Bug Fixed Version of the Paper in ICDE 2010

Optimal Tree Node Ordering for Child/Descendant Navigations

Atsuyuki Morishima [#1], Keishi Tajima [*2], Masateru Tadaishi [#3]

[#]*Graduate School of Library, Information, and Media Studies, University of Tsukuba*
1-2 Kasuga, Tsukuba, Ibaraki 305-8550 Japan
[1]mori@slis.tsukuba.ac.jp
[3]tada@slis.tsukuba.ac.jp

[*]*Graduate School of Informatics, Kyoto University*
Yoshida-Honmachi, Sakyo, Kyoto 606-8501 Japan
[2]tajima@i.kyoto-u.ac.jp

***Abstract*— There are many applications in which users interactively access huge tree data by repeating set-based navigations. In this paper, we focus on label-specific/wildcard children/descendant navigations. For efficient processing of these operations in huge data stored on a disk, we need a node ordering scheme that clusters nodes that are accessed together by these operations. In this paper, (1) we show there is no node order that is optimal for all these operations, (2) we propose two schemes, each of which is optimal only for some subset of them, and (3) we show that one of the proposed schemes can process all these operations with access to a constant-bounded number of regions on the disk without accessing irrelevant nodes.**

I. Introduction

There are many applications in which users interactively access huge tree data (e.g., file directories or huge XML data) by repeating *set-based navigations*. In set-based navigation, a user specifies one node and a type of navigation. Then the system retrieves all the nodes reachable from that node via that type of navigation. The user browses the retrieved nodes, and select one node in order to repeat a set-based navigation from it. For exploration of huge data, set-based navigations are more efficient than simple node-at-a-time navigations.

In such an interactive browse-and-traverse style of access, users rarely specify complex traverse conditions. In this paper, we focus on the following most basic tree navigations:

$$a \rightarrow X, \quad a \xrightarrow{*} X, \quad a \xrightarrow{l} X, \quad a \xrightarrow{l*} X.$$

$a \rightarrow X$ is an operation that retrieves all the children of a given node a, and $a \xrightarrow{*} X$ retrieves all the descendants of a. The other two operations traverse only edges with a given label l.

For interactive access, we need to process these operations efficiently. This is easy when data is stored on the main memory. We can construct a tree structure on the memory by using pointers. When data is huge and stored on the disk, however, we need a storage scheme that can process these operations with low I/O cost. In order to reduce I/O cost, we need to store the nodes in an appropriate order so that nodes that are accessed together by these operations are clustered.

Such an order is not trivial. Suppose we store nodes of a tree in the depth-first order on a disk, as shown in Fig. 1. Here, we assume each disk block can store three nodes. This

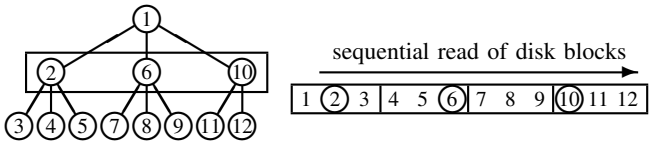


Fig. 1. Retrieval of children on the depth-first order storage

storage scheme can efficiently process $a \xrightarrow{*} X$ for any node a, because the descendants of a are always stored in consecutive positions, and we can read them out by one sequential access to a contiguous disk region without reading irrelevant nodes (except for those in the blocks at the both ends of the region). In that sense, the depth-first order is optimal for $a \xrightarrow{*} X$.

On the other hand, this storage scheme is not efficient for $a \rightarrow X$, because children of a are interleaved by their subsequent descendants. As a result, if the children have many subsequent descendants, we have to read many irrelevant nodes stored in the same disk blocks as the relevant ones. In addition, we have to read many incontiguous regions, which significantly degrade performance compared to access to a single contiguous region, because of the today's disk architecture and the prefetch by the OS. We can adhere to one sequential access, but then we have to read irrelevant blocks on the way. For example, for retrieving children of the node 1 (i.e., nodes 2, 6, 10) in Fig. 1, we have to read irrelevant nodes 1, 3, 4, 5, 11, 12. In addition, we have to access two incontiguous regions, or have to read an irrelevant block of nodes 7, 8, 9, if we use a single sequential access.

