Available online at www.sciencedirect.com

ScienceDirect

Data & Knowledge Engineering 59 (2006) 681–699

DATA & KNOWLEDGE ENGINEERING

www.elsevier.com/locate/datak

Indexing XML documents for XPath query processing in external memory

Qun Chen [a,*], Andrew Lim [a], Kian Win Ong [b], Jiqing Tang [a]

[a] *Department of Industrial Engineering and Engineering Management, Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong*

[b] *Department of Computer Science and Engineering, University of California at San Diego, 9500 Gilman Drive, La Jolla, CA 92093-0114, United States*

Received 6 May 2005; accepted 1 November 2005
Available online 1 December 2005

Abstract

Existing encoding schemes and index structures proposed for XML query processing primarily target the containment relationship, specifically the *parent–child* and *ancestor–descendant* relationship. The presence of *preceding-sibling* and *following-sibling* location steps in the XPath specification, which is the de facto query language for XML, makes the horizontal navigation, besides the vertical navigation, among nodes of XML documents a necessity for efficient evaluation of XML queries. Our work enhances the existing range-based and prefix-based encoding schemes such that all structural relationships between XML nodes can be determined from their codes alone. Furthermore, an external-memory index structure based on the traditional *B*+-tree, *XL*+-tree(XML Location+-tree), is introduced to index element sets such that all defined location steps in the XPath language, *vertical* and *horizontal*, *top-down* and *bottom-up*, can be processed efficiently. The *XL*+-trees under the range or prefix encoding scheme actually share the same structure; but various search operations upon them may be slightly different as a result of the richer information provided by the prefix encoding scheme. Finally, experiments are conducted to validate the efficiency of the *XL*+-tree approach. We compare the query performance of *XL*+-tree with that of *R*-tree, which is capable of handling comprehensive XPath location steps and has been empirically shown to outperform other indexing approaches.
© 2005 Elsevier B.V. All rights reserved.

Keywords: XML; XPath query language; External-memory index structure; *B*+-tree

1. Introduction

As XML gains unprecedented popularity as the standard format for presenting and exchanging information over the Internet, the XML database floats as a suitable, semi-structured alternative to store data. Documents in an XML database are usually parsed into ordered tree structures.

[*] Corresponding author.
E-mail addresses: qunchen@ust.hk (Q. Chen), iealim@ust.hk (A. Lim), okwin@ucsd.edu (K.W. Ong), ie05tj@alumni.ust.hk (J. Tang).

0169-023X/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.datak.2005.11.002

Various schemes have been proposed for indexing XML data, the most popular of which involves storing inverted lists of elements based on their XML tag. These elements are often augmented with additional labels. One such technique used is range labeling [1–3]. This scheme encodes each element, v, with a pair of integers $(LP(v), RP(v))$, in which $LP(v)$ and $RP(v)$ represent the left and right positions of v on the XML tree respectively. An element v is an ancestor of u iff $LP(v) < LP(u) < RP(u) < RP(v)$. Prefix labeling [4], as another example, labels each element with a unique string S such that an element v is the ancestor of u iff $S(v)$ is a prefix of $S(u)$. We note, however, that these indexes were designed primarily to facilitate the containment relationship evaluation. Although the containment relationship is indeed the most basic and important structural pattern, other structural relationships are just as relevant to XML query processing.

The XML Path Language (XPath) [5] has emerged as the de facto standard for navigating XML documents. The core addressing capability of XPath lies in the location path, which is used to locate nodes on an XML tree. A location path begins with a context node (not necessarily the root node), which serves as the starting point of the tree traversal, and consists of a series of location steps. Given a context node, a step's axis establishes the subset of document nodes that are reachable from this context node via the specified axis. In turn, this set of nodes provides the context nodes for the next location step. There are altogether 13 different axes defined in XPath, namely: *child*, *parent*, *descendant*, *ancestor*, *following-sibling*, *preceding-sibling*, *following*, *preceding*, *descendant-or-self*, *ancestor-or-self*, *self*, *attribute* and *namespace*. The axes' semantics are evident from their chosen names. We do note, however, the non-obvious property of the *following* axis that excludes the context node's descendants when locating nodes after it in document order. Likewise, the *preceding* axis excludes ancestors when locating nodes before the context node in document order.

Since attributes and namespaces can be simply treated as special instances of element nodes, axes *attribute* and *namespace* have essentially the same structural meaning as the axis *child* in the evaluation of the location path. The axis *self* has no evaluation cost. Axes *descendant-or-self* and *ancestor-or-self* are the same as axes *ancestor* and *descendant* respectively, with the addition of the context node. It would thus be the remaining four pairs of axes that are of primary interest to us. The pair of axes, *child* and *descendant*, represents the vertical top-down traversal. The pair *parent* and *ancestor* makes up the vertical bottom-up traversal. Axes *following-sibling* and *preceding-sibling* typifies the horizontal traversal. Lastly, axes *following* and *preceding* represent traversing the document in forward and backward order, and as shall be presented later, can be easily decomposed into a combination of the previous three kinds of traversals.

It is clear now that, besides the well studied containment relationship, the XPath language also specifies the sibling structural relationship between XML elements. The *preceding-sibling* and *following-sibling* axes enable the horizontal navigation among tree nodes, which we believe is an important query pattern for XML database. Therefore, sibling structural join, as well as the containment structural join, should be dealt with while building index structures for XML databases.

In this paper, we first enhance the traditional range-based and prefix-based labeling schemes to ensure all structural relationships specified in the XPath language between two nodes be determined from their codes alone. Then we proceed to propose an $B+$-tree based external-memory index structure, $XL+$-tree, which facilitates comprehensive types of structural navigation on XML trees. The $XL+$-tree targets three types of search problems corresponding to the top-down, bottom-up and horizontal navigations among XML tree nodes respectively. Let B denote the page size and k denote the total number of indexed entries. Our major results can be summarized as follows:

(1) *Analytical results* (regardless of the underlying encoding scheme, range or prefix). The *descendant* search operation takes $\mathbf{O}(\log_B k + \frac{ds}{B})$ worst-case disk accesses, where ds is the total size of descendants. The *child* search operation takes $\mathbf{O}(\log_B k + rd)$ worst-case I/Os, where rd is the number of disk pages storing children. Note that rd may not be equal to $\frac{ds}{B}$ because children may not be stored contiguously on the $XL+$-tree. The *following-sibling* and *preceding-sibling* search operations both take the $\mathbf{O}(\log_B k + rd)$ worst-case I/O cost as well. The *parent* search operation takes $\mathbf{O}(\log_B k)$ worst-case I/O cost, while the *ancestor* search takes $\mathbf{O}(lv \times \log_B k)$ I/O cost in the worst case, where lv is the level number of the input entry. For the update operations on $XL+$-tree, both the insertion and deletion operation take $\mathbf{O}(\log_B k)$ amortized I/O cost. The comparison of analytical performance between the $XL+$-tree approach and the

Table 1
Analytical performance comparison between *XL*+-tree and *B*+-tree

	XL+-tree	*B*+-tree
descendant search	$\mathbf{O}(\log_B k + \frac{ds}{B})$	$\mathbf{O}(\log_B k + \frac{ds}{B})$
child search	$\mathbf{O}(\log_B k + rd)$	$\mathbf{O}(\log_B k + \frac{ds}{B})$
parent search	$\mathbf{O}(\log_B k)$	$\mathbf{O}(\log_B k)$
ancestor search	$\mathbf{O}(lv \times \log_B k)$	$\mathbf{O}(\log_B k) + \frac{pbs}{B}$
following-sibling search	$\mathbf{O}(\log_B k + rd)$	$\mathbf{O}(\log_B k) + \frac{pas}{B}$
preceding-sibling search	$\mathbf{O}(\log_B k + rd)$	$\mathbf{O}(\log_B k) + \frac{pbs}{B}$
update (insertion)-amortized cost	$\mathbf{O}(\log_B k)$	$\mathbf{O}(\log_B k)$
update (deletion)-amortized cost	$\mathbf{O}(\log_B k)$	$\mathbf{O}(\log_B k)$

naive *B*+-tree approach (both index the left positions of entries) has been presented in Table 1, in which *pbs* is the total size of entries that are before the query entry in document order and *pas* is the total size of entries that are after the query entry in document order but not its descendants.

(2) *Experimental evaluation*. As far as we know, there is no external-memory index structure specifically designed for handling comprehensive Xpath location steps. The Xpath query accelerator, proposed in [18], encodes each node in an XML tree with a multi-dimensional descriptor and takes advantage of traditional *R*-tree and *B*-tree to support various Xpath locating processes on a relational engine. Since *R*-tree has been shown to outperform *B*-tree in their experiments, to validate the effectiveness of the *XL*+-tree, we compare its performance with that of *R*-tree. Our experiments on both benchmark and synthetic XML data demonstrate that the *XL*+-tree outperforms *R*-tree by wide margins in most cases in term of both I/O and CPU cost.

The rest of this paper is organized as follows. We present the XML data model and enhanced encoding schemes in Section 2. Section 3 is devoted to describing the search and update operations on the *XL*+-tree based on the range encoding scheme. The *XL*+-tree based on the prefix encoding scheme is discussed in Section 4. Preliminary experiment results are presented in Section 5. We discuss more related work in Section 6 and conclude this paper with Section 7.

2. XML data model and enhanced encoding schemes

An XML document is usually parsed into an ordered, labeled tree, with each node in the tree corresponding to an element, an attribute or text data. Edges between nodes represent element–subelement or element–attribute relationships. An example of the XML data model is shown in Fig. 1. An XML database consists of a forest of such trees.

