# Ctree: A Compact Tree for Indexing XML Data

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

## Abstract
In this paper, we propose a novel compact tree (Ctree) for XML indexing, which provides not only concise path summaries at the group level but also detailed child-parent links at the element level. Group level mapping allows efficient pruning of a large search space while element level mapping provides fast access to the parent of an element. Due to the tree nature of XML data and queries, such fast child-to-parent access is essential for efficient XML query processing. Using group-based element reference, Ctree enables the clustering of inverted lists according to groups, which provides efficient join between inverted lists and structural index group extents. Our experiments reveal that Ctree is efficient for processing both single-path and branching queries with various value predicates.

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

## General Terms
Algorithms, Measurement, Performance, Experimentation

## Keywords
XML index, path summary, XQuery evaluation, value index, Ctree

## 1. Introduction
With the growing popularity of XML, an increasing amount of information is being stored and exchanged in the XML format. XML is essentially a textual representation of the hierarchical (tree-like) data where a meaningful piece of data is bounded by matching tags, such as <name> and </name>. To cope with the tree-like structures in the XML model, several XML-specific query languages have been proposed recently (e.g., XPath, XQuery) to provide flexible query mechanisms. An XML query typically consists of two parts: structure constraints and value predicates. Structure constraints are usually represented by a tree, which can have either a single-path or multiple branches. Value predicates can be comparison predicates (e.g., >, <, =) or containment predicates (e.g., *contains*).

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.

First, it is expected that a structure index is to be a covering index such that it alone can answer both single-path and branching queries without consulting the XML data. Many previous approaches can be classified into the following three categories: 1) Path indexing [7][6][5][3], creates a path summary from XML data. Path indexing speeds up the evaluation of single-path queries. Path indexing greatly speeds up the evaluation of single-path queries but needs expensive join operations for processing queries with multiple branches. 2) Node indexing [13][23], indexes each data node by some numbering schemes. The structure relationship between a pair of nodes can be determined in constant time in node indexing, but relying on the pair-wised comparisons to answer a query sometimes is inefficient. 3) Sequence-based indexing [20][17], transforms both XML documents and queries into sequences, and evaluates queries based on sequence matching. It supports flexible queries without join operations, but false alarms may exist in query results since a sequence match is not necessarily a tree match.

Second, values in XML documents are usually heterogeneous, including many types of data such as date, number, and text. Little research has been done on using a variety of indexing types for the heterogeneous XML values. Some approaches index a value as an entire string, and some use stemming words and build inverted files for value indexing. Such uniform value processing is not suitable for the heterogeneous nature of XML values. For example, while stemming is applicable to the text of an article, stemming authors' names will produce undesirable results.

Finally, using global IDs such as pre-orders to refer to elements requires join operations between the matches for value predicates and structure constraints. Such references do not carry any semantic meaning, i.e., a pre-order itself does not give any information about its label path and tag name.

To address the above challenges, we propose:

- 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.
- 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.

- We propose a Ctree-based query processing method that can speed up query evaluation and prune search space at the earliest processing stage.

We have conducted a set of experiments to compare the effectiveness of Ctree with path indexing and node indexing approaches. Our study reveals that Ctree is about an order of magnitude faster on most test queries.

The paper is organized as follows. Section 2 introduces our XML data model and the definition of Ctree. In Section 3 we present Ctree properties. In Section 4 we overview a framework for building the Ctree index. Section 5 gives the Ctree-base query processing method. In Section 6 we present experimental results which show the effectiveness of Ctree. Section 7 reviews related works.

## 2. Introduction of Ctree

In this section, we will first present the XML data model and path summary, and then introduce Ctree.

### 2.1 XML Data Model

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* represents its 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.

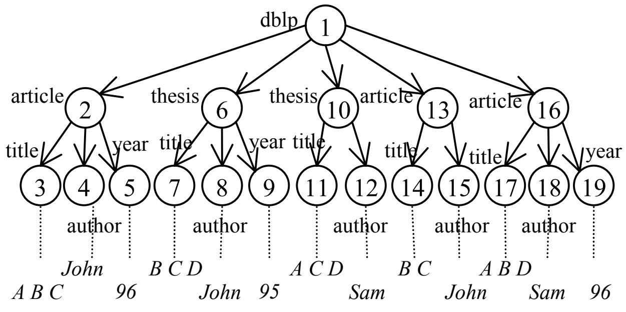

**Figure 1: An example of XML data tree $T_1$**

We now introduce the definitions of a label path and equivalent nodes, which are useful for describing a path summary and a Ctree.

**Definition 1** A *label path* for a node *v* in an XML data tree *D*, denoted by *L(v)*, is a sequence of dot-separated labels of the nodes on the path from the root node to *v*.

For example, node 8 in Figure 1 can be reached from the root node 1 through the path: 1→6→8. Therefore the label path for node 8 is *dblp.thesis.author*.

