Ctree: A Compact Two-level Bidirectional Tree for Indexing XML Data

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

ABSTRACT

Indexing XML data to facilitate query processing has been a popular subject of study in recent years. Most of previous studies can be classified into three categories: path indexing, node indexing and sequence-based indexing. Many of them cannot answer both single-path and branching queries with various value predicates very efficiently. In this paper, we propose a novel compact tree (Ctree) structure, which provides not only concise path summaries but also detailed element relationships, and a configurable index scheme based on data statistics. We develop an efficient Ctree-based method for processing a tree structure query with various value constraints. Efficiency of our method is achieved by: (1) summarizing a large database into a condensed structure view to prune irrelevant search space; (2) evaluating a tree structure query directly without expensive join operations; (3) using Ctree properties such as trivial groups and bi-direction to reduce query processing time; (4) using a new element-referring schema to avoid expensive join operations between the matches for value constraints and those for structure constraints. Our experiments reveal that Ctree is efficient in evaluating XML queries. It is about an order of magnitude faster than most previous methods.

1. INTRODUCTION

With the growing popularity of XML, more and more information is being stored and exchanged in the XML format [1]. 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[23], XQuery[24], etc.) to provide flexible query mechanisms. An XML query typically consists of two parts: structure constraints and value constraints. Structure constraints are usually represented by a tree structure, which can have a single-path or multiple branches. And value constraints can be some Database style (DB-style for short) predicates, such as equality or inequality. But they can also be some Information Retrieval style (IR-style for short) predicates such as containment predicates.

The semi-structure nature of XML data and the flexible mechanisms provided XML queries introduce new challenges to database indexing methods. An index on XML data's structures is expected to a covering index such that the index alone can facilitate the evaluation of both single-path and multiple branches tree-structure queries without expensive joins.

Currently, there are three major approaches for indexing XML data's structures. 1) *Path indexing* [6, 5, 4, 2, etc.]: create a path summary index from XML data. Path indexing greatly speed up the evaluation of single-path queries. But a path index either is too large for practical use or does not preserve enough details to answer branching queries. 2) *Node indexing* [11, 20]: create indexes on each node in an XML data tree by some numbering schemes. They are very efficient in determining the hierarchical relationships between any two nodes and support all kinds of queries. But node index approaches usually require expensive join operations to answer a query. 3) *Sequence-based indexing* [18, 15]: transform both XML documents and queries into sequences and evaluate queries based on sequence matching. These approaches support flexible queries without join operations. But they either may have false alarms in query results or require a lot of post-processing to guarantee result accuracy.

In addition, value indexing is important for efficient evaluation of XML queries with various value predicates. Many of previous approaches either have no value indexes or index the heterogeneous values uniformly. They either index a value as an entire string which is not efficient for containment value predicates, or as a set of words which is not efficient for equality or numerical range search. Such uniform value treatment and indexes are not suitable for the heterogeneous nature of XML values.

Finally, most previous approaches use global IDs such as preorder to refer elements. Such a referring schema is expensive to maintain when updating and requires join operations between the matches for structure constraints and those for value constraints.

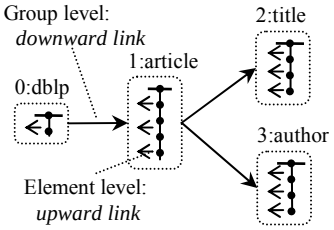


Figure 1: Ctree-A two level tree

In this paper, we address the above challenges in the following four aspects.

First, a novel, compact tree structure, called Ctree, is constructed. As shown in Figure 1, Ctree is a two-level bidirectional tree where an array in a group represents a list of elements and points to their parents. Ctree provides a concise structure summary at its group level and detailed element relationships at its element level. A Ctree is very compact in the index size compared to other index approaches. For example, for the DBLP dataset, the size of Ctree is about 67% of the size of node index and about 77% of the size

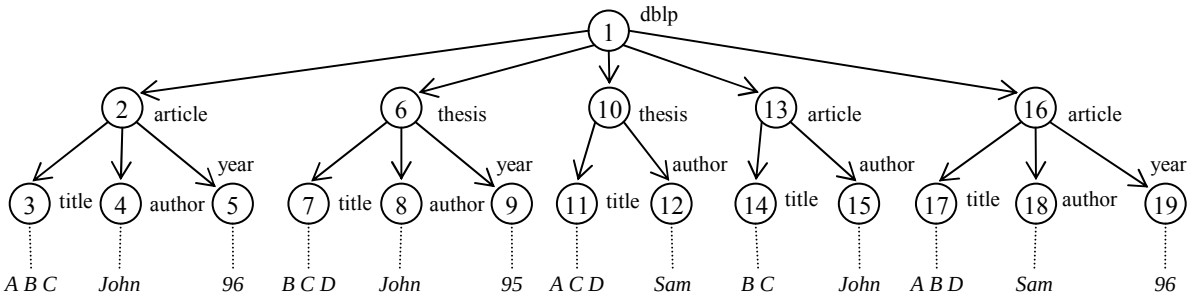


Figure 2: A Example of XML data tree

of path index. Further, Ctree properties such as trivial groups and bi-direction can speed up query evaluation. Therefore, Ctree is well-suited for indexing XML structure.

Second, inspired by relational databases where an expert designs schema and creates indices, we propose a user configurable index scheme for each group based on its statistics. We analyze basic data types in XSchema [22] and introduce five types of value index. We also propose a set of basic value treatments such as stemming, removing stop words, transferring to lower case, etc. Finally, the value statistics of each group are presented to users for configuring the proper value treatment options and index types.

Third, a Ctree-based query processing method is developed, which first locates a set of relevant groups on the group-level, then processes value constraints only on the nodes in relevant groups, which significantly narrows the value constraints, and finally evaluates element level structure constraints. This three-step strategy is very effective and prunes search space at early processing stage.

Fourth, instead of using global IDs, Ctree provides two types of element references: a relative reference and an absolute reference. This novel element-referring schema reduces the cost for updating and avoids the expensive join operation between the results from evaluating structure and value constraints.

We have conducted a set of experiments to compare the performance of the Ctree approach with path index, node index and sequence-based index. Our study shows that our Ctree approach is at least an order of magnitude faster than node index and also outperforms path index and sequence-based index.

The rest of the paper is organized as follows. Section 2 introduces our XML data model. It then provides the definition for Ctree, along with some properties. In Section 3 we present how to build Ctree along with the configurable value index. Section 4 develops the Ctree-base query processing method. In Section 5 we present experimental results that show the effectiveness of our approach. Section 6 reviews related works. We conclude our work in Section 7.

2. FOUNDATION

We describe XML data model in Section 2.1, then move to Ctree and its properties in Section 2.2.

2.1 XML Data Model and Query Model

We model an XML document as an ordered, labeled tree where nodes corresponds to elements and attributes and edges represent element inclusion relationships in the XML documents. To simplify our discussion, we assume that each node is a triple (*id*, *label*, <*value*>), where *id* uniquely identifies the node, *label* is the tag name of the node and the optional *value* is the value of its corresponding element (or attribute) in the XML document.

For example, Figure 2 shows a sample XML data tree which has 19 nodes numbered from 1 to 19 as shown by the circles with the nodes' labels shown beside the circles. The names of the nodes are shown beside the nodes. The value of a node is linked to the node by a dotted line. For instance, there are 13 values associated with the 13 leaf nodes respectively in Figure 2.

Note that in the paper, we treat an attribute as a subelement of an element and a reference IDREF as a special type of value.

We now introduce the terminologies for label path and equivalent nodes, which are useful for describing Ctree.

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

For example, node 8 in Figure 2 can be reached from the root node 1 through the path: node 1→ node 6 → node 8. Therefore the label path for node 8 is *dblp.thesis.author*.

