Faculty of Library, Information and Media Science, University of Tsukuba
Technical Report

I/O-Optimal Node Ordering Schemes for Set-Based Navigations in Trees

Keishi Tajima*, Atsuyuki Morishima**, Masateru Tadaishi***

*Graduate School of Informatics, Kyoto University
**Faculty of Library, Information and Media Science, University of Tsukuba
***Graduate School of Library, Information and Media Studies, University of Tsukuba

SLIS-TR-2014-002

***Abstract*— There are many applications in which users interactively access large tree data by repeating set-based navigations, i.e., by repeatedly selecting one (or several) node, retrieving a set of nodes connected to it, and again selecting one (or several) node among them. In this paper, we focus on the eight most fundamental operations in set-based navigation, which include neighbor/reachable, label-specific/wildcard, and forward/backward navigations. For efficient processing of these operations for large data stored on a disk, we need a storage scheme that clusters nodes that are accessed together by those operations. In this paper, (1) we show no storage scheme can be I/O-optimal for all these operations, (2) we show several node ordering schemes, each of which is I/O-optimal for some subset of them, (3) we show that one of the schemes can process all the forward operations with sequential access to a constant-bounded number of regions on the disk without accessing irrelevant nodes, and (4) backward operations can be efficiently processed on that scheme by using a standard cache technique. We also show that our storage scheme is compatible with several known techniques, such as, those for updates, that are important in practical applications. Finally, we give experimental results with synthesized and real data that confirm our theoretical results.**

I. Introduction

Edge-labeled trees are commonly used to organize large volume of data, both in many traditional applications, such as file directories, and in many recent applications, such as parsed corpus (e.g., tree banks), bioinformatics data (e.g., PDML data, Gene Ontology, and many other data with deeply nested structure [31], [6]), and XML serializations of various data. Many graph data in real applications are also sparse, tree-like graphs [37].

This paper focuses on the problem of I/O-efficient storage schemes for *navigation* on such data. Queries and navigation are the two fundamental ways for exploring huge data. While there has been much research on I/O-efficient storage schemes for queries, there has not been much research on that for navigation. Interactive navigation on huge data is, however, important in some applications. For example, in bioinformatics databases, interactive exploration is important because text annotations by other researchers require interpretation by the experts [33]. Astronomical/medical image databases also require interpretation by the experts.

Since simple node-at-a-time navigations are not sufficient for the efficient exploration of such huge data, many applications support *set-based navigations* (see, e.g., [9]). In set-based navigation, a user specifies a starting node and a condition on the edges to traverse. Then the system retrieves all the nodes reachable from that node through edges satisfying the given condition.

In an interactive browse-and-traverse style of access, users rarely specify complex conditions, and usually use only simple ones. In this paper, we focus on a symmetric set of the following eight most fundamental navigations, consisting of neighbor/reachable, label-specific/wildcard, and forward/backward navigations:

$$a \to X,\ \ a \xrightarrow{*} X,\ \ X \to a,\ \ X \xrightarrow{*} a,$$
$$a \xrightarrow{l} X,\ \ a \xrightarrow{l*} X,\ \ X \xrightarrow{l} a,\ \ X \xrightarrow{l*} a.$$

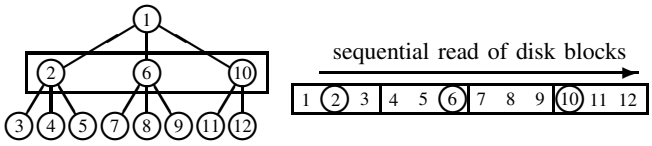


Fig. 1. Child retrieval on depth-first order storage

$a \to X$ is an operation that retrieves all nodes that are destinations of edges outgoing from a given node a. $a \xrightarrow{*} X$ retrieves all nodes reachable from a through paths of any length. $X \to a$ and $X \xrightarrow{*} a$ do the same, but with the direction of edges reversed. When the data is a tree, they retrieve the children, descendants, parents, and ancestors of a, respectively. The following four operations traverse only edges with a given label l. For example, $a \xrightarrow{l*} X$ retrieves the nodes reachable from a via traversing only l edges.

To allow users to interactively explore huge data, we need to process these operations efficiently. This is easy when the data is stored on the main memory. We can construct a tree or graph structure on the memory by using pointers. When the data is huge and stored on the disk, however, I/O cost is critical, and we need a storage scheme that clusters nodes that are accessed together by these operations.

Such a scheme is, however, not trivial. Suppose we store the nodes of a tree on a disk in depth-first order, as shown in Fig. 1. Here, we assume each disk block can store three nodes. This scheme is I/O-optimal for $a \xrightarrow{*} X$, because the descendants of some a are always perfectly clustered. 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).

On the other hand, this scheme is not efficient for $a \to X$, because children of a are interleaved by their subsequent descendants, and are not clustered. For example, for retrieving three children (nodes 2, 6, 10) of node 1 in Fig. 1, we have to read three blocks including irrelevant nodes 1, 3, 4, 5, 11, 12. In addition, we have to access two incontiguous regions, or if we use a single sequential access, we have to read an irrelevant block storing nodes 7, 8, 9 along the way.

Notice that the breadth-first order has a counter problem: children are clustered, but descendants are not. In fact, if we consider complex path queries, including multi-step paths, twig patterns, or value predicates, no storage scheme can cluster answers to all the possible queries.

Contribution. The main contribution of this paper is to answer an interesting question: Is there an ordering scheme that can cluster the answers quite well for the most fundamental set of set-based navigations? Our answer is that there *exists* a non-trivial ordering scheme to meet the property. We also show the ordering scheme is compatible with several existing techniques that are important in practical applications. It is worth noting that such a scheme is advantageous even in applications that support general path queries, if most queries actually issued by users are *shoot-and-pull* queries, i.e., queries that select a small number of nodes that satisfy the given conditions, and

then pull the nodes connected to these nodes.

Outline of the Paper. In Section III, we show that there is no node order that is I/O-optimal for all the operations explained above, and introduce three schemes, each of which is I/O-optimal only for some subset of them. We also show that one of these schemes can process all the forward operations with access to a constant-bounded number of disk regions, without accessing irrelevant nodes. Finally, we show that combination of that scheme and a standard cache technique can efficiently process backward operations. Section IV shows that our scheme is compatible with several existing techniques that are important in practical applications, e.g., that for updates. Section V gives experimental results with synthesized and real data to confirm our theoretical results.

II. Related Work

Tree partitioning [28], [34], [26], [5] or graph partitioning [36], [35], [11] is a problem to divide a tree/graph into disjoint connected subgraphs, called *clusters*, so that it minimizes total weights of inter-cluster edges while keeping total node weights in each cluster smaller than a given limit. They can be used to asign nodes of hierarchical data [28], [34], object-oriented data [36], or XML data [26], [5] to disk blocks.

In these studies, edge weights represent navigation workload, but such workload data may not be available. On the other hand, our storage schemes are always I/O-optimal for some of the basic operations above. In addition, tree/graph partitioning only consider whether nodes are in the same disk block or not. On the other hand, in our node ordering approach, related nodes are stored in consecutive blocks even when they do not fit in one block. Such consecutive blocks can be efficiently accessed by a sequential access.

[3] has proposed a clustering scheme for CAD/CAM data that is equivalent to $<_t$, the simplest scheme discussed in this paper. [32] also proposed a node numbering scheme for XML data that is similar to $<_t$ (although they store data in RDBMS, and do not discuss in what order they store data on the disks). However, as shown later, $<_t$ is I/O-optimal only for $a \rightarrow X$ and $a \stackrel{*}{\rightarrow} X$. In this paper, we show new node ordering schemes that support a wider range of operations, including label-specific navigations. In addition, we show not only the ordering schemes, but also the detailed storage schemes and efficient scan-based algorithms for navigation operations on those storage schemes. We also discuss backward navigation, navigations from multiple nodes, and updates.

[4] proposed a storage scheme for tree data on disks. Their scheme is also similar to $<_t$, but instead of linearly ordering nodes, it arrange nodes in two-dimensional disk space consisting of tracks and blocks, and it requires the modification to the disk access interface in the operating system layer.

[24], [27], [1], [23], [24] have shown storage schemes which store graph nodes on the disk in appropriate order so that we can efficiently compute transitive closures by a sequential scan. However, there is no research discussing node ordering schemes that can cluster both neighbors and transitive closures, and both label-specific and wildcard ones.

[21], [22] have proposed disk block prefetching strategies for object-oriented databases, which can be applied to other tree or graph data. In our ordering approach, we can improve disk access performance by a simple strategy that prefetches the following blocks of the requested block. Such a prefetch may be done by disk controllers or operating systems. Also notice that the cost of prefetching the following blocks is far smaller than that of prefetching blocks at somewhere else.

