Index XML Data Using Extended Order and Path Index

Guangming Xing
Department of Computer Science
Western Kentucky University
Bowling Green, KY 42101
Guangming.Xing@wku.edu

ABSTRACT

The eXtensible Markup Language (XML) is becoming a new standard for information representation and exchange over the Internet. How to index XML data for efficient query processing and XML transformation is an important subject in the XML community. In this paper, based on extended preorder indexing method, we add path information as part of the index. It is shown that the number of path joins can be reduced to the number of the "interested points", but not related the length of the path expression in a query. The extra space needed is about the same as extended preorder labeling method.

1. INTRODUCTION AND REVIEW

The eXtensible Markup Language (XML) is becoming a new standard for information representation and exchange over the Internet [8]. To retrieve XML data from data repository, various query languages are proposed [6, 2, 7]. One key component of all these query languages is XPath, which allows regular path expressions in a query. There are three parts in an XPath expression [4]:

1. Node test: which specifies the node type and expanded-name of the nodes selected by the location step;
2. Axis: which specifies the tree relationship between the nodes selected by the location step and the context node;
3. Predicate: which use arbitrary expressions to further refine the set of nodes selected by the location step.

Because wildcard can appear in node test and descendent axis can represent any sequence of elements between two location steps in an XPath expression, XPath has the power to express regular path expressions. Users can navigate through arbitrary long paths in the XML data by using regular path expressions which coincide with capabilities of arbitrary nesting in XML. Also using wildcard in regular path expressions allows the user to retrieve data from documents that are not well-structured or whose structure is not known. We can expedite the processing of SQL queries over relational data by using index over tables. But unlike relational data, XML data is difficult to index as it lacks of structure. There are two reported methods to index XML data: the first is called path index like DataGuide [6] and Index Fabric [3], and the other uses extended preorder to index path information as described in [5]. These two methods offer different approaches to handle regular paths. For path index, take Index Fabric [3] as an example, it can handle child axis efficiently as it indexes the tag information from the root to the leaf. But it can not handle regular path expression explicitly. Several techniques were proposed to handle this, these include using existing query, or path pruning based on the schema to convert the regular path expressions into definite path expressions without wildcard and descendent axis. We will briefly discuss these two methods in the next two sections. Interested reader should refer [3] and [5] for more details.

1.1 Extended Preorder Labeling

In [5], extended preorder labeling was proposed to index the path information for XML data. The idea is based on the fact that we can view an XML document as an ordered labeled tree (forest). For each node (element or attribute) in an XML tree, it is labeled with an integer pair $(order, size)$. We use $order(x)$ to denote the node x's $order$ and $size(x)$ to denote node size of the subtree rooted at x. The order and size is not exactly the same as traditional preorder traversal labeling. It is extended in the sense that it satisfies the following conditions that are typically only apply to the preorder labeling:

1. For node y with parent (or ancestor) x, $order(x) < order(y)$, and $order(y) + size(y) \leq order(x) + size(x)$. This means that node x properly covers the subtree rooted at y.
2. For sibling nodes x and y, if x is the predecessor of y in the preorder traversal, then $order(x) + size(x) < order(y)$, which means that x appears to the left of y.

Based on the above observation, the following theorem was stated in [5].

THEOREM 1. *For two given nodes x and y, x is an ancestor of y iff $order(x) < order(y) \leq order(x) + size(x)$.*

In contrast to path indexing methods, extended preorder indexing, on the other hand, does not support traversing from a node to its children (descendents) directly based on

the node test as in the original document stored in files, because there is no physical links between a node with its children and vice versa. Instead, different elements (and attributes) are stored in different "bag"s which can be implemented as a B^+-tree. When we need to go from E_1 to E_2, it does not matter whether it is child axis or descendent axis, (if child axis is insisted, we need to add level information in the index, this is an obvious extension to [5]) we just get all the elements stored in bag E_1 and E_2, based on theorem 1, we can efficiently determine the ancestor-descendent relation based on the index.

