A New Design for a Native XML Storage and Indexing Manager

Jihad Boulos and Shant Karakashian

Department of Computer Science,
American University of Beirut,
P.O.Box 11-0236 Riad El-Solh,
Beirut, Lebanon.
{boulos, smk09}@aub.edu.lb

Abstract. This paper describes the design and implementation of an XML storage manager for fast and interactive XPath expressions evaluation. This storage manager has two main parts: the XML data storage structure and the index over this data. The system is designed in such a way that it minimizes the number of page reads for retrieving any XPath expression results while avoiding the shortcomings of previous work on storing XML data where the index must adapt to the most frequent queries. Hence, the main advantage of our index is that it can handle any new XPath expression without any need for adaptation. We show comparable performance of our design by presenting path evaluation results of our index against those of the currently most known index on documents of different sizes.

1 Introduction

This paper presents a new storage manager and its indexing scheme for XML data. XML is becoming widely used for data exchange and manipulation in local but mostly distributed environments. It is becoming the foundation of the semi-structured and labeled graph data model where this data can be irregular and/or incomplete and consisting of atomic or composite elements that are nested in a hierarchical manner. This storage manager is part of the Al$\mathcal{X}$emist [1] project that we are designing and implementing for processing XQuery queries on stored and streaming XML data.

There have been many proposals for native XML storage managers and indexing schemes (*e.g.* [2, 3]). Contrary to the widely used approach in these systems where the storage keeps subtree nodes physically close to each other, our scheme only considers the depth and the element name for physical proximity. The logical structure information is preserved by a now classic element numbering scheme. We store XML elements through a breadth-first data layout instead of depth-first; this last approach is mostly adopted for storing and retrieving document subtrees in the same data pages, and this is probably the main reason for its adoption by most previous data layouts. However, we show that our breadth-first design does not have the subtree reconstruction performance hit

Y. Ioannidis et al. (Eds.): EDBT 2006, LNCS 3896, pp. 755–772, 2006.
© Springer-Verlag Berlin Heidelberg 2006

that one may imagine for the most typical queries. We use a layered approach for storing XML documents and building indexes on them, and this approach demonstrates comparable performances to those of the currently most known approaches for retrieving both document subtrees and elements only as results for XPath expressions. Our approach, however, does not need to force an adaptation of the index to the most frequent XPath expressions, but is most general and can handle any new expression.

XQuery is fast becoming the de-facto querying language for XML data. It is based on XPath querying and matching. A path expression matches a node in the XML tree if the path from the root to the node has the same sequence of labels as the path expression. Paths also might have ancestor-descendant (*i.e.* //), wildcard elements (*i.e.* /*) and branches. A naive evaluation of path expressions may require a complete scan of the document; this is too expensive for large documents where these documents do not fit in memory (the same also can be said about memory-resident documents). For this reason, we propose a structural summary index that has a tree structure, summarizing the nodes of the indexed XML document. The size of the index tree remains much smaller than the size of the document (more on that to come). Evaluating an XPath expression on the index and retrieving the resultant elements from the data pages prevent the scan of the whole document tree. Unlike other structural summary indexes (*e.g.* [4–7]) that require being adapted to the most frequent path expressions (FUP's), our index is a general one that can be used to evaluate any path expression without any pre-processing.

The rest of the paper is organized as follows: In Section 2 we mention some related work. We then describe in Section 3 our data layout design for XML documents and then in Section 4 the design of our index. In Section 5, our XPath expression matching algorithm is presented and then in Section 6 the results of some experiments are presented. We conclude in Section 7.

2 Related Work

Several mapping techniques have been proposed [8–11] for storing XML documents in flat tables of relational databases. [8–10] consider XML documents as graphs, and store these graph nodes with their edges in relational database tables. [11] presents a mapping that explores the XML data and creates a mapping to a relational database, separating the objects by their types. The rationale behind this mapping is to use the mature RDBMS technology in indexing and querying XML documents. For this purpose, [9] proposes algorithms for translating XPath expressions to SQL. [8] also proposes a similar conversion from Quilt to SQL.

XML database systems like [2, 3] store XML documents in native storage managers. These systems relieve the processing from additional layers for mapping the logical data to the physical layout, which eventually slows down query processing [12]. [2] stores subtrees in clustered physical pages, and [3] stores XML documents in pre-order traversal in order to cluster sub-elements together.

We also store the XML documents natively, but we store them in breadth-first order, without clustering the elements with their subelements. We cluster only elements of the same name and depth together.

A number of structural indexes for XML have been proposed (*e.g.* [4–7, 13]). These indexes are structural summaries of the target XML documents and are based on the notion of bisimilarity. Two nodes are bisimilar if they have identical label paths. Bisimilar nodes are grouped together in one index node. The 1-Index [14] was the first to propose the idea of bisimilarity. It can answer path expressions of any size without referring to the data nodes. The drawback of the 1-Index is that the size of the index depends upon the regularity of the data graph. For irregular data graphs, the size may become quite large, and the path evaluation time, being proportional to the size of the index node, may also become too high. In many cases where long path expressions are not used, the 1-index may not be necessary. The A(k)-Index was proposed to overcome these shortcomings and is based on the notion of k-bisimilarity. A(k)-Index can answer, without referring to the data graph, path expressions of lengths at most k, where this last value controls the resolution of the index and influences its size in a proportional manner. For large k values, the size of the index may grow to large sizes, and have the same problem as the 1-Index. For small values of k, the size of the index can be substantially smaller, but cannot handle long path expressions.

