Florida International University Technical Report **TR-2006-09-01**

Efficient Native Storage Systems for Semi-structured Data

Medha Bhadkamkar **Fernando Farfan** **Vagelis Hristidis** **Raju Rangaswami**

School of Computer Science
Florida International University
Miami, FL 33199

{medha,ffarfan,vagelis,raju}@cs.fiu.edu

Efficient Native Storage for Semi-structured Data

Medha Bhadkamkar Fernando Farfan Vagelis Hristidis Raju Rangaswami
School of Computing and Information Sciences,
Florida International University,
Miami FL 33199, USA.
{medha,ffarfan,vagelis,raju}@cs.fiu.edu

Abstract

Semi-structured data is becoming commonplace with examples such as XML, Bioinformatics suffix-trees, scientific computing data, and even generic directory-file hierarchies. Such semi-structured data must be stored on mass storage devices for persistence as well as cost-efficiency. Current approaches, which map semi-structured data to relational databases or simply use flat files, incur a mismatch between the structure of the data and the underlying storage device (disk drive). In this paper, we explore alternate native strategies for storing semi-structured data that match its access characteristics to those of disk drives, using XML data as a concrete case study. In particular, we present algorithms that, given semi-structured data and a disk drive, decide how to store the data on the drive in a way that will later allow efficient navigation and retrieval. We evaluate our proposed methods using the DiskSim disk simulator and benchmark XPath queries. The experimental results indicate savings of as much as 7X-34X in query execution time for an important class of navigational queries (which we call non-deep-focused class), compared to the baseline sequential layout of the XML data.

1 Introduction

In recent years, there has been a move towards storing more and more data in semi-structured form. The popularity of semi-structured data, proved by the success of XML, is due to the fact that tree- or graph- structures lend themselves naturally to organizing and visualizing large amounts of information. In addition to general-purpose XML databases, examples of applications dealing with large amounts of semi-structured data include BioInformatics suffix-tree alignments [13], scientific data grids [36], multi-resolution video [16], as well as directory-file hierarchies in general-purpose filesystems [32]. Although, there have been several recent efforts directed towards developing semi-structured data processing systems (e.g., Galax [5], Timber [26], XALAN [1], Natix [33] and XT [2]) and evaluating queries on semi-structured data (e.g., [7, 22]), the problem of efficiently storing semi-structured data on mass storage devices remains largely unexplored (Natix is an exception and we discuss later how we leverage and extend its native storage strategies).

To store and access semi-structured data, current approaches map them to tables (e.g, [40, 14]) or object-based storage abstractions [12] or simply use flat files. These approaches, however, ignore the specific characteristics of semi-structured data as well as those of disk drives. In particular, semi-structured data has *tree* (or *graph*) structure, whereas relational databases are structured tables and flat files are unstructured. On the other hand, disk drives store information in circular tracks that are accessed with mechanical seek and rotational delay overheads. As a result, such solutions result in sub-optimal accesses to semi-structured data. Given the abundance of data in semi-structured form today, there is an immediate need for re-examining the existing storage and access machinery for semi-structured data.

In this paper, we explore strategies to optimize the storage (placement) and retrieval of semi-structured data on disk drives by explicitly accounting for the mismatch between the structure of the data and the disk drive characteristics. Throughout the paper, we use XML as a concrete case of semi-structured data. In particular, we present algorithms that given the physical characteristics of a disk drive (number of tracks, rotational speed, seek time, etc.), place semi-structured data on the disk drive in a way that facilitates efficient navigation of the data by reducing access overheads. The proposed technique first addresses the problem of grouping nodes of semi-structured data so that they can be mapped to disk blocks. We develop and experimentally evaluate different grouping strategies, including strategies developed in previous work [29]. Second, our on-disk placement strategy for node groups optimizes common navigation operations (parent-to-child and node-to-next-sibling) on tree-structured data. In the case of XML, this in turn leads to efficient execution of XML queries on the data. For on-disk placement, we use the semi-sequential disk access technique proposed recently by Schindler et. al. [37]. We show that a naive usage of semi-sequential access, however, can lead to large seek times and unacceptable fragmentation of disk space. We propose an optimized strategy which reduces such overheads drastically.

The baseline storage strategy that we compare our approach against is sequential layout of semi-structured tree nodes in depth-first order, referred to as *default* storage strategy henceforth. To evaluate our work on optimizing navigation of XML data, we consider XPath [4] queries, which are the core navigation component of XQuery [3].

Paper contributions:

1. We explore native strategies to place semi-structured data on a disk drive, which closely match the characteristics of the data to that of the drive.

2. We study techniques for grouping tree nodes into supernodes which balance fragmentation and preservation of the tree-structure within the supernodes.

3. We experimentally evaluate different placement strategies (with various grouping techniques), for XML benchmark documents and queries [39, 17] using an instrumented DiskSim [11] disk simulator.