There have also been studies on storage schemes for XML databases. However, they focus on complex multi-step path queries either starting at the "root" node or starting at "any" node, while we focus on single-step navigations starting at a given single node. This difference makes efficient storage schemes for complex path queries and those for our set-based navigations completely different. For example, in the scheme proposed in [39], the answers to a query of the form p or $p//l$, where p is a simple (i.e., no branch) path query starting from the root node and l is a label, are always clustered in one contiguous disk region, but the answers to queries not starting from the root node are not clustered. In addition, no existing scheme can process $a \stackrel{l*}{\rightarrow} X$ efficiently.

There have also been much research on path-based storage schemes for XML data [29], [17], [12], [41], [10], [38], [7], [8]. They index, sort, or cluster nodes based on their paths from the root (or their suffixes, or the reversed path). As a result, these schemes are not necessarily I/O-optimal for our one-step navigations starting from an arbitrary given node.

A storage scheme proposed in [43] uses pre-order. Therefore, children of a node are interleaved by many descendants. They use information on the depth of nodes in order to skip those descendants. However, even if we skip them, because siblings are not clustered, if there are n child nodes, we have to read n disk blocks in the worst case.

There are also many path query processing schemes that scan nodes in pre-order [42], [2], [20]. These schemes avoid scanning some irrelevant nodes in order to reduce computation cost. Pre-order is, however, not I/O-optimal for retrieving children, as explained above. To solve this problem, [19] proposed an indexing scheme that uses two B-tree indices so that we can scan a tree in both depth-first and breadth-first order. Our scheme achieves the same benefit without maintaining two B-trees. Moreover, even if we can identify the answer nodes by using indices, if they are not clustered, we have to read many disk blocks, as explained above.

There has also been research on succinct data structure for trees that efficiently support navigation (e.g., [25], [16]), and research on the efficient implementation of DOM trees for XML [13], [14]. Those studies, however, consider only node-at-a-time navigations. There are some studies on succinct data structure for trees that support queries retrieving node sets, such as [15], [40], but they assume that the data fits in the memory, and do not discuss I/O-complexity.

III. Proposed Node Ordering

In this section, we first show a node ordering scheme that is optimal for both child and descendant navigations. Then

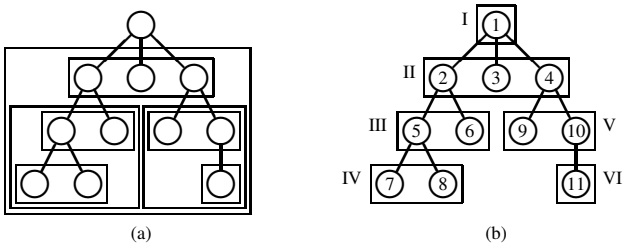


Fig. 2. Clustering of children and descendants

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 of the proposed schemes, the number of disk regions we need to access for those operations is bounded by a small constant. Finally, we also show that there is no good ordering scheme for ancestor navigations, and we devise a caching strategy that achieves efficient processing only with a small size of memory.

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

We start with a node order that is I/O-optimal for $a \rightarrow X$ and $a \stackrel{*}{\rightarrow} X$. The requirement is to cluster both children and descendants of every node. The boxes in Fig. 2(a) show the node sets to cluster in the tree. Note 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.

I/O-Optimal Ordering for $a \rightarrow X$ and $a \stackrel{*}{\rightarrow} X$: $<_t$
For any nodes n_1, n_2 in a tree t, (1) if n_1 (or n_2) is the root node, $n_1 <_t n_2$ (or $n_2 <_t n_1$), (2) 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, and (3) 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. □

That is, we group siblings, we sort the sibling groups in the depth-first order in t, and within each group we sort nodes in the 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 and descendants of the node 2 have consecutive numbers 5, 6, and 5, 6, 7, 8, respectively.

In the following, we call those numbers *addresses* of 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). If nodes have some data, they may be stored in each entry. Then we can process $a \rightarrow X$ and $a \stackrel{*}{\rightarrow} X$ by the procedures below:

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

Algorithm for $a \stackrel{*}{\rightarrow} X$:
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, in Fig. 2(b), we can retrieve descendants of node 2 by first retrieving its children, 5 and 6, and then retrieving the following nodes 7 and 8. In the following, N denotes a node set to retrieve, let B be the size of the disk blocks, and let I/O-complexity be the number of disk blocks to read in the worst case. Then the following holds.

***Theorem** 2:* The procedure for $a \rightarrow X$ and $a \stackrel{*}{\rightarrow} X$ above correctly retrieves the children and the descendants of a, and their I/O-complexities are $\lceil |N|/B \rceil + 1$, which are optimal. □

The procedures above retrieve children and descendants by scanning a single consecutive disk region without reading irrelevant nodes, except for the last node at which we stop the scan. Because the unit of real disk access is a disk block, we do not read an irrelevant block unless the node at which we stop happens to be the first node of some block. If necessary, we can prevent even such unnecessary access by storing in each block (1) a one-bit flag showing whether the last node in that block is a last sibling, and (2) a counter which shows how many ancestors of the last node in the block have that last node as the last descendant.

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

Next, we introduce $a \stackrel{l}{\rightarrow} X$ and $a \stackrel{l*}{\rightarrow} X$. Then the following theorem holds for $a \stackrel{l*}{\rightarrow} X$.

***Theorem** 3:* No node order can cluster answers to both $a \stackrel{l*}{\rightarrow} X$ and $a \rightarrow X$ without interleaving nodes. □

***Theorem** 4:* No node order can cluster answers to both $a \stackrel{l*}{\rightarrow} X$ and $a \stackrel{*}{\rightarrow} X$ without interleaving nodes. □

Fig. 4(a) illustrates the conflict between $a \stackrel{l*}{\rightarrow} X$ and $a \rightarrow X$. In this tree, $1 \rightarrow X$, $1 \stackrel{\lambda*}{\rightarrow} X$, $1 \stackrel{\mu*}{\rightarrow} X$, $1 \stackrel{\omega*}{\rightarrow} X$ 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, $a \stackrel{l*}{\rightarrow} X$ and $a \stackrel{*}{\rightarrow} X$ 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. For example, in Fig. 4(b), $1 \stackrel{\lambda*}{\rightarrow} X$ retrieves nodes 2 to 7, and $2 \stackrel{*}{\rightarrow} X$, $3 \stackrel{*}{\rightarrow} X$, $4 \stackrel{*}{\rightarrow} X$ 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. □

Therefore we have two choices: sacrificing $a \rightarrow X$ and $a \stackrel{*}{\rightarrow} X$, or sacrificing $a \stackrel{l*}{\rightarrow} X$. If we sacrifice $a \stackrel{l*}{\rightarrow} X$, a node order $<_t^l$ defined on a tree t, as below, gives an optimal scheme for the other three operations, i.e., $a \rightarrow X$, $a \stackrel{*}{\rightarrow} X$, and $a \stackrel{l}{\rightarrow} X$:

I/O-Optimal Ordering for $a \rightarrow X$, $a \stackrel{*}{\rightarrow} X$, $a \stackrel{l}{\rightarrow} X$: $<_t^l$
Given a tree t, let t' be the tree created from t by stable sorting

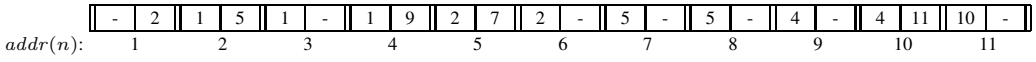
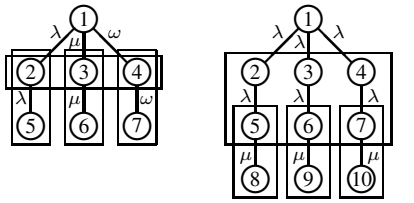
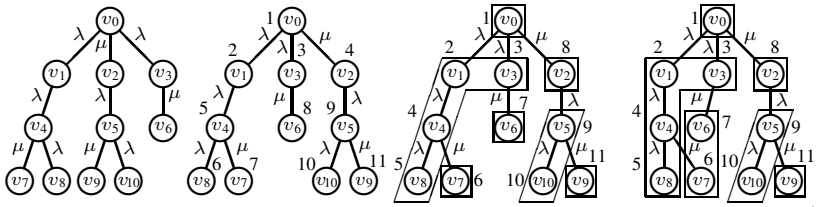


Fig. 3. Disk image of the tree data in the scheme based on $<_t$



(a) conflict with $a \to X$ (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 $<_t^{l*}$ (d) the order by $<_t^{l*\prime}$

Fig. 5. Ordering for edge-labeled trees

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 $n_1 <_{t'} n_2$. □

The only difference between $<_t^l$ and the previous $<_t$ is that we sort the children of each node primarily by their labels. For example, given a tree shown in Fig. 5(a), Fig. 5(b) shows a tree after the sorting of sibling nodes, and the numbers beside the nodes represent the final node ordering given by $<_t^l$.