To accommodate path expressions of various lengths, without unnecessarily increasing the size of the whole index, D(k)-Index [5] was proposed. This index can assign different k values for different index nodes. These k values are assigned to be in conformance with a given set of frequently used path expressions (FUP's). For parts of the index corresponding to parts of the data graph targeted by long path expressions, large values of k are assigned, and small values of k are assigned for parts corresponding to data targeted by short path expressions. To facilitate the evaluation of path expressions with branching, UD(k, l)-Index [6] was proposed. It is similar to A(k)-Index, but also imposes downwards similarity.

A D(k)-Index builds a coarser index than an A(k)-Index, but has the problem of over-refinement. [7] identifies four types of over-refinements, and proposes the M(k)-Index as a solution for all of them. M*(k)-Index (again in [7]) is an extension to the M(k)-Index and is in fact the combination of several M(k)-Indexes in such a way that each one of these has a different resolution. This design solves the problem of large scan space in the index, while the path coverage is not affected. The drawback of this design is inherited from the D(k)-Index and is the requirement to adapt to a given list of FUP's.

Our proposed U($*$)-Index (for Universal-generic) also uses the notion of bisimilarity in a relatively similar way to the 1-Index. However, and in order to overcome the problem of large search space for XPath evaluation on the index, we use a special labeling scheme of index nodes that enables the pruning of the search space. More importantly, our index does not need to be adapted to any particular list of FUP's; it has a uniform resolution and hence is more generic.

3 Data Layout

We describe in this section how an XML document is shredded in stored data pages. As we mentioned earlier, and contrary to previous approaches, we store XML elements (and their related attributes and text) according to their depths and names. We came in fact to this design after designing, implementing and testing several other approaches. We show in the experimental results section, and contrary to previously believed common sense, that the reconstruction of a result subtree is not as expensive as it was thought to be for the most common queries. This is mainly justified by the relatively few levels in most XML schemas (and hence documents). Moreover, as XPath expressions get longer, the height of their resulting trees get smaller, which reduces even more the cost of subtree reconstruction. For queries with large result sizes, subtree reconstruction is always quite expensive.

3.1 Data Shredding

To store an XML document in disk pages, we partition it by element depth and name. All elements named e at depth i belong to the same partition (called "extent" on the storage device). All elements belonging to the same extent are stored in a chain of contiguously clustered pages. With this scheme, all elements in a chain have the same depth and element name. The algorithm that builds these chains is not mentioned here due to space limits but is simple to explain. While scanning an original XML document, pages in memory are created and filled according to the conditions mentioned above (*i.e.* names and levels). When a page is filled, it is saved at its designated disk page in the extent dedicated to that chain of pages.

3.2 Enumeration

In order to preserve the structural information of the XML document in the stored pages, two position numbers are assigned to and stored with each element in the data pages: the start index ($sIndex$) and the end index ($eIndex$) [15]. The $sIndex$ of element e is the depth-first order of e in the document tree (starting with 0 for the root). An $sIndex$ is unique in a document. The $eIndex$ of element e is the greatest $sIndex$ in the subtree of e. If e is a leaf, then $sIndex = eIndex$. More than one element can have the same $eIndex$. The $eIndex$, $sIndex$ and the depth of an element are necessary and sufficient to reconstruct the XML document tree from the stored structure. The depth of an element does not need to be stored with an element since it is retrieved from the storage structure. Fig. 1 shows how a sample XMark [16] XML document fragment is numbered.

3.3 Internal Representation

Fig. 2 shows the internal page representation of the document shown in Fig. 1. A page holds variable sized elements, and is filled until the next element does not fit in. Each rectangle in Fig. 2 represents a disk page (other details of this

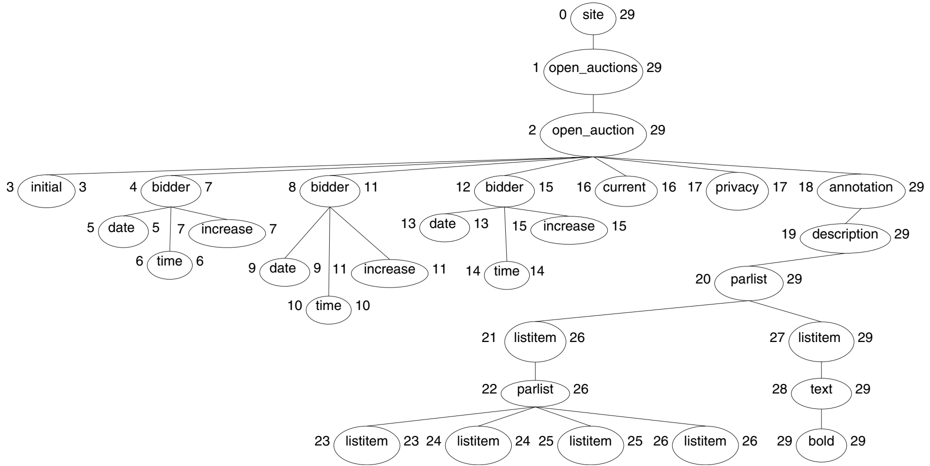


Fig. 1. A sample XMark-fragment document tree

Depth	Element		Page	Page
0	site	:	0(0)	
1	open_auctions	:	1(1)	
2	open_auction	:	2(2)	
3	initial	:	3(3)	
	bidder	:	4(4) 8(4) 12(4)	
	current	:	16(8)	
	privacy	:	17(9)	
	annotation	:	18(10)	
4	date	:	5(5) 9(5) 13(5)	
	time	:	6(6) 10(6) 14(6)	
	increase	:	7(7) 11(7) 15(7)	
	description	:	19(11)	
5	parlist	:	20(12)	
6	listitem	:	21(13) 27(16)	
7	parlist	:	22(14)	
	text	:	28(17)	
8	listitem	:	23(15) 24(15) 25(15)	26(15)
	bold	:	29(18)	

Fig. 2. The stored pages of the data in Fig. 1

figure are presented in the following section). For clarity, it is assumed that each page can hold a maximum of 3 elements. An element is stored with its name, *sIndex*, *eIndex* (these are the numbers to the left and to the right of nodes in Fig. 1), and eventually its attribute names and values, and its *text()* value. A crucial point to notice here is that elements are stored in page chains ordered by *sIndex*. The index tree for our running example is shown in Fig. 3 and is explained in the next Section.