The rest of the paper is organized as follows. Section 2 presents the system framework including its architecture and the model used for semi-structured data and their access. In Section 3, we present a native data-layout strategies for semi-structured data. In Section 4, we present strategies for organizing and grouping nodes in semi-structured tree data so that they can be mapped to disk blocks. In Section 5, we evaluate the proposed approach by comparing it against the default sequential layout. We survey related work in Section 6 and conclude in Section 7.

2 System Framework and Data Model

Storage Stack Figure 1(a) shows the current storage stack. The proposed storage stack (Figure 1(b)) builds a native Semi-Structured Storage (SSS) engine on top of the block I/O interface to provide native storage and access support for semi-structured data. The SSS engine employs disk profiling to perform low-level data layout on the disk drive [15]. Storage access modules (Filesystem, DB Engine, etc.) need to be minimally modified to use the SSS interface in order to efficiently store and retrieve semi-structured data, or bypass it for non-semi-structured data.

Semi-structured data We view a semi-structured data object as a labeled tree T, where each node v has a *label* $\lambda(v)$, which is a *tag* name for non-leaf nodes and a *value* for leaf nodes. Also, non-leaf nodes v have an optional set $A(v)$ of attributes, where each attribute $a \in A(v)$ has a name and a value. Note that our placement technique can also be applied to documents with cycles (e.g., ID-IDREF edges in XML); however, the navigation on such edges has not been optimized. Figure 2 shows an example of an XML document.

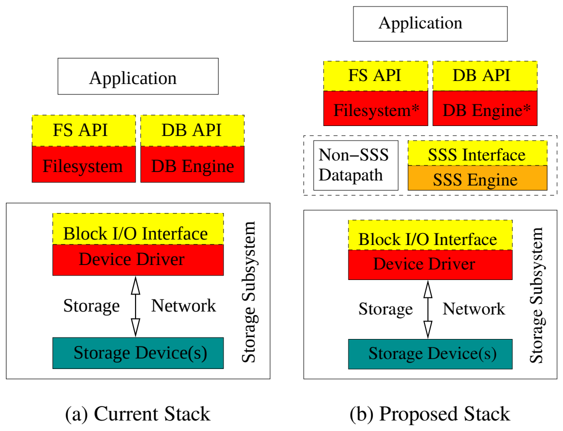


Figure 1. Storage stack modifcation.

We assume that each node has a pointer to its first child and its right sibling. This assumption is intended to make the comparison of our storage method to the default storage method more fair, allowing the default storage method to avoid reading the entire subtree of a node to access its right sibling.

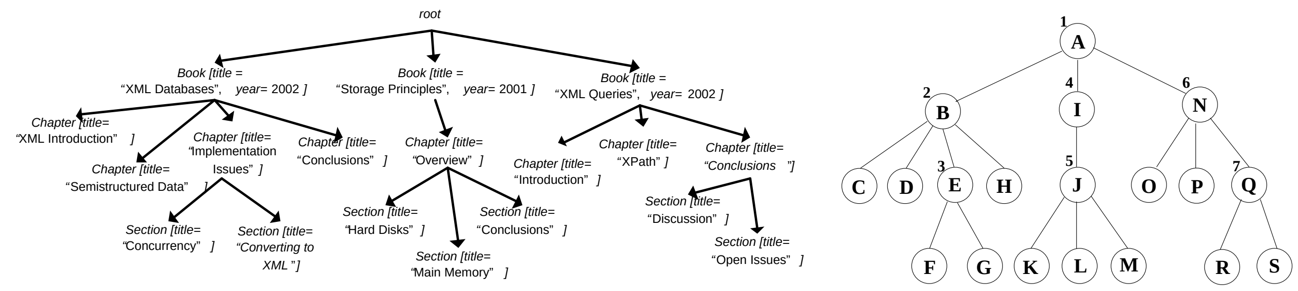
Indexes There is a large body of work on indexing semi-structured and XML data (e.g., [20, 23]). Our work focuses on exploring the effects of the data placement and therefore we only consider no-index query execution plans to make the comparison clear. The interplay of XML indexes with our placement method is an important issue which we leave as future work (see Section 7).

XML queries We adopt the “standard” XPath evaluation strategy [22] shown in Figure 3. Intuitively, this strategy processes an XPath query Q in a depth-first manner on the XML document, one step of Q ($Q.first$) at a time, and stores the intermediate results in a set S. In [10] we explain how optimizing XPath also leads to optimized XQuery.

Updates We do not directly tackle the problem of updates in this work. However, the techniques of [27] can be applied, where they focus on maintaining the supernode tree in the presence of updates.