**Definition 2** Nodes in an XML data tree *D* are *equivalent* if they have the same label path.

For example, nodes 8 and 12 in Figure 1 are equivalent since their label paths are the same *dblp.thesis.author*.

The parents of equivalent nodes share the same label path and thus are equivalent. For example, the parent nodes of 8 and 12 are nodes 6 and 10 respectively, which are equivalent.

### 2.2 Path Summary

For a data tree *D*, a *path summary* as in [7][2] is a tree on which each node is called a *group* and corresponds to exactly one label path *l* in *D*. The group contains all the equivalent nodes in *D* sharing the label path *l*. We call a path summary an *ordered path summary* if the equivalent nodes in every group are sorted by their pre-order identifiers.

For example, an ordered path summary for the XML data tree in Figure 1 is shown in Figure 2a. Each dotted box represents a group and the numbers in the box are the identifiers of equivalent data nodes. 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 [7].

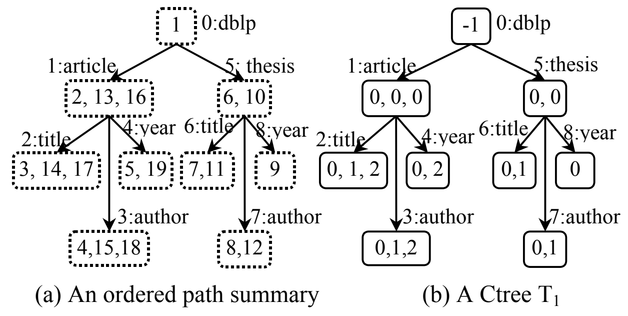

**Figure 2: The path summary and the Ctree for $T_1$**

As shown in past research, a path summary greatly facilitates the evaluation of single-path queries. For example, for a query $Q_1$, /*dblp*/*article*/*author*, the answers are data nodes 4, 15, and 18 because their label paths satisfy $Q_1$.

The path summary, however, does not preserve the hierarchical relationships among individual data nodes. Therefore, the path summary is unable to answer branching queries. For example, in Figure 2a, there is no information to indicate which data node in group 1 is the parent of data node 4 in group 3. Such node-level relationships are important for answering branching queries. For example, for a query "find *articles* under *dblp* with both *title* and *year*," i.e. /*dblp*/*article*[*title* and *year*] ($Q_2$), the path summary (Figure 2a) indicates that elements in group 1 are candidate answers but does not provide the information about which elements in the group actually have both *title* and *year*.

This motivated us to propose an index structure which provides not only a path summary but also detailed element-level relationships.

### 2.3 The data structure of Ctree

Ctree is a bi-level tree containing a *group level* and an *element level*. At the group level, Ctree provides a summarized view of hierarchical structures. At the element level, Ctree preserves detailed child-parent links. Each group in Ctree has an array mapping elements to their parents.

**Definition 3** A *Ctree* is a rooted tree where each node *g*, called a *group*, contains an array of *elements* denoted as *g.pid*[] such that:

1) Each group $g$ is associated with an *identifier* and a *name*, denoted by *g.id* and *g.name* respectively.

2) Edge directions are from the root to the leaves. If there is an edge from $g_1$ to $g_2$, then $g_1$ is called the *parent* of $g_2$ and $g_2$ is called a *child* of $g_1$. If there is a path from $g_1$ to $g_3$, then $g_1$ is called an *ancestor* of $g_3$ and $g_3$ is called a *descendant* of $g_1$.

3) An array index $k$ of *g.pid*[] represents an *element* in $g$, denoted by *g:k*. The value of *g.pid*[$k$] points to an element in *g's* parent $g_p$; and $g_p$:*g.pid*[$k$] is called the *parent element* of *g:k*.

4) For any two elements $g{:}k_1$ and $g{:}k_2$, if $k_1<k_2$, then $g.pid[k_1] \leq g.pid[k_2]$.

An *ordered Ctree* is a Ctree where sibling groups are ordered.□

In Ctree, the groups carry semantic meanings (label paths) and the array in a group stores child-parent links which provides fast access to the parents of a list of elements. This is in contrast to using individual child-to-parent pointers which will be very time consuming for a large XML data tree.

**Definition 4** For referring an element $k$ in group $g$, *g:k* is called a *absolute reference* and $k$ is called a *relative reference*.

For example, Figure 2b is a sample Ctree. There is an array in each group. The array values are shown in the box separated by a comma. The array indexes are the positions of the values numbered starting from 0. The two elements in group 4 are referred to by 4:0 and 4:1, whose values are 0 and 2 which are relative references for elements 1:0 and 1:2.

**Theorem 1** Every data tree $D$ has a unique Ctree $T_D$.

***Proof***: We can construct a $T_D$ in the following three steps:

1) Create a path summary $S$ for D.