2.1. Range-based encoding scheme

In the traditional range encoding scheme, positions of nodes in XML trees are represented by 3-tuple (*DocNo*, *LP*:*RP*, *lv*). *DocNo* is the identifier of document. The pair of *LP* and *RP* can be generated by doing a depth-first traversal of the tree and sequentially assigning a number at each visit. *lv* is the nesting depth of nodes in the tree.

With the range encoded representation of an XML tree, the containment structural relationship between tree nodes can be determined easily: (1) containment or ancestor–descendant: a tree node v, ($LP(v)$:$RP(lv)$, $lv(v)$), contains a tree node u, ($LP(u)$:$RP(u)$, $lv(u)$), iff $LP(v) < LP(u)$ and $RP(v) > RP(u)$; (2) direct containment or parent–child: a tree node v directly contains u iff $LP(v) < LP(u)$, $RP(v) > RP(u)$ and $lv(v) = lv(u) - 1$. One important advantage of this presentation is that checking an ancestor–descendant structural relationship is as easy as checking a parent–child structural relationship.

While the traditional range encoding of XML trees is sufficient to determine the *parent/child* and *ancestor/descendant* relationship between tree nodes, it does not capture the *preceding-sibling/following-sibling* relationship. The enhanced range encoding scheme represents each node with a three-dimensional descriptor:

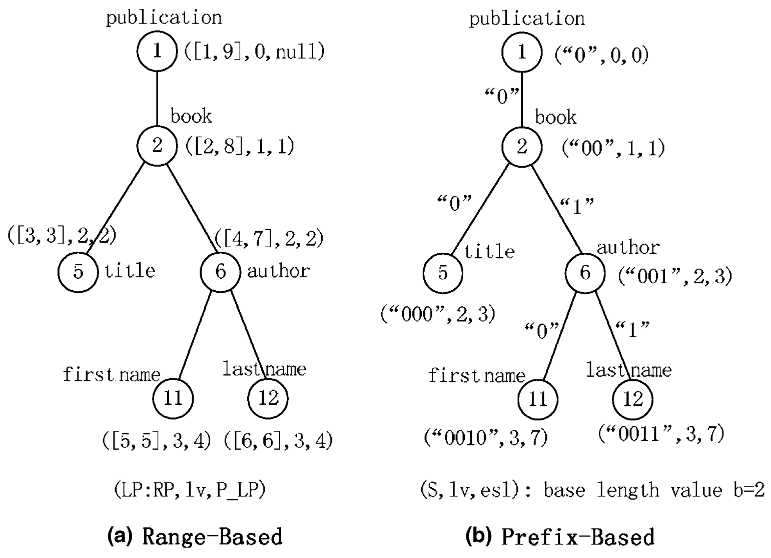

Fig. 1. An example XML data model and enhanced encoding schemes.

(*LP*:*RP*, *lv*, *PLP*), in which *PLP* is its parent node's left position. An example of the enhanced range encoding scheme is shown in Fig. 1(a). Note that we assume that nodes are from the same document and ignore the *DocNo* information from the descriptor. Extending it to handle nodes across multiple documents should be trivial. Based on this encoding scheme, the structural relationship between XML tree nodes can be determined as follows:

(1) *descendant*. Node u is a descendant node of v iff $LP(u) > LP(v)$ and $RP(u) < RP(v)$;
(2) *child*. Node u is a child node of v iff $LP(u) > LP(v)$, $RP(u) < RP(v)$ and $lv(u) = lv(v) + 1$;
(3) *ancestor*. Node u is an ancestor node of v iff $LP(u) < LP(v)$ and $RP(u) > RP(u)$;
(4) *parent*. Node u is a parent node of v iff $LP(u) = PLP(v)$;
(5) *following-sibling*. Node u is the following-sibling node of v iff $LP(u) > LP(v)$ and $PLP(u) = PLP(v)$;
(6) *preceding-sibling*. Node u is the preceding-sibling node of v iff $LP(u) < LP(v)$ and $PLP(u) = PLP(v)$;
(7) *following*. Node u is the following node of v iff $LP(u) > RP(v)$;
(8) *preceding*. Node u is the preceding node of v iff $RP(u) < LP(v)$.

Note that the *XL*+-tree is built using the *LP* values of node descriptors as keys. Given a context node v, the *following* location step can be accomplished by a simple range search that identifies those data nodes satisfying $LP > RP(v)$. Since node v's *preceding* nodes are defined to be those nodes with $LP < LP(v)$, but excluding v's ancestor nodes, the *preceding* location step can be accomplished by a range search followed by an *ancestor* search operation. Given an input descriptor $D(v)$, $(LP(v){:}RP(v), lv(v), PLP(v))$, the following six search problems corresponding to six basic location steps are critical to the XPath processing, therefore, are of primary interest to us.

- *Top-Down Search*: Descendant_Search($D(v)$) and Child_Search($D(v)$).
- *Bottom-Up Search*: Ancestor_Search($D(v)$) and Parent_Search($D(v)$).
- *Horizontal Search(RES)*: Following-Sibling_Search($D(v)$) and Preceding-Sibling_Search($D(v)$).

2.2. Prefix-based encoding scheme

In the prefix labeling scheme, we encode each node with a unique string S such that: (1) $S(v)$ is before $S(u)$ in lexicographic order iff node v is before node u in document order; (2) $S(v)$ is a prefix of $S(u)$ iff node v is the ancestor of node u. Informally, the document order in an XML tree orders its nodes corresponding to a sequential read of nodes by a preorder traversal. One simple example prefix scheme works as follows. We assign to the out-going edges of each node a set of prefix-free binary strings. From left to right, strings assigned to edges are in lexicographic order. Then, starting from the root and going down, we define the label of each node to be the concatenation of its parent's label and the string assigned to its incoming edge. Consider, for example, a node v has two children, v_1 and v_2, and v_1 is before v_2. We can assign string "00" to edge (v, v_1), string "01" to edge (v, v_2). So the label string of v_1, $S(v_1) = S(v) \bullet$ "00"; the label string of v_2, $S(v_2) = S(v) \bullet$ "01". The prefix labeling example of an XML tree is given in Fig. 1(b). Please note that the problem of how to label nodes in the XML tree using the shortest possible string in the static or dynamic setting is beyond the scope of this paper. We use the above-mentioned scheme to explain our results. However, our results are valid for any prefix labeling scheme satisfying the above two conditions.

As in the range-based labeling scheme, we record nodes' level numbers to distinguish the parent–child and ancestor–descendant relationship. Another parameter we keep for each node records lengths of the strings assigned to edges over its incoming path from the root. We have the following two observations:

(1) The basic component of strings, *character* (referred to as the *char* data type in most programming languages), has up to 256 distinct values; therefore, the strings with maximum length of 5 can represent up to 256^5 ($\gg$1 billion) distinct values. Thus, the length of strings assigned to edges in an XML tree can afford to be small.
(2) The maximal level of the XML tree can be expected to be small also. Authors in [4] said that the average depth of XML files collected by a crawler over the web is low; the trees are balanced with relatively high degrees. The popular DBLP document and Xmark benchmark data have the maximal level of $\leqslant$12.

Suppose that the maximal length of labeling strings assigned to edges of an XML tree is m. We set the base length value b to be $(m + 1)$. A node v at level k (with the root at level 0) has the incoming path of $v_0v_1 \dots v_k$, in which node v_0 is the root of the XML tree and $v = v_k$. And the length of the labeling string over edge $v_{i-1} \to v_i$, for each $1 \leqslant i \leqslant k$, is e_{i-1}. We add an integer parameter, the *edge string length* $esl = e_0 \times b^{k-1} + e_1 \times b^{k-2} + \cdots + e_{k-1} \times b^0$, to each node v's descriptor. Obviously, we can determine the values of all e_is from esl's value, specifically $e_i = \lfloor \frac{esl}{b^{((k-1)-i)}} \rfloor$ %(*modula*)b. The *edge string length* parameter will be used to extract label strings of a node' ancestors. This completes our enhanced prefix encoding scheme. Each node v on the XML tree is represented by a three-dimensional descriptor: (S, lv, esl), in which, lv is the nesting level of node v and esl is the *edge string length*. An example of the enhanced prefix encoding scheme is shown in Fig. 1(b).

Under this prefix encoding scheme, the structural relationships specified in the XPath language can be determined correspondingly. The function *prefix*(S, i) returns the string consisting of the first i characters of string S. We denote $S(v) < S(u)$ iff $S(v)$ is before $S(u)$ in lexicographic order, and $S(v) > S(u)$ iff $S(v)$ is after $S(u)$ in lexicographic order. The lengths of strings assigned to edges of node v's incoming path, are denoted as $e_0(v), e_1(v), \dots, e_k(v)$ $(k = lv(v) - 1)$ in the order from the root to v.

(1) *descendant*. Node u is a descendant node of v iff $S(v)$ is a prefix of $S(u)$;
(2) *child*. Node u is a child node of v iff $S(v)$ is a prefix of $S(u)$, and $lv(u) = lv(v) + 1$;
(3) *ancestor*. Node u is an ancestor node of v iff $S(u)$ is a prefix of $S(v)$ and $lv(u) < lv(v)$;
(4) *parent*. Node u is a parent node of v iff $S(u) = prefix(S(v), |S(v)| - e_k(v))$;
(5) *following-sibling*. Node u is the following-sibling node of v iff $S(u) > S(v)$, $lv(u) = lv(v)$, and *prefix*$(S(v), |S(v)| - e_k(v))$ is a prefix of $S(u)$;
(6) *preceding-sibling*. Node u is the preceding-sibling node of v iff $S(u) < S(v)$, $lv(u) = lv(v)$, and *prefix*$(S(v), |S(v)| - e_k(v))$ is a prefix of $S(u)$;
(7) *following*. Node u is the following node of v iff $S(u) > S(v)$, and $S(v)$ is **NOT** a prefix of $S(u)$;
(8) *preceding*. Node u is the preceding node of v iff $S(u) < S(v)$, and $S(u)$ is **NOT** a prefix of $S(v)$.