In this way, the depth-first order is I/O-inefficient for the retrieval of children. Notice that the breadth-first order has a counter problem: children are consecutive, but descendants are not. For efficient interactive exploration of huge data, we need a node ordering scheme that clusters answers to all the supported operations. When we consider general path queries, including multi-step path patterns, twig patterns, or value predicates, it is difficult to design such a scheme. However, when we concentrate only on the most fundamental operations explained above, there *exists* non-trivial ordering schemes that can cluster answers to these operations quite well.

In this paper, first, we show that there is no single node ordering scheme that is optimal for all the operations above. Then we propose two schemes, each of which is optimal only for some subset of them, and finally, but most important, we show that one of the proposed schemes can process all those operations with access to a constant-bounded number of regions on the disk, without accessing irrelevant nodes.

Notice that our ordering schemes are advantageous even in applications that support general path queries, if most queries actually issued by users are the fundamental operations above.

II. Related Work

There has been much research on indexing schemes and labeling schemes for path queries, but indexing or labeling schemes do not necessarily answer the question of how to order the nodes on the disk in order to minimize I/O cost.

There are also many processing schemes of path queries that scan nodes in the depth-first order [1], [2], [3]. These schemes avoid scanning some irrelevant nodes in order to reduce computation cost. The depth-first order is, however, not I/O-optimal for retrieving children as explained above. [4] has proposed a scheme which uses two B-tree indices so that we can scan a tree in both depth-first and breadth-first order. This approach, however, requires the cost to maintain two B-trees.

There has also been research on storage schemes for tree data, such as [5], [6], [7]. However, they focus on either multi-step path queries, or twig queries, and no research has studied storage schemes specializing in child and descendant queries starting from a single node, although they are the most frequently used operations in interactive data exploration.

Also notice that our operation set includes $a \xrightarrow{l*} X$. Although this is a very fundamental operation in set-based navigation, no existing scheme can process it efficiently.

[8] proposed a node ordering scheme that is equivalent to the scheme we use in the simplest case, i.e., when we only consider $a \rightarrow X$ and $a \xrightarrow{*} X$. Our contribution is to propose schemes that also support $a \xrightarrow{l} X$ and $a \xrightarrow{l*} X$.

There are also research on succinct data structure for trees supporting efficient navigation, and some studies, e.g., [9], [10], also support queries retrieving node sets. However, they assume data fits in the memory, and do not discuss I/O-cost.

III. Proposed Node Ordering

We first show a node order that is optimal for both child and descendant navigations. Then we show that there is no such an optimal scheme when we introduce two more operations that specify edge labels. We propose two schemes that are optimal for only some subset of them, and then show that in one proposed scheme, the number of disk regions we need to access for those operations is bounded by a small constant. In this paper, we omit the proofs of the theorems and the correctness of the algorithms for space limitation.

A. Child and Descendant Navigations: $a \rightarrow X$ and $a \xrightarrow{*} X$

Now we show an optimal node order for $a \rightarrow X$ and $a \xrightarrow{*} X$. The requirement is to cluster both the children and the descendants of every node. The boxes in Fig. 2(a) show the node sets to cluster in a tree. Notice that these boxes either include, are included by, or are disjoint with each other, i.e., they never partially overlap. Therefore, there exists a node order that clusters all these node sets. In fact, the node order $<_t$ defined on a tree t as below achieves the requirement.

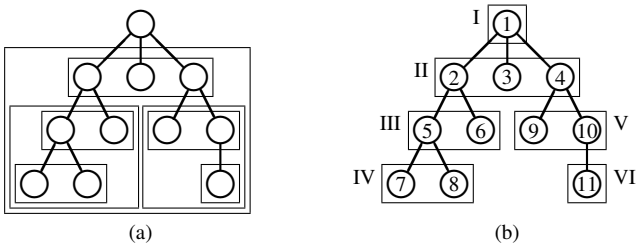


Fig. 2. Children/descendants of nodes and node order that cluster them

Optimal Ordering for $a \rightarrow X$ and $a \xrightarrow{*} X$: $<_t$

For any nodes n_1, n_2 in a tree t,

- if n_1 (or n_2) is the root node, $n_1 <_t n_2$ ($n_2 <_t n_1$, resp.).
- if n_1 and n_2 are siblings, $n_1 <_t n_2$ iff n_1 precedes n_2 in the sibling order in t,
- otherwise, $n_1 <_t n_2$ iff the parent of n_1 precedes the parent of n_2 in the depth-first order in t. □

In other words, we group siblings that share the same parent, we sort the groups in the depth-first order in t, and within each group we sort nodes in their sibling order. For example, in Fig. 2(b), the roman numbers I to VI designate the depth-first order of sibling groups, and the numbers 1 to 11 designate the order given by $<_t$. Then the following theorem holds for $<_t$:

Theorem 1: For any tree t and for any node a in it, its children and descendants (excluding a itself) have consecutive positions in the ordering defined by $<_t$. □

For example, children of the node 2 have consecutive numbers 5, 6, and its descendants have consecutive numbers 5, 6, 7, 8.