Definition 2 (Equivalent node) Two nodes in an XML data tree *D* are equivalent if they have the same labeled path.

For example, in Figure 2, node 8 and 12 are equivalent since their label paths are the same: *dblp.thesis.author*.

With the definition of equivalent node, we can infer that if two nodes are equivalent in an XML data tree *D*, then their parent nodes are equivalent in *D*. For example, the parent nodes of node 8 and 12 are node 6 and 10 respectively, which are equivalent nodes.

2.2 Ctree and its Properties

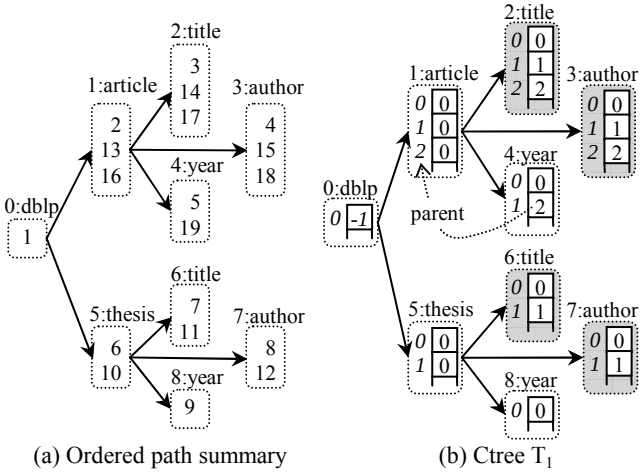
Now we are ready to introduce Ctree which is a compact, complete, and accurate tree representation for XML data tree. A Ctree is a two-level bidirectional tree: group level and element level. At group level, it provides path summary information and contains edges from parent groups to their child groups. At element level, a Ctree has links pointing from child elements to their parent elements.

Similar to most path index approaches, Ctree clusters equivalent nodes in *D* into groups where each group corresponds to a distinct label path. That is, if two nodes are equivalent, we put them into the same group in Ctree. Unlike traditional approaches where

nodes in a group are unordered, Ctree orders the nodes into an array by their preorder in *D*. An edge is added from group A to group B if A's label path is the direct prefix of B's label path.

For example, an ordered path summary for the XML data tree (Figure 2) is illustrated in Figure 3a. Each dotted box in Figure 3a represents a group and the numbers in a box are the identifiers (IDs) of the nodes of *D*. Each group has a label and a unique group identifier which are listed above the boxes. For example, nodes 2, 13, 16 in Figure 2 are in the same group 1 since their label paths are the same: *dblp.article*.

As shown in past research, such path summaries greatly facilitate the evaluation of single-path expressions by searching only the relevant part of a tree, and the answers can be found without any expensive join operations. For example, for a query (Q_1) */dblp/article/author*, the path summary implies that all the nodes in group 3, i.e. 4, 15, and 18, are the answers because their label paths are all *dblp.article.author*.



(a) Ordered path summary (b) Ctree T_1

Figure 3: Examples of path summary and Ctree for the XML data tree in Figure 2.

However, such path summaries are insufficient for answering branching queries due to their incompleteness. They do not preserve the hierarchical relationships among individual nodes in a data tree *D*. For example, with the path summary in Figure 3a, we cannot determine the hierarchical relationships between node 2 in group 1 and node 4 in group 3. Such hierarchical relationships are important for answering branching queries. For example, for a query (Q_2) "find an *article* element which is a subelement of *dblp* and has both subelement *title* and *year*", i.e. */dblp/article*[*title and year*], the path summary in Figure 3a indicates that nodes in group 1 are candidate answers. However, we cannot decide which nodes in group 1 can answer Q_2 unless the hierarchical relationships between individual nodes in group 1, 2, and 3 are provided.

To overcome this problem, we propose Ctree which uses array index to represent elements and the values in arrays to point to the parent elements. Figure 3b shows the Ctree T_1 for the XML data tree in Figure 2. Instead of keeping these "absolute" node IDs inside each group as shown in Figure 3a, a Ctree transforms such "absolute" IDs into their "relative" position within each group. For example, nodes 2, 13 and 16 in group 1 are transformed into 0, 1, and 2 which are their relative positions in group 1 respectively. Since such a relative position can be mapped to an array index, each group *g* in a Ctree *T* is associated with an array, denoted by *T*.grps[*g*].pid[]. Each index *k* of the array represents a node, called node-*k* for short, and the value *T*.grps[*g*].pid[*k*] points to its parent node. For example, *T*.grps[*4*].pid[*1*]=*2* indicates that node-*2* in group 1 is the parent of node-*1* in group 4 as shown by the dotted arrow line in Figure 3b. With the Ctree in Figure 3b, we can answer not only single path but also branching queries. For example, to answer the query */dblp/article*[*title and year*], the arrays associated with groups 2 and 4 contain 0 and 2 and imply that the node-*0* and node-*2* in group 1 are the answers.

By comparing Figure 3b with Figure 3a, the numbers in Figure 3a are in chaos in a sense while the numbers in Figure 3b are in orderly condition. For example, the numbers in shaded groups are the same as their array indices and can be removed. After removing these numbers, the Ctree is about half the size of the path summary. Furthermore, the equivalences of array indices and its values in the shaded groups can be used to speed up query processing.

The formal definition for a Ctree for an XML data tree *T* is defined as follows:

Definition 3 (**Ctree**) A Ctree for an XML data tree *D*, denoted as T_D, is an ordered labeled tree {G, E} where

1) G is the set of groups in T_D which have one-to-one relationships with the label paths in *D*. Each group, *g*, has a unique identifier and a label, denoted as *g.gid* and *g.label* respectively.
2) E is the set of directed edges in T_D that connect these groups. An edge is added from group g_1 to group g_2 if the label path of g_1 is the longest prefix of the that of g_2.
3) Each group *g* in T_D is associated with an array of integers, denoted as *g*.pid[], where the array index represents a list of equivalent nodes ordered by their preorder in *D* and the array values point to the corresponding parent elements.

The definitions 3(1) and 3(2) provide a path summary for an XML data tree *D*, while 3(3) adds to each group in a Ctree the hierarchical relationships among individual elements in *D* to guarantee its completeness. A Ctree has parent-to-child edges at the group level and provides child-to-parent links at its element level as illustrated by the dotted line in Figure 3b. Such a bidirectional tree makes Ctree superior to other indexing methods.

For a group g_c and its parent group *g*, a node in *g* may have no child in g_c. For example, the node-*1* in group 1 has no child in group 4. It is also possible that a node in *g* can have one or more child nodes in g_c. If every node in group *g* has exactly one child in g_c, we call g_c a trivial group, which is formally defined as follows:

Definition 4 (Trivial Group) A group *g* in a Ctree is called a trivial group iff it is the same size as its parent group and *g.pid*[*k*]=*k* ($\forall$k, 0≤*k*<*g*.size).

For example, groups 2, 3, 6 and 7 in Figure 3b are trivial groups. That is, every *article* (or *thesis*) element has exactly one sub-element *title* and *year* in Figure 2. Such information is valuable for query optimization. For example, for a query Q_3 "find an *article* element that has a sub-element *title* and a sub-element *year*", i.e. *//article*[*title and year*], the Ctree directly returns all nodes in group 1 as answers without further checking the element-level information.

The array pid[] in a trivial group can be removed from the Ctree without losing information since we know the function pid[*k*]=*k*. This will reduce the size of Ctree. In our experiments, we found that trivial groups are very common in various XML datasets. Therefore, the size of a Ctree is usually much smaller than that of its counterpart path index.