4 Structural Index

Our main contribution in this paper is a disk resident structural summary index for XML documents that we call U(∗)-Index. This index is constructed on an XML document independent of path expressions and how the XML data is stored in data pages. Hence, it evaluates path expressions without requiring any

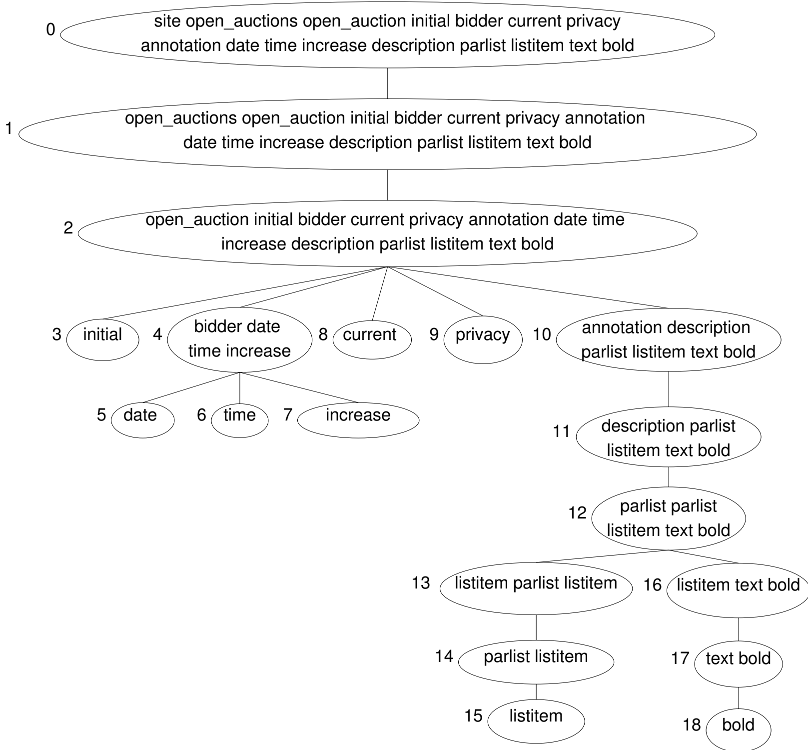


Fig. 3. The structure index of the document in Fig. 1

adaptation or index structure modification to specific path expressions and is detached from how the XML data is clustered in data pages.

Each index node points to a group of elements in data pages. All elements pointed to by an index node i have the same name, depth, *subtree label* (SL) and path label sequence from the document root. The depth of i is equal to the depth of the elements pointed to by i.

Definition 1 (Subtree Label (SL)). *The Subtree Label of a node i is the sorted list of all distinct element names in the subtree of i.* E.g. *the SL of index node corresponding to root "a" in document:* $< a >< c >< b/ >< b/ >< a/ >< /c >< /a >$ *is "abc". The SL of a leaf node is the empty string* ε.

For an index node i, the path label sequence from the root of the index tree to i is the same as the path label sequence of all elements pointed to by i from the root of the XML document. The parent of an element e pointed to by index node i is pointed to by the parent node of i. Similarly, the children of e are pointed to by index node(s) that are the children of i in the index tree. Algorithm 1. is a trivial algorithm that shows how the skeleton of an index is constructed and Algorithm 2. shows how an index node is labeled. These two algorithms are quite simple and clear. Algorithm 3. compresses the index by grouping the nodes

Algorithm 1. Index Construction

ConstructIndex ($DocumentRoot, indexRoot$)
1: $indexRoot.elementName = DocumentRoot.elementName$
2: $indexRoot.SL = \epsilon$
3: **for all** $child \in indexRoot.children$ **do**
4: $indexRoot$.addChild(e)
5: ConstructIndex ($child, e$)
6: **end for**
7: LabelIndex ($indexRoot$)
8: Compress ($indexRoot$)

Algorithm 2. Index Labeling

LabelIndex ($root$)
1: **for all** $child \in root.children$ **do**
2: LabelIndex ($child$)
3: $root.SL = child.elementName \cup root.SL$
4: $root.SL = child.SL \cup root.SL$
5: **end for**

Algorithm 3. Index Compression

Compress ($root$)
1: **for all** $e \in root.children$ **do**
2: **for all** $e' \in root.children$ **do**
3: **if** $e.elementName == e'.elementName$ & $e.SL == e'.SL$ **then**
4: $e.extent = e.extent \cup e'.extent$
5: $e.children = e.children \cup e'.children$
6: Delete e'
7: **end if**
8: **end for**
9: **end for**
10: **for all** $child \in root.children$ **do**
11: Compress ($child$)
12: **end for**

based on the criteria mentioned above (*i.e.* element name, depth, SL, and path label from root).

Fig. 3 shows the index tree for the XML document in Fig. 1. The names in the ovals are the name of the index node, followed by the SL. The numbers to the left of the nodes in Fig. 3 are fictitious and added only for clarity. These are the same numbers between parenthesis in Fig. 2 and are shown in this paper for the reader to make the connection between the index nodes and the data pages. *E.g.* the elements pointed to by the index node 4 have *sIndex* 4, 8, and 12.