2) Replace the collection $C$ of equivalent nodes in every group $g$ of $S$ with a array *g.pid*[] of the same size as $C$ and build the mapping $M$: $d \rightarrow g{:}k$ where $d$ is a data node and $k$ is its position in $C$.

3) Build element level pointers. Given $d$ is the parent of $d'$ in $D$, if $d \rightarrow g_1{:}k_1$ and $d' \rightarrow g_2{:}k_2$, then let $g_2{:}k_2$ point to $g_1{:}k_1$, i.e., $g_2.pid[k_2]= k_1$.

Using the above steps, only one $T_D$ can be constructed. From Section 2.2, we know there is a unique path summary $S$ for $D$ in step 1. In step 2, the mapping $M$ is uniquely determined. Since every node $d'$ has at most one parent, step 3 has only one assignment for each elements. Thus Theorem 1 holds. □

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 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 let 4:1 (19) point to 1:2 (16) in Figure 2b, i.e., the value of 4:1 is 2.

With the Ctree in Figure 2b, we can answer not only single-path queries but also branching queries. For example, for the query */dblp/article*[*title and year*], elements 1:0 and 1:2 are the answers since the boxes in groups 2 and 4 contain values 0 and 2. A Ctree has parent-child edges at the group level and provides child-parent links at its element level. Such a bidirectional tree makes Ctree better than other indexing methods.

## 3. Ctree Properties

### 3.1 Monotonic relationship

By the definition of Ctree, we know that the elements in a group are arranged in consistent with the order of their parents. In other words, for two elements $i$ and $j$ in a group $g$, if $i$ precedes $j$, then $i$'s children precedes $j$'s in every child group of $g$.

**Observation 1** *Monotonic property*: The values of a group's array are arranged in increasing order. That is, if $i<j$, then $g.\text{pid}[i] \leq g.\text{pid}[j]$.

This property enables us to use a binary search to locate the child elements of a given element. It also results in the contiguous property.

**Theorem 2** *Contiguous property*: Let $g'$ be a descendant group of $g$ and $E$ be a list of contiguous elements in $g$, then the descendant elements of $E$ are contiguous in $g'$.

***Proof*** Let $y=anc(x)$ be the function of mapping an element $x$ in $g'$ to its ancestor $y$ in $g$ as shown in Figure 3. Suppose Theorem 2 is not true, then there exists $d_1<d<d_2$ and $anc(d)$ is not in the range $a_1$ to $a_2$. Recursively applying the monotonic property, we know *anc* is an increasing function, i.e., $a_1 \leq anc(d) \leq a_2$. It conflicts with the assumption. Thus Theorem 2 holds.□

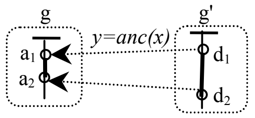

**Figure 3:** ***y=anc(x)*** **is an increasing function.**

In a situation where we want to find the descendants of a large number of contiguous elements, we only need to determine the upper and lower bounds of the descendants. For example, if we know $d_1$ and $d_2$ are descendants of $a_1$ and $a_2$, then all elements in the range of $d_1$ to $d_2$ are the descendants of the elements $a_1$ to $a_2$.

### 3.2 Regular and irregular group

Let $g$ be a group and $g_c$ be one of its child groups.

**Definition 6** If every element in $g$ has the same number of children in $g_c$, then $g_c$ is called a *regular group*. Otherwise, $g_c$ is called *irregular group*.

For example, in Figure 2b, groups 1, 2, 3, 5, 6, and 7 are regular groups. Group 4 and 8 are non-regular groups. The root group (group 0) is a regular group since it has no parent.

Such information is useful for query optimization. For example, for a query "find *article* elements that have child elements *title* and child elements *year*", i.e. *//article*[*title and author*]($Q_3$), the Ctree directly returns all elements in group 1 as answers without further checking the element-level links since groups 2 and 3 are regular groups.

The array in a regular group can be removed since the content of the array can be inferred from group sizes (number of elements in a group). Therefore, a Ctree not only provides more information but is also smaller in size than a path summary. In Figure 2b, we only need to keep the 3 numbers in group 4 and group 8 and can remove the 16 numbers in the other 6 groups (shaded). This significantly reduces the size of a Ctree. As we will see in Section 6, regular groups are common in XML datasets.

## 3.3 Group-based element reference scheme

Most previous approaches use global pre-orders for referring elements. In contrast, Ctree uses group-based element reference (i.e.., *g:k*) which provides the following advantages:

- Feasible in supporting stepwise early pruning of a large search space. To determine a candidate answer *g:k*, we can first determine a relevant group *g* to eliminate irrelevant groups and then determine a relative reference *k*.
- Efficient in processing of value predicates. Value inverted files can be sorted by *g:k*, i.e., first by *g* and then by *k*, so that values of equivalent nodes are clustered together. Many previous methods, however, sort inverted files by pre-orders, which requires a scan of all the elements in an inverted file to determine which elements satisfy a given label path.
- Ease in differentiating the heterogeneous XML values. Values of the elements in a group are usually of the same type. For example, the values for the elements in group 4 (*dblp.article.year*) are all numbers, while the values for *dblp.article.title* are all strings.
- Efficient in maintaining XML data updates. Inserting an element *e* into a group *g* in a Ctree affects only the elements after *e* in *g* and has no effect on other groups. Using the global pre-order approach, inserting an element *e* into XML data incurs updating the identifiers of all the elements after *e*.