Depending on the context, we can refer to a node in a Ctree group by two types of references: absolute reference and relative reference which are defined as follows:

Definition 5 (Absolute Reference and **Relative Reference)** The absolute reference to the $(k+1)^{th}$ node in a group g ($0 \le k < g.size$) is denoted as *g.gid:k* and its relative reference is *k*.

For example, the absolute reference to the second node in group 4 in Figure 3b is 4:1. If the referring context is clear, its relative reference is *1*.

Theorem 1 For any group g in a Ctree, the values in g.pid[] are in increasing order, i.e. $\forall i, j\ (0 \le i < j < g.size)\ g.pid[i] \le g.pid[j]$.

This can be inferred from Definition 3(3) where the equivalent nodes in a group are ordered by their preorder in *D* and the continuous preorder numbers.

As shown in Figure 4, group g_1 is the ancestor of group g_2, i.e., there is a group level path from g_1 to g_2.

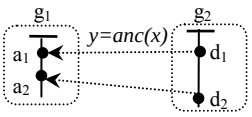


Figure 4: *y=anc(x)* is an increasing function.

Theorem 2 The ancestor-child function y=anc(x) from g_2 to g_1 is an increasing function.

Theorem 2 can be inferred by recursively using Theorem 1. The theorem means that $a_1 \le a_2$ whenever $d_1 < d_2$ in Figure 4. This property can be used for query processing.

3. A Ctree-based XML INDEXING

The architecture for a Ctree-based XML indexing and querying system is illustrated in Figure 5. There are three function parts in the system architecture: a *Scan* module, an *Index Builder* and a *Query Processor*. There are also three data parts: the source *XML Data*, the *Configuration* file, and the *Ctree Index Data*.

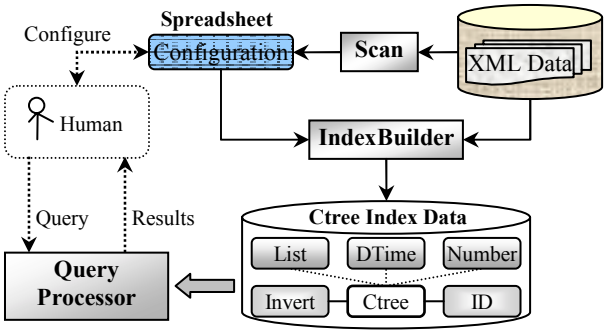


Figure 5: Architectural Overview of Ctree.

A Ctree index for XML documents is created by three steps. First, XML data is sent to the *Scan* module that collects statistical information about the data. This information is stored in an Excel spreadsheet and presented to a user. Second, based on the statistics information, a user selects a set of index options for each group of equivalent nodes. Third, based on the user's index configurations, the *Index Builder* constructs a Ctree and builds value indexes on the data.

When a query comes in, the *Query Processor* evaluates the query based on the Ctree structure and value indices, and returns the query results to the user. We present the Ctree-based query evaluation process in Section 4.

In this section, we first describe the index configuration process. Then we introduce five value index types supported in our system. We present how the Index Builder creates a Ctree on XML raw data based on a user's index configurations in section 3.3. Finally we discuss how to use a Ctree to update the XML data and the Ctree simultaneously.

3.1 Index Configurations

Previous XML indexing approaches create indexes uniformly on each node, each path, or each sequence of XML documents. No consideration is given to the heterogeneous nature of XML data model which allows any user-defined tags. Tags in an XML document usually can be classified into three categories: (1) Semantic Tags; (2) Presentation Tags; and (3) Annotation Tags. Semantic Tags describe the semantics of corresponding elements, such as <*title*>, <*author*> and <*year*> in Figure 6. Presentation tags sense the tone of displaying, such as <*scp*> which informs a browser to display the text bounded by <*scp*> and </*scp*> in lower cases. Finally, the text bounded by annotation tags represents a document author's comments and annotations, such as <*note*> in Line 9. The heterogeneous tags in an XML document call for a non-uniform indexing scheme where some individual tags and some annotations (i.e. element tags and their values) may be ignored during indexing.

```
01 <article key="books/abc/webb03">
02    <author> Webb </author>
03    <title> A K<scp>NOWLEDGE</scp> Based Web Data
04       Integration and Exchange </title>
05    <year> 2003 </year>
06    <pages> 180 -200 </pages>
07    <body>
08       <sec>The problem of information integration
09          <note>see reference 2</note> and exchange …
10       </sec>
11    </body> </article>
```

Figure 6: An Example XML Document.

Similarly, the heterogeneous element values in XML documents call for a non-uniform value index scheme to support the diverse value constraints in XML queries.

To this end, we propose a user configurable index framework which allows a user to specify ignorable individual tags and annotations. It also allows a user to specify how to index an element's value: whether to index an element's value as a whole string, or as a bag of tokens, or as a number. More importantly, our configurable index framework allows a user to specify how to treat an element's value, such as whether to remove words with no discriminative power, i.e. stop words, and whether to transform words to its canonical form to support keyword searching and to reduce index size.

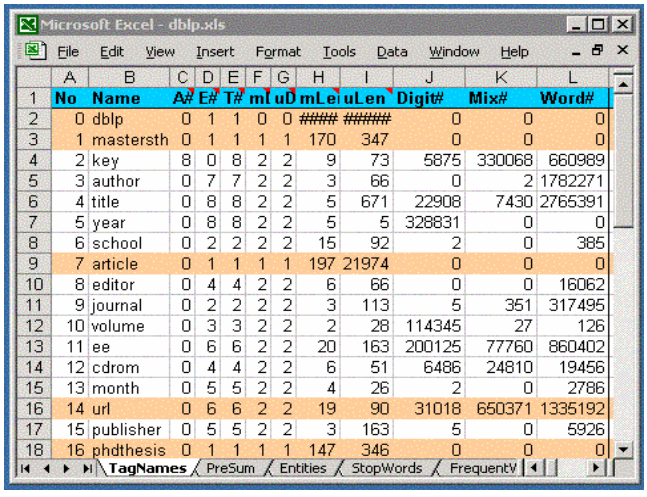
3.1.1 Scan

To facilitate index configurations, we use a *Scan* module to collect statistics information from XML documents while parsing. This statistics information is stored in a spreadsheet for ease of browsing and configuration.

Since equivalent nodes in an XML data tree usually have similar structure and value characteristics, a *Scan* module collects a set of statistics information about a group of equivalent nodes. The statistics information is stored in an Excel's worksheet (*PreSum*). Each row in the worksheet stores some statistics information about a group of equivalent nodes as listed below:

1) *Group Label*: the label of the group, such as *title*.
2) *Group depth*: the depth of the group.
3) *# of nodes*: the number of equivalent nodes in the group
4) *Max value length* (*mLen*): maximal value length for the nodes in the group.
5) *Min value length* (*mLen*): Minimal value length for the nodes in the group.
6) *Digit#*: number of tokens composed by pure digits in the values of nodes in the group. A token is defined as a set of continuous digits or letters or mixed digits and letters without delimiting spaces.
7) *Word#*: number of tokens composed by pure letters in the values of the nodes in this group.
8) *Mixed#*: number of tokens composed by mixed digits and letters in the values of the nodes in this group.

The items 1)-3) summarize a group's structure characteristics while 4)-8) summarize a group's value characteristics. The group statistics can be further grouped by tag names. For example, the grouped statistics information for the DBLP dataset collected by the Scan module is presented on the worksheet named *TagNames* as shown in Figure 7. This statistics information can help a user to select index options for each group.