4.1 Index Node

An index node i in U(∗)-Index is identified by its name, depth and SL. The name of an index node is the name e of the elements it points to (they all have the same name e). The depth is equal to the depth of these element(s), and the SL of that index node corresponds to the SL of the data nodes pointed to by i. No more than one index node can exist having the same name, SL and parent. Every index node has at least one pointer to an element in the data pages. With every pointer, i also contains the *sIndex* and *eIndex* of the pointed element. *E.g.* the node 4 of the index in Fig. 3 is of depth 3 and of name "bidder"; All elements pointed to by it are also necessarily and exclusively in the pages of elements of depth 3 and name "bidder" (Fig. 2). However, it is not necessarily true that all elements in element pages of the same depth and the same name be pointed to by the same index node (*e.g.* elements "listitem" of depth 6 in Fig 2). Fig. 4 shows in a more visual manner how the nodes of an XML document are mapped to the nodes of their index.

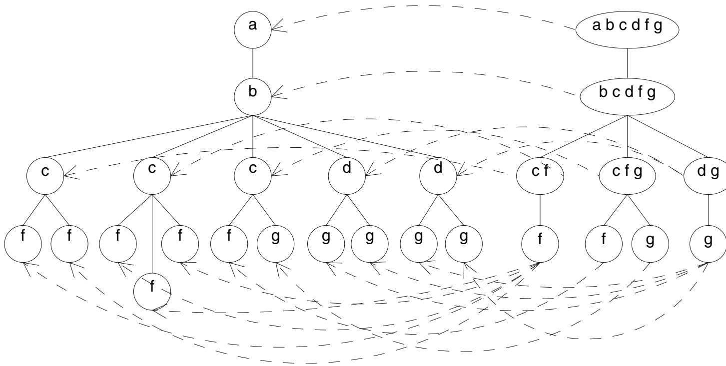


Fig. 4. Sample XML document mapping to its index

4.2 Index Disk Pages

At the implementation level, there are two different internal page structures that make an index. Each one of these page types makes a chain of contiguously clustered disk pages. These two page types are respectively shown on the left and right in Fig. 5. The first chain contains for each index node the name, SL, the pointers to its children index nodes, a count of the number of instances for that element, and a pointer to the location in the second chain holding the pointers to the element(s) in the XML document storage. The order of index nodes stored in this chain is the depth-first ordering of the index tree. In the second chain, we store the pointers to elements in the XML document storage (*i.e.* data pages). With each pointer P, we store the *sIndex* and *eIndex* of the element pointed to by P.

CI	Name	SL	Children	PC	SPL
0	site	open_auctions ...	1	1	0
1	open_auctions	open_auction initial ...	2	1	1
2	open_auction	initial bidder current ...	3, 4, 8, 9, 10	1	2
3	initial			1	3
4	bidder	date time increase	5, 6, 7	3	4
5	date			3	7
6	time			3	10

CI	Name	SL	Children	PC	SPL
7	increase			3	13
8	current			1	16
9	privacy			1	17
10	annotation	description parlist ...	11	1	18
11	description	parlist listitem text ...	12	1	19
12	parlist	parlist listitem text ...	13, 16	1	20

CI	Name	SL	Children	PC	SPL
13	listitem	parlist listitem	14	1	21
14	parlist	listitem	15	1	22
15	listitem			3	23
16	listitem	text bold	17	1	27
17	text	bold	18	1	28
18	bold			1	29

CI	sIndex	eIndex	P
0	0	29	
1	1	29	
2	2	29	
3	3	3	
4	12	15	
5	4	7	
6	8	11	
7	5	5	
8	9	9	
9	13	13	

CI	sIndex	eIndex	P
10	14	14	
11	10	10	
12	6	6	
13	15	15	
14	11	11	
15	7	7	
16	16	16	
17	17	17	
18	18	29	
19	19	29	

CI	sIndex	eIndex	P
20	20	29	
21	21	26	
22	22	26	
23	24	24	
24	25	25	
25	26	26	
26	23	23	
27	27	29	
28	28	29	
29	29	29	

Fig. 5. Sample index layout

Within this second chain of pages, entries are logically grouped together according to the pointers from the first type of pages and their counters; each group belongs to an index node. Physically the pages in the second chain contain entries where each one of these makes a sequence of four integers. Fig. 5 shows in fact how the conceptual index tree is stored in disk pages. Each block represents a physical disk page. The three wide blocks belong to the first chain, and the three other blocks belong to the second chain. Columns “CI” in both page types denote a logical address for that tuple. Column “Children” shows the “CI’s” of the children nodes, “PC” the number of pointers in the group associated with the index node in the second chain. “SPL” shows the position of the group of pointers in the second chain. The small arrows in the last column of the second chain stand for the pointers pointing to the XML data. The groups are ordered in the same order as the index nodes in the first chain. Each group constitutes the *extent* of its index node.

4.3 Index Size

The cost of path expression evaluation is directly proportional to the size of the index tree. The limit size of the index tree depends upon the number of distinct element names and the maximum depth of a specific XML document. An index node i can have a maximum of $n2^n$ children (n if children are leafs), where n is the number of distinct element names in the subtree rooted by i. This limit is reached when for every distinct element name “m” in the subtree of i, there are 2^n subtrees, rooted by 2^n “m”’s, all children of i, with each “m” having a distinct SL. This limit is unlikely to be reached for the following reasons: first, it is unlikely for a node to have a big number of children with distinct names; and second, it is unlikely for elements, with parents of the same name and of the same depth, to have a big number of distinct SL.

5 Path Expression Evaluation

The semantics of XPath makes it necessary to fetch as the result of an XPath expression the subtree rooted at the target element from the queried XML document. However, for XQuery several XPath expressions may exist in the "*For-Let-Where*" parts of the query and only predicates on the target elements (and/or on their attributes and *text()*) in these expressions are applied; hence, there is no need to fetch the whole subtrees for these expressions. Only expressions in the "*Return*" clause of an XQuery need the resulting subtrees if what is required is not explicitly stated as an element name and/or its *text()* value, or attributes for some elements. Our storage manager and U(∗)-Index return both types of required results.