Similar to the case of the range encoding scheme, nodes *preceding* a given node v are those nodes whose labeling strings are smaller than $S(v)$, but excluding node v's ancestors. Thus, the *preceding* location step can be solved by performing a range string search followed by a string search corresponding to the *ancestor* location step. Nodes *following* a given node v should have a labeling string larger than $S = \mathit{prefix}(S(v), |S(v)| - e_k(v)) \bullet \omega$, in which $k = lv(v) - 1$, $\bullet$ is a concatenation operator and ω is an imaginary character larger than any other character. Therefore, the *following* location step actually amounts to a range search on strings. The three search problems under the prefix encoding scheme (*PES*), which correspond to six basic XPath location steps, can be presented similarly as under the range encoding scheme.

It is interesting to note that while the top-down search problem is similar to the string prefix search problem well studied in previous literature; the bottom-up and horizontal search problems are specific to the XPath evaluation.

3. The *XL*+-tree based on range encoding scheme

The *XL*+-tree under the range encoding scheme is an extension of the *B*+-tree index data structure, in which node descriptors are stored on leaf disk pages and all leaves are linked sequentially. Entries are sorted according to left positions (*LP*). In the XPath specification, each location step usually comes with a node test, specifically an element tag test. Therefore, in our design, an *XL*+-tree is built for each tag in XML documents. In case that there are so many distinct tags that *XL*+-trees may flood the XML query engine, node descriptors with different tags can be actually indexed on a single *XL*+-tree; but it has a composite key, (*tag*, *LP*). For convenience of explanation, we will focus on the *XL*+-tree built for a single tag in this section. Extending it to handle multiple tags is straightforward.

The overall structure of *XL*+-tree is shown in Fig. 2. Each entry in the *XL*+-tree leaf pages consists of the descriptor and two pointers, one referring to its immediate preceding sibling and the other referring to its immediate following sibling. The structure of the *XL*+-tree's internal node is the same as in the *B*+-tree except that we store two additional integers on each reference to its child page. These two integers record the minimal and maximal level(*lv*) of entries in the corresponding subtree. As it will be shown later, the pair of additional integers is for efficiently identifying the *first child/sibling* of a given context node; the pair of pointers in each entry is for facilitating the horizontal navigation.

To make reading easier, the notations used in this paper are summarized in Table 2.

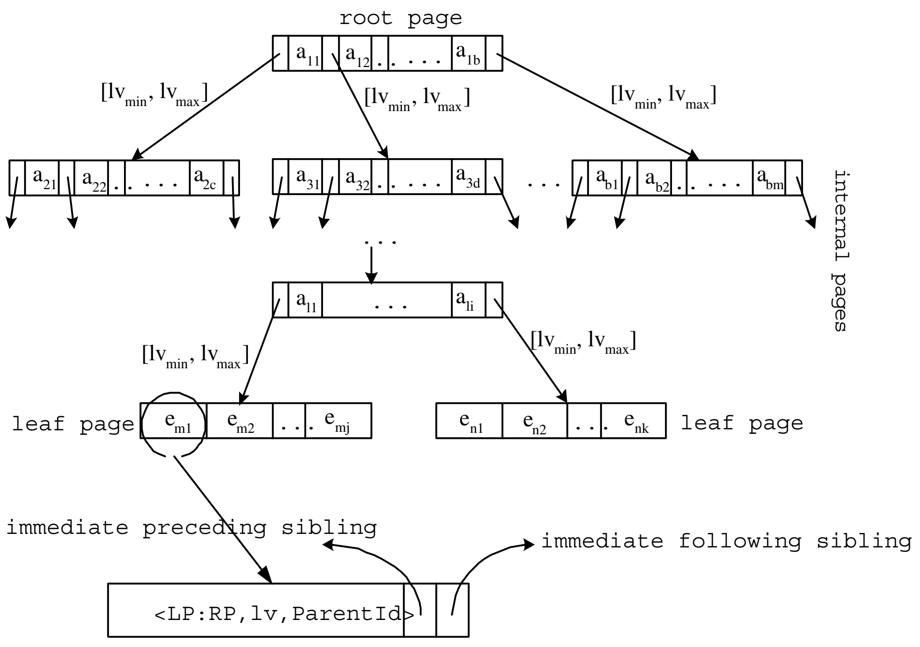
root page
a_{11} a_{12} . . . . a_{1b}
$[lv_{min}, lv_{max}]$
$[lv_{min}, lv_{max}]$
$[lv_{min}, lv_{max}]$
a_{21} a_{22} . . . . a_{2c}
a_{31} a_{32} . . . . a_{3d}
a_{b1} a_{b2} . . . . a_{bm}
internal pages
a_{l1} . . . a_{li}
$[lv_{min}, lv_{max}]$
$[lv_{min}, lv_{max}]$
leaf page
e_{m1} e_{m2} . . . e_{mj}
e_{n1} e_{n2} . . . e_{nk}
leaf page
immediate preceding sibling
immediate following sibling
<LP:RP,lv,ParentId>

Fig. 2. The overall structure of *XL*+-tree.

Table 2
List of notations

Symbol	Meaning
v or v_i	Data node labeled V
u or u_i	Data node labeled U
e or e_i	Entry on the XL+-tree
$LP(v)$	Data node v's left position
$RP(v)$	v's right position
$lv(v)$	v's nested level on XML tree
$lv_{\min}$	Minimal level number stored over a page reference
$lv_{\max}$	Maximal level number stored over a page reference
$PLP(v)$	v's parent node's left position
$S(v)$	v's prefix label string
$esl(v)$	v's edge string lengths in the prefix encoding scheme
$D(v)$	v's node descriptor
ds	Number of v's descendants
rd	Number of disk pages storing search results
p_i	Page on the XL+-tree
pos_j	Position on the page of XL+-tree
PR_j	Page reference on the XL+-tree

3.1. Search operations on XL+-tree

Given a target node descriptor, $D(v)$, its position in the XL+-tree is defined to be the position of the leftmost entry whose LP is $\geqslant LP(v)$. We denote its position by (p_i, pos_j), with p_i representing the leaf disk page and pos_j representing the position on this disk page. The procedure, *Find_Position*$(D(v))$, which identifies $D(v)$'s position, can be implemented by repeatedly performing the binary search on keys stored on pages of the XL+-tree. Its details are omitted here since it is the standard operation on the traditional B+-tree.

3.1.1. Top-down search: descendant and child

Since $D(v)$'s descendent are entries satisfying $LP(v) < LP < RP(v)$, the *Search_Descendant*$(D(v))$ operation amounts to a range search on the XL+-tree. It can simply be implemented by the *Find_Position*$(D(v))$ operation followed by sequentially scanning entries until $LP \geqslant RP(v)$. Therefore, the *Search_Descendant*$(D(v))$ operation's worst case I/O cost is $\mathbf{O}(\log_B k + \frac{ds}{B})$, in which ds is the number of descendants.

For the *Search_Child*$(D(v))$ operation, we have the observation that, once $D(v)$'s first child is found, its other children can be identified by following the *following-sibling* pointers. The naive approach to find $D(v)$'s first child is through *Find_Position*$(D(v))$ followed by a sequential scan. Unfortunately, it has the same worst-case I/O cost as the *Search_Descendant*$(D(v))$ operation. In case that there are a lot of $D(v)$'s descendent before its first child, this approach's efficiency may suffer. In the scenario of Fig. 3, it needs to scan B_d pages before finding the first child. Instead, we present a procedure that takes $\mathbf{O}(\log_B k)$ I/Os in the worst case to identify $D(v)$ first child. Our approach takes advantage of pairs of integers stored over page references on the XL+-tree. First, we have the following lemma.

Lemma 1. *$D(v)$'s first child $D(u)$, if it exists, is the first (leftmost) entry satisfying $LP > LP(v)$ and $lv = (lv(v) + 1)$ on the XL+-tree; and all entries with $LP > LP(v)$ but before $D(u)$ have level of $lv > lv(v) + 1$.*

Proof. Since $D(v)$'s first child $D(u)$ satisfies $LP(v) < LP(u) < RP(v)$, all entries before $D(u)$ but with $LP > LP(v)$ also satisfy $LP(v) < LP < RP(v)$. Thus they are all $D(v)$'s descendants. □

The procedure of identifying $D(v)$'s first child involves two phases: (1) a top-down search; (2) if necessary, backtracking and another top-down search. The first top-down search begins with the root page of XL+-tree and recursively advances to the next target page. It goes through two steps on each internal page. Firstly, it chooses the *leftmost* page reference, PR_i, whose corresponding subtree has a range of keys (LPs) $(LP_{\min}, LP_{\max}]$ satisfying $LP(v) < LP_{\max}$. Note that the values of $LP_{\min}$ and $LP_{\max}$ are two delimiting integers of each page reference; thus this search process can be accomplished through a binary search over delimiting keys as