In the following, we call those numbers the *addresses* of the nodes, and write $addr(n)$ to denote the address of the node n. We store nodes on a disk in the order of $<_t$, and for each node n, we store the address of its parent, denoted by $parent(n)$, and the address of its first child, denoted by $firstChild(n)$. Fig. 3 shows how we store the tree in Fig. 2(b). Then we can process $a \rightarrow X$ and $a \xrightarrow{*} X$ by the procedures below:

Algorithm for $a \rightarrow X$:

1) scan the node entries starting at $firstChild(a)$, and
2) stop the scan at a node n s.t. $parent(n) \neq addr(a)$. □

Algorithm for $a \xrightarrow{*} X$:

1) retrieve the children of a by the procedure above, and
2) continue to read the following nodes, until we reach a node n s.t. $parent(n) < firstChild(a)$. □

For example, suppose we retrieve descendants of node 2 in Fig. 2(b). We first scan its children 5 and 6. Then we proceed to 7 and 8. During that scan, the addresses of already found descendants are in the range from 5 to the current address of the scan. When we reach node 9, its parent is 4, which is not within that range. Therefore the node 4 is not a descendant of 2, and therefore, node 9 is not a descendant, either.

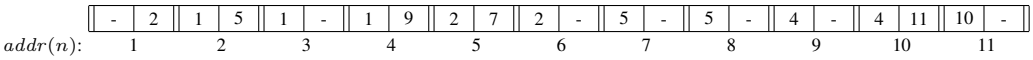
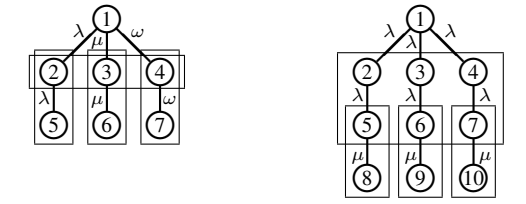
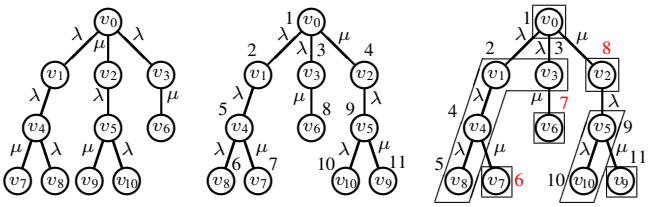


Fig. 3. Disk image of the tree data



(a) conflict with $a \rightarrow X$ (b) conflict with $a \xrightarrow{*} X$

Fig. 4. Conflicts caused by $a \xrightarrow{l*} X$



(a) the original tree (b) the order by $<_t^l$ (c) the order by $<_t^{l*}$

Fig. 5. Ordering for edge-labeled trees

The procedures above can retrieve children and descendants by scanning a single consecutive region on the disk without reading irrelevant nodes, except for the last node at which we stop the scan. In real disk access, because the unit of disk access is a disk block that includes many nodes, we do not read a irrelevant block unless the node at which we should stop happens to appear in the first entry of some block.

If necessary, we can prevent such unnecessary access by storing in each block (1) a flag showing whether the last node in the block is a last sibling, and (2) the number of ancestors of the last node in the block which have that last node as the last descendant. Here, we omit the details for space limitation.

B. Edge Labels: $a \xrightarrow{l} X$ and $a \xrightarrow{l*} X$

Now we introduce $a \xrightarrow{l} X$ and $a \xrightarrow{l*} X$. When we have $a \xrightarrow{l*} X$, and also have either $a \rightarrow X$ or $a \xrightarrow{*} X$, no order can cluster answers to both operations without interleaving nodes.