We also modify the storage scheme. In each node entry, we store a pointer $\mathit{firstChild}(a, l)$ for each label l, which points to the first child reachable via that label. Child pointers in each entry are stored in the dictionary order of the labels. 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 the binary file because each node entry in this scheme has a variable length and we cannot use $addr(n)$ as pointers. When the given data is very homogeneous, i.e., when every node has almost the same set of outgoing edge labels, fixed length tuples including null pointers may be more efficient.

On this data representation, $a \to X$ and $a \xrightarrow{*} X$ are processed by the same procedure as before, except that each entry 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 $\mathit{firstChild}(a, l)$,
2) stop the scan when we reach an entry of n s.t. either
   - $addr(n) = \mathit{firstChild}(a, l')$ where l' is the label of the next child pointer in a, or
   - $\mathit{parent}(n) \neq addr(a)$. □

We again have the theorem below (the proof is omitted):

***Theorem** 5:* The procedures above for $a \to X$, $a \xrightarrow{*} X$, $a \xrightarrow{l} X$ are correct, and their I/O-complexities are $\lceil |N|/B \rceil + 1$, which are optimal. □

On the other hand, this representation is very inefficient for $a \xrightarrow{l*} X$, as shown by the theorem below:

***Theorem** 6:* On this storage scheme, the I/O-complexity of $a \xrightarrow{l*} X$ is $|N|$, where N is the set of the answer nodes. □

Proof Outline: In the worst case, each relevant node may appear alone in the middle of a different sibling group. □

Another choice is to choose $a \xrightarrow{l*} X$, sacrificing $a \to X$ and $a \xrightarrow{*} X$. Before defining a node order that is optimal for them, we define a couple of concepts. First, 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. Next, we define *unilabeled clusters* in t as subgraphs created from the maximal unilabeled connected subgraphs, by removing their root nodes. We call a removed node *the original root* of the corresponding unilabeled cluster. Notice that each unilabeled cluster forms a forest whose roots are siblings in the original tree t. We also regard the root node of t always forms a unilabeled cluster including only itself. The unilabeled clusters of a tree then disjointly classify all the nodes in it.

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:

I/O-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 $<_t^l$. For any nodes n_1, n_2 in t, let c_1, c_2 be the unilabeled clusters including n_1, n_2, and let m_1, m_2 be the first nodes in c_1, c_2 in the depth-first order in t', respectively. Then (1) if $c_1 = c_2$, $n_1 <_t^{l*} n_2$ iff $n_1 <_{t'} n_2$, (2) if $c_1 \neq c_2$, $n_1 <_t^{l*} n_2$ iff m_1 precedes m_2 in the depth-first order in t. □

In other words, we sort the unilabeled clusters in t' in the depth-first order of their first nodes, and within each unilabeled cluster, we sort nodes in the order of $<_{t'}$.

Fig. 5(c) illustrates $<_t^{l*}$ defined on the tree in Fig. 5(a). The seven boxes in the figure represent unilabeled clusters. For example, v_1, v_3, v_4, v_8 form a unilabeled cluster, whose original root is v_0 and whose first node is v_1. We sort these seven clusters in the depth-first order of their first nodes, and within each cluster, nodes are sorted by $<_t$. The numbers beside the nodes represent the node order given by $<_t^{l*}$. In this ordering, for any tree t and its node a, the answer nodes of $a \xrightarrow{l} X$ or $a \xrightarrow{l*} X$ have consecutive positions.

We also modify the storage scheme. We change the node order to $<_t^{l*}$, and in each entry of the node n, we store its label, which we denote by $label(n)$. Although it is possible to process $a \xrightarrow{l*} X$ without $label(n)$ as explained later, here we show an algorithm that uses it because that algorithm is simpler, and we need to store $label(n)$ anyway when we introduce backward navigations later. Now we show how to process $a \xrightarrow{l} X$ and $a \xrightarrow{l*} X$ on the data representation above.

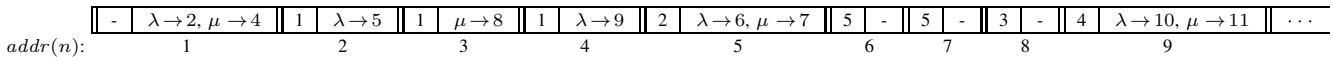


Fig. 6. Disk image in the scheme based on $<_t^l$

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 a node n s.t. either
   - $parent(n) \neq addr(a) \wedge parent(n) < firstChild(a, l)$ or
   - $label(n) \neq l$. □

Then, the following theorem holds for these algorithms.

***Theorem** 7:* The procedures above for $a \xrightarrow{l} X$ and $a \xrightarrow{l*} X$ are correct, and their I/O-complexities are $\lceil |N|/B \rceil + 1$, which are optimal. □

Proof Outline: Let l' be the label of a, and let c be the unilabeled cluster including $firstChild(a, l)$. In either $a \xrightarrow{l} X$ or $a \xrightarrow{l*} X$, all the nodes to retrieve are within a single cluster c, and are given consecutive positions starting at $firstChild(a, l)$ because nodes within a unilabeled clusters are sorted by $<_t$.

In $a \xrightarrow{l} X$, the node n that immediately follows the last l-child of a is either (1) some node in c which is not a child of a, (2) the first node in the next unilabeled cluster whose original root is not a, or (3) the first node in the next unilabeled cluster whose original root is a. In Case (1) or (2), $parent(n) \neq addr(a)$, and in Case (3), n is the node pointed by the next child pointer of a.

In $a \xrightarrow{l*} X$, the node n that immediately follows the last l-descendant of a is either (1) some node in c which is not a descendant of a, (2) the first node in the next cluster whose original root is not a l-descendant of a, or (3) the first node in the next cluster whose original root is a l-descendant of a. In Case (1) or (2), $parent(n) \neq a \wedge parent(n) < firstChild(a, l)$, and in Case (3), $label(n) \neq l$ because otherwise n should be in c, which is a contradiction. □

For example, suppose we process $v_0 \xrightarrow{\lambda*} X$ in Fig. 5(c). We start the scan at $firstChild(v_0, \lambda)$, i.e., v_1, and proceed to v_3, v_4, and v_5. When we reach v_6, $label(v_6) = \mu \neq \lambda$ (Case (3)). On the other hand, when we process $v_4 \xrightarrow{\mu*} X$, we start the scan at v_7, and when we reach v_6, $parent(v_6) = 3 \neq addr(v_4) = 4 \wedge parent(v_6) = 3 < firstChild(v_4, \mu) = 6$ (Case (2)).

The algorithm above uses $label(n)$, but as mentioned before, we can compute $a \xrightarrow{l*} X$ even if we do not store $label(n)$. In that case, in order to detect Case (3) in $a \xrightarrow{l*} X$, while scanning the l-descendants of a, we examine the child pointers in those entries for any l' s.t. $l' \neq l$, and record the smallest address among them. Then, if the scan reaches the recorded address, it means we get out of the cluster, and we should stop.

$<_t^{l*}$ has one significant advantage. Although $<_t^{l*}$ is not "optimal" for the other operations, $a \rightarrow X$ and $a \xrightarrow{*} X$, we can process them quite efficiently by the following procedures:

Algorithm for $a \rightarrow X$:

Repeat $a \xrightarrow{l} X$ for all l s.t. a has a l-child. □

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 when we reach a node n s.t. either
   - $parent(n) \neq addr(a) \wedge parent(n) < firstChild(a, l)$ or
   - $label(n) \neq l$.
6) Scan the node entries starting at $minAdd$.
7) Stop the scan at n s.t. $parent(n) \neq addr(a) \wedge parent(n) < firstChild(a, l)$. □

***Theorem** 8:* The procedures above for $a \rightarrow X$ and $a \xrightarrow{*} X$ are correct, and their I/O-complexities are $\lceil |N|/B \rceil + 2L - 1$ and $\lceil |N|/B \rceil + 3$, respectively. □

Proof Outline: Let l be the label of a, and let c be the unilabeled cluster including a. $a \rightarrow X$ is trivial. In $a \xrightarrow{*} X$, the node set to retrieve is the union of the following four sets:

S_1: l-descendants of a. They appear within the cluster c.
S_2: Further descendants of nodes in S_1. They appear in other clusters, whose original roots are nodes in S_1.
S_3: l'-descendants of a for some $l'(\neq l)$. For each l', they appear in another cluster, whose original root is a.
S_4: Further descendants of nodes in S_3. They are in yet other clusters, whose original roots are nodes in S_3.