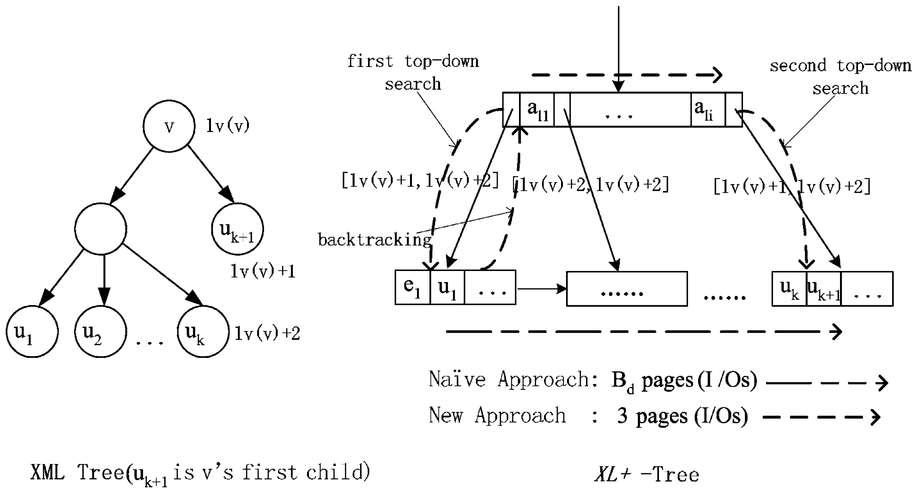

Fig. 3. A working instance of searching $D(v)$'s first child.

on the traditional B+-tree. Secondly, if the chosen page reference's range of levels $[lv_{\min}, lv_{\max}]$ contains $lv(v)+1$, which means that $D(v)$'s first child is *probably* in this subtree, it advances to the next-level page following this page reference; otherwise, it is the end of the first top-down search. Note that in the case that $lv_{\max} < lv(v)+1$, if it is known that some entry in this subtree has the left position(LP) equal to $LP_{\max}$, we can conclude that $D(v)$ has no child in the XL+-tree. This condition is satisfied if all delimiting keys stored on internal pages are keys of entries stored on leaf pages. In the following description, as in the traditional B+-tree context, we assume such guarantee. If a leaf page is reached, entries stored on it should have a LP range $(LP_{\min}, LP_{\max}]$ satisfying $LP_{\min} \leqslant LP(v) < LP_{\max}$, and a level range $[lv_{\min}, lv_{\max}]$ containing $(lv(v)+1)$. However, it does not guarantee that the first (leftmost) entry with $LP > LP(v)$ and $lv \leqslant lv(v)+1$, which according to Lemma 1, either is $D(v)$'s first child or indicates that there is no $D(v)$'s child, is on this leaf page. Therefore, the procedure continues to identify the leftmost entry, $D(w)$, with $LP > LP(v)$ and sequentially scan entries after $D(w)$ on this leaf page. If an entry with $lv \leqslant lv(v)+1$ is found, either it is $D(v)$'s first child or we can conclude that $D(v)$ has no child in the XL+-tree. Otherwise, if all entries after $D(w)$ (including $D(w)$) have level of $lv > lv(v)+1$ and the last entry of this leaf page has the left position of $LP < RP(v)$, we invoke the second phase of the procedure.

The second phase involves probably backtracking on the top-down search of the first phase and then another top-down search. If the first phase ends at a leaf page, the second phase firstly backtracks to the leaf page's parent page. Then it continues to sequentially consider the page references after the current one. There are totally six possible cases:

(1) The page reference's maximal level, $lv_{\max} < (lv(v)+1)$. In this case, it can be concluded that $D(v)$ has no child entry since according to Lemma 1, all entries before $D(v)$'s first child but with $LP > LP(v)$ should have level larger than $(lv(v)+1)$.
(2) The page reference's minimal level, $lv_{\min} > lv(v)+1$ but $LP_{\max} \geqslant RP(v)$. In this case, it can be concluded that no $D(v)$'s child entry exist in the XL+-tree.
(3) $lv_{\min} > lv(v)+1$ and $LP_{\max} < RP(v)$. In this case, it can be concluded that no $D(v)$'s child is in this subtree. It continues to consider the next page reference.
(4) $lv_{\min} \leqslant (lv(v)+1) \leqslant lv_{\max}$ but $LP_{\min} \geqslant RP(v)$. In this case, again it can be concluded that $D(v)$ has no child entry since any $D(v)$'s child entry should satisfy $LP < RP(v)$.
(5) $lv_{\min} \leqslant (lv(v)+1) \leqslant lv_{\max}$ and $LP_{\min} < RP(v)$. In this case, $D(v)$'s first child is *probably* in this subtree; it indicates the end of backtracking and the beginning of the second top-down search.
(6) The end of this page is reached. It backtracks to the current page's parent page. If the current page is the root page of XL+-tree, it can be concluded that no $D(v)$'s child exists.

In the second phase, another top-down search is required only when Case 5 occurs. We also have the observation that in the second top-down search, all entries in the subtree satisfy $LP > LP(v)$. Therefore, the procedure always sequentially scans page references or entries on the current page beginning with the leftmost one. The operations upon page references on the internal page are similar to the six cases just described. Cases 1, 2 and 4 indicate that $D(v)$'s first child does not exist. If case 5 is encountered, it advances to the next-level page following the current page reference. The search operation on the leaf page is also the same as in the first phase except that it again scans from the leftmost entry and there should be some entry with level of $lv \leqslant lv(v) + 1$.

It is now obvious that in the worst case, the number of page accesses the above procedure invokes is $\mathbf{O}(\log_B k)$ (for the first top-down search) $+ \mathbf{O}(\log_B k)$ (for the backtracking) $+ \mathbf{O}(\log_B k)$ (for the second top-down search) $= \mathbf{O}(\log_B k)$. By simply following the *following-sibling* pointers, we achieve the claimed $\mathbf{O}(\log_B k + rd)$ worst case I/O cost for the *Search_Child*($D(v)$) operation, in which rd is the number of pages where $D(v)$'s children are stored. Note that this result *asymptotically* improves the result of the naive solution that takes $\mathbf{O}(log_B k + \frac{ds}{B})$ I/Os in the worst case, in which ds is the number of $D(v)$'s descendants, since $\lceil \frac{ds}{B} \rceil \geqslant rd$. A working example is also provided in Fig. 3. Note that instead of scanning B_d pages to find $D(v)$'s first child as required by the naive approach, our proposed procedure takes only one backtracking step and one additional top-down search, totally two additional page accesses, to achieve the purpose.

3.1.2. Horizontal search: preceding and following sibling

To search $D(v)$'s preceding (or following) siblings, we have the observation that once $D(v)$'s *first* preceding (or following) sibling is identified, its other preceding (or following) siblings can be tracked through entries' *preceding-sibling* (or *following-sibling*) pointers. We have the following Lemma, whose proof is omitted because of its similarity to Lemma 1.

Lemma 2. *$D(v)$'s first following sibling $D(u)$, if it exists, is the leftmost entry satisfying $LP > RP(v)$ and $lv = lv(v)$ on the XL+-tree; and all entries with $LP > RP(v)$ but before $D(u)$ have level of $lv > lv(v)$. Similarly, $D(v)$'s first preceding sibling $D(w)$, if it exists, is the rightmost entry satisfying $LP > LP(v)$ and $lv = lv(v)$ on the XL+-tree; and all entries with $LP > LP(v)$ but after $D(w)$ have level of $lv > lv(v)$.*

The strategy of searching $D(v)$'s first preceding or following sibling on the *XL+-tree* is similar to that of searching $D(v)$'s first child. It also involves two phases: the first phase of a top-down search, and if necessary, the second phase of backtracking and another top-down search.

Consider the procedure for identifying $D(v)$'s first following sibling. The first top-down search finds the *leftmost* page reference with $LP_{\max} > RP(v)$, PR_i, on each internal page. If PR_i's level range contains $lv(v)$, the search advances to the page of next level. Otherwise, it invokes the second phase and sequentially scans other page references after PR_i. There are totally four possible cases:

(1) $lv_{\max} < lv(v)$. It can be concluded that $D(v)$ has no following sibling.
(2) $lv_{\min} > lv(v)$. $D(v)$'s first following sibling cannot be in this subtree, continue to the next page reference.
(3) $lv_{\min} \leqslant lv(v) \leqslant lv_{\max}$. $D(v)$'s first following sibling is probably in this subtree; this case indicates the end of backtracking.
(4) The end of page is reached. It backtracks to the current page's parent page; if the current page is the root page of *XL*+-tree, no $D(v)$'s following sibling exists.

As in the procedure of identifying the first child, if a leaf page is reached in the first top-down search, it determines whether the leftmost entry with $LP > RP(v)$ and $lv \leqslant lv(v)$ is in this page. If yes, either $D(v)$'s first following sibling is found or it is concluded that no $D(v)$'s following sibling exists. Otherwise, it backtracks to the leaf page's parent page. The operations upon page references on internal pages are basically the same as described above. The second top-down search is also the same as the first one except that it always begins with the leftmost page reference (or entry) on pages and case 4 should never occur.

The procedure of identifying $D(v)$'s first preceding sibling should be straightforward since it is actually symmetric to the procedure of identifying $D(v)$'s first following sibling. The first top-down search find the *rightmost* page reference with $LP_{\min} < LP(v)$, PR_i, on each internal page. If PR_i's level range contains $lv(v)$, the search advances to the page of next level. Otherwise, it invokes the second phase and sequentially scans other page references *before* PR_i in the *backward* manner. Four possible cases are as follows:

(1) $lv_{\max} < lv(v)$. It can be concluded that $D(v)$ has no preceding sibling.
(2) $lv_{\min} > lv(v)$. $D(v)$'s first preceding sibling cannot be in this subtree, continue to previous page reference.
(3) $lv_{\min} \leqslant lv(v) \leqslant lv_{\max}$. $D(v)$'s first preceding sibling is probably in this subtree; this case indicates the end of backtracking.
(4) The end of page is reached. It backtracks to the current page's parent page; if the current page is the root page of *XL*+-tree, no $D(v)$'s preceding sibling exists.

The backtracking and second top-down search can also be accomplished in the similar way. We do not describe further details since they are obvious.