To handle the location steps in an XPath expression, the following three path joins was proposed in [5]:

1. Element-Element Join
2. Element-Attribute Join
3. Kleene-Closure

For more detail about these three kinds of join, please refer the original paper.

Another advantage of the extended preorder labeling method which is not mentioned in [5] is that it can handle a path expression in either top-down direction (descendent axis) or bottom-up direction (ancestor axis), as determining the ancestor relation is the same as determining the descendent relation. But for Index Fabric, it is difficult to process a path expression in bottom-up direction, as there is no explicit link between an element and its ancestors unless we add a sperate parent link (but we still can not solve the ancestor axis problem).

One disadvantage of using extended preorder labeling is that it prefers short path expression. Suppose we have a path expression like the following:

$$E_1//E_2//E_3//E_4//E_5[@A = v]$$

From the algorithm presented in [5], we have to start from the set of elements of E_1, then get E_2, E_3, E_4, E_5 and those E_5 with attribute A of value v. Our question is, could we eliminate these Element-Element joins and just use a single Element-Attribute join instead? The answer is yes as long as E_5 is the only point we are interested in. In fact we do not have to process those E_5 elements that do not have $E_1//E_2//E_3//E_4$ as prefix. Unlike other indexing methods preferring more definite path expression, a shorter path expression (even with more descendent axis and wildcard) will make the query more efficient. We will propose a solution to handle this problem, and this is the main contribution of this paper.

1.2 Path Index Review

Various path indexing methods such as DataGuide and Index Fabric were proposed to expedite XML query processing. Both methods offer good performance when the path is fixed (no wildcard or descendent axis) as navigating on the data is just following the path. But when there is wildcard or descendent axis in the path expression, there is no explicit method to navigate the data efficiently except traversing the whole data tree (path pruning is still a possible solution when schema is available).

Consider the following XML data segment from ACM SigModRecord at `http://www.acm.org/sigmod/record/xml`..

```
<SigmodRecord>
 <issues>
 <issue>
  <volume>15</volume>
  <number>2</number>
 <articles>
 <article>
  <title articleCode="152037">A formal view integration
  method</title>
 <authors>
  <author AuthorPosition="02">Bernhard Convent</author>
  <author AuthorPosition="01">Joachim Biskup</author>
  </authors>
  </article>
</issue>
</issues>
<SigmodRecord>
```

For the above XML data segment, we have the following path index:

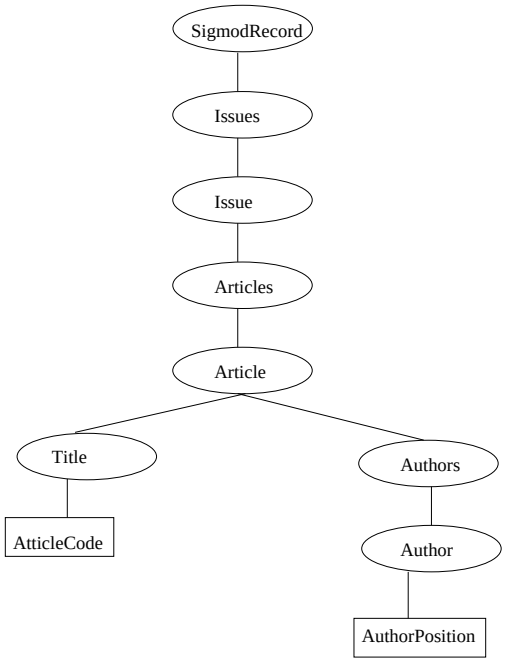

Figure 1: Path Index for SigMod Data

2. COMBINED METHOD

Although XML tags consume considerable amount of space for XML data, the size of the DataGuide (unique paths) tends to be small, regardless of the DTD and real data size [9]. We can get this from the above example, as we can have as many data entries as we like, but the the size of the DataGuide is very small as illustrated by Figure 2.