## 3.4 Element ordering in Ctree

Since XML data are usually stored in files, we need to know the element ordering in a Ctree. For two elements $k_1$ and $k_2$ in the same group *g*, we know $g{:}k_1$ precedes $g{:}k_2$ in *D* if $k_1<k_2$. Suppose $g_1{:}k_1$ and $g_2{:}k_2$ are from different groups, let $g{:}k_1'$ and $g{:}k_2'$ be their ancestors in group *g* which is the lowest common ancestor group of $g_1$ and $g_2$. The ordering of the two elements can be determined by the following definition.

**Definition 5** $g_1{:}k_1$ precedes $g_2{:}k_2$, denoted as $g_1{:}e_1 \ll g_2{:}e_2$, if

1) $k_1'<k_2'$, or
2) $k_1'=k_2'$ and $g_1.gid<g_2.gid$.

For example, in Figure 2b, 2:0«4:0 since they have the same ancestor 1:0 and 2 is less than 4; 4:0«2:1 since 4:0's ancestor (1:0) precedes 2:1's (1:1).

By definition 5, an ordered Ctree implies a total ordering of elements. In the case that elements from different groups have the same ancestor, we use group IDs to infer the ordering of elements. In the situation that equivalent data nodes have inconsistent ordering of sub-elements, the inferred ordering may be incorrect. For example, if one *article* has a sub-element *title* before a sub-element *author* but another *article* has the reverse order, then no ordered Ctree can be built because we cannot assign *gids* to the two sub-groups *author* and *title*. In such cases, we can use elements' start positions (see section 4) for determining their ordering.

## 4. Building Ctree Index

We first present a framework for building Ctree index. Then we briefly discuss Ctree index data and value index searching.

### 4.1 A framework for building Ctree index

XML data contains not only heterogeneous values but also tags with different purposes such as semantic tags (e.g., *title*, *author*) and presentation tags (e.g., *bold*, *italic*). Thus we use a configurable indexing framework to build Ctree index, which allows user to specify index options such as ignorable tags, value treatments and value indexing types.

Figure 4 shows the system structure for the Ctree index which consists of three function parts (*Scan*, *IndexBuilder*, and *Query Processor*) and three data parts (*XML Data* and optional *schema*, *Spreadsheet*, and *Ctree Index Data*).

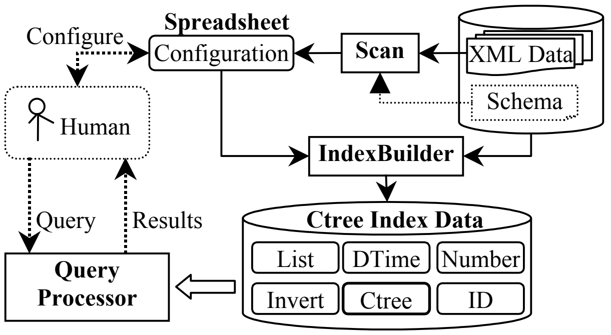

**Figure 4: System structure of Ctree index**

Note that the user is not required to set indexing options since the default indexing options are automatically generated based on some rules of data statistics. Ctree index can be built in three steps.

1. The *Scan* module collects the structure and value characteristics from an XML dataset and extracts schema information if it exists. The *Scan* also proposes indexing options for each group of equivalent data nodes based on data features and schema.
2. A user reviews the proposed indexing options and makes proper adjustments to finalize the index configurations.
3. Based on the index configurations, the *Index Builder* constructs a Ctree and builds value indexes for the XML dataset.

When a query arrives, *Query Processor* evaluates the query based on *Ctree Index Data* which includes both structure index and five value indexes, and returns query results to the user. We present a Ctree-based query evaluation process in Section 5.

### 4.2 Ctree index data

Ctree index data can be modeled in relational tables so that it can be implemented in relational database. For the simplicity of tabular data, Ctree can also be transformed into native XML databases or be stored in disk files with the B+ index for fast access.

Ctree structure index can be mapped into four tables: *Elements*, *Groups*, *CtreeDB*, and *ElmPosLen*. The *Elements* table stores the mappings from elements to their parents. The *Groups* table stores the group-level tree by *gid*, *sub_num*(the number of descendant groups), *level* (the depth of the group), and *pgid* (parent group). It also stores the group name and the label path (*lp*). The *CtreeDB* table has one row for each Ctree describing the main features of the Ctree including the Ctree name, the file group (*fgrp*), the number of groups, and elements. The *fgrp* indicates in which group the element IDs are the same as XML file IDs. In other words, each element in the *fgrp* is associated with an XML document. The *ElmPosLen* table records the position and length of each element, which is useful for retrieving the element.