From the above description, procedures for identifying $D(v)$'s first preceding or following sibling both take $\mathbf{O}(\log_B k)$ I/Os in the worst case. Therefore, both *Search_Preceding-Sibling*($D(v)$) and *Search_Following-Sibling*($D(v)$) operations can be accomplished consuming $\mathbf{O}(\log_B k + rd)$ I/O cost in the worst case, in which rd is the number of pages storing $D(v)$'s preceding or following siblings. A working instance of identifying $D(v)$ first following sibling is presented in Fig. 4. It takes $\mathbf{O}(\log_B k)$ I/Os. Note that the naive approach, which searches the leftmost entry with $LP > RP(v)$ and then conducts a sequential scan, takes $\mathbf{O}(\log_B k + B_d)$ I/Os.

3.1.3. Bottom-up search: ancestor and parent

Concerning the *Search_Parent*($D(v)$) and *Search_Ancestors*($D(v)$) operations, we have the following lemma.

Lemma 3. *$D(v)$'s ancestor at level $lv_a \leqslant (lv(v) - 1)$, if it exists, is the rightmost entry at level lv_a and with $LP < LP(v)$ on the XL+-tree; and all entries after it but before $D(v)$ have level of $lv > lv_a$.*

Obviously, the *Search_Parent*($D(v)$) operation amounts to the key (equal to $PLP(v)$) search operation on the *XL*+-tree.

To facilitate the *Search_Ancestor*($D(v)$) operation, we record all distinct levels of entries indexed by an *XL*+-tree. As claimed in Section 3.2, XML trees' maximal depth can be expected to be small; thus number of distinct levels on an *XL*+-tree is also small. The overall idea of conducting the *Search_Ancestor*($D(v)$) operation is similar to that of other search operations. Intuitively, it repeatedly searches, in the decreasing order of lv_a, the *rightmost* entry of level $lv_a (lv_a < lv(v))$ before $D(v)$. It involves multiple repetitions of the top-down search followed by the backtracking.

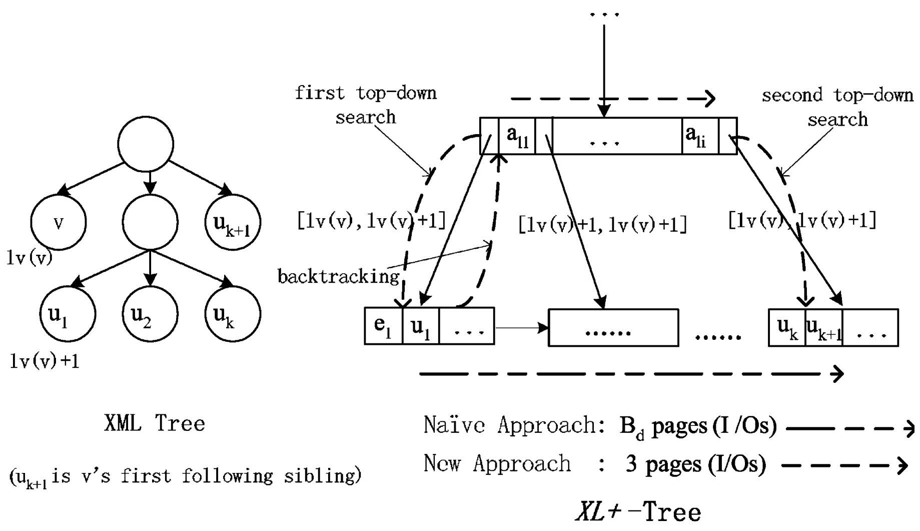

Fig. 4. A working instance of searching $D(v)$'s first following sibling.

Its first top-down search recursively identifies the *rightmost* page reference satisfying $LP_{\min} < LP(v)$ on internal pages. If it reaches a leaf page, all entries with $LP < LP(v)$ are sequentially scanned in the *backward* manner. If the minimal level of these entries is $lv_m < lv(v)$, according to Lemma 3, we have the conclusion that $D(v)$'s ancestors of level $lv_m \leqslant lv < lv(v)$ should be among them if they exist. The procedure then continues to identify $D(v)$'s ancestors of level $lv < lv_m$. It backtracks to the current leaf page's parent page and scans page references sequentially in the *backward* manner before the current page reference. In the case that the first top-down search ends at some internal page because $lv_{\min} \geqslant lv(v)$, it simply continues to consider previous page reference sequentially. If the encountered page reference's $lv_{\min}$ is less than lv_m, it stops the backtracking and begins the second top-down search. The second top-down search similarly identifies the rightmost page reference with $lv_{\min} < lv_m$ on internal pages. Note that beginning with the second top-down search, all entries in the corresponding subtree have $LP < LP(v)$. Therefore, the top-down search should reach a leaf page and the minimal level(lv) of entries on this leaf page should be $lv_{\min} < lv_m$. According to Lemma 3, we have the conclusion that all $D(v)$'s ancestors of level $lv \in [lv_{\min}, lv_m)$ should be on this leaf page. Therefore, the procedure scans all entries on this leaf page in the *backward* manner. If it encounters an entry with level of $lv' < lv_m$, we have the conclusion that either this entry is $D(v)$'s ancestor of level lv' or $D(v)$ has no ancestor of level lv'. If an entry of level $lv_{\min}$ is encountered, the scanning process on this leaf page stops. The value of lv_m is now reset to be $lv_{\min}$ and another round of backtracking and top-down search begins. The procedure continues this process until the value of lv_m reaches the smallest level of entries indexed by the *XL*+-tree. Since each round of backtracking and top-down search reduces the value of lv_m by at least one, the maximal number of rounds required by the operation is $(lv(v) - 1)$. Therefore, the *Search_Ancestor*($D(v)$) operation takes $\mathbf{O}(lv(v) \times \log_B k)$ I/O cost in the worst case. A working instance of the *Search_Ancestor*($D(v)$) operation is also provided in Fig. 5.

3.2. Update operations on XL+-tree

When the entry of a new element is inserted or deleted from the *XL*+-tree, the pointers of related entries and the level ranges stored over page references need be maintained effectively. It turns out that both deletion and insertion operations upon *XL*+-tree take the amortized I/O cost of $\mathbf{O}(\log_B k)$.

As described in [19], we use *slot*s to store entries's positions on leaf pages. The advantage of implementing *slot*s is that when a new entry is inserted into a leaf page and positions of all entries after it in this page are shifted forward, we only need to update position values of shifted entries stored in slots; since the *preceding-sibling* or *following-sibling* pointers actually refer to slots, they do not need to be updated upon such shifting.

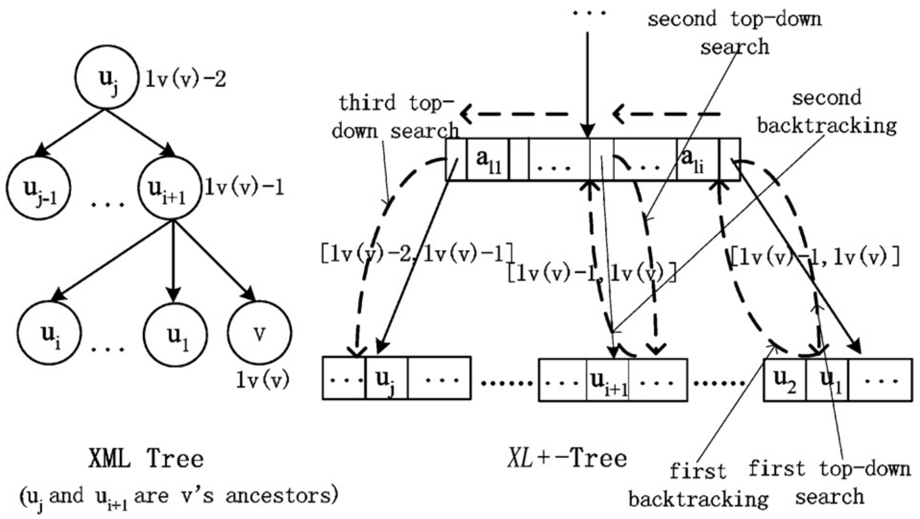

Fig. 5. A working instance of searching $D(v)$'s ancestors.

A new entry $D(w)$ can be inserted at the right position on XL+-tree just as on a typical B+-tree. $D(w)$'s immediate preceding and following siblings in the XL+-tree can also be identified using presented search operations with the worst-case I/O cost of $\mathbf{O}(\log_B k)$. To maintain the level ranges over page references, we check the level range over the page reference pointing to the leaf page where $D(w)$ is inserted. If the level range $[lv_{\min}, lv_{\max}]$ contains $D(w)$'s level $lv(w)$, it remains unchanged and no other level range on the XL+-tree needs to be updated; otherwise, either $lv_{\min}$ or $lv_{\max}$ should be updated to accommodate $lv(w)$ and such update should be recursively propagated to the current page' parent page. Note that only level ranges of page references on the path from the root page to the target leaf page can be affected by the insertion operation. Therefore, in the worst case, the required I/O cost to maintain level ranges is $\mathbf{O}(\log_B k)$. Finally, if an insertion operation results in the overcapacity of a leaf page, this leaf page needs to be split into two. For each moved entry, pointers to it should be updated properly. Since each entry is only referred by its immediate preceding sibling or following sibling entry, only constant I/Os are required to update pointers referring to each shifted entry. Additionally, only page references on the paths from the root page to two new sub-page can be affected by such splitting. Therefore, the amortized I/O cost of the insertion operation is $\mathbf{O}(\log_B k)$.