3 XML Tree Storage

In this section, we present our on-disk data placement strategies for XML data. These strategies are generic and can be applied to work with other non-XML, semi-


root
Book [title = "XML Databases", year= 2002]
Book [title = "Storage Principles", year= 2001]
Book [title = "XML Queries", year= 2002]
Chapter [title= "XML Introduction"]
Chapter [title= "Semistructured Data"]
Chapter [title= "Implementation Issues"]
Chapter [title= "Conclusions"]
Chapter [title= "Overview"]
Chapter [title= "Introduction"]
Chapter [title= "XPath"]
Chapter [title= "Conclusions "]
Section [title= "Concurrency"]
Section [title= "Converting to XML "]
Section [title= "Hard Disks"]
Section [title= "Main Memory"]
Section [title= "Conclusions"]
Section [title= "Discussion"]
Section [title= "Open Issues"]
1 A
2 B
4 I
6 N
3 E
5 J
7 Q
C D H O P F G K L M R S

(a) Sample XML Document

(b) XML Tree

Figure 2. A Sample XML document and corresponding tree.

```
procedure process-location-step(n0, Q) {
 /* n0 is the context node;
    query Q is a list of location steps */
 node set S := apply Q.first to node n0;
 if (Q.tail is not empty) then
  for each node n in S do
   process-location-step(n, Q.tail);
}
```

Figure 3. Standard XPath evaluation strategy.

structured data. We first introduce a basic tree-structured placement strategy which illustrates our approach. We then present an improved and optimized variant of the basic strategy, which addresses the shortcomings of the basic strategy. Finally, we address several practical issues that must be considered when implementing the proposed placement strategies.

3.1 Basic Tree-structured Placement

The limitation of the default storage method is that it is optimized only for accessing the XML tree in depth-first order. For example, for the XML tree in Figure 2(b) (created by replacing the labels with node ids in the XML tree of Figure 2(a)), the nodes would be stored sequentially in alphabetical order. If the XML file is accessed in strictly depth-first order, such a placement scheme would be optimal.

However, the navigation of the XML tree in answering XPath (or XQuery) queries displays the following characteristics: (a) nodes are accessed along any path from the root to a leaf of the XML tree, and (b) siblings are often accessed together, without accessing their descendants. The default layout of the nodes would translate to random accesses (and therefore poor IO performance) for both the above accesses, except for the leftmost path or traversals along leaf levels.

Based on the above observations, we design our basic XML layout strategy, *tree-structured placement*. To simplify the presentation of the algorithm we assume that each XML node occupies an entire disk block. This assumption is removed in Section 4 where we discuss in detail the grouping methods we employ to minimize internal fragmentation within disk blocks while maintaining the tree structure of the XML file.

In the basic tree-structured placement, nodes are placed on the disk starting from the outermost available track (we choose the outermost track due to its higher bandwidth, favoring the more frequently accessed higher levels of the tree). In particular, we first place the root node v on the block with the smallest logical-block-number (LBN), on the outermost available track of the disk. Second, we place its children sequentially on the next *free* track such that accessing the first child u of v after accessing v results in a *semi-sequential access*. This is accomplished by choosing a block for u rotationally skewed from v such that when accessing u after accessing v, the rotational delay incurred is zero. Further, accessing a non-first child from a parent node involves a semi-sequential access to reach the first child and a short rotational-delay based on the child index. The children of the first-child of the root node are then placed on the next available track, once again in a rotationally-optimal fashion relative to their parent. Next, the grandchildren of the first child of the root are placed following a similar approach, and so on. This order of placement corresponds to a depth-first ordering (DFO). Although there are alternate strategies to determine the order of placing the nodes, we use DFO due to its drastically shorter average semi-sequential access times. This is explained by the localization of the numberings for each subtree in DFO. For the XML tree in Figure 2(b), the numbers above each node (ignoring the leaf nodes) illustrate the DFO numbering.

Example 3.1 *Figure 4 shows the placement of the XML tree of Figure 2(b) on a disk platter. To simplify presentation, we assume that the disk has a single platter with a single surface (and consequently a single disk head). Furthermore, we assume that the rotational skew between tracks is the seek-distance × quarter-rotation. The asterisked blocks in each track immediately before the fi rst-child represent the rotational skew between a parent and its fi rst-child.*

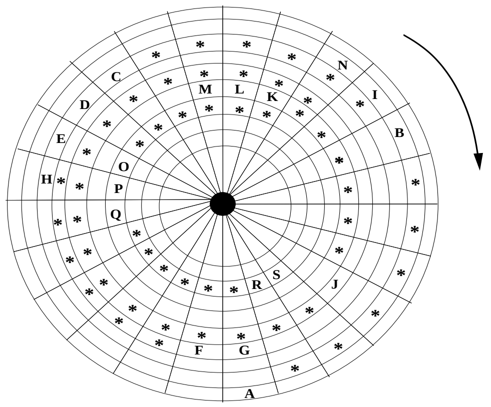


Figure 4. Basic tree-structured strategy.

Figure 5 outlines the algorithm for tree-structured placement. Notice that the leaf nodes of T are not numbered in the ordering and hence are not returned by `getNextNode()`, which is when the placement algorithm terminates.