S_1 is a subset of c. Each of the other clusters is either entirely included in the answer, or not included at all. In addition, clusters including nodes in S_2, S_3, S_4 are all given consecutive positions in the disk because clusters are sorted by the depth-first order of their first nodes. Therefore, we can process $a \xrightarrow{*} X$, in essence, by the following two scans:

(i) scan a subregion of c that stores nodes in S_1, and
(ii) scan all the clusters that are subsets of S_2, S_3, or S_4, which are stored in consecutive positions.

In the latter scan, we should start the scan at:

1) if a has child pointers for some $l'(\neq l)$, the earliest cluster among those pointed by them, and
2) if a has no child pointer for $l'(\neq l)$, the earliest cluster whose original root is some l-descendant of a. □

For example, on the tree in Fig. 5(c), we process $v_4 \xrightarrow{*} X$ by (i) retrieving its λ-descendants that appear at consecutive positions in one cluster (only v_8 in this case), and (ii) scanning a sequence of the clusters that store the other descendants of v_4 (only the cluster including v_7 in this case). In this case, we start the second scan at v_7 which is pointed by the next child pointer of v_4. This corresponds to Case (1) above. On the other hand, we process $v_1 \xrightarrow{*} X$ by (i) retrieving its λ-descendants that are stored in consecutive positions in one cluster (v_4

and v_8), and (ii) retrieving their further descendants in other clusters by scanning nodes starting at v_7. This corresponds to the Case (2) above, and we start the scan at the node pointed by $firstChild(v_4, \mu)$.

In this way, in this storage scheme, the number of disk regions required to access for processing $a \rightarrow X$ is bounded by the number of labels, which is a small constant in most practical cases, and that for processing $a \xrightarrow{*} X$ is at most 2, which is not "optimal," but near-optimal.

In this paper, we focus on the set-based navigations, and do not consider path queries including more than one steps. In fact, our ordering scheme based on $<_t^{l*}$ is efficient for the set-based navigations, but not optimal for longer path queries. For example, if we process a path query $v_0 \xrightarrow{\lambda*} \cdot \xrightarrow{\mu*} X$ on the tree in Fig.5(c), its answers, i.e., v_6 and v_7, are in separate clusters in our scheme, and there may be many nodes between them if there is a large subtree rooted by v_7.

To see the tradeoff between efficiency for one-step navigations and that for longer path queries, we show a variation of the previous scheme, which is a mixture of our idea of $<_t^{l*}$ and DataGuide [17]. In this scheme, we recursively apply vertex contraction starting from the root node. Here we only show its informal definition for the sake of space limitation.

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

Given a tree t, we sort the siblings as before and obtain t'. Then, for each label l, we merge all the l-descendants of the root node into one node, and create a contracted tree. Then we recursively apply the same vertex contraction to each subtree rooted by the children of the root. The nodes that are merged into one node are stored in consecutive positions as a cluster. Those clusters are stored in the depth-first order, and the nodes in each cluster are sorted by $<_t$. However, when nodes in a cluster form a forest whose roots do not share the same parent in the original tree, we sort the nodes in that cluster by assuming these roots are the children of some virtual node, and their sibling order is the depth-first order of the parents of these root nodes in the original tree. □

Fig. 5(d) shows an example of this scheme. First we merge v_1, v_3, v_4, v_9 that are reachable through λ-edges from the root, into a contraction node. We recursively apply the same procedure to the subtrees, and merge v_6 and v_7 that are reachable through μ-edges from that contraction node. Notice that v_6 and v_7, which are the answers to the query $v_0 \xrightarrow{\lambda*} \cdot \xrightarrow{\mu*} X$, are in the same cluster in this scheme. Because the nodes in that cluster form a forest whose roots, i.e, v_6 and v_7, do not share the same parent, we assume that they are children of a virtual node when sorting the nodes in this cluster. The final node ordering is shown by the numbers beside the nodes.

As shown by the example of $v_0 \xrightarrow{\lambda*} \cdot \xrightarrow{\mu*} X$ above, and also shown in the experiments in Section V, this scheme is efficient for path queries consisting of more than one steps in many cases, especially when they start from the root node. This scheme, however, does not guarantee a constant bound on the number of disk regions to access for $a \xrightarrow{*} X$ or $a \xrightarrow{l*} X$, except when the query starts from the root.

TABLE I
NUMBER OF DISK REGIONS TO ACCESS

	$a \rightarrow X$	$a \xrightarrow{*} X$	$a \xrightarrow{l} X$	$a \xrightarrow{l*} X$
$<_t$	1	1	$\|N\|$	$\|N\|$
$<_t^l$	1	1	1	$\|N\|$
$<_t^{l*}$	L	2	1	1

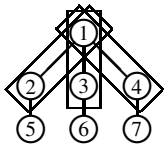


Fig. 7. Conflicts in ancestor clustering

Discussion: The numbers of regions we need to access for each operation in ordering scheme $<_t$, $<_t^l$, and $<_t^{l*}$ are summarized in Table I, where N is the set of the answer nodes, and L is the number of distinct label names occurring under a node. Although $<_t^l$ is optimal for the three operations, it can be 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 (as long as the number of distinct labels is a small constant). Therefore, $<_t^{l*}$ is preferable in most cases. There is a trade-off between efficiency of single-step navigations and that of longer path queries, and if long path queries starting from the root are the main concern, the last scheme based on $<_t^{l*'}$ may be more efficient.

C. Parent and Ancestor Navigations

Next, we discuss the remaining four backward navigations. Unfortunately, the following theorem holds.

***Theorem** 9:* No node order can cluster ancestors of every node at the same time. □

That is, there is no good clustering scheme for ancestors.

Proof. Fig. 7 illustrates the problem. The three boxes in this figure show the ancestors of the node 5, 6, 7, which are $\{1, 2\}$, $\{1, 3\}$, and $\{1, 4\}$. Obviously, it is impossible to serialize the nodes without decomposing any of these boxes. □

However, there is an algorithm that allows us to retrieve ancestors in a constant number of disk accesses if we can use a small amount of cache memory and additional disk spaces. It takes a commonly used approach: First, we store in the cache the nodes that are frequently included in the answers to ancestor retrieval. Second, when retrieving ancestors of a, we access p disk blocks preceding the block including a, where p is a predefined small constant. Then, we can extract all the ancestors from these preceding blocks and the cache.

Our problem, however, is that it is *not trivial* whether we can develop an algorithm that chooses nodes to cache working with our novel ordering scheme. Here we show there exists such an algorithm: First, for each node n, we compute $overflowsize(n)$, which is the number of a s.t. $X \xrightarrow{l*} a$ retrieves n, and the p preceding blocks of a do not include n. Then we store in the memory cache the nodes n having large values for $overflowsize(n)$. The $overflowsize(n)$ can

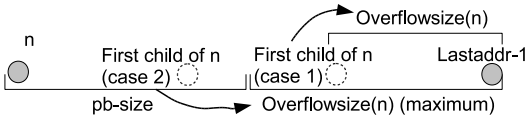


Fig. 8. Computing $overflowsize(n)$

be computed by the function `overflowsize(`n`,`$laddr$`,`pb`)` shown below. Here, for simplicity, we assume that we use the storage scheme based on $<_t^l$, and the descendants of each node is stored in one region. When we use the scheme based on $<_t^{l*}$, the descendants of each node is stored in two regions, and therefore, we repeat the similar computation twice.

```
1. long overflowSize(n, laddr, pb) {
2.   long overflowSizeMax = ( laddr - addr(n) ) - pb;
3.   if (overflowSizeMax > 0) {
4.     return min(overflowSizeMax, laddr - firstChild(n));
5.   else return 0;
6. }
```

This function takes as parameters a node reference n, the address $laddr$ next to the end of the subtree rooted by n (e.g., in the tree in Fig. 2(b), if $n = 2$, then $laddr = 9$), and pb, the number of nodes stored in the p preceding blocks.

Fig. 8 illustrates how $overflowsize(n)$ works. Suppose the first child of n is far from n and the p preceding blocks of the first child do not include n (Case 1). In this case, the overflowsize is computed by `laddr - firstChild(n)`. In Case 1, if the first child is closer to n, the overflowsize becomes larger. Once the first child is close enough that its p preceding blocks contain n (Case 2), the overflowsize stops growing, and the maximum size is computed by Line 2. Then we take the smaller one of the two values.