Next we turn to the deletion operation. After an entry $D(w)$ is deleted from XL+-tree, its immediate preceding sibling entry's *following-sibling* pointer should be redirected to its immediate following sibling; and its immediate following sibling entry's *preceding-sibling* pointer should be redirected to its immediate preceding sibling. Maintaining the level ranges over page references is similar to what was described in the insertion operation. If the deleted entry $D(w)$'s level satisfies $lv_{\min} < lv(w) < lv_{\max}$, in which $lv_{\min}$ and $lv_{\max}$ are minimal and maximal levels recorded over the page reference pointing to the leaf page where $D(w)$ is stored before deletion, the level range $[lv_{\min}, lv_{\max}]$ over this page reference does not need to be changed; thus no other level ranges in XL+-tree needs to be updated. Otherwise, $lv(w)$ is equal to $lv_{\min}$ or $lv_{\max}$. In this case, we need to sequentially scan all remaining entries on this leaf page to determine if there is any one with level of $lv(w)$. If there is an entry of level $lv(w)$, the level range over this page reference again does not need to be updated and it also indicates the end of the process to maintain level ranges. Otherwise, the target level range should be updated correspondingly and such update is propagated up one level. On each internal page, the level range over the page reference pointing to it is set to be $[lv_i, lv_a]$, where lv_i is the minimal of all $lv_{\min}$s over page references initiating from this page and lv_a is the maximal of all $lv_{\max}$s. This process is continued until the level range of the target page reference remains unchanged or the root page of XL+-tree is reached. It is not hard to see that the I/O cost of this procedure to maintain level ranges on XL+-tree is $\mathbf{O}(\log_B k)$. If a deletion operation results in the undercapacity of a leaf page, it should be merged with another leaf page or some entries from another leaf page should be moved onto this leaf page. For each moved entry, it takes only constant I/Os to maintain pointers. Note that only page references over paths from the root page to the affected pages (at most two) need to be updated in the worst case. Therefore, the amortized I/O cost of the deletion operation on XL+-tree is $\mathbf{O}(\log_B k)$.

We conclude this subsection with the following theorem, whose proof is straightforward from our above analysis.

Theorem 1. *The amortized I/O cost of the insertion and deletion operation on the XL+-tree are both* $\mathbf{O}(\log_B k)$.

4. The *XL*+-tree based on prefix encoding scheme

The XL+-tree based on the prefix encoding scheme has exactly the same structure as the one based on the range encoding scheme. The only difference is that entries on XL+-tree are represented by descriptors of format, (S, lv, esl). Keys on the XL+-tree are label strings (S) instead of left positions (LP) and entries on leaf pages are sorted in the increasing lexicographic order of label strings. All the analytical results of I/O cost concerning the search and update operations under the range encoding scheme also apply under the prefix encoding scheme.

In this section, we focus on the potential improvements of search operations on the XL+-tree as a result of the richer information provided by the prefix encoding scheme. Even though these improvements are not so significant as to result in *asymptotically* improved worst-case I/O cost, they may reduce the I/O and CPU cost of search operations on XL+-tree.

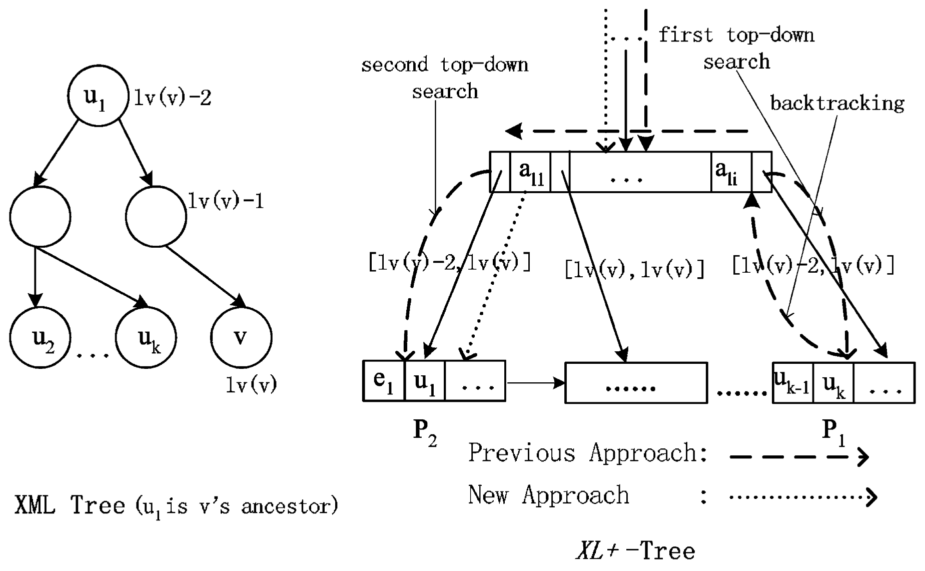

Fig. 6. The new approach of searching $D(v)$'s ancestor under the prefix encoding scheme.

Under the prefix encoding scheme, the *Search_Ancestor*$(D(v))$ operation can be accomplished by conducting multiple key searches since label strings of $D(v)$'s ancestors can be extracted from $D(v)$. Since the *XL*+-tree also stores the level range over each page reference, an additional requirement is enforced while advancing from one page to the next-level page: $[lv_{\min}, lv_{\max}]$ should contain the level(lv) of target entry; otherwise, it can be concluded that no such entry exists in the *XL*+-tree. The potential improvement of the new approach can be illustrated by the example in Fig. 6. The previous approach requires to read the leaf page P_1 into main memory and then scan the entries before $D(v)$ on this page; next, it backtracks to P_1's parent page and reads the second leaf page P_2 into main memory; finally it scans all entries on P_2 in the backward manner to identify $D(v)$'s ancestor $D(u_1)$. In contrast, the new approach only requires to search the label string of $D(u_1)$ in the *XL*+-tree. Its first advantage is that, it do not need to read P_1 into main memory, but directly reads P_2, which has the result $D(u_1)$, into main memory. Secondly, searching on the P_2 page can be accomplished through the binary search, which is more CPU efficient than the linear scanning search.

The second potential improvement is on the *Search_Following-Sibling*$(D(v))$ and *Search_Preceding-Sibling*$(D(v))$ operation. Note that under the range encoding scheme, two nodes's non-overlapping intervals $[LP, RP]$ gives no clue about their sibling relationship, which can only be determined by checking their *PLP*s. Under the prefix encoding scheme, label strings of $D(v)$'s following or preceding siblings should have the label string of $D(v)$'s parent as their prefix. We denote the label string of $D(v)$'s parent as $S(v_p)$. Any entry with the label string larger than $S(v_p) \bullet \omega$ thus cannot be $D(v)$'s following sibling. This observation can be exploited to further prune the search space. Note that the similar strategy has been used under the range encoding scheme to prune search space while searching $D(v)$'s first child. Over there, any entry with $LP > RP(v)$ cannot be $D(v)$'s first child. Therefore, while searching $D(v)$'s first following sibling, if we encounter some page reference with $S_{\min} \geqslant S(v_p) \bullet \omega$, it can be concluded that no $D(v)$'s following sibling exists in the *XL*+-tree.

5. Experimental evaluation

In this section, we experimentally evaluate the performance of the *XL*+-tree on both the benchmark and synthetic XML data. The two datasets we use are:

(1) *Xmark Benchmark Data*. It comes from the XML benchmark project [8], which simulates activities of an auction site. An XML documents of 200 MB are generated through the provided data generator.
(2) *Synthetic XML Data*. We use the IBM XML data generator to generate this synthetic data of size 20 MB according to the DTD definition in Fig. 7. Note that same-label nodes represent the same element

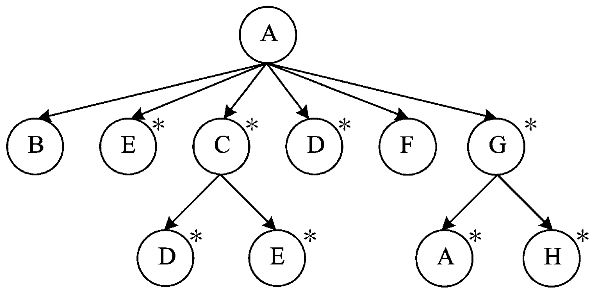

Fig. 7. The DTD definition of synthetic data. *Note*: B, D, E, F, H are of PCDATA typre.

definition. The asterisk (*) at the right-top of label nodes specifies the zero-or-more numerical relationship. This DTD is deliberately designed such that the resulting XML data has the following properties: (1) the first D-labeled child of an A-labeled data node a_i may not be right after the position of a_i on the XL+-tree T_D indexing D-labeled data nodes; (2) the first D-labeled following-sibling of an E-labeled data node e_i may not be right after the position of e_i in T_D; (3) the A-labeled ancestors of a H-labeled data node h_i may not be right before the position of h_i on the XL+-tree T_A.

We compare the performance of the XL+-tree with that of the R-tree approach used in [18]. In [18], data nodes on an XML tree are represented as multi-dimensional data points based on their *pre* and *post* positions. Specifically, we represent each data node v by ($pre(v)$, $pos(v)$, $par(v)$), in which $par(v)$ is v's parent node's *pre* position. Note that, since we do not differentiate attribute nodes from element nodes and the XL+-tree or R-tree indexes same-label nodes, we do not include the additional two dimensions used in [18], $att(v)$ and $tag(v)$, in our representation. We implement both structures on the TPIE platform (written in C++) [24], which is a software environment for external-memory algorithms. To fully explore the potential of R-tree approach, we also run the queries in the *batch* mode on R-trees. Instead of searching next location nodes from the current context nodes one by one, we bound a group of data points in a multi-dimensional box, which is then run on the R-tree. As a result, in the batch running mode, the query process involves one additional step: validating returned entries from R-tree. Depending on the type of locating axis, we also optimize the validation algorithm accordingly. We first sort the data points of current context nodes by some appropriate dimension in the increasing order and then validate the returned entries one by one. Suppose that we want to validate the returned entry u,

(1) *child*: data points are sorted by *pre*; the validation is accomplished through the binary search of $par(u)$.
(2) *parent*: data points are sorted by *par*; the validation is accomplished through the binary search of $pre(u)$.
(3) *preceding-sibling*: data points are sorted by *par*; the validation is accomplished through the binary search of $par(u)$.
(4) *following-sibling*: data points are sorted by *par*; the validation is accomplished through the binary search of $par(u)$.
(5) *descendant*: data points are sorted by *pre*; the validation linearly scans data points until $pre > pre(u)$.
(6) *ancestor*: data points are sorted by *pre*; the validation identifies the first data point with $pre > pre(u)$ and then linearly scans the list until $pos > pre(v)$.

