Using a Compact Tree to Index and Query XML Data

Qinghua Zou, Shaorong Liu, Wesley W. Chu
Computer Science Department
University of California – Los Angeles
{zou,sliu,wwc}@cs.ucla.edu

ABSTRACT
Indexing XML is crucial for efficient XML query processing. We propose a compact tree (Ctree) for XML indexing, which provides not only concise path summaries at group level but also detailed child-parent relationships at element level. Based on Ctree, we are able to measure how well XML data is structured. We also propose a three-step query processing method. Its efficiency is achieved by: (1) summarizing large XML data structures into a condensed Ctree; (2) pruning irrelevant groups to significantly reduce the search space; (3) eliminating join operations between the matches for value predicates and those for structure constraints and (4) using Ctree properties such as regular groups to reduce query processing time. Our experiments reveal that Ctree is an effective data structure for managing XML data.

Categories and Subject Descriptors
E.1 [**Data Structures**]: trees

General Terms
Algorithms, Measurement, Design, Performance

Keywords
Path summary, compact tree, XML index, XQuery processing, early pruning, inverted file clustering

1. INTRODUCTION
XML indexing is the key to the efficiency of XML query processing. The semi-structured nature of XML data and the flexible mechanisms of XML queries introduce new challenges to the existing database indexing methods. In this paper, we propose: (1) A novel compact tree, called Ctree, for indexing XML structures. Ctree is a two-level tree which provides a concise structure summary at its group level and detailed child-parent links at its element level which can provide fast access to elements' parents. Thus Ctree is an efficient index for processing the structure constraints of XML queries. (2) Group-based element reference instead of using global IDs. This enables us to cluster the entries in value inverted files by groups, which provides efficient evaluation of value predicates on a relevant Ctree group. The group-based element reference also facilitates the differentiation of the heterogeneous XML values by their groups and enables us to cluster similar element values and index them accordingly. (3) A Ctree-based query processing method. It can speed up query evaluation and prune search space at the earliest processing stage.

Copyright is held by the author/owner(s).
CIKM'04, NOVEMBER 8-13, 2004, WASHINGTON, DC, USA.
ACM 1-58113-874-1/04/0011.

2. CTREE
We model an XML document as an ordered labeled tree where nodes correspond to elements, and edges represent element-inclusion relationships. A node is represented by a triple (id, label, value), where *id*, *label* and *value* represent the node's identifier, tag name, and optional value respectively. For example, Figure 1 shows a sample XML data tree which has 19 nodes with identifiers in the circles and labels beside the circles. To differentiate values from sub-elements, we link a value to its corresponding node by a dotted line.

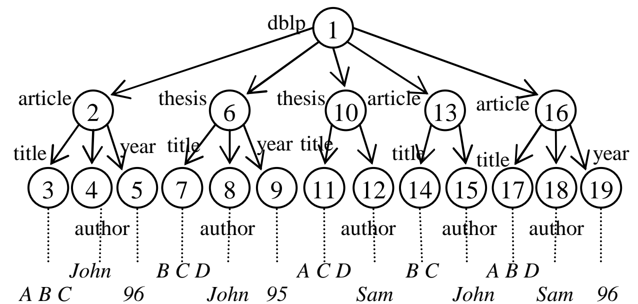
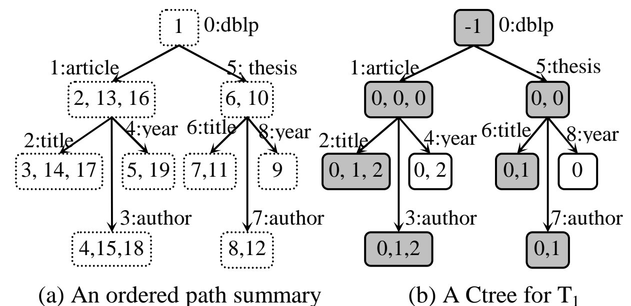


Figure 1: An Example of XML data tree T_1

For a data tree *D*, a path summary [1] is a tree in which each node corresponds to exactly one label path and contains all the equivalent nodes that share the label path. For example, a path summary for the XML data tree in Figure 1 is shown in Figure 2a. Each dotted box contains a group of node ids. Each group has a label and an identifier listed above the group. For example, data nodes 2, 13, 16 are in group 1 since their label paths are the same: *dblp.article*. Every data tree has a unique path summary [3].



(a) An ordered path summary (b) A Ctree for T_1

Figure 2: The path summary and the Ctree for T_1

As shown in past research, a path summary greatly facilitates the evaluation of single-path queries. For example, for a query Q1, */dblp/article/author*, the answers are data nodes 4, 15, and 18 because their label paths satisfy Q1. Path summary, however,

does not preserve the hierarchical relationships among individual data nodes. Therefore, it is unable to answer branch queries [2].

Thus, we propose Ctree which is a two-level tree containing a group level and an element level. At the group level, a Ctree provides a summarized view of hierarchical structures. At the element level, it preserves detailed child-parent links. Each group in a Ctree has an array mapping elements to their parents. For example, Figure 2b is the Ctree for the T_1 in Figure 1. Each group contains an array whose values are shown in the box separated by a comma and are indexed by nonnegative integers, called relative element ids. A relative element id together with a group id (gid) is called an element id. For example, the two elements in group 4 are referred to by 4:0 and 4:1, whose values 0 and 2 are relative element references for elements 1:0 and 1:2.