Auxiliary Methods:

```
Node getNextNode()
/* returns one node at a time in ascending order */
Track getFirstFreeTrack() /* smallest free track */
place(LBN lbnFirst,NodeList L)
/* place children nodes L starting from lbnFirst */
LBN findSemiSequential(LBN parent, int t)
/* returns the LBN n on track t such that
access to t from parent is semi-sequential. */
```

Tree-structured Placement Algorithm:

```
1. placeInTrack(getFirstFreeTrack(),0,root(tree));
2. while (more nodes) {
3.   n = getNextNode();
4.   t = getFirstFreeTrack();
5.   L = empty;
6.   L -> add(children(n));
7.   lbnFirstChild = findSemiSequential(n.lbn, t);
8.   place(lbnFirstChild,L); }
```

Figure 5. Tree-structured placement algorithm.

The following example illustrates the execution of an XPath query when the XML document has been placed using the tree-structured placement algorithm.

Example 3.2 *Consider the XPath query* $root/Book[title = \text{“}XML\ Queries\text{”}]/Chapter[title = \text{“}XPath\text{”}]$ *on the XML document of Figures 2(a) and 2(b). The following table shows the sequence of node accesses to answer this query using the evaluation strategy of Figure 3 and the types of disk accesses they correspond to for the default and the tree-structured placement. Notice that the tree-structured placement incurs a semi-sequential access instead of sequential access in two cases, but this is outweighed by a sequential instead of random access in two other cases.*

nodes	*A*	*B*	*I*	*N*	*O*	*P*	*Q*
default	*rand*	*seq*	*rand*	*rand*	*seq*	*seq*	*seq*
tree	*rand*	*semis*	*seq*	*seq*	*semis*	*seq*	*seq*

Discussion The parent-to-first-child and node-to-next-sibling operations optimized by our placement are critical for the following reasons. In theory, these two navigation operations have been shown to be sufficient for answering any XQuery query [21]. In practice, the two main approaches for native XQuery evaluation are navigation-based (e.g., Galax [5]), and native algebraic-based (Timber [26], Niagara [34], Tamino [8], X-Hive [6]). Galax evaluates XQuery queries using optimized XML Core, which is based on nested for-loops, which translate to our two operations. Likewise, the algebraic operators (e.g., select in TLC algebra of Timber) themselves are typically implemented using the two proposed operations.

The basic XML layout strategy, as is obvious in Figure 4, results in severe external fragmentation of disk space (internal fragmentation within a disk block is discussed in Section 4), which, among other things, increases the average seek time. Next, we present an optimization of the basic tree-structured layout strategy that reduces external fragmentation as well as random seek times drastically.

3.2 Optimized Tree-structured Placement

The key idea in the *optimized tree-structured placement* is the use of *non-free tracks* for placing the children for a given parent node. The optimized placement strategy allows further flexibility by not requiring the first child to be placed at the exact rotationally-optimal block, but rather allows placing the first-child anywhere within a *rotationally-optimal track-region* (defined below).

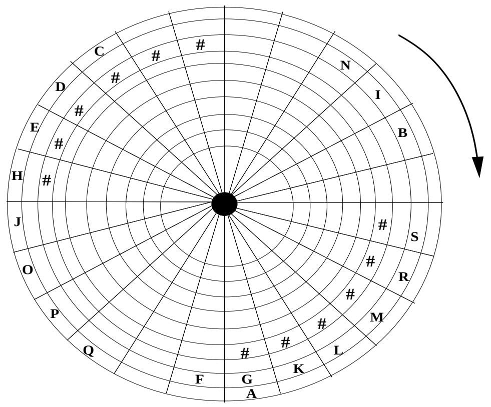


Figure 6. Optimized Strategy.

The optimized placement strategy is less restrictive than the basic tree-structured placement strategy in two ways: (1) it allows placing children on a *non-free track*, and (2) it does not require the first-child to be placed at the rotationally-optimal block, but rather allows placing the first-child anywhere within a rotationally-optimal track-region as defined next.

We define a *track-region* as a contiguous list of N_{tr} disk-blocks along a track. The blocks within a track-region, therefore, are also sequential in the logical address space (LBN space) of the disk. Given a parent node u and a target track t, we define the *rotationally-optimal track-region* for u on track t as the track-region of size N_{tr} blocks starting from the block where the disk head lands when seeking to track t starting from u. In Figure 6, two rotationally-optimal track-regions (N_{tr}=6) for parent node 'S' are marked using the # symbol. To place the children nodes for u, a set of *candidate* rotationally-optimal track-regions are chosen close to u, which can lie in either side of the parent track. The optimized placement algorithm chooses the track-region closest to u with sufficient free space to house the children of u. Other than this variation, the optimized placement algorithm proceeds to place the XML tree similar to the basic placement algorithm.

In the above placement description, the choice of the track-region size (N_{tr}) is a critical factor. Increasing the track-region size gives the placement algorithm more opportunity to reduce fragmentation and consequently reduce random-seek overhead between node accesses, but it also increases the average rotational-delay incurred during parent-to-child node-traversals. This is an important tradeoff to be considered when choosing N_{tr}. In our experiments, we choose N_{tr} as a quarter of the track-size, which worked well for the benchmark XML documents.