Our machine features the OS of Linux 2.4 and a Pentium 2.2 GHz processor. Three query loads, which correspond to the top-down, bottom-up and horizontal navigations respectively, are tested on each dataset. The query loads for Xmark data are randomly generated from its DTD definition and each consists of 10 binary patterns. The query loads for the synthetic data are presented in Table 3.

5.1. XL+-tree vs R-tree

Since it is observed in our experiments that additional cache above 1 MB has little effect on the overall performance of the *XL+-tree* and R-tree on both datasets, all presented results of I/Os and running time

Table 3
Queryloads on synthetic data

Top-down patterns	//A/*child::*D, //A/*descendant::*D
Bottom-up patterns	//D/*parent::*A, //H/*ancestor::*A
Horizontal patterns	//B/*following-sibling::*D, //F/*preceding-sibling::*E

are virtually independent of the size of available cache. Their comparative I/O performance and running time on both datasets are presented in Figs. 8–11. In them, *R*-tree (*k*) represents the batch query mode on the *R*-tree with the size of batch set to be the total capacity of *k* pages. Since our experiments show that on the *R*-tree, the

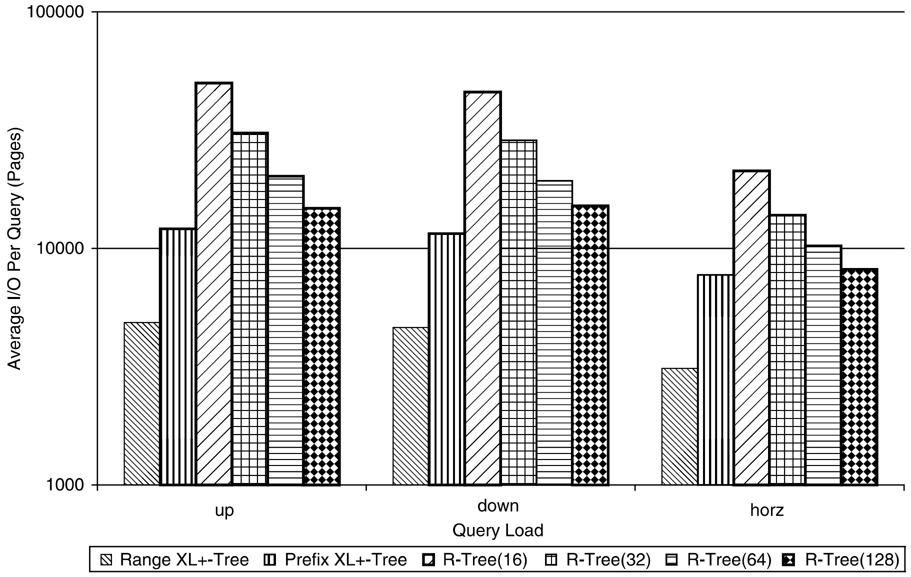

Fig. 8. I/O performance on Xmark data.

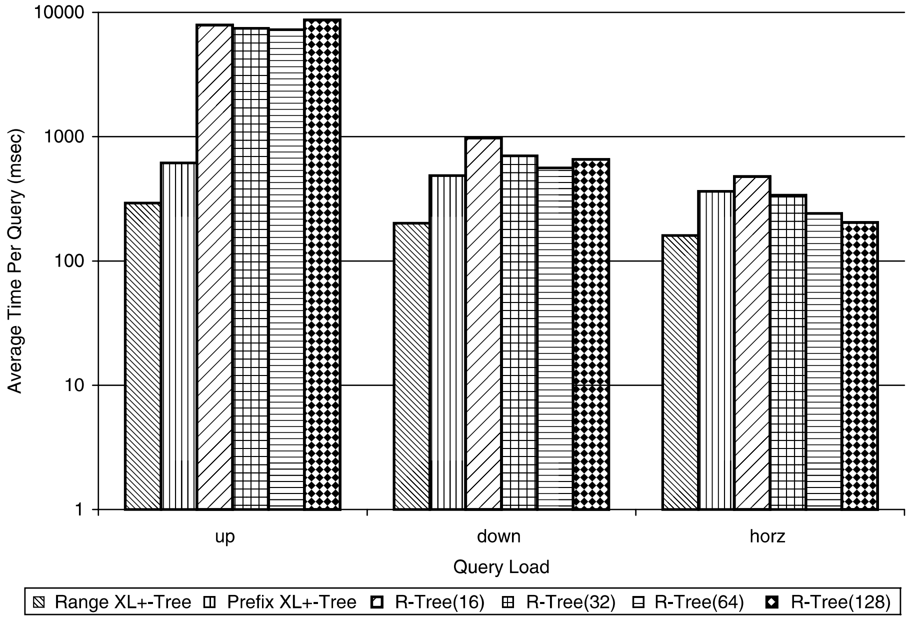

Fig. 9. Combined I/O and CPU performance on Xmark data.

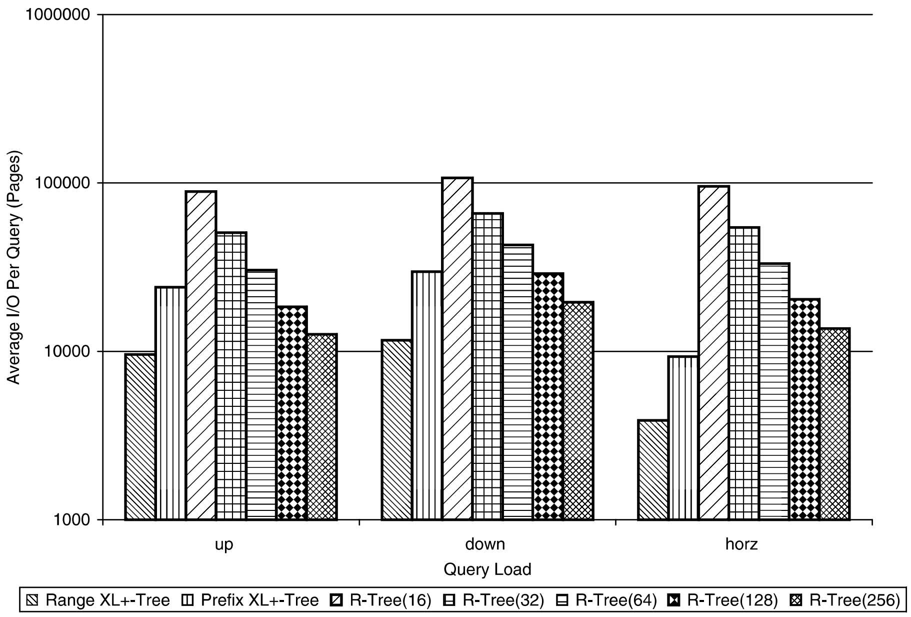

Fig. 10. I/O performance on synthetic data.

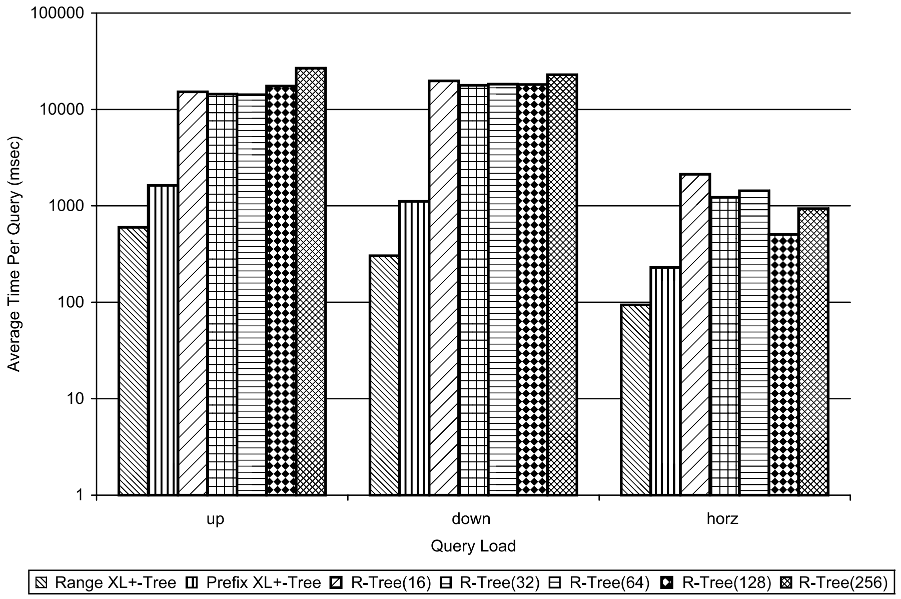

Fig. 11. Combined I/O and CPU performance on synthetic data.

batch approach performs significantly better than the one-node-at-a-time approach, only results of the batch *R*-tree approach are presented. Note that the vertical axes of all figures follow a logarithmic scale, since there are marked differences in performance.