Fig. 4(a) illustrates the conflict between $a \xrightarrow{l*} X$ and $a \rightarrow X$. In this tree, $1 \rightarrow X$, $1 \xrightarrow{\lambda*} X$, $1 \xrightarrow{\mu*} X$, $1 \xrightarrow{\omega*} X$ retrieve nodes $\{2,3,4\}$, $\{2,5\}$, $\{3,6\}$, and $\{4,7\}$, respectively. The boxes in Fig. 4 represent these node sets, and obviously we cannot serialize the nodes without decomposing any of these boxes.

On the other hand, $a \xrightarrow{l*} X$ and $a \xrightarrow{*} X$ never conflict when they start from the same node, because the answer to the former is the subset of the latter. When they start from different nodes, however, they may conflict, as shown in Fig. 4(b). In this example, $1 \xrightarrow{\lambda*} X$ retrieves nodes 2 to 7, and $2 \xrightarrow{*} X$, $3 \xrightarrow{*} X$, $4 \xrightarrow{*} X$ retrieve $\{5,8\}$, $\{6,9\}$, $\{7,10\}$, respectively. The boxes in Fig. 4(b) represent these node sets, and obviously, there is no node ordering that agrees with all these boxes.

Because $a \xrightarrow{l*} X$ conflicts with $a \rightarrow X$ or $a \xrightarrow{*} X$, we need to either sacrifice $a \xrightarrow{l*} X$, or sacrifice $a \rightarrow X$ and $a \xrightarrow{*} X$. If we sacrifice $a \xrightarrow{l*} X$, a node order $<_t^l$ defined on a tree t, as below, is optimal for the other operations, $a \rightarrow X$, $a \xrightarrow{*} X$, $a \xrightarrow{l} X$:

Optimal Ordering for $a \rightarrow X$, $a \xrightarrow{*} X$, $a \xrightarrow{l} X$: $<_t^l$

Given a tree t, let t' be the tree created from t by stable sorting of siblings by the labels of their incoming edges. Then for any nodes n_1, n_2 in t, $n_1 <_t^l n_2$ iff $n_1 <_{t'} n_2$. □

For example, given a tree in Fig. 5(a), Fig. 5(b) is the sorted tree, and the numbers beside the nodes represent the node order given by $<_t^l$. Then we have the following theorem.

***Theorem** 2:* For any tree t and its node a, answer nodes of $a \rightarrow X$, $a \xrightarrow{*} X$ or $a \xrightarrow{l} X$ have consecutive positions in $<_t^l$. □

However, $<_t^l$ is very inefficient for $a \xrightarrow{l*} X$, as shown below:

***Theorem** 3:* Given a tree t and its node a, answer nodes of $a \xrightarrow{l*} X$ appear, in the worst case, in N detached positions in the ordering $<_t^l$, where N is the number of answer nodes. □

This is obvious because each relevant node may appear alone in the middle of a distinct sibling group.

We also modify the storage scheme. We store nodes in the order of $<_t^l$, and for each node, we store a pointer $firstChild(a,l)$ for each label l, which points to the first child reachable via l. Child pointers of each node are stored in the dictionary order of l. Fig. 6 shows the disk image in this scheme. In Fig 6, a pointer to a node n is represented by $addr(n)$ for simplicity, but in the real implementation, we use the byte offset in this image, because each entry in this scheme has a variable length and we cannot use $addr(n)$ as pointers.

On this data representation, $a \rightarrow X$ and $a \xrightarrow{*} X$ are processed by the same procedure as before, except that each node may have many child pointers, and we follow the first one among these. $a \xrightarrow{l} X$ can be processed by the procedure below:

Algorithm for $a \xrightarrow{l} X$:

1) scan the node entries starting at $firstChild(a,l)$, and
2) stop the scan when we reach a node n s.t. either (1) $addr(n) = firstChild(a,l')$ where l' is the label of the next child pointer in a, or (2) $parent(n) \neq addr(a)$. □

Next, we consider the second choice, i.e., choosing $a \xrightarrow{l*} X$ and sacrificing $a \rightarrow X$ and $a \xrightarrow{*} X$. First, we define a couple of concepts. We define the *maximal unilabeled connected subgraphs* of a tree t as the maximal connected subgraphs of t that include only one kind of edge label. Notice that they always form trees. Then we define *unilabeled clusters* as subgraphs created from the maximal unilabeled connected subgraphs, by removing their root nodes. (deleted) Each unilabeled cluster forms a forest. We also regard the root node of t always forms a unilabeled cluster including only itself. The unilabeled clusters of t then disjointly classify all the nodes in t.