The various data types in XML data require multiple value indexing types. Therefore we propose five types of value indexes (*Invert*, *List*, *Number*, *DTime* and *ID*) which can be extended for new requirements. The *Invert* uses the table *Words* to map a word to an identifier (*wid*) which minimizes storage overhead by eliminating replicated strings and computational overhead by eliminating expensive string comparisons. The table *Hits* stores the occurrences and positions (*pos*) of words (*wid*) in XML elements (*gid:eid*). Similarly, we use two tables *Phrases* and *List* for the value type *List*. For *Number*, *DTime*, and *ID*, we use three tables *Number*, *DTime*, and *Idref* respectively to record element values which are transformed to the corresponding types in the indexing phase.

The *XMLfiles* stores all the XML documents of the Ctree which are required if a user wants to look up the source of an element.

### 4.3 Value index searching

All the five value indexes support a *search*(*value, gid?*) operation where the question mark indicates an optional input parameter, and the *value* and *gid* corresponds to a specific value and a certain group respectively. The *search* operation returns a list of absolute elements (when the *gid* is not specified) or relative elements (when the *gid* is specified). Since the *Invert* value index is clustered by (*wid*, *gid*, *eid*), the operation *search*(*wid*, *gid*) can be computed very efficiently once the *value* is mapped to a *wid*. For Ctree-based query processing, we first determine *gid* to eliminate irrelevant groups by group-level structure mapping and then evaluate value predicates (Section 5). Thus, the returned answers and the I/O cost are reduced.

In order to examine the details of XML elements, we need to refer to the XML source. We are able to retrieve an element from an XML document since Ctree stores the elements' location in the document.

## 5. Query Processing

There are many possible ways to evaluate a query on a Ctree. For example, previous query processing based on a node index is also applicable to Ctree since the table *ElmPosLen* provides the position and length for each element. To take full advantages of Ctree characteristics, we propose a Ctree-based query processing method. Let us first introduce our query model.

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 query node returned, *target node*, which is emphasized with a box. For example, Figure 5 is a tree representation of the following query ($Q_4$):

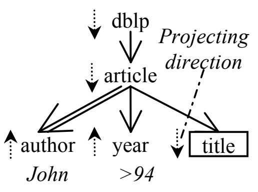

**Figure 5: A query tree**

*/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 (Section 5.3).