Microsoft Excel - dblp.xls

File Edit View Insert Format Tools Data Window Help

	A	B	C	D	E	F	G	H	I	J	K	L
1	No	Name	A#	E#	T#	mL	uD	mLe	uLen	Digit#	Mix#	Word#
2	0	dblp	0	1	1	0	0	####	#####	0	0	0
3	1	mastersth	0	1	1	1	1	170	347	0	0	0
4	2	key	8	0	8	2	2	9	73	5875	330068	660989
5	3	author	0	7	7	2	2	3	66	0	2	1782271
6	4	title	0	8	8	2	2	5	671	22908	7430	2765391
7	5	year	0	8	8	2	2	5	5	328831	0	0
8	6	school	0	2	2	2	2	15	92	2	0	385
9	7	article	0	1	1	1	1	197	21974	0	0	0
10	8	editor	0	4	4	2	2	6	66	0	0	16062
11	9	journal	0	2	2	2	2	3	113	5	351	317495
12	10	volume	0	3	3	2	2	2	28	114345	27	126
13	11	ee	0	6	6	2	2	20	163	200125	77760	860402
14	12	cdrom	0	4	4	2	2	6	51	6486	24810	19456
15	13	month	0	5	5	2	2	4	26	2	0	2786
16	14	url	0	6	6	2	2	19	90	31018	650371	1335192
17	15	publisher	0	5	5	2	2	3	163	5	0	5926
18	16	phdthesis	0	1	1	1	1	147	346	0	0	0

TagNames | PreSum | Entities | StopWords | FrequentV

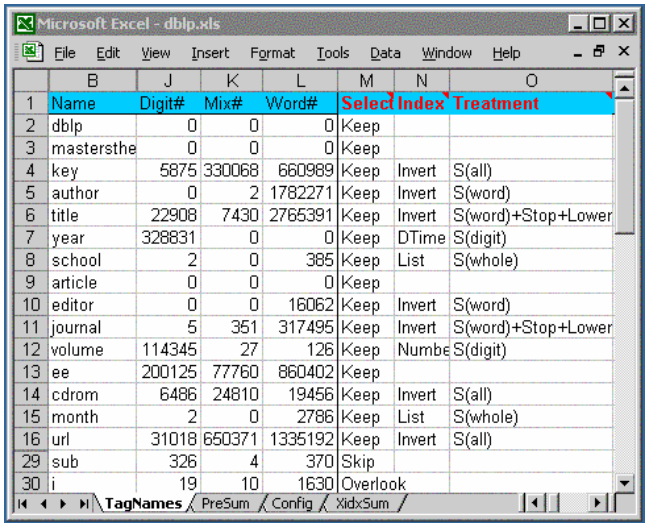
Figure 7: Collecting statistics from source XML data.

3.1.2 Configuration

With the collected statistics information, a user can specify a tag index type, and value index type, and select a set of value treatment options for each group, which are described in detail as follows:

- *Tag index type*: To configure how to index a tag. We currently support three tag index types: 1) *keep*, which means that a user wants to keep these tags during indexing; 2) *overlook*, which means that a user wants to ignore these individual tags but keep their values during indexing; 3) *skip*, which means a user wants to ignore the whole annotation, i.e. both tags and values, during indexing.

An example of index configurations for the DBLP dataset is illustrated in Figure 8. The option *overlook* on the cell M30 (column M, row 30) means that the tags <*i*> and </*i*> will be overlooked during indexing. The option *Skip* on the cell M29 means that the elements named *sub* will be skipped as a whole.


Microsoft Excel - dblp.xls

File Edit View Insert Format Tools Data Window Help

	B	J	K	L	M	N	O
1	Name	Digit#	Mix#	Word#	Select	Index	Treatment
2	dblp	0	0	0	Keep		
3	mastersthe	0	0	0	Keep		
4	key	5875	330068	660989	Keep	Invert	S(all)
5	author	0	2	1782271	Keep	Invert	S(word)
6	title	22908	7430	2765391	Keep	Invert	S(word)+Stop+Lower
7	year	328831	0	0	Keep	DTime	S(digit)
8	school	2	0	385	Keep	List	S(whole)
9	article	0	0	0	Keep		
10	editor	0	0	16062	Keep	Invert	S(word)
11	journal	5	351	317495	Keep	Invert	S(word)+Stop+Lower
12	volume	114345	27	126	Keep	Numbe	S(digit)
13	ee	200125	77760	860402	Keep		
14	cdrom	6486	24810	19456	Keep	Invert	S(all)
15	month	2	0	2786	Keep	List	S(whole)
16	url	31018	650371	1335192	Keep	Invert	S(all)
29	sub	326	4	370	Skip		
30	i	19	10	1630	Overlook		

TagNames | PreSum | Config | XidxSum

Figure 8: Index configuration for DBLP dataset

- *Value index type*: To choose a value index type to index the values of a group. Currently six options are available: 1) *No index*, if a user specifies this value index type for a group, then the values of this group will not be indexed; 2) *Invert*; 3) *Number*; 4) *DTime*; 5) *List*; and 6) *ID*. The latter five choices will be explained in detail in Section 3.2.
- *Value treatment options*: To specify a set of treatments to process the values of a group before indexing. Currently, four treatment functions are supported in our system. 1) *S Function:* Provides four options *digit*, *word*, *all* and *whole* for the user to define which kinds of tokens to index. 2) *Stop Function.* Removes the words with weak discriminative powers such as articles and pronouns. 3) *Lower Function.* Transforms a value into lower cases before indexing. 4) *Stemming Function.* Transforms a value into its canonical forms before indexing. For example, "clustering", "clusters" and "clustered" will be transformed to "cluster".

For example, the option on the cell *O7* (column *O*, row *7*) in Figure 8 means that a user is only interested in indexing the digits tokens in values for elements *year*. The option on the cell O8 means that a user wants to index the values for elements *school* as a whole.

Note that a user can configure index options for a tag which are applied to a set of groups of the tag name. A user can also make

configurations for a specific group, which has higher priority than the tag-level configurations.

3.2 Value Index Types

The heterogeneous nature of element values in XML documents calls for multiple value index types according to the values' characteristics. In this paper, we propose five value index types: *Invert*, *List*, *Number*, *DTime* and *ID* to support values of common XML data types in the XML schema [22], such as *xs:string* and *xs:decimal*, etc., and some special data values such as values for IDREF attributes. Each value index type supports a common search function:

List *Search* (*gid*, *predicate*)

That is, given a group identifier and a value predicate, the function returns a list of elements that satisfy the value predicate. This search function plays an important role in our Ctree-based query evaluations (Section 4). For example, suppose a user is interested in finding books authored by "*David*" in the DBLP dataset, i.e. /*book*/*author*[*contains*(., "*David*")] in XQuery. The Ctree-based query evaluation algorithm first locates the group 88 for the path /*book*/*author* and then calls the search function: *Search(88, "David")* which returns only *27* authors in group 88 with "*David*" in their names. In contrast, most previous indexing approaches return the *13,218* author names containing "*David*" since their inverted value indices use global element IDs, and then join the *13,218* author names with the results for /*book*/*author*.

The *Invert* and *ID* value index types are global in that they are shared among groups. A Ctree has at most one *Invert* value index and one *ID* value index. The other three value index types (i.e. *List*, *Number* and *DTime*) are local in that each of them is associated with a particular group. A Ctree can have several List, Number and DTime value indices. Global value index types use absolute element references and local value index types use relative element references.