If we store all the nodes s.t. $overflowsize(n) > 0$ in the cache, we can retrieve ancestors only by accessing p preceding blocks, i.e., by a constant number of disk access. We write $cachesize(t, pb)$ to denote the necessary cache size for that. It is difficult to estimate $cachesize(t, pb)$ for arbitrary tree, so here we give $cachesize(t, pb)$ for a perfect m-tree. In this case, the size of a subtree is $\frac{1}{m}$ of the tree rooted by its parent. When $\frac{|t|}{m^n} \leq pb$, the subtrees whose root are level-n nodes are contained in the p preceding blocks, which means the required cache size is $O(m^n)$. Therefore, $cachesize(t, pb) = \frac{|t|}{pb}$. If $pb = 500$ (i.e., the preceding blocks contain 500 nodes), the required memory size is $1/500$ of the required disk size.

If we do not have enough memory size to cache those $cachesize(t, pb)$ nodes, we can store them on the disk, which forms a smaller tree, and recursively apply the same technique to the tree. Then we only need $1/500^2$ size of the memory in exchange for one more disk access to p blocks and an additional $1/500$ disk space. This dramatically reduces the required memory. For example, we need only 4Mbyte memory (and additional 2G disk) for 1T byte database.

IV. Compatibility with Other Techniques

One important question is whether our ordering schemes can work well with some existing techniques that are important or even necessary in some applications. This section shows that popular approaches to the three important problems—handling sparse graphs, navigations starting at multiple nodes, and update of sorted data—can work well with our schemes. We omit the discussion on edge labels, but the result shown here can be generalized to data with edge labels.

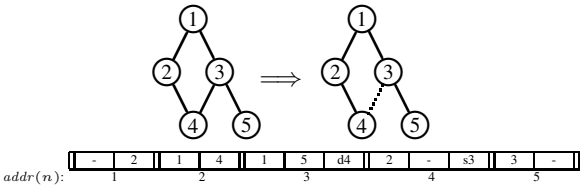


Fig. 9. Non-tree edges and the disk image

A. Sparse Graphs

As in many studies on sparse graphs, we decompose a sparse graph into a spanning tree and *non-tree edges*, i.e., the edges that are removed when we construct the spanning tree. Then we store in the source (destination) nodes of non-tree edges pointers to the destinations (sources), which are distinguished from the ordinary pointers to parents and children. For example, given a tree in Fig. 9, we construct a spanning tree by regarding the edge from 3 to 4 as a non-tree edge, and we store additional pointers in the entry of 3 and 4 in the disk image as shown at the bottom of Fig.9.

To process a navigation q from a against a graph with non-tree edges, we compute a node set A' s.t. if we evaluate q starting from each node in A' on the spanning tree of the graph, and merge their results, we get the result of q starting from a on the graph. For example, from $X \xrightarrow{*} 4$, we get $A' = \{4, 3\}$ s.t. the union of the results of $X \xrightarrow{*} 4$ and $X \xrightarrow{*} 3$ is the result of the original query.

To compute A', we construct the following table from the non-tree edges and keep it in the memory.

$$nt\text{-}edge(s, d, d\_region)$$

Here s and d are the source and destination addresses of a non-tree edge, and d_region is the range (a pair of addresses) of the subtree rooted by d. The size of the table is $O(|nt\text{-}edge|)$, which we assume is small as mentioned above.

From each tuple (s, d, d_region) in the table, we obtain two kinds of rules: a forward expansion rule $(s \rightarrow d)$ and a backward expansion rule $(d\_region \rightarrow s)$. The former says that, if we start a forward traversal from a node n whose region includes s, then we eventually jump to another subtree rooted by d. The latter says that when we start a backward traversal from a node in the d_region, we eventually jump to node s.

Suppose we process $a \xrightarrow{*} X$ for some a. Then let $A_0 = \{a\}$, and we obtain A_{i+1} by applying forward expansion rules to the nodes in A_i, and we get $A' = A_n$ if $A_n = A_{n+1}$. Similarly, A' for $X \xrightarrow{*} a$ is computed by using backward expansion rules. When we compute the union of the results, duplicate elimination is done in the way explained in Section IV-B.

B. Multiple Starting Nodes

Here we consider navigations with more than one (but not too many to fit in the memory) starting nodes. We write $A \rightarrow X$ to denote a child navigation with multiple starting nodes, i.e., a retrieval of $\bigcup_{a \in A} a \rightarrow X$. Similarly, we write $A \stackrel{*}{\rightarrow} X$, $X \rightarrow A$, and $X \stackrel{*}{\rightarrow} A$ to denote the other types of navigations with multiple starting nodes. Notice that the answers to those queries are unions of the answers for all the starting nodes, and that they are different from the structural joins whose answers are pairs (a,b) that have the given relationship.

For efficient evaluation of them, we need to consider two issues: redundant disk access and the cost of duplicate elimination. A naive approach to avoid them is to keep information on the nodes that have already been processed, but its space complexity is linear to the size of the answer, which can be the entire data in the worst case. Interestingly, our ordering scheme allows us to reduce the required memory size.

Each type of navigation has its own strategy for avoiding such redundancies. Here we explain only the algorithms for $A \stackrel{*}{\rightarrow} X$ and $X \stackrel{*}{\rightarrow} A$ below because $A \rightarrow X$ and $X \rightarrow A$ are much easier than these. In the following explanation, we assume the data is a tree, but when the data is a graph, we can apply the same technique as in Section IV-A.

$\boldsymbol{A \stackrel{*}{\rightarrow} X}$: Here we assume that the size of A is small enough to fit in the main memory. Then we sort nodes in A in the memory in the order of the address of their first child. This sorting maximize the possibility of sequential access. Notice that the descendants of each starting node are stored in a contiguous region on the disk, and two regions for two starting nodes are either distinct or included, but never partially overlaps. Given this property, and the fact the nodes in A are sorted in the order of the start addresses of their descendant regions, it is sufficient to keep only one address pair $(start, end)$ of a region to check whether the descendants of the later starting nodes are subset of the former results. Below is the algorithm that evaluates $A \stackrel{*}{\rightarrow} X$ without redundancy, based on this idea. If the address of the first child of the next starting node is included in the current region, then that starting node is skipped (Line 4). Otherwise, we evaluate it and replace the current region with the region of the new result (Lines 5-6).

```
1. CurrentExtent =(null, null)
2. for each a in A {
3.    firstaddr= a.firstchildaddr();
4.    if (currentextent contains firstaddr) continue;
5.    process a and get (firstaddr, lastaddr);
6.    CurrentExtent= (firstaddr, lastaddr);
7. }
```

$\boldsymbol{X \stackrel{*}{\rightarrow} A}$: $X \stackrel{*}{\rightarrow} A$ follows parent pointers until it reaches the root node, and repeat it for each $a \in A$. To avoid visiting the same node later in the traversal from another a, we record one *stop node* for each a. Initially, the stop node for each a is the root node (Line 1 in the algorithm below). Then we evaluate $X \stackrel{*}{\rightarrow} a$ for each a, and obtain only "new" answers, i.e., nodes between a and a child of the stop node, excluding the stop node or its ancestors (Line 3-4). Then we update the stop node for a remaining starting node `aa` if it is a descendant of some of those new answer nodes (Line 7-8). Note that the current `stopNode[aa]` can never be equal or a descendant of the new answer n.

Note that the start and end addresses of a subtree rooted by n can be obtained by looking at $firstChild(n)$, and also $firstChild(n')$, where n' is n's following sibling. When n is the last sibling, we need to look at the following sibling of n's parent instead and, if it is also the last sibling, we repeat this to more distant ancestors, up to the root node in the worst case. In the algorithm below, however, we only need the regions of subtrees rooted by some node on the retrieved path `P`. In that case, we have already accessed all their ancestors before that. Therefore, if we keep the necessary information for the ancestors that are the last sibling, we can compute the region of some new n without extra disk access.

```
 1. stopNode[1..|A|] = initialized by root(start, end);
 2. for each a in A {
 3.   P = a node sequence from (exclusive) stopNode[a] to a;
 4.   // update stop nodes for the remaining a's;
 5.   for each remaining node aa in A {
 6.     for each node n on P in the parent-to-child order {
 7.       if (aa is not within n(s,e)) break;
 8.       stopNode[aa]=n;
 9.     }
10.   }
11. }
```

We show the complexity of navigations with multiple starting nodes. Analysis of $A \rightarrow X$ and $A \stackrel{*}{\rightarrow} X$ is easy. In the followings, N is the set of nodes to retrieve as defined before.