### 5.1 Ctree-based query processing

After a query is transformed into a tree *Q*, we can evaluate it using Ctree index data *T* in three steps as shown in Figure 6.

```
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 6: A Ctree-based query processing algorithm**

First, it locates a set of frames matching *Q*'s tree structure, 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). For example, there is one frame consisting of groups (0, 1, 3, 4, 2) in the Ctree (Figure 2b) for $Q_4$, which match query nodes (*dblp*, *article*, *author*, *year*, *title*) respectively. Notice that by assigning *gid* 3 to the query node *author*, we exclude other elements which also have the tag name *author* (e.g. elements in group 7) and thus reduce search space. The *FrameFinder* finds frames in a top-down fashion starting from candidate groups for the root of the query tree down to the leaves.

Second, for each frame, it evaluates value predicates using value indexes to determine which elements satisfy the predicates (Line 3). As discussed in Section 4, all value types support the *Search(value, gid?)* operation. For example, there are two value predicates in $Q_4$: *author*="John" and *year*>94. For the first predicate, it calls *Search("John", 3)* since the query node *author* is mapped to group 3 in step 1. Elements 3:0 and 3:1 (i.e. data nodes 4 and 15 in Figure 1) are retuned. Similarly, element 4:0 (i.e. data node 5) is returned for the second value predicate.

Finally, it evaluates element level structure constraints and returns the query results to the user. For example, for the frame (0, 1, 3, 4, 2) for $Q_4$, the second step of our query processor determines that elements {3:0, 3:1} and {4:0} satisfy value constraints. Now the last step is to determine which elements in the target group 2 can answer $Q_4$. The answers can be determined by projecting relevant elements from other nodes to the target node. The projecting direction for a query node can be either downward or upward depending on its position in the query tree. If a query node is an ancestor of the target node, its projecting direction is downward. Otherwise, it is upward. For example, in Figure 5, the projecting directions for *dblp*, *article*, and *title* are downward while those for *author* and *year* are upward. Since Ctree stores detailed child-to-parent relationships, upward projecting is straight forward by joining common parent elements. Downward projecting can be done similar to upward projecting by keeping track of children.

The proposed Ctree-based query processing algorithm has the following advantages:

- Locating frames at the group level prunes a large number of irrelevant groups at an early stage. If there is no group-level match, it returns empty answer in step 1. For example, for a query *//article[author ="John"]/address* on DBLP, it will return no matches at the first step since the path *//article/address* does not exist in the Ctree of the DBLP dataset. Many previous approaches, however, require a set of expensive join operations to return the no answer.
- Evaluating a value predicate based on a group significantly reduces the possible matches and improves the efficiency of combining the matches for value predicates and structure constraints.
- Using an array for fast mapping elements to their parents facilitates the evaluation of element level structure constraints. Such a fast access to elements' parents is essential for efficient XML query processing since the tree nature of XML data and queries is rooted on sharing common parents.

## 6. Performance Evaluation

To validate the efficiency of Ctree in XML indexing, we tested two datasets DBLP [11] and XMARK[25], and compared the performance of Ctree with that of some previous methods. We implemented Ctree and several other methods in C# for XML indexing. Experiments were run on a 2.8 GHz PC-compatible machine with 1GB of RAM running Windows XP.

Ctree and value indexes can be resided in relation database or disks as in [24] where we have used Ctree to index INEX'03. In our experiment, Ctree and value indexes are compiled into objects which reside in memory, and thus queries can be answered without any I/O access. For comparison purpose, we have implemented a path index method similar to Index Fabric [6] and a node index method similar to XISS [13] where values are indexed by inverted files. We load each index data into the RAM before testing so that no IO operations for reading the index data are required.

### 6.1 XML test dataset characteristics

Table 1 shows the properties of the two datasets DBLP and XMARK. The configuration files for specifying index options can be downloaded from our website. The main characteristics of these datasets are illustrated below.

DBLP is a popular computer science bibliography dataset with a maximal depth of 6 and over 3.7 million of elements. XMARK is a synthetic on-line auction dataset which is relatively deep and contains over 2 million elements.

**Table 1: Characteristics of test datasets**

| | DBLP | XMARK |
|---|---|---|
| Size (MB) | 134 | 117 |
| Max depth | 6 | 12 |
| Group# | 119 | 548 |
| Regular group# | 42 | 164 |
| Group name# | 36 | 77 |
| Elm# | 3733320 | 2048193 |
| Elm# in regular group | 1640391 | 1326827 |
| Percentage | 43.9% | 64.8% |

It is interesting to note that there are about half the elements in the two datasets belonging to regular groups (43.9% for DBLP and 64.8% for MARK). Regular groups reduce index size and optimize query processing.

Table 2 shows the space requirements for three indexing approaches: Ctree, Index Fabric and XISS. We noticed that XISS requires slightly more space on both datasets than Index Fabric.