- *Invert Type*: This type treats a value as a bag of tokens and creates a mapping from each token to a list of absolute element references whose values contain this token. These absolute references are sorted in the ascending order; that is, references are first sorted by their group identifiers and then by relative references. This facilitates a group-based search.
- *List Type*: Some groups have only a limited number of distinct values, for instance, a person's education. For such values, we create a unique value identifier for each distinct value and associate the identifier with a list of elements having this value.
- *Number Type*: Numerical values are indexed by a Number type, which sorts numerical values of a group in ascending order. Number type supports numerical range search. More specifically, given a numerical range (*a*, *b*), it returns a list of relative node references whose values are greater than *a* and less than *b*.
- *DTime Type*: The DTime type transfers a string expression of date or time into an integer and creates indexes on the integers, which minimizes index size and computation overhead by eliminating string comparisons.
- *ID Type*: The ID type is used to index the special IDREF values in XML documents. It creates a map for a referring node to a referred node, where the nodes are represented by their absolute element references but not the IDREF values.

3.3 Constructing a CTree

3.3.1 Ctree Index

To support efficient and scalable query processing, Ctree index stores the following mapping information:

- The *nameHt*: *Tag name→nid*. A hash table maps each tag name to a unique name identifier (*nid*). Using identifiers rather than string minimizes index size and computational overhead.
- The *nid2Gids*[][]: *nid→gids*. An array, for a given tag name identifier *nid*, returns a list of group identifiers (*gids*) that correspond to the tag name.
- The *pathHt*: *path→gid*. A hash table, for a given label path, returns the group identifier corresponding to the label path. For example, it returns group 3 for the label path *dblp.article.author* in the Ctree T_1 in Figure 3b. This mapping can facilitate the evaluation of simple path expression queries.

At the group level, Ctree stores a set of attributes (*gid, size*, *depth, nid*, *pgid*, *pid*[], *pos*[], *len*[]) where *gid* and *size*, representing the group identifier and the number of descendant groups, can be used to determine the ancestor-descendant relationship between any pair of groups in constant time. For example, for groups g_1 and g_2, g_1 is an ancestor of g_2 if and only if $g_1.gid < g_2.gid \leq g_1.gid + g_1.size$. The *pgid* points to the parent group. At element level, for each element, Ctree stores the relative reference of its parent element, its position, and its length in the arrays *pid*[], *pos*[], and *len*[] respectively.

3.3.2 IndexBuilder

Algorithm 1 shows the *IndexBuilder* which takes a file path, where a set of XML documents reside, and a configuration Excel file for the documents as input and creates a Ctree for the documents. Line 1 reads the configuration directly from the Excel file and uses it to build an empty instance of Ctree. Line 2 opens an XML file one at a time and reads it with a SAX parser. Lines 3 to 15 show the main event processing.

Lines 4 to 6 process an open tag recording element information such as positions and degrees and resetting its text content. If the current group *grp* is configured as *Select*, then Ctree group *cgrp* is pointed to *grp*. For a text node, it simply records the text as in line 7. For a close tag, if the configuration of *grp* is *Overlook*, then add its text is added to its parent element as in lines 10 and 11; if it is *Select*, then add the element information into the Ctree and process its text for adding into the value index.

When finished with reading all XML documents, *Index Builder* builds the value index as shown in line 16. Then the Ctree is fully created and can be serialized onto hard disk for reuse.

```
Input:  P,  a file path where XML documents reside.
        C,  an Excel file for index configuration
Output: T,  Ctree index object with value index
1  Read C and build a Ctree object T to fill in information.
2  For each XML file, open a reader r and read till the end
3    Case (open tag):
4      Record position and degree into →info[r.dep]
5      Current group→grp and reset text[r.dep];
6      If (grp is configured to Select) then cgrp = grp;
7    Case (text node): Add r.text to text[r.dep];
8    Case (close tag):
9      If (grp is configured to Skip) break.
10     If (grp is configured to Overlook)
11       Add text[r.dep] to text[r.dep-1] and break;
12     Add element info to the Ctree T;
13     If (cgrp is configured to have a value index)
14       Process text[r.dep] and add into the value index.
15   cgrp=cgrp→parent;
16 Build value index for each group;
17 Return T
```

Algorithm 1: Indexer: building Ctree with value index based on user configuration

3.4 Updating Ctree

There has been previous works on updating XML data [17] and some works on updating XML index [9,2], but how to use XML index to assist updating both XML data and XML index simultaneously has not been addressed in previous research. We propose to use Ctree to query and update XML source data. Due to space limitations, we only discuss inserting and removing a single element using a Ctree.

```
Input:  D, a source XML document to add an element
        T, A Ctree to be updated
Output: Updated D and T.
//Add element e into group g at position k
1  Insert e at proper position in D.
2  Insert the info of e into group g at position k.
3  For each g's child c, if k≤c.pid[i], then c.pid[i]= c.pid[i]+1.
4  Update value index for elements of g with reference ≥k.
5  Update affected elements' positions and lengths info.
```

Algorithm 2: Add an element into a XML file using Ctree.

Algorithm 2 shows the main steps to insert an element *e* into group *g* at the position *k* in a Ctree *T*. It first gets a proper position and inserts the element into the XML document. Then information is inserted into *T* as in line 2. Line 3 updates the element level links for the child groups of *g*. Line 4 updates the part of the value index affected by the changes in group *g*'s references. Finally, we update the positions and lengths information for the affected elements. Similarly, Algorithm 3 illustrates the main steps to remove an element *e* from group *g* at the position *k* in the Ctree *T*.

Previous works use global IDs to refer elements. Updating an element from a XML document affects not only a large number of the remaining elements' IDs but also their corresponding positions and lengths information in a XML document. Furthermore, changing an element's ID incurs updating the value index for the element. There also has been some previous work that uses dynamic labeling schemes to reserve some IDs for each node. Updating an element in a XML document does not affect other elements' IDs when there is a spare ID at the updating point, but it still requires updating other information such as positions and lengths.

```
Input:  D, a source XML document to contain an element
        T, A Ctree to be updated
Output: Updated D and T.
//Remove element e from group g at position k
1  Remove e at its position in D.
2  Remove the info of e from group g at position k.
3  For each g's child c, if c.pid[i]≥k, c.pid[i]= c.pid[i]-1.
4  Update value index for elements of g with reference ≥k.
5  Update affected elements' positions and lengths info.
```

Algorithm 3: Remove an element from a XML file using Ctree

In contrast, A Ctree has several important advantages when updating XML documents:

- A Ctree changes fewer element references when updating an XML document than previous methods. For example, inserting a new element into a group in a Ctree affects only the elements after the inserting point in that group and has no effect on other groups.
- The fast access of parent elements in a Ctree enables it to use relative element positions (the offsets within their parents) which can significantly reduce the number of affected elements and thus reduce the costs for updating the positions and lengths information.
- Finally, using relative reference and private value indices reduces the value index updating costs.

4. QUERY EVALUATION

In this paper, we model an XML query *Q* as a tree T_Q where each node corresponds to an element (or attribute) in Q and an edge represents an element-inclusion relationship. An edge in a T_Q can be either a parent-child (PC) edge, denoted as "/", or an ancestor-descendant (AD) edge, denoted as "//". Value constraints in *Q* become value constraints of corresponding nodes in T_Q. We assume that each query has an element or attribute that a user wants to return and we call its corresponding node in T_Q *target node*. To distinguish a target node from other nodes in T_Q, we emphasize it with a box. For example, Figure 9 is a tree representation of the following query (Q_4):

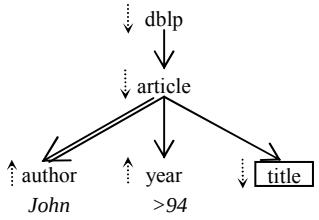


Figure 9: Tree pattern query

/*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 a descendant element *author* containing "*John*" and a sub-element *year* with value greater than 94.