Figure 6 shows the placement of the XML tree of Figure 2(b) on a hard disk (platter) using the optimized strategy. Again, we assume that the platter rotates in the clockwise direction. The assumptions of track skew are also the same as for the basic strategy. In the optimized placement, since track-regions can be filled with children of various nodes, the external fragmentation (measured in Section 5) is drastically reduced compared to the basic tree-structured placement. [10] elaborates on the details of this approach further.

3.3 Practical System Implementation Issues

In implementing the strategies presented above, several practical issues must be considered. First, the above placement scheme assumes that a single, contiguous partition, large enough to accomodate the XML data is available. This assumption is realistic since database systems typically allocate a large continous disk partition.

Second, after a tree node is read from the disk drive, a non-negligible CPU think time is typically required before the next IO request is issued. We address this isssue as follows. If the next request is for a sibling node (stored sequentially in our approach), then on-disk prefetching mechanisms ensure that this node is prefetched into the on-disk cache. However, if the next request is for a child node (stored semi-sequentially), then during computation time, the disk would have already rotated by an amount proportional to the CPU thinktime and hence no semi-sequential access would be possible. To address this, we skew the first child by an additional rotational delay, slightly more than the average CPU think time. This ensures that in most cases, the semi-sequential nature of child node accesses will be preserved.

Third, the proposed strategy would work well when processing a single query at a time. However, if there are multiple queries issued concurrently by different processes or users, then the resulting interleaving IOs are likely to degrade sequential or semi-sequential accesses to random ones. This problem is not specific to XML databases and is equally prominent in traditional relational database and filesystem acceses. Techniques at the disk scheduling layer such as *anticipatory scheduling* [25], which group together requests from a single process and minimize the effects of multiple interleaved IO request streams, address this issue well. In the rest of the paper we assume a single query is submitted at a time, and leave the problem of placement for multi-query evaluation as future work.

4 Supernode Trees

In this section, we first lay the foundation for grouping nodes in an XML tree T to form *supernodes*. Each supernode occupies an entire disk block. Next, we describe how to organize the supernodes into a supernode tree structure T_S. The placement strategies of Section 3 are then applied on the supernode tree instead of the node tree.

Grouping Nodes into Supernodes The grouping of the XML tree nodes to form supernodes determines the internal fragmentation of disk blocks. In principle, we would like to include as many nodes as possible in a supernode. Furthermore, it is desirable to group adjacent nodes of T in the same supernode, so that navigating among these nodes requires only one disk access. We consider three grouping approaches (details can be found in [10]): (a) the *sequential grouping*, where nodes are added to a supernode in a depth-first manner until the supernode is full (used in the default placement strategy), (b) the *tree-preserving grouping*, which differs from the sequential one in that no cycles of supernodes are allowed by enforcing an additional constraint when adding nodes to a group so that cycles do not exist in the resulting supernode graph (this increases internal fragmentation but preserves the tree-structure), and (c) the *Enhanced Kundu Misra (EKM) grouping* [29], where the tree is first converetd to a binary tree followed by a bottom-up grouping strategy.

Building Supernode Trees The organization of the supernodes into a supernode tree, T_S, is critical because this tree determines the placement of the supernodes on the disk drive according to the algorithms presented in Section 3. Hence, it is desirable to preserve the tree-structure of T in T_S. That is, if a parent-child pair of nodes in T is split to different supernodes, then it is preferable to split it to two adjacent supernodes in T_S. Based on the grouping strategies described above, we consider four supernode tree organization strategies (for details see [10]):

1. The *sequential supernode list*, which corresponds to the default placement strategy, uses sequential grouping to form supernodes. It is merely a linked-list of supernodes in the order in which the supernodes were formed.

2. The *tree-preserving supernode tree*, which corresponds to the *tree-preserving*[1] *tree-structured*[2] *placement* in Section 5, uses the tree-preserving grouping to form supernodes. The supernode tree is formed by adding edges between two supernodes S_i, S_j if there is an edge between two nodes $v_i \in S_i, v_j \in S_j$ in T. Notice that due to the nature of tree-preserving grouping no cycles can occur.

3. The *sequential supernode tree*, which corresponds to the *sequential tree-structured placement algorithm* in Section 5, uses the sequential grouping to form supernodes. Then, the supernode tree is created by adding edges between pairs of supernodes S_i, S_j if there is an edge between two nodes $v_i \in S_i, v_j \in S_j$ in T and adding the edge will not create a cycle.

4. The *EKM supernode tree* builds a tree on the EKM supernodes. Again no cycles exist due to the nature of EKM grouping.

5 Experiments

This section evaluates the suitability of our approach for placing XML data on disk drives. We used the Disksim [11] disk simulator for our evaluations, instrumenting it to provide the additional interface: `<LBN> findSemiSequential(LBN parent, int cyl, int track)` which returns an LBN X on <cyl,track> such that access to X from parent is semi-sequential. The optimized tree-structured and the default placement algorithms were implemented in C and integrated with the instrumented DiskSim code. The grouping algorithms of nodes to supernodes were implemented as a separate module.