Now we define a node order $<_t^{l*}$ on a tree t, which is optimal for $a \xrightarrow{l} X$ and $a \xrightarrow{l*} X$, as follows:

Optimal Ordering for $a \xrightarrow{l} X$, $a \xrightarrow{l*} X$: $<_t^{l*}$

Let t' be the tree created from t by sorting siblings as in the definition of $<_t^l$. For any nodes n_1, n_2 in t, let f_1, f_2 be the unilabeled clusters including n_1, n_2, let m_1, m_2 be the first nodes in f_1, f_2 in the depth-first order in t', and let l_1, l_2 be

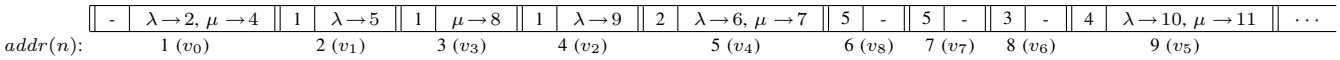


Fig. 6. Disk image in the storage scheme based on $<_t^l$

the labels of the incoming edges of n_1, n_2, respectively. Then

- if $f_1 = f_2$, $n_1 <_t^{l*} n_2$ iff $n_1 <_{t'} n_2$,
- if $f_1 \neq f_2$, $n_1 <_t^{l*} n_2$ iff m_1 precedes m_2 in the depth-first order in t'. □

Fig. 5(c) illustrates $<_t^{l*}$ defined on the tree in Fig. 5(a). The boxes in the figure represent unilabeled clusters. For example, v_1, v_3, v_4, v_8 form a unilabeled cluster with a label λ, and its first node is v_1. We sort seven unilabeled clusters in this tree in the depth-first order of their first nodes. Within each unilabeled cluster, nodes are sorted by $<_t$. Thus the numbers beside the nodes represent the node order given by $<_t^{l*}$. The theorem below shows that $<_t^{l*}$ is optimal for $a \xrightarrow{l} X$ and $a \xrightarrow{l*} X$.

Theorem 4: For any tree t and its node a, the answer nodes of $a \xrightarrow{l} X$ or $a \xrightarrow{l*} X$ have consecutive positions in $<_t^{l*}$. □

We again modify our storage scheme. We store nodes in the order of $<_t^{l*}$, and for each node n, we store its label, denoted by $label(n)$. Although we can process all the operations without $label(n)$, here we show simpler algorithms for $a \xrightarrow{l*} X$ and $a \xrightarrow{*} X$ that use it because of space limitation. First, we can process $a \xrightarrow{l} X$ and $a \xrightarrow{l*} X$ as follows:

Algorithm for $a \xrightarrow{l} X$: The same procedure as before. □

Algorithm for $a \xrightarrow{l*} X$:

1) scan the node entries starting at $firstChild(a, l)$, and
2) stop the scan at n s.t. either (1) $parent(n) \neq a \wedge parent(n) < firstChild(a, l)$, or (2) $label(n) \neq l$. □

For example, suppose we process $v_4 \xrightarrow{\mu*} X$ in Fig. 5(c). We start the scan at $firstChild(v_4, \mu)$, i.e., v_7, and proceed to v_6. When we reach v_6, $parent(v_6) = 3 < firstChild(v_4, \mu) = 6$. On the other hand, when we process $v_2 \xrightarrow{\lambda*} X$, we start the scan at v_5, and when we reach v_9, $label(v_9) = \mu \neq \lambda$.

Although $<_t^{l*}$ is not "optimal" for the other two operations, $a \rightarrow X$ and $a \xrightarrow{*} X$, $<_t^{l*}$ has the following good property:

Theorem 5: For any tree t and its node a, the answers to $a \rightarrow X$ are clustered in, at most, L regions in $<_t^{l*}$, where L is the number of distinct labels on outgoing edges of a, and the answers to $a \xrightarrow{*} X$ are clustered in, at most, 2 regions. □

We can actually evaluate $a \rightarrow X$ and $a \xrightarrow{*} X$ by accessing only L and 2 regions by the following procedures:

Algorithm for $a \rightarrow X$:

Repeat $a \xrightarrow{l} X$ for all l s.t. a has $firstChild(a, l)$. □

Algorithm for $a \xrightarrow{*} X$:

1) Let l be $label(a)$, and let $minAdd$ be MAXINT.
2) If a has child pointers for some $l' (\neq l)$, let l' be the first one among them in the dictionary order, and let $minAdd$ be $firstChild(a, l')$.
3) Scan the node entries starting at $firstChild(a, l)$.
4) When scanning n, if $firstChild(n, l') < minAdd$ for some $l' (\neq l)$, let $minAdd$ be $firstChild(n, l')$.
5) Stop the scan at a node n s.t. either (1) $parent(n) \neq a \wedge parent(n) < firstChild(a, l)$, or (2) $label(n) \neq l$. Let $maxL$ be $addr(n) - 1$.
6) Scan the node entries starting at $minAdd$.
7) Stop the scan at n s.t. $parent(n) \neq a \wedge parent(n) < firstChild(a, l)$ or $maxL < parent(n) < minAdd$. □

For example, in Fig. 5(c), we process $v_1 \xrightarrow{*} X$ by (i) retrieving its λ-descendants (v_4, v_8) that are stored in consecutive positions in one cluster, and (ii) retrieving their further descendants in other clusters by scanning nodes starting at $minAdd$, which is set to $firstChild(v_4, \mu) = v_7$ when we scan v_4.

In this way, in this storage scheme, the number of disk regions we need to access for processing $a \rightarrow X$ is the number of distinct labels of the children of a, which is usually a small constant, and that for processing $a \xrightarrow{*} X$ is at most 2.

IV. Conclusion

This paper studies how we should order nodes of labeled trees on the disk for efficient processing of children/descendant and wildcard/label-specific navigations. We showed that there does not exist an ordering scheme which is optimal for all these operations, and we proposed two node orders $<_t^l$ and $<_t^{l*}$. Although $<_t^l$ is optimal for the three operations, it is quite inefficient for $a \xrightarrow{l*} X$. On the other hand, $<_t^{l*}$ is optimal only for $a \xrightarrow{l} X$ and $a \xrightarrow{l*} X$, but it guarantees that the nodes to retrieve in $a \rightarrow X$ and $a \xrightarrow{*} X$ are clustered in a small number of disk regions. Therefore, $<_t^{l*}$ is preferable in most cases.

Acknowledgment

This research was partially supported by the Grant-in-Aid for Scientific Research (#20700076, #21013004) from MEXT, Japan, and Research Projects of Graduate School of Library, Information, and Media Studies, University of Tsukuba, Japan.

References

[1] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman, "On supporting containment queries in relational database management systems," in *SIGMOD*, 2001, pp. 425–436.
[2] S. Al-Khalifa, H. V. Jagadish, J. M. Patel, Y. Wu, N. Koudas, and D. Srivastava, "Structural joins: A primitive for efficient XML query pattern matching," in *ICDE*, 2002, p. 141.
[3] T. Grust, M. van Keulen, and J. Teubner, "Staircase join: Teach a relational DBMS to watch its (axis) steps," in *VLDB*, 2003, pp. 524–525.
[4] T. Grust, J. Rittinger, and J. Teubner, "Why off-the-shelf RDBMSs are better at XPath than you might expect," in *SIGMOD*, 2007, pp. 949–958.
[5] C.-C. Kanne and G. Moerkotte, "Efficient storage of XML data," in *ICDE*, 2000, p. 198.
[6] W. Wang, H. Jiang, H. Wang, X. Lin, H. Lu, and J. Li, "Efficient processing of XML path queries using the disk-based F&B index," in *VLDB*, 2005, pp. 145–156.
[7] N. Zhang, V. Kacholia, and M. T. Özsu, "A succinct physical storage scheme for efficient evaluation of path queries in XML," in *ICDE*, 2004, pp. 54–65.
[8] J. Banerjee, W. Kim, S.-J. Kim, and J. F. Garza, "Clustering a DAG for CAD databases," *IEEE Trans. Software Eng.*, vol. 14, no. 11, pp. 1684–1699, 1988.
[9] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan, "Compressing and searching XML data via two zips," in *WWW*, 2006, pp. 751–760.
[10] R. K. Wong, F. Lam, and W. M. Shui, "Querying and maintaining a compact XML storage," in *WWW*, 2007, pp. 1073–1082.