Our experiments show that compared with the range-based *XL*+-tree, the prefix-based *XL*+-tree has a worse performance in term of either the I/O cost or the running time on all settings. Note that since the prefix-based *XL*+-tree actually shares the same structure as the range-based one, the observed performance differences result from the underlying encoding schemes instead of the index structure design. Between the range-based *XL*+-tree and the *R*-tree, it is clear that on either dataset, the *XL*+-tree performs considerably better than the *R*-tree approach in term of both I/O and running time. On both datasets, increasing the batch size of the *R*-tree approach consistently results in the reduced I/O cost; however, the overall running time may increase as the validation consumes more CPU time. On the Xmark data, the running time of the *up* and *down* queryloads increase as the batch size is increased from 64-pages to 128-pages. On the synthetic data, the running time of the *up* and *down* queryloads also increase as the batch size is increased from 64-pages to 128-pages, the running time of the *horizontal* queryload increase as the batch size reaches 256-pages. Finally, we have the observation that the prefix *XL*+-tree also performs better than the *R*-tree approach in term of both the I/O cost and running time in most cases.

6. More related work

The range labeling scheme was first used to index XML tree in [1]. We note that the later proposed durable numbering scheme [2,10,11], which is more friendly to update operations, is also range-based. Given two range labeled element sets, it has been shown that the containment structural join can be performed in the linear I/O and CPU cost [3]. Later on, with the help of the advanced *B*+-tree [7], its performance was improved to be sublinear because irrelevant descendants or ancestors can be skipped. Similar approaches have also been successfully applied in the more complicated twig pattern XML queries [6,9]. Unfortunately, only the containment relationship was considered in these works.

We note that there are much less work on the prefix-based labeling scheme. The solution proposed in [16] encodes paths as strings and inserts them into a special index called Index Fabric. It is worth pointing out that the Index Fabric is actually a compact path summary of the XML tree. Its indexing technique is not the type of the prefix-based encoding we discussed in this paper.

Our idea of storing level ranges over page references on the *XL*+-tree was inspired by [17], where authors proposed a succinct XML physical storage for efficiently matching *next-of-kin* (NoK) patterns with only *parent/child* and *preceding-/following-sibling* relationships. Their technique represents an XML tree as a string on the external memory, and stores the minimal and maximal levels of nodes on each page.

Theoretical aspects of labeling the tree-structured data in the static or dynamic settings were studied in [4,20–22]. Specifically, they considered how to encode nodes in the tree using the shortest labels such that we can decide the ancestor–descendant relationship between two nodes from their labels only.

The existing index structures to manipulate the external-memory strings, such as inverted files [23], *B*-tree [12,14] and its variants (prefix *B*-trees [13] and string *B*-tree [15]), mainly target the prefix search and substring search problems. Especially, [15] assumed that strings are arbitrarily long and addressed the string search problems on *B*-tree where strings are represented by their logical pointers in external memory; their proposed technique can also be applied on the *XL*+-tree based on the prefix encoding scheme. Note that these data structures did not consider new string search problems specific to Xpath query processing.

7. Conclusion

In this paper, we enhance the traditional range-based and prefix-based encoding schemes for XML documents and based on them, propose an external-memory index structure, the *XL*+-tree, which efficiently implements the comprehensive location steps specified in the Xpath query language. We analyze the I/O performance of both the search and update operations on *XL*+-tree. Finally, our experimental evaluation results on the benchmark and synthetic data validate the effectiveness of the *XL*+-tree proposal.

References

[1] C. Zhang, J. Naughton, D. Dewitt, Q. Luo, G. Lohman, On supporting containment queries in relational database management systems, in: Proceedings of the 20th ACM International Conference on Management of Data (SIGMOD), 2001, pp. 425–436.

[2] Q. Li, B. Moon, Indexing and querying XML data for regular path expressions, in: Proceedings of 27th International Conference on Very Large Data Bases (VLDB), 2001, pp. 361–370.

[3] D. Srivastava, S. Al-Khalifa, H.V. Jagadish, N. Koudas, J.M. Patel, Y. Wu, Structural joins: a primitive for efficient XML query pattern matching, in: Proceedings of the 18th International Conference on Data Engineering (ICDE), 2002, pp. 141–152.

[4] E. Cohen, H. Kaplan, T. Milo, Labeling dynamic XML trees, in: Proceedings of the 21st ACM International Conference on Principles of Database Systems (PODS), 2002, pp. 271–281.
[5] J. Clark, S. DeRose, XML path language, in: W3C Recommendation 16 November, 1999. Available from: <http://www.w3.org/TR/xpath>.
[6] N. Bruno, N. Koudas, D. Srivastava, Holistic twig joins: optimal XML pattern matching, in: Proceedings of the 21st ACM International Conference on Management of Data (SIGMOD), 2002, pp. 310–321.
[7] S.-Y. Chien, Z. Vagena, D. Zhang, V. Tsotras, C. Zaniolo, Efficient structural joins on indexed XML documents, in: Proceedings of 28th International Conference on Very Large Data Bases (VLDB), 2002, pp. 263–274.
[8] R. Busse, M. Carey, D. Florescu, M. Kersten, A. Schmidt, I. Manolescu, F. Waas, The XML Benchmark Project. Available from: <http://monetdb.cwi.nl/xml/index.html>.
[9] H.F. Jiang, W. Wang, H.J. Lu, Holistic twig joins on indexed XML documents, in: Proceedings of 29th International Conference on Very Large Data Bases (VLDB), 2003, pp. 273–284.
[10] S-Y. Chien, V.J. Tsotras, C. Zaniolo, Efficient management of multiversion documents by object referencing, in: Proceedings of 27th International Conference on Very Large Data Bases (VLDB), 2001, pp. 291–300.
[11] S-Y. Chien, V.J. Tsotras, C. Zaniolo, D. Zhang, Efficient complex query support for multiversion XML documents, in: Proceedings of the 8th International Conference on Extending Database Technology (EDBT), 2002, pp. 161–178.
[12] R. Bayer, C. McCreight, Organization and maintenance of large ordered indexes, Acta Informatica 1 (3) (1972) 173–189.
[13] R. Bayer, K. Unterauer, Prefix *B*-trees, ACM Transactions on Database Systems 2 (1) (1977) 11–26.
[14] D. Comer, The ubiquitous *B*-tree, Computing Survey 11 (1979) 121–137.
[15] P. Ferragina, R. Grossi, The string *B*-tree: a new data structure for string search in external memory and its applications, Journal of ACM 46 (2) (1999) 236–280.
[16] B.F. Cooper, N. Sample, M.J. Franklin, G.R. Hjatlason, M. Shadmon, A fast index for semistrucured data, in: Proceedings of 27th International Conference on Very Large Data Bases (VLDB), 2001, pp. 341–350.
[17] N. Zhang, V. Kacholia, M.T. Ozsu, A succinct physical storage scheme for efficient evaluation of path queries in XML, in: Proceedings of the 20th International Conference on Data Engineering (ICDE), 2004, pp. 54–65.
[18] T. Grust, Accelerating XPath location steps, in: Proceedings of the 21st ACM International Conference on Management of Data (SIGMOD), 2002, pp. 109–120.
[19] R. Ramakrishnan, J. Gehrke, Database Management Systems, third ed., McGraw-Hill, 2002.
[20] S. Abiteboul, H. Kaplan, T. Milo, Compact labeling schemes for ancestor queries, in: The 12th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2001, pp. 547–556.
[21] S. Alstrup, T. Rauhe, Improved labeling scheme for ancestor queries, in: The 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2002, pp. 947–953.
[22] H. Kaplan, T. Milo, R. Shabo, A comparison of labeling schemes for ancestor queries, in: The 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2002, pp. 954–963.
[23] N.S. Prywes, H.J. Gray, The organization of a multilist-type associative memory, IEEE Transactions on Communication and Electronics 68 (1963) 488–492.
[24] The TPIE Project. Available from: <http://www.cs.duke.edu/tpie/>.

Qun Chen obtained his bachelor degree in Management Information Systems from Tsinghua University of Beijing, China, in 1998. After a brief period working as a software engineer, he began his postgraduate study in computer science at National University of Singapore in 1999 and received his Ph.D. degree in 2004. He is currently a research associate of Department of Industrial Engineering and Engineering Management at the Hong Kong University of Science and Technology. His research interests are in querying and indexing semi-structured databases.

Andrew Lim obtained his Ph.D. degree in 1992 from the University of Minnesota. From 1992 to 1997, he was a system developer, project leader and consultant to many large private and government organizations in Singapore. From 1997 to 2002, he was an associate professor of Computer Science at the National University of Singapore. At the present moment, he is an associate professor at the Department of Industrial Engineering and Engineering Management in Hong Kong University of Science and Technology. Andrew's interests include algorithms, software components, framework and architecture for Global Supply Chain management. He has published more than 185 papers in premier international conferences and journals. Professor Lim is also the Director of the HKUST Logistics and Supply Chain forum, and the founding director of the Logistics and Supply Chain Institute (China). He has consulted to a large number of companies in Hong Kong, Singapore and United States.

Kian Win Ong received his bachelors in Computer Engineering from the National University of Singapore in 2003. After working on various XML indexing structures in the Hong Kong University of Science and Technology, he is currently a computer science postgraduate student at University of California at San Diego. His research interests are in databases, including optimizations for semi-structured data, novel application frameworks and schema evolution.

Jiqing Tang received his Bachelor Degree in Electronic Engineering in 2003 from Zhejiang University. From September 2003, he studied in Department of Industrial Engineering and Engineering Management at the Hong Kong University of Science and Technology and received his master degree in Industrial Engineering in January 2005. He is currently working at the high-tech company NVIDIA in Hong Kong. His research interests include operations research, integer programming, artificial intelligence and databases.