Based on this observation, if we can organize the data based on the path (absolute path starting from the root element) as well as label the XML data using extended preorder as presented in [5], we could combine the advantages of both methods. As DataDuide is a tree, there is a unique path from the root to each node. So for a tree with n nodes, there are n unique paths from the root. Based on this idea, the running time of our algorithm is only related to the num-

ber of "interested points" that will be defined later, but not related to the length of the XPath expression in an XQuery. As the first step to handle this problem, we only consider the following three axis: child, descendent and attribute.

To process a path expression, we need to convert it to a regular expression whose alphabet is the set of tags. To facilitate the processing, we can think a tag is a special character called designator as used in [3].

For ease of description, we call each element that has predicate associated with it as an "interested point" (those elements and attributes for output must be "interested point" also because we need to pick those elements out, but we will not consider that case here as we only talk about the XPath here, that's the reason why we call it "interested point" not just use predicate). "Interested points" are those nodes whose presence makes the difference when we process a path expression.

We call two segments adjacent if there is no "interested point" in between. Based on the definition of "interested point", each path expression has a number of such "interested points". For the prefix of a path expression ending with that "interested point" but without all previous interested points, we call it a segment. It is obvious the number of "interested points" is equal to the number of segments.

These segments P will be used to filter the elements (or attributes) whose path P are generated by these path segments. Take the following path expression as an example:

$$E_1/E_2[@A_2 = v_2]//E_3/E_4/E_5[@A_5 = v_5]$$

There are two "interested points" and

$$E_1E_2A_2$$

$$E_1E_2E_3E_4E_5A_5$$

are all possible segments for the above path expression.

Then the next question is how can we use this index to do faster XML query processing. Now let us consider the following XPath expression:

$$E_1/E_2[@A_2 = v_2]//E_3/E_4/E_5[@A_5 = v_5]$$

There will be four Element-Element joins and two Element-Attribute joins if we use the extended preorder labeling naively. But, we really do not need to make Element-Element Join between E_1 and E_2, E_3 E_4, E_4 and E_5. We only need to compute the ancestor relation for those E_2 and E_5 as they are the elements that we are interested in.

The idea is to have only one path join for two path segments here, as we have two "interested points" based on the selection of $@A_1 = v_2$ and $@A_5 = v_5$. (One segment for each "interested point" and predicate). Instead of going through the path joins one by one, we just do the join for two adjacent segments.

The remaining question is how to determine which element (attribute) matches the "interested point" in the path expression? We treat the path expression as a regular expression in which we omit the predicate but leaves the other parts untouched. Then we can just use traditional regular expression membership [1] to determine whether or not the absolute path for the element (attribute) can be generated by the path expression. For the above example, we can view the whole path expression as the following expression: $E_1E_2A_2$ and $E_1E_2 * E_3E_4E_5A_5$. We will explain this in more detail when we present the algorithm.

We can apply the following procedure to get the result:

1. Convert the tags in the regular path expression to designators and regular path expression into a regular expression;
2. Find all "interested points";
3. Divide the regular expression into segments based on the "interested points";
4. Convert each segment to a regular expression;
5. Find the bag that hold the the last element (or attribute) of the segment using regular expression membership procedure as in [1];
6. Do the Element-Element or Element-Attribute join as specified in [5].

Take the above expression

$$E_1/E_2[@A_2 = v_2]//E_3/E_4/E_5[@A_5 = v_5]$$

as an example, we follow the above procedure working on the expression as follows:

1. If we treat $E_1, E_2, E_3, E_4, E_5, A_2, A_5$ as designators, do nothing for this step;
2. As we have two predicate associated with A_2 and A_5, so A_2 and A_5 are two "interested points" in this expression;
3. Based on the "interested points": $[@A_2 = v_2]$ and $[@A_5 = v_5]$, and we have two segments:

   $$E_1/E_2[@A_2 = v_2]$$

   and

   $$E_1/E_2//E_3/E_4/E_5[@A_5 = v_5]$$

   it is easy to convert them into regular expressions as $E_1E_2A_2$ and $E_1E_2 * E_3E_4E_5A_5$
4. Use the regular expression membership procedure to determine which element's absolute path from the root can be generated by the regular expressions;
5. Use Element-Element, Element-Attribute, and Kleene-Closure Joins as presented in [5].