To evaluate our algorithms, we generated XML files of various sizes using the XMark generator; each file corresponds to an XML tree. To demonstrate the strengths and weaknesses of our approach, we classified XPath queries into two categories–a subset of each class is shown in Table 1. To compute the query set of Table 1, we adopted the

[1] with respect to grouping

[2] with respect to placement algorithm

#	Query
D1	$/site/closed\_auctions/closed\_auction/annotation/$ $description/parlist/listitem/text/keyword$
D2	$/site/people/person/watches$
D3	$/site/open\_auctions/open\_auction/annotation/$ $description/text/keyword$
D4	$/site/people/person/address/country$
D5	$/site/regions/australia/item/description/text/$ $emph$
D6	$/site/people/person/*/business$
D7	$/site/closed\_auctions/closed\_auction/*/$ $description$
D8	$/site/regions/*/item/description/text$
N1	$/site/open\_auctions/open\_auction$
N2	$/site/closed\_auctions$
N3	$/site/regions/australia$
N4	$/site/closed\_auctions/closed\_auction$
N5	$/site/regions/*/item$
N6	$/site/*/australia$
N7	$/site/open\_auctions/$ $open\_auction[@id =' open\_auction0']/bidder$
N8	$/site/regions/asia/item[@id =' item4']/mailbox/$ $mail/from$

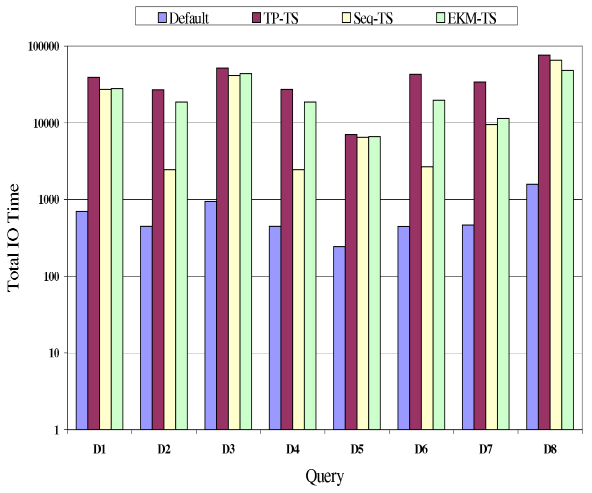
Table 1. XPath queries for the deep-focused (D) and the non deep-focused (N) classes.

performance-sensitive queries from XPathMark, but omitted the ones that check for features supported by XPath (e.g., *Q18: /comment()*).[3] This greatly reduced the number of queries. To compute reliable results we added more queries with similar properties (depth, number of conditions, selectivities).

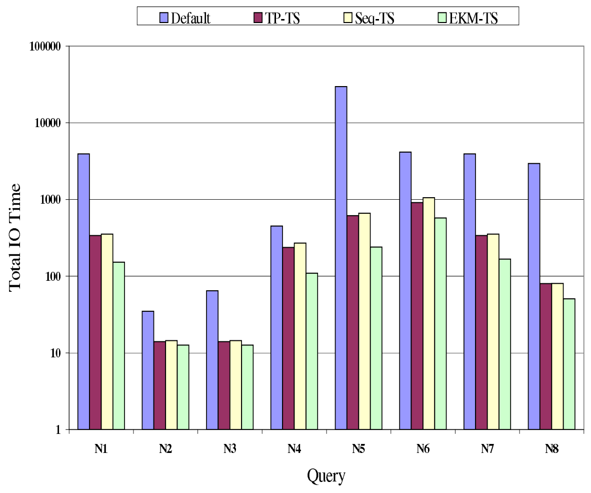
The two query classes we used are *deep-focused queries* and *non deep-focused queries*. The former class describes the special class of XPath queries that navigate entire subtrees of the XML tree (queries $D1, \ldots, D8$ in Table 1). The reason we classify based on this criterion is that the default strategy is optimized only for the special class of deep-focused queries and is sub-optimal for all other queries. Notice also that only the supernode-granularity navigation matters for overall IO performance, and not the node-granularity navigation. Hence, queries like $D2$, which do not access leaf nodes, are included in the first category since they access supernode leaves; the *watches* subtree is very small and fits in less than one supernode. The latter class of queries, non deep-focused, represents all queries that do not belong to the former class. We show that for non deep-focused queries our proposed placement methods are superior.

In the experiments below, we compare IO times for answering XML queries for four different XML layout strategies, corresponding to the supernode tree organizations of Section 4: *default*, *tree-preserving tree-structured* (TP-TS), *sequential tree-structured* (Seq-TS), and *EKM tree-structured* (EKM-TS).

[3] The latter is the main focus of XPathMark.



(a) Deep-focused queries



(b) Non-deep-focused queries

Figure 7. Total IO times in logarithmic scale for various XML placement strategies.