Every data tree has a corresponding Ctree, which can be created in two steps: (1) create a path summary; (2) replace node ids with the positions of their parents. For example, in Figure 2a, the positions of 2, 13, and 16 in group 1 are 0, 1 and 2. Thus they are mapped to elements 1:0, 1:1 and 1:2 respectively. Similarly, 5 and 19 in group 4 are mapped to 4:0 and 4:1. Since 16 is the parent of 19 in Figure 1, we replace 19 (4:1) with 2 (the relative element id for node 16) as shown in Figure 2b.

With the Ctree in Figure 2b, we can answer not only single-path queries but also branch queries. For example, for the query */dblp/article[title and year]*, elements 1:0 and 1:2 are the answers since the relative element ids 0 and 2 are contained both in group 2 and in group 4. An element id in Ctree contains path information (group id) which makes Ctree more efficient in query processing than other indexing methods.

3. QUERY PROCESSING

We model an XML query Q as a tree where nodes are the tags in Q and edges represent axes with a single arrow for a child axis "/" and a double arrow for a descendant axis "//". Filters in Q are represented by value predicates of the corresponding nodes. We assume that each query has only one return node as in the box. For example, Figure 3 represents of the following query (Q4):

/dblp/article [contains(.//author, "John") and year> 94]/title

In this example, a user is interested in titles of the articles under *dblp* which have descendant elements (*author*) containing "John" and sub-elements (*year*) with a value greater than 94. The dotted arrow beside the node indicates the result's projecting direction.

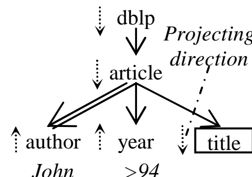


Figure 3: A query tree

After a query is transformed into a tree Q, we can evaluate Q using Ctree index data T in three steps as shown in Figure 4.

First, it locates a set of frames for Q, where each frame is an assignment of Ctree groups in T to the query nodes in Q that satisfy the structure of Q at the group-level (Line 1). The FrameFinder (Fig 4) finds frames in a top-down fashion starting from candidate groups for the root of the query tree down to the leaves.For example, there is one frame consisting of groups (0, 1, 3, 4, 2) in the Ctree (Figure 2b) for Q4, which are matches to query nodes (*dblp*, *article*, *author*, *year*, *title*) respectively. Notice that by assigning gid 3 to author, we exclude other elements which also have the tag name author (e.g. group 7) and thus reduce search space.

Input: *T*, a Ctree with value index
Q, a query tree
Output: A list of elements in *T* that satisfy the *Q*.
QueryProcessor(*T*, *Q*)
1 Evaluate group level structure constraints:
Call *FrameFinder* to get a list of frames.
2 For each frame, do
3 Evaluate value constraints on the frame.
4 Evaluate element level structure constraints:
Call *ElmEvaluator* to a list of matched elements;
5 Output the list of elements;

Figure 4 A Ctree-based query processing algorithm

Second, for each frame, the query processing algorithm evaluates value predicates using value indexes to determine which elements satisfy the predicates (Line 3). Value indexes support the Search(value, gid?) operation. For example, there are two value predicates in Q4: author="John" and year>94. For the first predicate, it calls Search("John", 3) since author is mapped to group 3 in step 1. Elements 3:0 and 3:1, data nodes 4 and 15 in Figure 1, are retuned. Similarly, element 4:0, data node 5, is returned for the second value predicate.

Finally, it evaluates element level structure constraints and returns the results. For example, the second step for Q4 determines that elements {3:0, 3:1} and {4:0} satisfy value constraints. The answers can be determined by projecting relevant elements from group 3 and 4 to the target group 2. We first project groups 3 and 4 upward to group 1 and get the answer {1:0} since the element 1:0 is the parent of both 3:0 and 4:0. Then we project group 1 downward to group 2 and return the result {2:0} since element 2:0 is the only child of the element 1:0.

4. CONCLUSIONS

In this paper, we propose a compact tree, Ctree, for indexing XML data. Ctree is a two-level representation of an XML data tree: group level and element level. The group level provides concise path summaries and the element level provides detailed child-parent relationships in an XML data tree. The group-based element reference facilitates stepwise early pruning, efficient value processing. Furthermore, Ctree is able to capture one-to-one parent-child relationships (the shaded box in Figure 2b) Our experimental studies [3] reveal that Ctree significantly outperforms other index methods in query processing.

5. REFERENCES

[1] R. Goldman and J. Widom. *Dataguides: Enabling query formulation and optimization in semistructured databases*. In *VLDB*, 1997.

[2] S. Liu, Q. Zou, and W. W. Chu. *Configurable Indexing and Ranking for XML Information Retrieval.* In *SIGIR*, 2004.

[3] Q. Zou, S. Liu, and W. Chu. *Ctree: A Compact Tree for Indexing XML Data*. In *WIDM*, 2004.