**Table 2: Index file size (M bytes)**

| Size (MB) | Ctree | Index Fabric | XISS |
|---|---|---|---|
| DBLP | 78.1 | 102.9 | 117.5 |
| XMARK | 27.5 | 46.3 | 54.2 |

Ctree requires significantly less space than Index Fabric and XISS for two main reasons. First, both DBLP and XMARK have a large number of elements in regular groups for which Ctree does not need to keep element-level links. Second, the multiple value index types in Ctree also reduce some space overhead. For example, representing a string "$1,234,567.99" by a number reduces value index size.

### 6.2 Experimental results on DBLP dataset

We use the same set of queries for DBLP as in ViST [20] with some slight changes on value predicates as shown in Table 3.

**Table 3: Sample queries for DBLP**

| | Description | Answer# |
|---|---|---|
| $Q_1$ | /inproceedings/title | 212,273 |
| $Q_2$ | /book/author[contains(., "David")] | 27 |
| $Q_2$ | /*/author[contains(., "David")] | 13,218 |
| $Q_4$ | //author[contains(., "David")] | 13,218 |
| $Q_5$ | /article[contains(./author,"David") *and* ./year=1995] | 258 |
| $Q_6$ | /article[contains(./author,"David") *and* ./year≥1995] | 2,195 |

The performances of the three indexing approaches are illustrated in Figure 7. $Q_1$ is a single path query without value predicates. Ctree and Index Fabric have similar performances for $Q_1$ while XISS takes a longer time since it requires join operations.

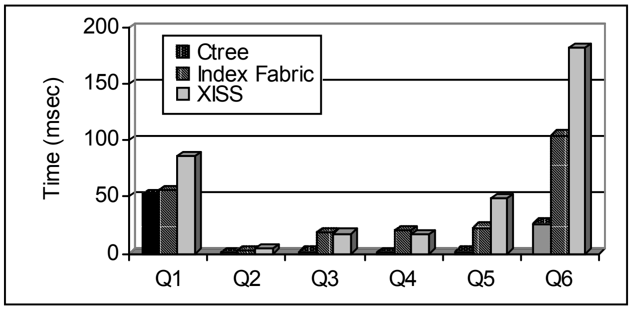

**Figure 7: Performance comparison of Ctree with Index Fabric and XISS on DBLP**

For the other queries, Ctree significantly outperforms Index Fabric and XISS. One reason is that all the five queries have a value predicate containing "*David*" which slows down both Index Fabric and XISS. This is because both Index Fabric and XISS require a join operation between the elements for *author* and the elements containing "*David*."

Since $Q_2$ has only 27 answers, the clustered value inverted file results in a speedup in the performance of Ctree by two orders of magnitude. For $Q_3$ and $Q_4$, Index Fabric and XISS took 15 times more time to get the answers than Ctree since they require many join operations for processing the wildcard or the AD edge ("//"). $Q_5$ and $Q_6$ are queries with two branches and involve numeric predicates. Ctree again significantly outperforms the other two methods. The smaller the answer list, the more the performance gains Ctree achieves since the time for composing the answer list is constant for all the methods.

### 6.3 Experimental results on XMARK dataset

In this set of experiments, we also tested all the 20 benchmark queries in XMARK, and compared Ctree's performances with those of Index Fabric and XISS. Due to space limitations, we randomly picked six of them as shown in Table 4. $Q_2$ and $Q_4$ are designed to test the performance of ordered access with $Q_2$ for element indexes and $Q_4$ for tag orders. $Q_{15}$ and $Q_{16}$ are to evaluate the performance of long path traversals. $Q_{18}$ is a converting applications and $Q_{20}$ is for aggregations.

**Table 4: Sample queries for XMARK**

| | Description | Answer# |
|---|---|---|
| $Q_2$ | Return the initial increases of all open auctions. | 10,830 |
| $Q_4$ | List the reserves of those open auctions where *person18829* issued a bid before *person10487*. | 2 |
| $Q_{15}$ | Print the keywords in emphasis in annotations of closed auctions. | 180 |
| $Q_{16}$ | Return the IDs of those auctions that have one or more keywords in emphasis. | 160 |
| $Q_{18}$ | Convert the currency of the reserve of all open auctions to another currency. | 5,922 |
| $Q_{20}$ | Group customers by their income and output the cardinality of each group. | 1 |

As shown in Figure 8, for all six queries we randomly picked, Ctree outperforms the other two methods by at least one order of magnitude due to fast access to elements parents and group-based value index. For the queries $Q_4$, $Q_{18}$ and $Q_{20}$, Index Fabric and XISS are further slowed down by the lack of proper value indexes for numbers. This is because Ctree index creates various value index types while the other two methods have only string data type.

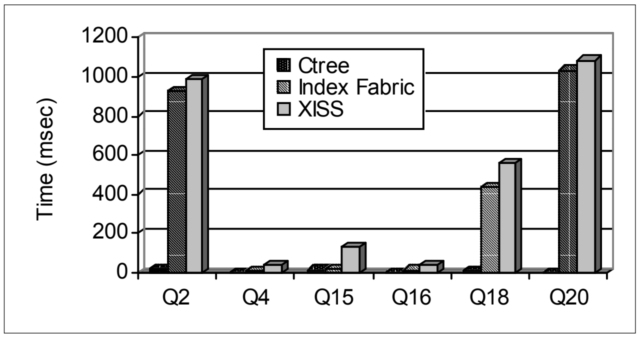

**Figure 8: Performance comparison of Ctree with Index Fabric and XISS on XMARK**

## 7. Related Works

Indexing and querying XML data is one of the major research fields in recent years. There are currently three major approaches for indexing XML data: node indexing, path indexing and sequence-based indexing.

Node index approaches [13][23] create indexes on each node by its positional information within an XML data tree. Such index schemes can determine the hierarchical relationships between a pair of nodes in constant time. Also, they use a node as a basic query unit, which provides great query flexibility. Any tree-structure query can be processed by matching each node in the query tree and structurally joining these matches. Structural join algorithms[18][4][14] have been proposed recently to support efficient query answering.

Path index approaches create path summaries for semi-structured data to improve query efficiency. DataGuides [7] indexes each distinct raw data path to facilitate the evaluation of simple path expressions. The Index Fabric approach [6] indexes frequent query patterns which may contain "//" or "*", in addition to raw data paths. APEX [5] and D-(k) [3] are two adaptive path approaches that apply data mining algorithms to mine frequent paths in the query workload and build indexes accordingly. In case of changes in the query workload, the structure summaries are updated accordingly. The structure summary of D-(k) is also adaptive to updates in XML documents. To handle all kinds of branching path expressions, F&B index approach [1] indexes each edge in an XML data tree both forward and backward. But it is usually too big to be practical. To overcome this problem, F+B approach [8] reduces index size by ignoring unimportant tags and edges, limiting the depths of branching queries.