Path expressions requiring only target elements are evaluated using the index only, without accessing the document data pages. Attributes and *text()*'s of these target elements are fetched from data pages. U(∗)-Index returns pointers to the locations of these elements in data pages and, for an element, it costs an average of one disk access to fetch its attributes and *text()*. Since elements at the same document level and with the same name are clustered together in data pages, we have here a high cache hit ratio. This is the main advantage of our data shredding layout.

The result of a path expression evaluation on the index contains exactly all the matches in the document. Consider an XML document D. Let I be the U(∗)-Index built on D. For any index node $i \in I$, the path label sequence from the root of I to i is the same as the path label sequence of all elements e in D that are pointed to by i. Also, all elements $e \in D$ satisfying a path expression p are pointed to by index nodes i's that satisfy p. Thus all i's with path label sequence from root of I to i satisfying a path expression p contain exactly all the matches of p in D (the statement is made in both directions to stress the point that it is equivalent to the *if and only if* condition). We explain next how our U(∗)-Index is used in matching simple and branching expressions.

5.1 Evaluating Simple Expressions

Algorithm 4. is called recursively to evaluate an XPath expression. We next explain how this matching process is done on a sample XPath expression. Consider the *path* expression */site//open_auction/∗/time* to be evaluated on the index in Fig. 3. Line 1 of the algorithm parses the *path* to get *separator*1="/", *label*="*site*", *separator*2="//" and *remainingPath*= "*open_auction/∗/time*".

First, the algorithm checks if the index node under consideration (*indexRoot* in Algorithm 4.) has the same label as the head of *path* (line 3). After the check passes for "*site*", it proceeds to check for branches in *path* (lines 4–10). If no branches are found in the path expression, it proceeds to line 11; otherwise, it pushes these branches on a stack. If the *label* is the leaf of *path*, it adds the extent (calling Algorithm 5.) of the node *indexRoot* to the results (lines 11–12). While the *remainingPath* is not empty, it prepares to evaluate *remainingPath* on *indexRoot*'s children.

Algorithm 4. XPath Evaluation

```
Evaluate (path, indexRoot)
 1: separator1.label.separator2.remainingPath = path
 2: branchAdded = 0
 3: if indexRoot.elementName == label then
 4:    while separator2 = '[' do
 5:       branch=getBranch(remainingPath)
 6:       '['...']'.remainingPath = remainingPath
 7:       pushBranchToStack(branch, indexRoot)
 8:       branchAdded = branchAdded + 1
 9:       separator2.remainingPath = remainingPath
10:    end while
11:    if remainingPath = ϵ then
12:       result = result ∪ returnExtent(indexRoot)
13:    else
14:       if subtreeContains(indexRoot, remainingPath) then
15:          for all child ∈ indexRoot.children do
16:             result = result ∪ Evaluate (separator2.remainingPath, child)
17:          end for
18:       end if
19:    end if
20: end if
21: if separator1 == '//' then
22:    if subtreeContains(indexRoot, path) then
23:       for all child ∈ indexRoot.children do
24:          result = result ∪ Evaluate (path, child)
25:       end for
26:    end if
27: end if
28: while branchAdded ≥ 0 do
29:    popBranchFromStack()
30:    branchAdded = branchAdded − 1
31: end while
32: return result
```

Since every index node has a list of element names in its subtree (SL), Algorithm 6. is called in line 14 to check for any possibility of matches for the *remaining Path* in the subtree of *indexRoot*. Algorithm 6. checks for every *label* in *path* its availability in the index subtree rooted by *indexRoot*. This check is done by simply scanning the SL of *indexRoot*. This algorithm helps in pruning unpromising branches in the index and hence speeds up a lot the matching process.

On our running example, the *remainingPath* has labels "*open_auction*" and "*time*", and both are found in the SL "*open_auctions* ..." of node 0 in Fig. 3, so we recursively call *Evaluate* on the children of *indexRoot* with *separator*2. *remainingPath* which is currently "//*open_auction*/∗/*time*". In this recursive call, *separator*1= "//", *label*= "*open_auction*", *separator*2= "/" and *remaining* − *Path*= "∗/*time*". The condition in line 3 fails, because *indexRoot.element*

Algorithm 5. Return Extent

returnExtent ($indexNode$)

1: **for all** ($branch, stackIndexNode$) in stack **do**
2:   **if** $\neg$ Evaluated($branchExtent[branch]$) **then**
3:     $branchExtent[branch]$=Evaluate ($branch, stackIndexNode$)
4:   **end if**
5: **end for**
6: **for all** $e \in indexNode.extent$ **do**
7:   $matched = true$
8:   **for all** ($branch, stackIndexNode$) in stack **do**
9:     **if** $\nexists\ b \in branchExtent[branch] \mid \exists\ p \in stackIndexNode.extent$ *and* $e.sIndex \geq p.sIndex$ *and* $e.eIndex \leq p.eIndex$ *and* $b.sIndex \geq p.sIndex$ *and* $b.eIndex \leq p.eIndex$ **then**
10:       $matched = false$
11:     **end if**
12:   **end for**
13:   **if** $matched = true$ **then**
14:     $result = result \cup e$
15:   **end if**
16: **end for**
17: return $result$

Algorithm 6. Subtree Contains Elements

subtreeContains ($indexRoot, path$)

1: **for all** $label \in path$ **do**
2:   **if** $label \notin indexRoot.SL$ **then**
3:     return $false$
4:   **end if**
5: **end for**
6: return $true$