We take disk caching into account in all our experiments, specifically assuming that the cache is large enough to hold all nodes along the path from the root to a single leaf node. This is a reasonable assumption for XML trees, which are typically short. As a result, we do not include duplicate nodes encountered during the depth first traversal of the XML tree. It is noteworthy that such caching reduces the number of random accesses equally in all three placement strategies, since the navigation of nodes for answering a query is exactly the same regardless of the placement strategy.

Access time of placement methods Figure 7 shows (in logarithmic scale) the IO times for each query, for the two classes of queries, deep-focused (Di) and non deep-focused (Ni). This experiment considered each query for an XMark file with scaling factor $f = 0.5$ (50MB). For the deep-focused class of queries, he default placement strategy performs consistently better than the others, since it can retrieve entire subtrees more efficiently. On the other hand, for the non-deep-focused query class, the performance of the default placement strategy is consistently worse than the tree-structured variants (TP-TS, Seq-TS, and EKM-TS). For this query-class, a large number of accesses are non-sequential for the default placement, since complete subtree accesses are few. In [10], where we split the cost to the various disk access times (rotate, seek, transfer), we show that we have significantly reduced average rotational delays for the tree-structured placement strategies compared to the default strategy. Overall, the EKM-TS placement strategy performs better due to its lower internal fragmentation and tree-structure preservation property; it results in IO times which are 3X-124X better than the default strategy. Between the remaining strategies, TP-TS performs better on an average, since it better preserves the original XML tree-structure.

Fragmentation We now measure the internal and external fragmentation incurred by the grouping and placement algorithms respectively. Figure 8 shows the internal fragmentation of disk block space with the three grouping algorithms, *sequential*, *tree-preserving*, and *EKM*. As expected, the sequential grouping algorithm has little internal fragmentation as it can freely add nodes to a supernode as long as adding the next node does not violate the block-size restriction. Disk blocks are not occupied completely if the supernode space left is smaller than the size of the next XML node. The tree-preserving grouping places further restrictions on grouping for preserving the XML tree-structure in supernodes and incurs additional internal fragmentation (as much as 55%). We argue that considering the fact that current disk drives are bound more by IO access time than by IO capacity, trading capacity for improving access is acceptable.[4] The internal fragmentation with EKM is very close to that in for sequential grouping. The EKM algorithm has the flexibility that allows selecting as partition any of a node's many subtrees, thereby obtaining a more optimal result for this procedure. Our tree-preserving grouping algorithms lack this flexibility, and can only add the next node to the current supernode in an in-order fashion.

Figure 9 shows the external fragmentation results for the data placement strategies. The default strategy incurs zero external fragmentation as it places the supernode list sequentially on the disk. The tree-preserving tree-structured placement strategy (TP-TS) and the sequential tree-structured placement strategy (Seq-TS) incur external

[4]Higher internal fragmentation also offers more update flexibility; handling updates, however, is left to a future study.

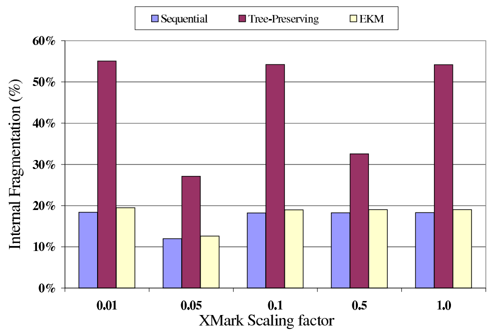


Figure 8. Internal fragmentation.

fragmentation of less than 28%, while that of the EKM-TS is higher at around 42% in some cases. However, we once again contend that these numbers are acceptable, following the arguments mentioned above. The EKM-TS placement strategy incurs the highest external fragmentation. This is because in EKM-TS, the fanout of nodes is less in the top levels (closest to root) of the tree and is higher in the lower levels than that in the other strategies. If the fanout of a tree is higher at a greater depth, it is more difficult to find contiguous free space to place all the children on the partially occupied tracks using the optimized placement strategy. Consequently the children are placed on new tracks, thereby increasing the external fragmentation.

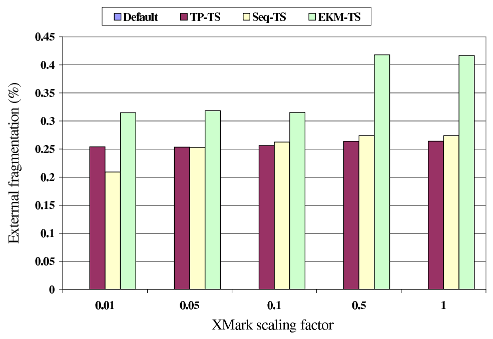


Figure 9. External fragmentation.

Sensitivity to drive characteristics To evaluate the effect of drive characteristics and performance on the XML placement strategies, we conducted a sensitivity study of IO access time for representative disk-drive models. The drive models chosen, shown in Table 2, were the Seagate Barracuda, Seagate Cheetah 9LP, Seagate Cheetah 4LP, and the HP C3323A as representative of four performance classes of disk drives: *base, fast rotating and fast seeking, fast rotating,* and *slow rotating* respectively. A disk block is of size 512 bytes. Figure 10 shows the average (across queries in a query-class) total IO times (in logarithmic scale) for the two query classes for an XMark file with $f = 0.5$ with the various hard disk models.