Sequence-based indexing approaches [20][17] transform XML documents and queries into structure-encoded sequences. They leverage on the well-studied sub-sequence matching techniques to find query answers. Since sequence index approaches use the entire query tree as the basic query unit, they avoid the expensive join operations and support any tree-structure XML queries.

In addition, value indexes are essential for efficient processing queries with value conditions. Most previous works create inverted indexes on XML values. We also create inverted index on values but we further cluster inverted indexes according to their incoming label paths or groups. During our manuscript preparation, we notice that [11][21]combine some information retrieval techniques with XML indexing, which is similar to our value index approach.

## 8. Conclusion

In this paper, we proposed a compact index tree, Ctree, for indexing XML data. Ctree provides concise path summaries at its group level and detailed child-parent relationships in arrays at its element levels. The array in a group provides direct mapping from elements to their parents. Such a fast child-to-parent access is essential for efficient branching query processing. We proposed a Ctree-based query processing method that enables early pruning of a large search space. Ctree is able to capture one-to-one parent-child relationships (regular groups) and has monotonic relationships, which can be used to speed up query evaluation. In addition, instead of using global IDs, we proposed group-based element reference which facilitates stepwise early pruning, efficient value processing, heterogeneous values differentiation, and efficient XML data updating. We conducted a set of experiments to evaluate the effectiveness of Ctree. Our studies reveal that Ctree significantly outperforms Index Fabric and XISS for all the tested queries. We have also successfully applied Ctree index to an application example in INEX'03.

## Acknowledgement

This research is supported by NSF IIS ITR Grant # 0219442.

## References

[1] S. Abiteboul, P. Buneman, and D. Suciu. Data on the web: from relations to semistructured data and XML. *Morgan Kaufmann Publishers*, Los Altos, USA, 1999.

[2] A. Arion, A. Bonifati, G. Costa and S. D Aguanno. Ioana Manolescu, Andrea Pugliese: Efficient Query Evaluation over Compressed XML Data. *EDBT*, 2004.

[3] Q. Chen, A. Lim, and K. Ong. D(k)-Index: An adaptive Structural summary for graph-structured data. *ACM SIGMOD*, 2003.

[4] S.-Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo. Efficient Structural Joins on Indexed XML Documents, *VLDB*, 2002.

[5] C. Chung, J. Min, and K.Shim. APEX: An adaptive path index for XML data. *ACM SIGMOD*, 2002.

[6] B. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason, and M. Shadmon. A fast index for semistructured data. *VLDB*, 2001.

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

[8] R. Kaushik, P.Bohannon, J. Naughton, and H. Korth. Covering indexes for branching path queries. *ACM SIGMOD*, 2002.

[9] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting Local Similarity for Indexing Paths in Graph-Structured Data. *ICDE*, 2002.

[10] R. Kaushik, P. Bohannon, J. F Naughton, and P. Shenoy. Updates for Structure Indexes. *VLDB*, 2002.

[11] R. Kaushik, R. Krishnamurthy, J. F. Naughton and R. Ramakrishnan. On the Integration of Structure Indexes and Inverted Lists. *SIGMOD*, 2004.

[12] Michael Ley. DBLP database web site. http://www.informatik.uni-trier.de/ley/db.

[13] Q. Li and B.Moon. Indexing and querying XML data for regular path expressions. *VLDB*, 2001.

[14] H. Jiang, H. Lu, W. Wang, and B.C. Ooi. XR-Tree: Indexing XML Data for Efficient Structural Joins. *ICDE*, 2003.

[15] T. Milo, and D. Suciu. Index structures for path expression. *ICDT*, 1999.

[16] S. Nestorov, J. Ullman, J. Wiener, and S. Chawathe. Representative objects: concise representations of semistructured, hierarchical data. *ICDE*, 1997.

[17] P. Rao, and B. Moon. PRIX: Indexing and querying XML using Prunfer sequences, *ICDE*, 2004.

[18] D. Srivastava, S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, and Y. Wu. Structural joins: A primitive for efficient XML query pattern matching. *ICDE*, 2002.

[19] I. Tatarinov, Z.G. Ives, A.Y. Halevy, D.S. Weld. Updating XML. *SIGMOD*, 2001.

[20] H. Wang, S. Park, W. Fan, and P. S Yu. ViST: A dynamic index method for querying XML data by tree structures. *SIGMOD*, 2003.

[21] F. Weigel, H. Meuss, F. Bry and K. U. Schulz. Content-Aware DataGuides: Interleaving IR and DB Indexing Techniques for Efficient Retrieval of Textual XML Data. *ECIR,* 2004.

[22] M. Yoshikawa, T. Amagasa, T. Shimura, and S. Uemura. XRel: A path-based approach to storage and retrieval of XML documents using relational databases. *ACM Transaction on Internet Technology*, 1(1):110-141, August 2001.

[23] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman. On supporting containment queries in relational database management systems. *ACM SIGMOD*, 2001.

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

[25] XMARK(The XML-benchmark project) http://monetdb.cwi.nl/ xml.

[26] INEX(Initiative for the Evaluation of XML Retrieval) http://inex.is.informatik.uni-duisburg.de:2003/.