***Theorem** 10:* The time, space, and I/O complexities for both $A \rightarrow X$ and $A \stackrel{*}{\rightarrow} X$ are $O(|A|log|A| + |N|)$, $O(|A|)$, and $\lceil |N|/B \rceil + 2|A| - 1$, respectively. □

Proof Outline: We need to read the nodes in A and sort them. The answer for each a exists in a contiguous region on the disk as explained before, but we have $|A|$ regions, each of which is for some a, and they may not be contiguous. □

For $X \stackrel{*}{\rightarrow} A$, here we assume we have enough memory for the memory cache explained in Section III-C.

***Theorem** 11:* The time, space, and I/O complexities for both $X \rightarrow A$ and $X \stackrel{*}{\rightarrow} A$ are $O(|A|^2 + |A|h + |N|)$ where h is the height of the tree, $O(|A| + cachesize(t, pb\text{-}size) + pb\text{-}size)$, and $|A|$, respectively. □

Proof Outline: In $X \stackrel{*}{\rightarrow} A$, we need to traverse parent pointers only $O(|N|)$ times because we eliminate duplicates. We also need to update the stop nodes. In the worst case, all the stop nodes are updated every time, but update can occur at most $|A|h$, which is the total length of paths between the root and a. In the memory, we need to store the stop nodes, and here we assume the node cache is larger than $cachesize(t, pb\text{-}size)$, which guarantees only one disk access for each $X \stackrel{*}{\rightarrow} a$. □

C. Updates

In the schemes proposed in this paper, it is crucial to keep the order of nodes sorted in the storage. This section shows that a relatively simple combination of a B-tree like structure and a lazy update mechanism works sufficiently in our scheme. Although more sophisticated and efficient approaches are also applicable, such discussions are beyond the scope of this paper.

The idea of keeping the data in secondary storages sorted is quite common in relational database systems. The difficulty

in implementing sorted relations is how to insert new tuples to disk blocks that are already full. We can take an approach similar to that in [3]: we store nodes in fixed-sized blocks, where each block is between 50% and 100% full like B-trees. If the blocks become full, or have less than 50% of the space full, block merging and splitting occur, in order to keep the node replacement to a small extent.

The problem here, however, is that we need to maintain pointers to other nodes in compliance with the node replacement. So we adopt a logical/physical address approach [18]: we do not update the old physical pointers immediately, but we keep a mapping table for computing the correct physical addresses from the old values. The old incorrect pointers will be updated at some later time when the system is idle.

Note that, in the pointer-update process, we only correct pointers that still have old values to new values one by one. Therefore it can be done incrementally: we can update only one pointer, and defer updating the remaining old pointers to the next idle time. Therefore, the cost of this pointer-update process is negligible in practice. The details of the update mechanism and the experimental results showing its effectiveness are given in [30].

V. Evaluations

Although the I/O efficiency of the proposed schemes was theoretically shown in the previous sections, it is important to experimentally confirm the theoretical results. This section gives the results of our experiments. First, we conducted experiments using synthesized data in order to clarify the characteristics of the proposed schemes. Next, we conducted experiments using real data in order to see the effectiveness in the real settings. The experiments were run on Windows XP SP2 on a PC with a Pentium M 1.73 GHz and 256MB memory. Programs are written in Java 1.6.0. The size of the read buffer is 4096 bytes, and the average size of each tree (or graph) node on the disk is 26 bytes.

A. Analyzing Characteristics

A1. Comparison with the Depth-First Order. First, we measured the elapsed time for the four wildcard navigations against a tree stored in two different ordering schemes: depth-first order and the order $<_t^{l*}$. (The comparison among $<_t$, $<_t^l$, $<_t^{l*}$ will be given in the next experiment.) On the depth-first order scheme, we used the stack-tree join [2] and staircase join [20]. We chose the two approaches based on depth-first order, because other existing approaches, which are not based on depth first order, either require workload information, or focus on queries starting from the root node (or any nodes) as explained in Section II. Staircase join is known as one of the most efficient algorithms on depth-first node order.

Settings. We synthesized various sizes of perfect n-trees: those with $\frac{n^h-1}{n-1}$ nodes where $(n,h) = (2,17),(2,18),(2,19),(4,9),(4,10),(4,11),(8,6),(8,7),(8,8)$. The largest one contains about 2,400,000 nodes. We randomly picked up 100 nodes out of the nodes at depth four, and use them as the starting nodes for navigations.

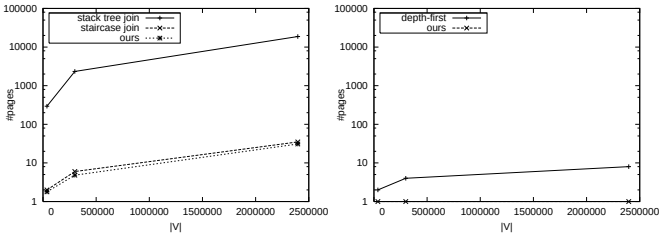


Fig. 11. #Pages accessed in descendant navigations (left) and #Pages that contain answer nodes in child navigations (right)

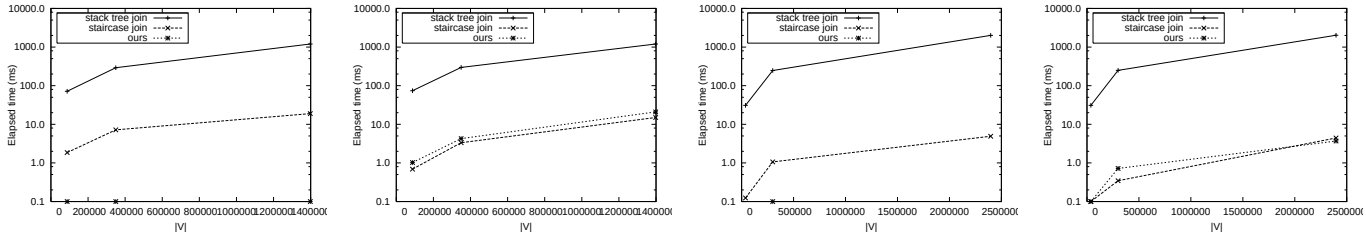
TABLE II
The Number of Regions for Single-Step Navigations

operation	#nodes in the result	$<_t$	$<_t^l$	$<_t^{l*}$	$<_t^{l*\prime}$
$a \xrightarrow{l1} X$	5	5	1	1	1
$a \xrightarrow{l1*} X$	488,281	488,281	97,656	1	1
$a \rightarrow X$	10	1	1	2	2
$a \xrightarrow{*} X$	111,111,111	1	1	2	572

Results and Discussions. In the experiment, the order $<_t^{l*}$ showed the similar behavior to that of the depth-first order with the staircase join, expect for child navigations. For space limitation, we only show the most interesting results, which compare the three algorithms in the child and descendant navigations. (The results not shown here are given in [30]). Figure 10 shows the average time for each query against the data whose fanout is 4 or 8. As the result suggests, the depth-first order is not efficient for child navigations, since children are interleaved by irrelevant nodes. Staircase joins are efficient for descendant navigations because they partition the pre/post plane [20] effectively so that they can access only relevant nodes for descendant navigations, but stack-tree join have no such skipping mechanism (Figure 11 shows the numbers of pages read by each algorithm). Therefore the complexities of navigations by stack-tree joins are always in $O(D)$ where D is the data size. On the other hand, $<_t^{l*}$ is efficient for both child and descendant navigations because the complexity is in the order of answer size for all types of navigations.

Even in child navigations on a storage scheme based on the depth-first oder, if we have appropriate indices, we can avoid accessing irrelevant blocks. However, in a scheme based on the depth-first order, even if we have indices, we cannot avoid reading irrelevant nodes stored in the same blocks as the relevant nodes, as explained before. To confirm these two, we counted the number of pages that have at least one answer node for child navigations in each scheme, i.e., the depth-first and $<_t^{l*}$. The result is shown in Figure 11 (right). This result shows that for child navigations, the number of pages that contain answer nodes is far smaller compared with the entire data, even in the depth-first order. However, the number of pages including answer nodes in our scheme, is much more smaller than that in the depth-first order.

