⟩
ancestor-or-self	⟨[0, pre(v)]	, [post(v), ∞)	, *	, false	, * ⟩
following	⟨(pre(v), ∞)	, (post(v), ∞)	, *	, false	, * ⟩
preceding	⟨[0, pre(v))	, [0, post(v))	, *	, false	, * ⟩
following-sibling	⟨(pre(v), ∞)	, (post(v), ∞)	, par(v)	, false	, * ⟩
preceding-sibling	⟨[0, pre(v))	, [0, post(v))	, par(v)	, false	, * ⟩
attribute	⟨(pre(v), ∞)	, [0, post(v))	, pre(v)	, true	, * ⟩

***Figure** 3: R-tree based index structure and Xpath axes α and their corresponding query windows window(α,v) (context node v)*

But in fact, this method has just focused on processing 4 main axes: ancestor, descendant, preceeding, following axes (based on the MBR property) and the remains require further processing. All axis query windows in the two-dimensional pre/post plane depend on a range of selection in the pre as well as the post dimension. Thus if the nodes in the window are determined via two independent range query, large query windows generally leaded to numerous false hits during scan.

After getting all nodes in the windows query preceeding, it requires remarkable steps to search the required nodes for sibling that will waste search space for complex queries (overlapping problem) and lack the ability for the position-based queries (location predicate).

3 Xp tree structure

3.1 Overview

When designing an access method, we not only have to be aware of the nature of the data, but must also know the types of queries and the method are to be used. Typically the R-tree-based indexing method has solved most of cases for the simple queries but a problem not addressed by using that method is the siblings and the complex queries with predicate especially with location parameter.

From that observation, we realized that all the data concentrate on the diameter of the plane but the previous method did not concerned about that, it just has focused on processing 4 axes and the remains require further processing. For example, for query preceding sibling and complex queries such as: ***descendant[2]/preceding-sibling[3]*** [3][4]. It means that, the required node in the third position of the preceding sibling of the second descendant of the context node. R-tree-based indexing takes time or event hardly to solve that kind of query.

3.2 Index structure

Xp-tree is fundamentally different from the previous access method (R-tree based method). Xp-tree applies a different insertion/split strategy to achieve the sibling relationships of the XML data easily, while not compromising the space discrimination capabilities of the index too much. New method will be more efficient than previous work because of using pointers that will reduce the number of windows query for searching and save time.

With the Xp-tree, we aim for an access method that strictly preserves sibling trajectory. Hereafter, trajectory means the sibling nodes are nodes in the same level and have the same parent. As such the structure of the Xp-tree is actually a set of leaf nodes, each containing a partial children of one parent, organized in a tree hierarchy. In other words, siblings of one parent are distributed over a set of disconnected leaf nodes. As we shall see later on when discussing about query processing, it is necessary to be able to retrieve siblings based on their sibling identifier. We choose a double linked list that connects leaf nodes through previous sibling and following sibling. Moreover we use one more pointer to connect from a node to its parent. Therefore we use 3 pointers in a one node to connect to previous sibling, next sibling, and its parent. This ensures that from every node we can trace quicky its relationship.

For example, in the case of Xmark XML Benchmark document, after numbering (for simplify, in here we only use preorder value), all the nodes will be represented in the Xp-tree structure as belows:

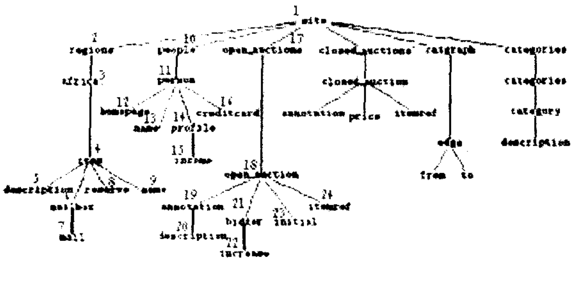


Figure 4: Element hierarchy of Xmark XML Benchmark document instances

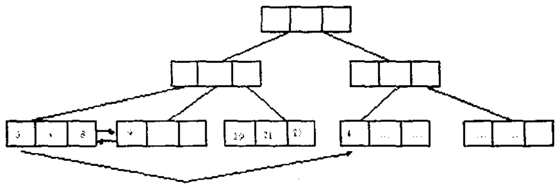


Figure 5: Xp-tree in tree structure and its data's representation on leaf node

3.2.1 Algorithms

The goal is to keep the sibling trajectory of the XML data. The insertion algorithm is rarely simple. To insert a new entry, we simply have to find the leaf node that contains its predecessor (previous sibling) with the same level in the trajectory. We start by traversing the tree from the root and step into every child node that overlaps with the MBR of the context node. We choose the leaf node containing nodes connected to the new entry.

```
Algorithm Insert(N,E)

1.  Invoke FindNode(N,E) to find a leaf node N' containing
    the sibling predecessor of the new context node entry E
    to be inserted
2.  if node N' is found,
3.     if (N' has space) not full then
4.        Insert new context node, E into N'.
5.     else
6.     CreateNewLeafNode(E) to create new leaf node
       for new context node, E, insert newly created leaf
       node into tree
7.     endif
8.  else
9.     CreateNewLeafNode(E) to create new leaf node
       for new context node, E, insert newly created leaf
       node into tree
10. endif

Algorithm CreateNewLeafNode(E)

Steps up the tree until a non-full parent node, Q is found.
Traverse the right-most path from this node, Q to reach
the non-leaf parent node P at the level 1.
1. if non-leaf node P is not full the newly created leaf
   node created is inserted into node P.
2. else
3. Split the non-leaf node P by creating a new non-leaf
   node R at level 1 and this new non-leaf node R has the
   new leaf node created previously as its first child.
4. Split of non-leaf nodes may propagate upwards the tree as
   the upper non-leaf nodes may become full.
5. endif
```

Figure 6: Insert Algorithms

3.2.2 Query Processing

In this part, we briefly depict some algorithms that are designed for specific cases: sibling queries. The remains axes (based on the majors axes of Xpath) are not mentioned in this paper due to the limitation of paper.

```
Algorithm SiblingQuery(N, E, RESULT)

1. Invoke Findnode(N, E) to find node N' which contains entry E
2.   if N' is found
3.     for each entry E' of N'
4.     Add E' to RESULT
5.     if Following sibling pointer F is valid,
       Invoke FollowingSiblingQuery(NF, RESULT), where NF
       is the childnode of N pointed to by F
6.     if preceding sibling pointer P is valid
       Invoke PrecedingSiblingQuery(NP, RESULT), where NP
       is the childnode of N pointed to by P.
7. else
8.   This node does not exist
9. endif
```

Figure 7: Algorithm for sibling query

The linked lists of the Xp-tree allow us to retrieve connected nodes without searching. We have two possibilities: a connected node can be in the same leaf node or in another node. If it is the same, finding it is trivial. If it is in another node, we have to follow the next (previous) pointer to the next (previous) leaf node. It is similar with the case of finding parent-child relationship

4 Conclusion

This work has been primarily motivated by the need for an efficiently Xpath index structure that would be capable to support the whole family of Xpath in an adequate manner as well as searching process especially for complex queries that contain predicate statement. Thus, in this paper, we just only point out idea, and design a new algorithms for enlarging that specific target.

Theoretically, the Xpath queries could gain from advanced query processing techniques like spatial-based indexing method because of its features. Observation of this kind, together with the cost estimation procedures could lead to a rather pragmatic cost model for Xpath queries. It will be interesting to compare this approach to other models.

References

[1] *Quanzhong Li, Bongki Moon.* Indexing and Querying XML Data for Regular Path Expression. In *Proceeding of the 27th VLDB Conference,* Roma, Italy, 2001

[2] *Tove Milo, Dan Sucio* Index Structure for Path expressions. In *Proc. of the Int'l Conf. on Database Theory*, page 277-295, 1999

[3] *Dan Olteanu, Holger Meuss, Tim Furche.* Xpath: Looking forward. In Proceeding of EDBT Workshop on XML Data Management, 2002

[4] *Kumar Gupta, Dan Suciu.* Stream Processing of Xpath Queries with Predicates. In SIGMOD Conference, San Diego, CA, June 2003

[5] *Torsten Grust.* Accelerating Xpath Location Steps. In SIGMON Conference, 2003