$Name$ = "*open_auctions*" and $label$ = "*open_auction*". Since $separator1$="//", the *label* "*open_auction*" is to be evaluated for all descendants of *indexRoot*. Line 24 recursively calls *Evaluate* on the children of *indexRoot* to evaluate "//*open_auction*/∗/*time*" after checking for the possible availability of "*open_auction*" and "*time*" in *SL* of *indexRoot* which is "*open_auction* ..." (node 1 in Fig. 3) in line 22. *indexRoot* "*open_auctions*" has only one child "*open_auction*" that matches the *label* in the following recursive call. The condition in line 14 is satisfied, and *Evaluate* is called with $separator2.remainingPath$ "/∗/*time*" for the children of index node named "*open_auction*". For all calls of *Evaluate* on the children of "*open_auction*" with $path$ = "/∗/*time*", the condition in line 3 would evaluate to *true*, since *label* is a wildcard, but the condition in line 14 will fail for all but "*bidder*", since only index node "*bidder*" has "*time*" in its *SL*. The recursive call of *Evaluate* will continue for index nodes "*date*", "*time*" and "*increase*" with path expression "/*time*". The condition in line 3

will fail for "*date*" and "*increase*" and only "*time*" will jump to line 12 because $remainingPath = \epsilon$. Algorithm 5. is then called and it simply returns the extents of "*time*".

5.2 Evaluating Branching Expressions

Evaluating branches in an XPath expression happens only when the target index node for that expression (*i.e.* the index node corresponding to the leaf element in the expression) has been reached. This evaluation is cached until the depth-first scan of the index tree backtracks the branching index node. Again, running an example here may be the most appropriate way to explain how branches are evaluated. Let's consider a simpler version of our previous example and add a branching expression to it so it becomes */site//open_auction/bidder*[*./date*]*/time*.

After encountering the "*/date*" branching in line 4 of Algorithm 4., it is pushed onto a global stack along with the branching index node "*bidder*". The matching process proceeds as it was explained in the previous section, up until a "*time*" element matches. At that time, and while fetching the extent of "*time*" in Algorithm 5., the "*/date*" branching is evaluated on the subtree rooted by "*bidder*", if it hasn't been evaluated previously. For each extent of "*time*", Algorithm 5. checks in line 9 if it matches a "*date*" extent with the same parent "*bidder*"; if so, it adds that "*time*" extent to the results. When the recursive call that pushed that branch onto the stack reaches line 28 (*i.e.* backtracks the branching node), it pops up that branch from the stack and deletes it.

5.3 Analysis

We present in this subsection some analysis on the average size of a U($*$)-Index for a certain XML document and on the average evaluation cost for an XPath expression relative to its characteristics.

Index Size: Consider a schema for an XML document of depth d and let μ be the average schema child count, and ν be the average schema optional descendant count. Let $\alpha \in [0..1]$ be a parameter associated with the presence of the optional descendants; when $\alpha = 1$ then the appearance of the optional elements in subtrees makes all possible combinations (*i.e.* 2^ν), and when $\alpha = 0$ then the appearance of optional elements in subtrees makes a single combination. α is almost always very close to 0. Let $\mathcal{X}$ be the average number of nodes in the index tree; then

$$\mathcal{X} = \sum_{i=0}^{d} (\mu 2^{\alpha\nu})^i \tag{1}$$

Evaluation Cost: Let β be the number of nodes visited to evaluate a sub-path expression p_j of the format "*separator label*" from an expression $p = p_1 p_2 \ldots p_n$ on an index node i, where *separator* can be "/" or "//", and "*label*" can be an

element name or a wildcard. Let d_i be the depth of that index node i and d_e be the depth of the deepest occurrence of element named *label*; then

$$\beta(p_j) = \begin{cases} 2^{\alpha\nu} & \text{if } label \neq * \text{ AND } separator = / \\ \mu 2^{\alpha\nu} & \text{if } label = * \text{ AND } separator = / \\ \sum_{x=d_i}^{d_e} (2^{\alpha\nu})^{x-d_i} & \text{if } label \neq * \text{ AND } separator = // \\ \sum_{x=d_i}^{d_e} (\mu 2^{\alpha\nu})^{x-d_i} & \text{if } label = * \text{ AND } separator = // \end{cases}$$

The overall cost of evaluating a path expression of depth d_e is:

$$\prod_{j=1}^{n} \beta(p_j) \tag{2}$$

For path expressions with branching, the additional cost for a branch is:

$$\sum_{\forall x} \beta(p_j)\beta(p_b) \tag{3}$$

where p_j is the separator and branching node label, p_b is the separator and the branch leaf label and x is the number of index nodes matching the branching node.

Although one can see that these formulae have exponential complexities, they are rarely exponential in reality because in most cases $\alpha \rightarrow 0$ and both μ and ν tend to 1 as the processing goes down the index tree.

6 Experimental Results

We present in this Section the performance results of our proposed data shredding scheme and its U(∗)-Index. The implementation of these two components went through several refinements, and we also implemented the D(k)-Index in order to compare its execution times against those of U(∗)-Index and to cross-check the correctness of our implementation by making sure that both indexes returned the same results for about 70 queries. The codes for the data page manager, U(∗)-Index and D(k)-Index were all implemented in Java and made about 15,000 lines of code. Both data loading and index building were performed from and into normal disk files. We conducted our experiments on a Pentium 4 machine with 3.2GHz processor and 512MB RAM, running Linux Fedora. The set of path expressions that were used in our experiments were always evaluated with an initially empty buffer. We ran in fact a large set of experiments where different parameters were varied. For lack of space, we only show here some representative results, but we would like to mention that with all the variations in the experiments, most results went in the same direction. For all performance results, the average of 5 runs for a query is reported here.