A2. Comparison between Our Ordering Schemes. Next, we compare our ordering schemes. In this comparison, we use a 10-ary perfect tree with the depth 10. Among 10 children of



(fanout=4, $a \rightarrow X$) (fanout=4, $a \overset{*}{\rightarrow} X$) (fanout=8, $a \rightarrow X$) (fanout=8, $a \overset{*}{\rightarrow} X$)

Fig. 10. Comparison with the depth-first order

TABLE III
THE NUMBER OF REGIONS FOR MULTI-STEP NAVIGATIONS

operation	#nodes in the result	$<_t^{l*}$	$<_t^{l*'}$
Root $\overset{l1}{\rightarrow} \cdot \overset{l2}{\rightarrow}$ X	25	6	2
Root $\overset{l1}{\rightarrow} \cdot \overset{l2*}{\rightarrow}$X	2,441,405	6	3
Root $\overset{l1*}{\rightarrow} \cdot \overset{l2}{\rightarrow}$ X	2,441,405	175,782	3
Root $\overset{l1*}{\rightarrow} \cdot \overset{l2*}{\rightarrow}$X	23,803,711	175,782	3

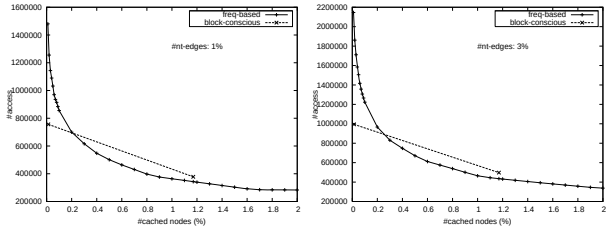


Fig. 12. Performance gain by memory cache

each node, 5 have label $l1$, and 5 have label $l2$. We store this tree in three schemes, $<_t$, $<_t^l$, and $<_t^{l*}$, and in each scheme, we counted the numbers of disk regions we need to access for the four types of navigations starting from a node at the second level. Table II shows the type of navigations, the number of nodes contained in their results, and the number of disk regions storing them. As explained in Section III-B, $<_t^{l*}$ guarantees that the number of disk regions is either a small constant or bounded by the number of labels for these operations.

On the other hand, Table III compares the number of disk regions accessed by path queries starting at the root node, and consisting of two steps. As suggested by this table, if we mostly use path queries that start at the root node, and include more than one steps, $<_t^{l*'}$ would be the better choice.

A3. The Effect of Caching Strategy. Next, we evaluate the effect of our caching strategy for ancestor navigations. One issue in our approach is we cannot efficiently compute the $overflowsize$ when the data is not a tree. However, our experiment explained below shows that $overflowsize$ computed for the spanning tree is sufficient as an approximation.

First, we synthesized perfect binary trees with $2^{19}-1$ nodes, and add 1% or 3% non-tree edges by randomly choosing node pairs. Then we compared the number of disk regions to access for ancestor navigations in the following settings: (1) The ideal frequency-based cache, where we compute the accurate

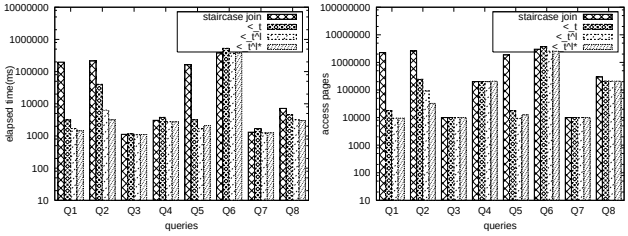


Fig. 13. Exp. B1 (Tree 1): Elapsed time (left) and #pages accessed (right)

frequency of the appearance of each node in the ancestor sets, and store the nodes with high-frequency; (2) The normal (1-level) approach that uses the approximation computed by the spanning tree, and uses enough memory cache to guarantee only one disk access; (3) The 2-level approach, where we use the approximation by the spanning tree, store the 1-level cache in the disk, and construct the 2nd-level cache against it. The scheme requires an additional 144KB of disk space. For a fair comparison, we read the preceding blocks in all the settings. We count the number of regions for 262,144 queries, each of which computes $X \overset{*}{\rightarrow} a$ for each leaf node a in the graph.

Fig. 12 shows the result. The x-axis is the size of memory cache compared to the size of the graph data. Two ends of the line for the block-conscious approach represent the scheme (2) and (3). As the figure shows, the scheme (1) is superior to the scheme (2) because the former reflects access frequencies more accurately. However, when the cache size is small, the scheme (1) shows rapid degradation in performance. On the other hand, the 2-level block-conscious approach shows good results, and it requires only one more disk access, and additional disk space only about 0.014% of the size of the graph data, which means 147MB memory for 1TB graph data.

B. Evaluation Using Real Data

B1. Real Data with Edge-labels. Since we found that the staircase join is superior to the stack-tree join in Experiment A1, we compared our method to the staircase join in a real data setting. We compared them in the elapsed time and the number of pages read for the eight navigations against a tree. Since staircase joins do not support the closure for edge labels, we simulated it by multiple label-specific child/parent navigations.

Settings. We used two trees, each constructed from a file server and the Gene Ontology, respectively. The first tree was constructed from a file server of our research group. Edges of

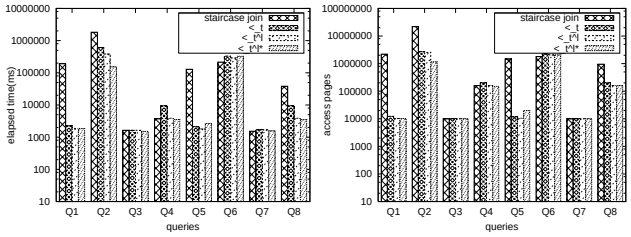


Fig. 14. Exp. B1 (Tree 2): Elapsed time (left) and #pages accessed (right)

the tree has the following two types of labels: (1) "subdir" for edges reaching directory (intermediate) nodes, and (2) "contains" for edges reaching file (leaf) nodes. The average fanout is 26.3, and the maximum depth of the tree is 23. Then, we executed eight types of navigations (Q1: $d \overset{subdir}{\rightarrow} X$, Q2: $d \overset{subdir*}{\rightarrow} X$, Q3: $X \overset{subdir}{\rightarrow} d$, Q4: $X \overset{subdir*}{\rightarrow} d$, Q5: $a \rightarrow X$, Q6: $a \overset{*}{\rightarrow} X$, Q7: $X \rightarrow a$, Q8: $X \overset{*}{\rightarrow} a$), and measured elapsed time. Here, d and a represent a directory and any node, respectively. We chose the starting nodes for navigations as follows: For "downward" (child/descendant) navigations, we randomly picked up 100 nodes out of the tree nodes at the second level down from the root, and for "upward" (parent/ancestor) navigations, we randomly picked up 100 nodes out of the nodes at the second level up from the maximum depth.

The second tree was constructed from the Gene Ontology; its DAG structure was expanded to a tree by copying the nodes with multiple incoming edges. Note that the original structure can be preserved by adding the "equiv" edges between the nodes expanded from the same nodes. The average fanout is 5.1, and the maximum depth of the tree is 19. The tree has "is_a" edges between internal nodes and has seven different labels for the node properties. We executed the eight navigations in the same setting as the first tree except that the "is_a" label is used instead of the "subdir."

Results and Discussions. Figures 13 and 14 show the result. Each graph shows the sum of the elapsed time or the number of accessed pages over all executions. The results for wildcard navigations (Q5 to Q8) are compatible with the results of Experiment A1: The staircase join is not good especially at processing child navigations Q5. As explained in [20], a node skipping technique can be applied to the ancestor navigation by the staircase join, although slightly less effective compared to that for the descendant navigation. The result for Q8 shows that effect. Among the results for label-specific navigations (Q1 to Q4), the staircase join is not efficient for label-specific descendant navigation, since it has to be implemented by label-specific child navigation. For Q2, $<_t^{l*}$ is about twice faster than $<_t^{l}$, while for Q5, $<_t^{l}$ is better than $<_t^{l*}$.

To summarize, the order $<_t^{l*}$ allows us to implement an efficient algorithm that exploits the locality nature of set-based navigations and showed good results both in the elapsed time and in the number of accessed pages.

B2. Sparse Graph. Then, we evaluated our method in the elapsed time for navigations on a graph. Similarly to the directory tree above, the graph was constructed from a file server of our research group and its operation log. The graph is a tree-like sparse graph that consists of the directory tree structure and various relationships among files, such as the copy relationships and file references (HTML hyperlinks, and references from tex files to images). In addition, each file node has various property nodes associated to it, such as file name and file size nodes. In the graph, the number of nodes is 4,997,898, the number of edges is 5,490,665 (i.e., edges/nodes ratio is about 1.1), and the number of label names is 11 (subdir, refersTo, copy, filename, etc.).