4.1 CTSearch: Ctree-based query processing

We propose a novel Ctree-based query evaluation CTSearch (Algorithm 4), which processes a query, T_Q, in the three steps:

First, a Group Level Frame Finder (Section 4.2) locates a set of frames, where each frame is an assignment of Ctree groups to the query nodes that satisfy a query's tree-structure. For example, there is one frame consisting of groups (0, 1, 3, 4, 2) in the Ctree (Figure 3b) for Q_4, which are matches to query nodes (*dblp*, *article*, *author*, *year*, *title*) respectively.

Input: T, a Ctree with value index
T_Q, a query tree
Output: A list of elements in T that satisfy the 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 *StruEvaluator* to a list of matched elements;
5 Output the list of elements;

Algorithm 4: CTSearch: Ctree-based query processing.

Second, CTSearch evaluates the query's value constraints using value indices. As discussed in Section 3, all value indices implement a *Search* function with two parameters: a group identifier and a value predicator. The *Search* function returns a list of nodes in the group that satisfy the value predicator. For example, there are two value constraints in Q_3: *author*="John" and *year*>94. For the first value constraint, CTSearch calls the function *Search(3, "John")* since group 3 is a match for query node *author* in step 1. Nodes 3:0 and 3:1 are retuned. That is, node 4 and 15 in Figure 2, satisfy the value predicator. Similarly, node 4:0 (i.e. node 5 in Figure 2) is returned by evaluating the second value constraint on value indices.

Finally, CTSearch evaluates element level structure constraints (Section 4.3) and returns the query results to the user. This step can be done by analyzing element level pointers.

This three-step query evaluation strategy offers several important advantages:

- It is very efficient as it only searches the relevant part of the tree hierarchy with the guidance of a Ctree's group-level representation.
- It supports early pruning. If there is no answer available, it terminates at the earliest possible stage. For example, for a query *//article*[*author ="John"*]*/address* on DBLP, our CTSearch will return zero match at the first step since the path *//article/address* does not exist in the Ctree for the DBLP dataset. However, traditional node index approaches require a set of expensive join operations, such as a join operation between the 11629 matches for node *article* and 716488 matches for node *author* in DBLP.
- Evaluating value constraints based on a frame significantly reduces the possible matches.

4.2 Group Level Frame Finder

There are two kinds of nodes in a query tree T_Q: fixed nodes and variable nodes. A query node, *u*, is a fixed node if the path from the query root to *u* does not contain "//" and "*", such as node *dblp*, *article*, *year* and *title* in Figure 9. A query node, *u*, is a variable node if the path from the query root to *u* contains "//" or "*", such as node *author* and *year* in Figure 9.

A fixed query node can have at most one group match in a Ctree, which can be easily determined by the hash table *pathHt*, because every group in a Ctree represents a unique label path. For example, group 2 in Figure 3b is the match for node *title* whose label path is *dblp.article.title*. Similarly, the matches for query nodes *dblp*, *article* and *year* are group 0, 1, 4 respectively.

Unlike a fixed node, a variable query node can be matched to more than one group in a Ctree. For a variable query node, we can determine its candidate matches by first finding its name identifier nid_1 from the hash table *nameHt* and then retrieving the list of candidate group identifiers from the array *nid2Grps*[][] with nid_1, i.e. *nid2Grps*[nid_1][]. After identifying candidate matches for all the variable nodes in T_Q, we can apply structural join algorithms to get frames [3][16][12]. For example, the variable query node *author* in Q_4 has two candidate matches in the Ctree (Figure 3b): group 3 and group 7. Since the parent query node *article* is matched to group 1, group 3 is the only choice for *article* and the frame for T_{Q4} is {0, 1, 3, 4, 2}.

Algorithm 5 shows the *FrameFinder* which locates all frames in a Ctree. It first determines the matches for the fixed query nodes in T_Q (Line 1). Line 2 processes the candidate matches for each variable node in T_Q. A structural join algorithm is used to determine the match for each variable node in T_Q (line 3). Finally line 4 outputs a list of frames.

Input: T, a Ctree with value index
T_Q, a query execution tree
Output: F, A list of mappings from *gids*→ T_Q.*nodes*.
//
1 Get the *gid* for each fixed node by checking the hash table *T.pathHt*; if not existed, then return ∅.
2 Assign candidate *gids* to each variable node by the label of the node.
3 Apply structural join algorithms to determine the matches for each variable node.
4 Return a list of assignments.

Algorithm 5: FrameFinder: mapping *gids*→query nodes.

4.3 Element Level Structure Match

The last step in CTSearch evaluates element-level structure constraints and returns relevant elements in the groups matching a query's target node, which we call *target groups*. For example, for T_{Q4} (Figure 9) and the Ctree T_1 (Figure 3b), the first two steps of CTSearch determine that all the elements in groups {0, 1, 2} and elements {3:0, 3:1} and {4:0} satisfy the group-level structure and value constraints. Now the last step of CTSearch should return relevant elements in group 2 which can answer T_{Q4}.

Relevant nodes in the target groups can be determined by projecting results for non-target query nodes to the target groups. Depending on a query node's position within a query tree, a projection direction for a query node can be either downward or upward. A query node's projection direction is downward if it is on the path from the root node to the target node, such as node *dblp* and *article* in T_{Q4}. Otherwise, its projection direction is upward, such as node *author* and *year* in T_{Q4}.

Algorithm 6 shows the *StruEvaluator* which, given a query tree T_Q and a frame in a Ctree after the first two steps' processing, returns a list of nodes in the target group that are the answers to T_Q. Line 1 computes a projection direction for each query node in

T_Q as discussed above. Line 2 classifies query nodes in T_Q into two ordered lists: query nodes with upward projection directions, *UN*, and query nodes with downward projection directions, *DN*. Nodes in *UN* are ordered by the ascending post-orders in T_Q so that each node, *u*, can project their results to its parent node, *p(u)*, before *p(u)* projects its results upward. Similarly, nodes in *DN* are ordered by the ascending pre-orders in T_Q so that each node, *v*, can project their results to its child node or direct descendant, *c(v)*, before *c(v)* projects its results downward. For example, for T_{Q4} in Figure 9, *UN*= {*author*, *year*} and *DN* = {*dblp*, *article*, *title*}. Line 3 projects the results of the nodes in *UN* upward into their parents. (Section 4.3.1) Line 4 projects the results of the nodes in *DN* downward to their child in *DN*. (Section 4.3.2)

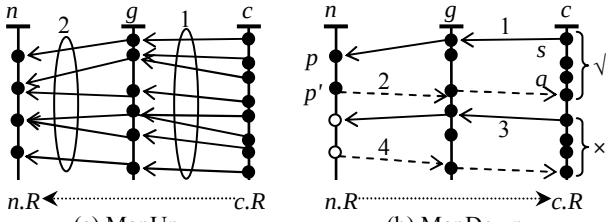
Input: T_Q, a query tree with a target node u_t
F, a frame after Step 1& 2 and g_t is the target group
Output: A list of nodes in g_t that are answers for T_Q.
//
1 Determine the projection direction for each query node.
2 Let ***UN*** = {v_1, v_2, …, v_m} be the list of query nodes with upward projection directions, ordered by their post-orders in T_Q; ***DN*** = {u_1, u_2, …, u_n} be the list of query nodes with downward projection directions, ordered by their pre-order in T_Q.
3 For each node in ***UN***, project its result upward to its parent node by calling MapUp function.
4 For each node in ***DN***, project its result downward to its child in *DN* by calling MapDown function.
5 Return the list of relevant nodes in the target node.

Algorithm 6: StruEvaluator: Evaluating element level structure constraints.

4.3.1 MapUp Function

Algorithm 7 describes the *MapUp* function, which projects the results of a query node *c* upwards to its parent query node *n* so that the results of *n* will be updated to reflect the structure constraint from the node *c*.