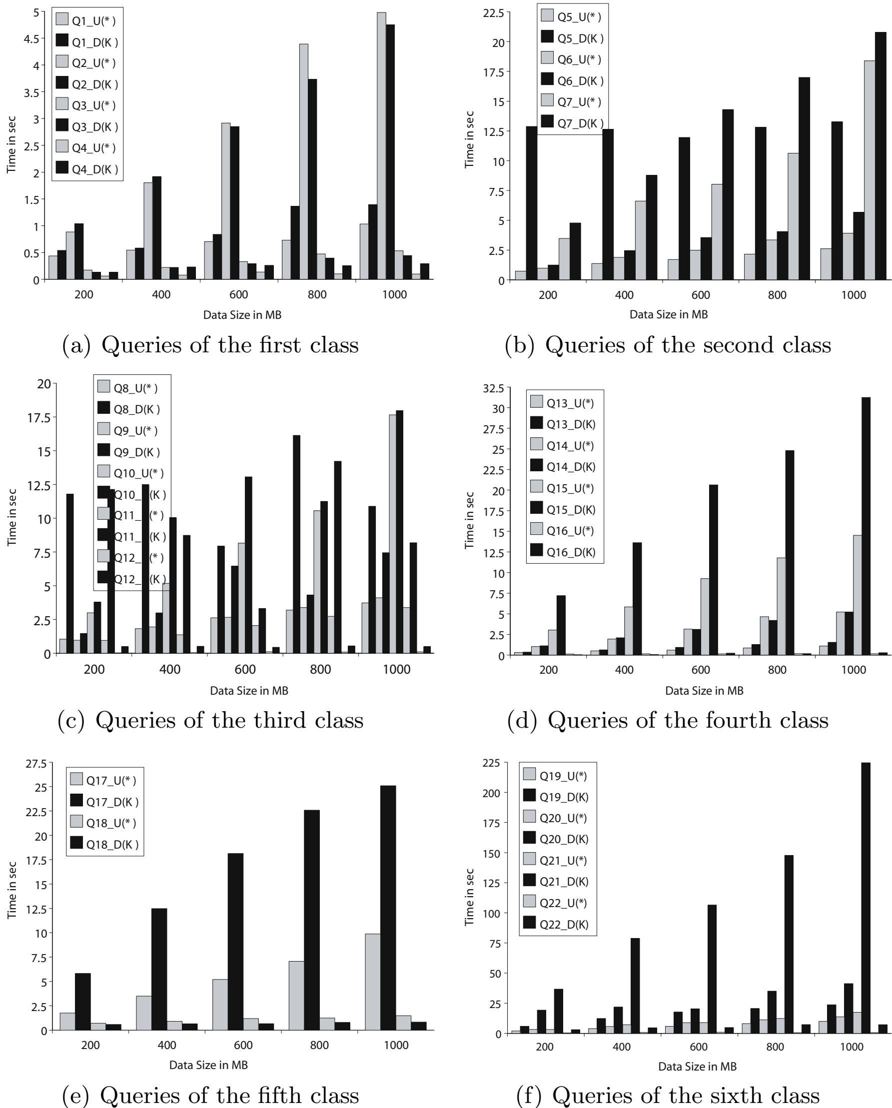
Datasets: We used as our data and path expressions the XMark benchmark [16] that is becoming widely used for its typical and representative data and queries. We generated documents of sizes 200, 400, 600, 800 and 1000 MB and loaded them into the storage manager and then built their indexes. The D(k) index was also built on the same data layout as U($*$), so that this data layout does not affect the performance comparison of the two indexes. For both U($*$)-Index and D(k)-Index, their sizes were about 25% of the XML document sizes.

Path expressions: We extracted a list of path expressions from XMark [16], and modified some of them to have path expressions with single, double and triple // along with their normal /; we also introduced path queries with wildcards and branches. The list of these expressions is shown in the technical report of this paper [1]. We also divided these expressions into six different classes according to the count of //, $*$, and branching present in an expression. Although we ran our experiments with about 70 XPath expressions, we report the results here of only 22 expressions for lack of space and similarity in the results.

Performance comparison of U($*$) versus D(k): Fig. 6 shows the results of both D(k) and U($*$) when evaluating the different queries on the five different data sizes. From this figure we can state that when the expressions are relatively simple (*e.g.* query classes 1, 2, and 3) the performance of both D(k) and U($*$) are comparable, with a slight advantage for U($*$). However, when the queries become complex (*e.g.* query classes 4, 5, and 6) U($*$) largely beats D(K) up to an order of magnitude on some queries. The technical report shows next to each XPath expression the number of elements returned by that expression on the 200 MB data size; these numbers justify some high execution times for some expressions. We must mention here that we ran all the reported queries on D(k) in advance and it took several minutes of processing every time to adapt to a query. The results reported here for D(k) are those after the adaptation pre-processing took place. For U($*$), and as we mentioned earlier, no adaptation is needed.

Subtree reconstruction: Fig. 7(a) shows a comparison for the execution times of sample queries from the six different classes when the required results are only the target elements in the XPath expressions versus the subtree reconstruction for these elements. The figure plots the execution times for only two data sizes since for higher sizes an extrapolation can easily be made by the reader. For these expressions, the subtree reconstruction times are quite high, but when the sizes of these subtrees are carefully looked at (see Table 1), one can understand these costs. These result sizes are very large and not typical, and are to a certain extent extreme cases. For Q18, for example, where the returned element is a leaf element in the original XML document and hence does not have a subtree, the cost of accessing the data pages is almost negligible.