TABLE IV
EVALUATION USING REAL DATA (GRAPH)

operation	#nodes	#region	elapsed time (ms)	meaning
$f \overset{copy*}{\rightarrow} X$	9.25	4.17	0.834	copied files
$d \overset{subdir}{\rightarrow} X$	6.82	3.39	0.656	sub-directories
$f \rightarrow X$	8.8	9.06	1.344	file properties
$X \overset{copy*}{\rightarrow} f$	2.00	1.0	21.874	original copies

We randomly picked up 100 such nodes in the graph, executed different navigations, and measured elapsed time. Table IV shows the result. Here, f and d represent a file and directory node, respectively. Each number in the table is the average values over the 500 (100$\times$ 5 executions) results. Note that the number of disk regions is kept relatively small when the graph is sparse. The elapsed time has a strong relationship with the number of disk regions, except for the last query. This is because the existence of non-tree edges requires us to recursively expand an operation before the execution, but the current naive implementation of nt-edge table is tuned only for forward expansion rules, and takes $O(n)$ time complexity for each backward expansion.

VI. CONCLUSION

This paper studies the problem of how we should order nodes of trees or sparse graphs on the disk for efficient processing of set-based navigations. We focused on the eight most common navigation operations, which include neighbor/transitive, label-specific/wildcard, forward/backward navigations, and showed that there does not exist an ordering scheme which is optimal for all these operations, but we show a couple of schemes that are optimal for some subset of them, and most importantly, we showed that in one of the proposed ordering scheme, i.e., $<_t^{l*}$, nodes to retrieve in any of those navigations are clustered in a small constant-bounded number of regions on the disk. We also showed that the proposed ordering schemes are compatible with variations of well-known solutions for related and important problems. Future work includes the discussion on the evaluation of other types of operations, e.g., how we combine our scheme and existing join-based path query evaluation algorithms in order to support multi-step queries.

REFERENCES

[1] R. Agrawal and J. Kiernan. An access structure for generalized transitive closure queries. In *Proc. of ICDE*, pages 429–438, 1993.

[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 *Proc. of ICDE*, page 141, 2002.

[3] J. Banerjee, W. Kim, S.-J. Kim, and J. F. Garza. Clustering a DAG for CAD databases. *IEEE TOSE*, 14(11):1684–1699, 1988.

[4] M. Bhadkamkar, F. Farfán, V. Hristidis, and R. Rangaswami. Storing semi-structured data on disk drives. *ACM TOS*, 5(2), 2009.

[5] R. Bordawekar and O. Shmueli. An algorithm for partitioning trees augmented with sibling edges. *Inf. Process. Lett.*, 108(3):136–142, 2008.

[6] P. Buneman, S. B. Davidson, K. Hart, G. C. Overton, and L. Wong. A data transformation system for biological data sources. In *Proc. of VLDB*, pages 158–169, 1995.

[7] J. Cheng, G. Yu, G. Wang, and J. X. Yu. PathGuide: An efficient clustering based indexing method for XML path expressions. In *Proc. of DASFAA*, pages 257–264, 2003.

[8] I.-H. Choi, B. Moon, and H.-J. Kim. A clustering method based on path similarities of xml data. *Data Knowl. Eng.*, 60(2):361–376, 2007.

[9] V. Christophides, D. Plexousakis, M. Scholl, and S. Tourtounis. On labeling schemes for the semantic web. In *Proc. of WWW Conf.*, pages 544–555, 2003.

[10] C.-W. Chung, J.-K. Min, and K. Shim. Apex: an adaptive path index for XML data. In *Proc. of SIGMOD*, pages 121–132, 2002.

[11] E. Demir, C. Aykanat, and B. B. Cambazoglu. A link-based storage scheme for efficient aggregate query processing on clustered road networks. *Inf. Syst.*, 35(1):75–93, 2010.

[12] Y. Diao, P. M. Fischer, M. J. Franklin, and R. To. YFilter: Efficient and scalable filtering of xml documents. In *Proc. of ICDE*, page 341, 2002.

[13] DTM. `http://xml.apache.org/xalan-j/dtm.html`.

[14] R. Edwards and S. Hope. Persistent DOM: An architecture for XML repositories in relational databases. In *Proc. of IDEAL*, pages 416–421, 2000.

[15] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Compressing and searching XML data via two zips. In *Proc. of WWW Conf.*, pages 751–760, 2006.

[16] R. F. Geary, R. Raman, and V. Raman. Succinct ordinal trees with level-ancestor queries. *ACM TALG*, 2(4):510–534, 2006.

[17] R. Goldman and J. Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *Proc. of VLDB*, pages 436–445, 1997.

[18] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[19] T. Grust, J. Rittinger, and J. Teubner. Why off-the-shelf RDBMSs are better at XPath than you might expect. In *Proc. of SIGMOD*, pages 949–958, 2007.

[20] T. Grust, M. van Keulen, and J. Teubner. Staircase join: Teach a relational DBMS to watch its (axis) steps. In *Proc. of VLDB*, pages 524–525, 2003.

[21] W.-S. Han, K.-Y. Whang, and Y.-S. Moon. A formal framework for prefetching based on the type-level access pattern in object-relational DBMSs. *IEEE TKDE*, 17(10):1436–1448, 2005.

[22] Z. He and A. Marquez. Path and cache conscious prefetching (PCCP). *VLDB J.*, 16(2):235–249, 2007.

[23] K. A. Hua, J. X. W. Su, and C. M. Hua. Efficient evaluation of traversal recursive queries using connectivity index. In *Proc. of ICDE*, pages 549–558, 1993.

[24] Y. E. Ioannidis, R. Ramakrishnan, and L. Winger. Transitive closure algorithms based on graph traversal. *ACM TODS*, 18(3):512–576, 1993.

[25] J. Jansson, K. Sadakane, and W.-K. Sung. Ultra-succinct representation of ordered trees. In *Proc. of SODA*, pages 575–584, 2007.

[26] C.-C. Kanne and G. Moerkotte. A linear time algorithm for optimal tree sibling partitioning and approximation algorithms in Natix. In *Proc. of VLDB*, pages 91–102, 2006.

[27] P.-A. Larson and V. Deshpande. A file structure supporting traversal recursion. In *Proc. of ACM SIGMOD*, pages 243–252, 1989.

[28] J. A. Lukes. Efficient algorithm for the partitioning of trees. *IBM Journal of Research and Development*, 18(3):217–224, 1974.

[29] T. Milo and D. Suciu. Index structures for path expressions. In *Proc. of ICDT*, pages 277–295, 1999.

[30] A. Morishima, K. Tajima, and M. Tadaishi. Optimal node ordering schemes for set-based navigations in trees and graphs (full version). Technical report.

[31] Z. M. Özsoyoğlu, G. Özsoyoglu, and J. Nadeau. Genomic pathways database and biological data management. *Animal Genetics*, 37(Suppl. 1):41–47, Aug. 2006.

[32] S. Prakash, S. S. Bhowmick, and S. K. Madria. SUCXENT: An efficient path-based approach to store and query XML documents. In *Proc. of DEXA*, pages 285–295, 2004.

[33] U. Rost and E. Bornberg-Bauer. TreeWiz: interactive exploration of huge trees. *Bioinformatics*, 18(1):109–114, Jan. 2002.

[34] M. Schkolnick. A clustering algorithm for hierarchical structures. *ACM TODS*, 2(1):27–44, 1977.

[35] S. Shekhar and D.-R. Liu. CCAM: A connectivity-clustered access method for networks and network computations. *IEEE TKDE*, 9(1):102–119, 1997.

[36] M. M. Tsangaris and J. F. Naughton. On the performance of object clustering techniques. In *Prof. of SIGMOD*, pages 144–153, 1992.

[37] H. Wang, H. He, J. Yang, P. S. Yu, and J. X. Yu. Dual labeling: Answering graph reachability queries in constant time. In *Proc. of ICDE*, page 75, 2006.

[38] H. Wang, S. Park, W. Fan, and P. S. Yu. Vist: a dynamic index method for querying XML data by tree structures. In *Proc. of SIGMOD*, pages 110–121, 2003.

[39] 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 *Proc. of VLDB*, pages 145–156, 2005.

[40] R. K. Wong, F. Lam, and W. M. Shui. Querying and maintaining a compact XML storage. In *Proc. of WWW Conf.*, pages 1073–1082, 2007.

[41] M. Yoshikawa, T. Amagasa, T. Shimura, and S. Uemura. XRel: a path-based approach to storage and retrieval of XML documents using relational databases. *ACM TOIT*, 1(1):110–141, 2001.

[42] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman. On supporting containment queries in relational database management systems. In *Proc. of SIGMOD*, pages 425–436, 2001.

[43] N. Zhang, V. Kacholia, and M. T. Özsu. A succinct physical storage scheme for efficient evaluation of path queries in xml. In *Proc. of ICDE*, pages 54–65, 2004.