Since the edge between *n* and *c* on T_Q can be an ancestor-descendant relationship, there are some groups in between the two groups assigned to *n* and *c*, i.e. *n.gid* and *c.gid*. Figure 10(a) shows a case where one group *g* links the two groups in a Ctree *T*.



(a) MapUp (b) MapDown

Figure 10: Illustration of MapUp and MapDown where there is an ad edge from *n* to *c* and only one group *g* in between

MapUp starts from group *c.gid* and projects the results one group upward at each loop until the group *n* is reached as shown in lines 1 to 7. Line 8 intersects the two element lists. Note that no mapping operation is required for a trivial group as shown in line 2, which reduces the computing time.

For example, the results for *author* in T_{Q4}, elements {3:0, 3:1} in the Ctree T_1 (Figure 3b.) are projected to *article* whose current results are {1:0, 1:1, 1:2}. The updated results for *article* are {1:0, 1:1}. Then, the result {4:0} for *year* is projected to *article* whose current results are {1:0, 1:1}. Since element 1:0 is the parent of element 4:0, the updated result for *article* is {1:0}.

Input: *T*, a Ctree with value index
n, a query node in query tree.
c, a child node of *n*.
Output: *n.R*, updated the results of the query node *n*..
1 for(*g*=*c.gid*, *R*=*c.R*; *g*!=*n.gid*; *g*=*g*→*parent*) do
2 if(*g* is a trivial group) continue;
3 for(*i*=0, *a*=new ArrayList(), *old*=*-1*; *i*<*R*.Count; *i*++)
4 if(*R*[*i*]→*pid*!=*old*)
5 *old* = *R*[*i*]→*pid*;
6 *a*.Add(*old*);
7 *R*=*a*;
8 *n.R*=*n.R* ∩ *R*;

Algorithm 7: MapUp: mapping the results from a query node up to its parent node.

4.3.2 MapDown Function

After the nodes in *UN* are projected upward, the *StruEvaluator* algorithm iteratively projects the results for the query nodes in *DN* downward until it reaches the target node.

For example, for T_{Q4}, *DN* = {*dblp*, *article*, *title*}. First, the MarkDown function is called to project the results {0:0} for *dblp* to its child node *article*, whose current result is {1:0}. Since element 0:0 is the parent of element 1:0, the updated result for *article* are still {1:0}. Then we project the {1:0} to the target query node *title*, whose current results are {2:0, 2:1, 2:2}. Since element 2:0 is the only child of element 1:0, the answer is {2:0}.

Input: *T*, a Ctree with value index
n, a query node in query execution tree.
c, a child node of *n*.
Output: *c.R*, updated with the MapDown results.
1 for(*i*=0, *R*=new ArrayList(); *i*<*c.R*.Count;) do
2 The mapping point from *c.R[i]* into *n*→*p*;
3 if(*p* in *n.R*)
4 The maximal continous number from p in *n.R*→*p*;
5 The maximal descendant id of p in *c.grp*→*q*;
6 while(*c.R[i]*<*q*) *R*.add(*c.R[i++]*);
7 else
8 The maximal continous number from *p not* in *n.R*→*p*;
9 The maximal descendant id of *p* in *c.grp*→*q*;
10 while(*c.R[i]*<*q*) *i*++;
11 *c.R*= *R*;

Algorithm 8: MapDown: mapping the results from a query node down to a child node.

Figure 10(b) illustrates the MapDown algorithm as shown in Algorithm 8. It starts from the first element *s* in *c.R* and follows the links to determine the ancestor *p* in the group *n.gid*. For the case *p* is in *n.R*, line 4 finds *p'* with the maximal continuous number in *n.R* from *p*, line 5 determines the maximal descendant *q* in the group *c.grp* of *p'* which can be done by a binary search, and line 6 adds the elements between *s* and *q* in *c.R* into the output list *R*. Lines 8 to 10 show the case *p* is not in *n.R* where we can overlook a list of elements in *c.R* as marked by "×" in Figure 12(b). The correctness of the MapDown algorithm can be proved

by Theorem 2. Intuitively, for any element between *s* and *q* in Figure 10b, its projected point on group *n* will be between *p* and *p'*. Therefore, if the elements [*p*, *p'*] are in *n.R*, then the elements in *c.R* in the range [*s*, *q*] will satisfy the structure constraints from *n*; and if the elements [*p*, *p'*] are not in *n.R*, then the elements in *c.R* in the range [*s*, *q*] will not satisfy the structure constraints from *n*.

5. EXPERIMENTAL RESULTS

We have implemented Ctree in DotNET C# for XML indexing. We have also implemented a path index method similar to DataGuide*[1] [6], and a node index method similar to XISS [11], a sequence index approach similar to ViST[18] in C# for comparison purposes. Experiments were run on a 2.8 GHz PC-compatible machine with 1GB of RAM running Windows XP. To focus on the comparison of query execution time, we loaded each index data into the RAM before testing so that no IO operation for reading the index data was required.

For our experiments, we used public XML databases DBLP [10] and the XML benchmark database XMARK [21]. DBLP is a popular computer science bibliography database and XMARK is a synthetic on-line auction database. Both of them are widely used in benchmarking XML index methods. The characteristics of these two datasets are illustrated in Table 1.

Table 1: Characteristics of the Datasets DBLP and XMARK

Dataset	Size (MB)	Max Depth	Element#	Element# in trivial groups	Percent
DBLP	134	6	3,736,406	1,311,532	35%
XMARK	117	12	2,048,193	1,255,826	61%

As shown in Table 1, the DBLP dataset is relatively shallow with a maximal depth of 6 and contains about 3.7 millions of elements including attributes. XMARK is relatively deep with a depth of 12 and has about 2 million elements. It is interesting to note that there are large percentages of elements belonging to trivial groups, about 35% on DBLP and 61% on MARK. Since trivial groups can be removed from a Ctree and be used for speeding up evaluating structure constraints (Section 4.4), a Ctree is much smaller and more efficient than previous approaches.

5.1 Index Size

Figure 11 shows the space requirements for four indexing approaches: Ctree, Data Guide*, XISS, and ViST. ViST requires decomposing the DBLP dataset into documents at the depth 1 in order to keep document sequences short. Since the content values in XMARK datasets are quite heterogeneous and it is hard to choose the proper depth to decompose, we have not done any comparison with ViST on XMARK.

We noticed that XISS incurs the most space overhead on the two datasets since it builds indices on each element. Data Guide* requires more space than ViST and Ctree.

[1] Since DataGuide cannot answer branching path expressions without accessing the original XML data, we implemented the DataGuide index with an additional array mapping from an element's ID to its end position to make it a covering index for branching path expression queries.

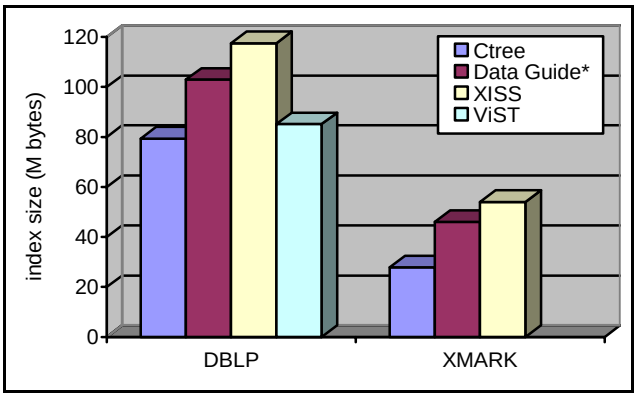


Figure 11: Comparison of the index size of different methods over DBLP and XMARK.

A Ctree requires the least space due to two reasons. First, a Ctree does not need to keep element-level links for trivial groups. As we can see from Table 1, the percentages of elements in trivial groups are quite large because of the common one-to-one relationships in both datasets. Second, the diverse value index schemes in Ctree also reduce of space overhead. For example, transforming a string "$1,234,567.99" into a number reduces index size.