Top 500 results: Another set of results that we show here are related to the cost of fetching the first 500 matching elements for different expressions from the six query classes. Fig. 7(b) compares for six different expressions the execution times



(a) Queries of the first class

(b) Queries of the second class

(c) Queries of the third class

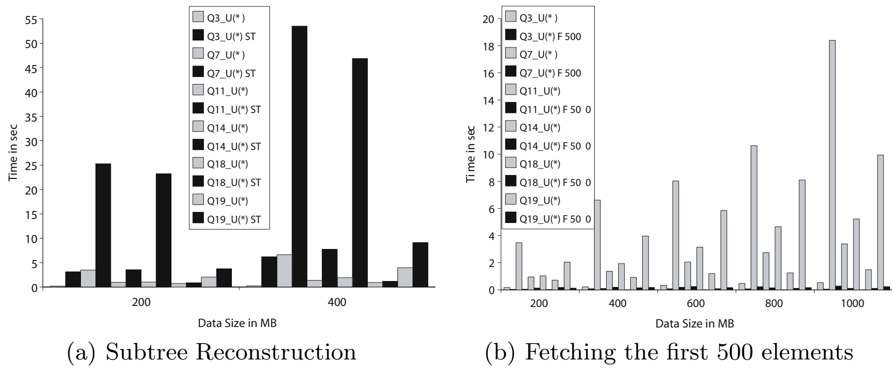
(d) Queries of the fourth class

(e) Queries of the fifth class

(f) Queries of the sixth class

Fig. 6. Performance of both D(k) and U($*$) on the different data sizes of XMark

when the whole results are returned versus the times needed to return the first 500 matching elements for each expression. From that graph, we can observe how the response times remained within the realm of few tens to few hundreds of milliseconds when only the top 500 elements for all the queries were returned and when the data set was scaled up to 1GB. This shows that the scalability of U($*$)-Index is not greatly affected by the size of the XML data, but mostly by its number of levels and the variations in element names and their relative positions.



(a) Subtree Reconstruction (b) Fetching the first 500 elements

Fig. 7. Performance of U(∗) for subtree reconstruction and top 500 results

Table 1. Number of Returned Elements on 200 MB

	Elements Only	Element with Subtree
Q3	4400	113768
Q7	121537	635919
Q11	33287	100243
Q14	51000	662212
Q18	3600	3600
Q19	26401	21596140

7 Conclusion

We presented in this paper the design, implementation and performance results of a native storage manager for XML data. A tightly coupled structural index to that layout was also presented. Our experimental results validate this design and show the comparable performances of U(∗)-Index relative to the D(k)-index. The major advantage of U(∗)-Index relative to the D(k)-index is its generality and non-need for adaptation to frequent queries.

Although not mentioned in this paper, we already augmented our structural index with an inverted index—that is quite similar to a B-tree—to fetch results of XPath expressions having value predicates on element texts and attribute values. We intend in a coming paper to publish results on the performance of this inverted index when used with our structural index. We are currently in the process of implementing another structural index that has been recently published and plan to compare its performance against our U(∗)-Index.

References

1. Boulos, J., Awada, R., Abdel-Kader, R., Hashem, A., Karakashian, S., El-Sebaaly, J.: The Al$\mathcal{X}$emist project, (http://www.cs.aub.edu.lb/boulos/AlXemist.htm)
2. Fiebig, T., Helmer, S., Kanne, C.C., Moerkotte, G., Neumann, N., Schele, R., Westmann, T.: Anatomy of a native XML base management system, The VLDB Journal, 11(4), 292-314 (2002)

3. Jagadish, H.V., Al-Kalifa, S., Chapman, A., Lashmanan, L.V.S., Nierman, A., Paparizos, S., Patel, J.M., Srivastava, D., Wiwatwattana, N., Wu, Y., Yu, C.: Timber: A native XML database, The VLDB Journal, 11(4), 274-291 (2002)
4. Kaushik, R., Shenoy, P., Bohannon, P., Gudes, E.: Exploiting local similarity for indexing paths in graph-structured data, IEEE ICDE (2002)
5. Chen, Q., Lim, A., Ong, K.W.: D(k)-index: An adaptive structural summary for graph-structured data, ACM SIGMOD (2003)
6. Wu, H., Wang, Q., Yu, J.X., Zhou, A., Zhou, S.: UD(k and l)-index: An efficient approximate index for XML data, China, Fourth Int. Conf. Web-Age Information Management, WAIM, Springer-Verlag LNCS 2762 (2003)
7. He, H., Yang, J.: Multiresolution indexing of XMl for frequent queries, IEEE ICDE (2004)
8. Manolescu, I., Florescu, D., Kossmann, D., Xhumari, F., Olteanu, D.: Agora: Living with XML and relational, 26th VLDB Conference (2000)
9. Tatarinov, I., Viglas, S.D., Beyer, K., Shanmugasundaram, J., Shekita, E., Zhang, C.: Storing and querying ordered xml data using a relational database system, ACM SIGMOD (2002)
10. Florescu, D., Kossmann, D.: Storing and querying XML data using an rdbms, IEEE Data Eng. Bull., 22(3), pp. 27-34 (1999)
11. Deutsch, A., Fernandez, M., Suciu, D.: Storing semistructured data with stored, ACM SIGMOD (1999)
12. Kanne, C.C., Moerkotte, G.: Efficient storage of XML data, IEEE ICDE (2000)
13. Chung, C.W., Min, J.K., Shim, K.: APEX: An adaptive path index for XML data, ACM SIGMOD (2002)
14. Milo, T., Suciu, D.: Index structures for path expressions, Proceeding of the 7th International Conference on Database Theory (1999)
15. Al-Khalifa, S., Jagadish, H.V., Koudas, N., Patel, J.M., Srivastava, D., Wu, Y.: Structural joins: A primitive for efficient XML query pattern matching, IEEE ICDE (2002)
16. Schmidt, A., Waas, F., Kersten, M., Carey, M.J.: Xmark: A benchmark for XML data management, 28th VLDB Conference (2002)