One comment is we do not need Element-Attribute Joins in most cases as this is included in the attribute's path. This will also improve the efficiency for query processing.

3. IMPLEMENTATION ISSUES

Due to time constraints, this method has not been fully implemented in an XML database management system, so there is no experimental comparison between this method and other existing methods.For implementation, we can have the same system architecture illustrated by Figure 2 as XISS in [5] by adding an intermediate layer, called path index to refine the elements (attributes) with the same name. We will have separate storage for those elements (attributes) having the same path, so this intermediate layer works as DataGuide while can handle regular path expression efficiently.

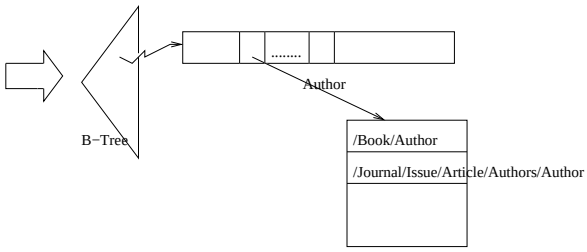

Figure 2: System Architecture

To process a path expression, first we follow the link to all the paths ending with the same element (or attribute) name. Then for each path p, we determine whether or not it can be generated by path expression P. If it is not, then we can just omit this bag of elements. If yes, we will use the same algorithm presented in [5] for Element-Element, Element-Attribute and Kleene-Closure join.

4. CONCLUSION AND FUTURE WORK

We presented a method using extended preorder labeling and path information to index XML data. It is shown that the running time does not depend on the length of the path expression in the query, but the number of "interested point"s. It combines the advantages of both path indexing method like DataGuide [6] and order indexing method like extended preorder [5]. Although this is a simple combination of these two existing indexing method, but to the author's knowledge, this has not been used in any known system to index XML data. It will be interesting to fully implement this method and compare the performance with the existing methods: Index Fabric and Extended Preorder.

5. REFERENCES

[1] J. U. A. Aho, J. Hopocroft. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass., 1974.

[2] D. F. D. Chamberlin, J. Robie. Quilt: An xml query language for heterogeneous data sources. In *Proceedings of WebDB 2000 Conference, Lecture Notes in Computer Science*, Springer-Verlag, 2000.

[3] M. F.-e. F. Cooper, N. Sample. A fast index for semistructured data. In *Proceedings of the 27th VLDB Conference*, Roma, Italy, 2001.

[4] S. D. J. Clark. *XPath Language (XPath)*. Version 1.0 W3C Recommendation, http://www.w3.org/TR/xpath, 1999.

[5] B. M. Q. Li. Indexing and querying xml data for regular path expressions. In *Proceeding of the 27th VLDB Conference*, Roma, Italy, 2001.

[6] J. W. R. Goldman, J. McHugh. From semistructured data to xml: Migrating the lore data model and query language. In *Proceedings of the 2nd International Workshop on the Web and Databases (WebDB '99)*, Philadelphia, Pennsylvania, 1999.

[7] M. F.-e. S. Boag, D. Chamberlin. *XQuery 1.0: An XML Query Language*. W3C Working Draft, http://www.w3.org/TR/xquery/, 2002.

[8] C. S.-M. E. M. T. Bray, J. Paoli. *Extensible markup language (XML) 1.0 second edition*. World Wide Web Consortium, 2000.

[9] M. O.-D. S. T. Green, M. Miklau. Processing xml streams with deterministic automata. In *Proceedings of ICDT*, 2003.