5.2 Comparison on DBLP Dataset

Table 2 lists 6 queries that were tested on the DBLP dataset and have ascending complexity. Figure 12 summarizes the query performance of four index methods: Ctree, DataGuide*, XISS and PRIX. We notice that for most of the queries, Ctree significantly outperforms the other three approaches.

Table 2: Sample queries over 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

Q1 is a single path query and there are no value constraints involved. The Ctree and DataGuide* approaches have similar query performance, while it takes longer for XISS approach as it requires join operations. The value constraints in Q2 slow down the DataGuide* approach against the Ctree approach, although both approaches have similar time requirement for processing structural constraint. In the Ctree, indexes for a specific value, such as "*David*", are sorted according to element's absolute reference, which puts the elements of the same group together. Processing value constraints in Ctree is similar to range search in an ordered list. On the other hand, DataGuide* and XISS require expensive join operations between two ordered lists when processing value constraints. Since Q3 and Q4 uses wildcards or "//" edges and involves a large number of join operations, DataGuide* and XISS take 15 times longer than Ctree. Q5 and Q6 are queries with two branches and again Ctree significantly outperforms the other two approaches.

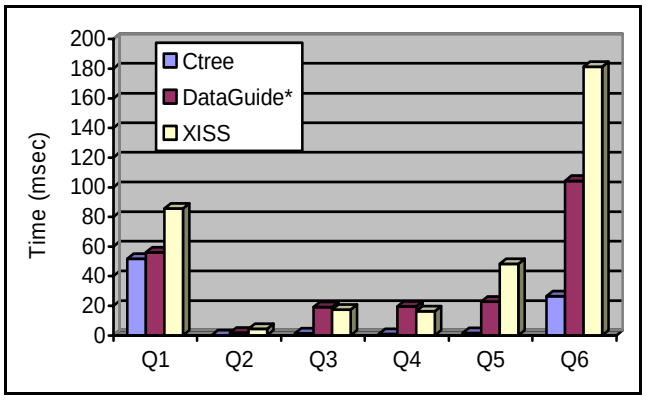


Figure 12: Comparing Ctree with path index and node index on DBLP in relative time for each query

5.3 Comparison on XMARK

We choose six XMARK [21] benchmark queries for our comparison study as shown in Table 3. Q2 and Q4 are designed to test the performance of ordered access: Q2 for element index and Q4 for tag order. Q15 and Q16 are to evaluate the performance of long path traversals. Q18 is to compare function application. Q20 is for aggregations.

Table 3: Sample queries over 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

Figure 13 shows the comparison results. For all the six queries, Ctree outperforms the other two methods by an order of magnitude.

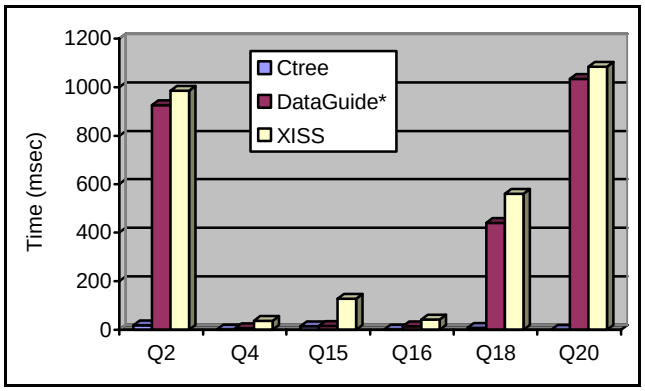


Figure 13: Comparing Ctree with path index and node index on XMARK in relative time for each query.

5.4 Comparison with ViST

We also compared Ctree with ViST on DBLP using the four queries (table 4). The first two queries contain only structure constraints while the last two queries also contain value constraints. Table 4 also shows the number of answers for each query.

Table 4: Queries for comparing Ctree with ViST

	Description	Answer#
Q_1	/inproceedings/title	212,273
Q_2	//www[author, url]	7
Q_3	//article/author="*Jim Gray*"	35
Q_4	//title="*A Query Language for XML.*"	1

Table 5 shows the performance comparisons between Ctree and ViST the four queries listed in Table 4. Ctree correctly answers all the queries. ViST only answers queries Q_1 and Q_2 correctly and contains false positives in the answers for Q_3 and Q_4. Ctree performs significantly better than ViST for queries Q1, Q2 and Q3 and has comparable performance for Q4.

Table 5: Comparison with sequence-based approach ViST

	Ctree	ViST		
	Time (*ms*)	Time (*ms*)	Answer#	False Alarm#
Q_1	51.7	123.6	212,273	0
Q_2	5.4	20.8	7	0
Q_3	26.9	267.2	43	8
Q_4	20.3	33.8	8	7

6. Related Work

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: path indexing, node indexing and sequence-based indexing.

Node index approaches [11, 20] create indexes on each node by its positional information within an XML data tree. Such an 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 then structurally joining these matches. Quite some structural join algorithms [16, 3, 12, etc.] have been proposed lately to support efficient query answering.

Path index approaches create path summaries for semi-structured data to improve query efficiency. DataGuides [6] indexes each distinct raw data path to facilitate the evaluation of simple path expressions. The Index Fabric approach [5] indexes frequent query patterns which may contain "//" or "*", in addition to raw data paths. APEX [4] and D-(k) [2] 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 practically useful. To overcome this problem, F^+B approach [7] reduces index size by ignoring unimportant tags and edges, limiting the depths of branching queries.

Sequence-based indexing approaches [18, 15] 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 whole query tree as the basic query unit, they avoid the expensive join operations and support any tree-structure XML queries.

7. CONCLUSIONS

In this paper, we have proposed Ctree, a compact two-level bidirectional tree, for indexing XML data. Ctree provides concise path summaries at its group level and detailed element relationships at its element levels, which makes it an efficient covering index for both single-path and branching queries. And Ctree is very compact because it uses the very common one-to-one relationships in XML data to reduce index size.

Unlike most of previous approaches that index values uniformly, we propose a configurable index scheme which supports five value index types and several value treatment options. As in relational databases where an expert designs schema and creates index, we believe that user configurable index scheme satisfies the heterogeneous XML data model much better than a uniform index scheme.

Instead of using global IDs, we have proposed two types of element references: an absolute reference and a relative reference when its corresponding group is clear in the context. This group-based element-referring schema avoids join operations between structure and value evaluation results. And it is easily adaptable to updates in XML documents.

We have implemented a Ctree-based search method called *CTSearch* which evaluates an XML query in three-steps: 1) locating relevant groups in a Ctree; 2) evaluating value constraints on the nodes in relevant groups; and 3) evaluating element-level structure matches. We also have studied Ctree's performance in comparison with several influential XML indexing methods. Our performance study shows that in most situations *CTSearch* processes queries at least an order of magnitude faster than previous methods.

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, CA 94022, USA, 1999.

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

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

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

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

[6] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, pages 436-445, 1997.

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

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

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

[10] Michael Ley. DBLP database web site. http://www.informatik.uni-trier.de/ ley/db, 2000.

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

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

[13] T. Milo and D. Suciu. Index structures for path expression. In *Proceedings of 7th International Conference on Database Theory (ICDT)*, pages 277-295, January 1999

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

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

[16] 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. In ICDE 2002

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

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

[19] 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.

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

[21] XMARK: The XML-benchmark project. http://monetdb.cwi.nl/ xml, 2002.

[22] XML Schema http://www.w3schools.com/schema/default.asp

[23] XPath. http://www.w3.org/TR/xpath

[24] XQuery http://www.w3.org/TR/xquery