Again, as expected, for the special class of deep-focused queries the default placement strategy performs better than the other strategies benefiting from optimized sub-tree retrievals. However, for all other queries the tree-structured placement strategies perform better for all disk models, offering as much as 7X-34X reduction in average IO time for answering queries. This underscores the importance of native layout strategies for XML data.

6 Related Work

Storage of XML data has received attention in the last few years due to the popularity of XML. However, most work has focused on storing XML in relational DBMSs or in flat files with indexes. The former approach (e.g., [14, 40]) has been the most popular due to the success and maturity of the relational DBMSs. The latter approach (e.g., [31, 30]) is based on storing the XML document as a flat file and building separate indexes on top. Unlike our approach, these strategies do not use native layout of XML data and are limited to the generic optimization strategies built into relational databases and filesystems.

The problem of native storage of XML data has been addressed by Kanne et al. [27, 28] and System RX [9], where XML documents are stored by first splitting the XML tree into a tree of pages, where each page corresponds to a disk block. In this manner, they reduce the number of blocks read to traverse the tree. Furthermore, [27] shows how updates can be handled when XML nodes are grouped into disk pages (similarly to our grouping algorithms of Section 4), which is complementary to our work. The above studies, however, ignore the physical characteristics of operation of the disk drive and views it as just a list of pages. On the other hand, we investigate how to exploit detailed information about the disk drive and use this information to minimize overheads such as seek-time and rotational-delay.

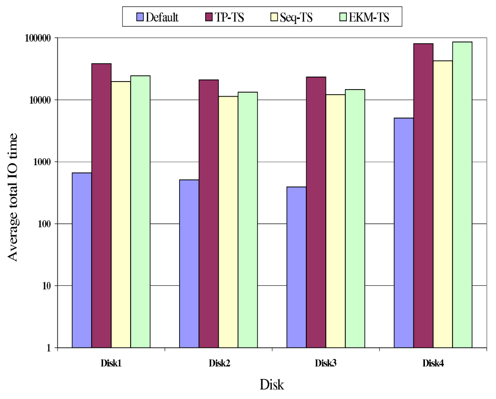
Wang et al. [42] present a technique to efficiently store Forward and Backward (F&B) Indexes [30] using clustering. As mentioned earlier, indexing is complementary to our work which focuses on the storage aspect.

Given the restrictive block IO interface, the clear case for a more expressive interface been made before [18]. Systems such as [19, 41, 24] use intelligence from upper layers of the storage stack inside storage devices to improve overall IO performance. Our work can use such systems if deployed, incorporating storage techniques for semi-structured data into disk firmware. Gray-box techniques for partially controlling layout of filesystem data [35] is an approach similar to ours, applied to sequentially accessed file data.

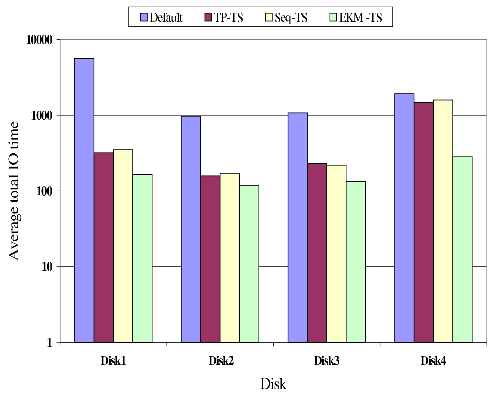
Recent work by Schlosser et al. [38] uses the idea of semi-sequential access for efficient storage of multi-dimensional data. This work is different from ours since multi-dimensional data is still structured and can afford efficient layout based on fixed attribute cardinality, not possi-

#	Disk model	Disk type	RPM	Stroke [ms]	Transfer [MBps]	Track Size [sectors]	Cylinders
1	Barracuda	Base	7200	16.679	10-15	119-186	5172
2	Cheetah 9LP	Fast disk	10045	10.627	19-28.9	167-254	6962
3	Cheetah 4LP	Fast rotate	10033	16.107	15-22.1	131-195	6581
4	HP C3323A	Slow rotate	5400	18.11	4.0-6.6	72-120	2982

Table 2. Characteristics of experimented disk drive.



(a) Deep-focused queries



(b) Non-deep-focused queries

Figure 10. Sensitivity of IO access times to changing disk drive characteristics (logarithmic scale).

ble with semi-structured data. Further, with semi-structured data, grouping of multiple data elements to be stored on a disk block is non-trivial due to the variable size of the data elements. Finally, the access patterns are different for semi-structured data than structured data, where access is along data dimensions. These issues make the problem of storing semi-structured data significantly different than that for structured multi-dimensional data.

Atropos [37] exploits the physical properties of disk drives and uses semi-sequential accesses to store relational databases. Our work targets data that has a tree structure, quite different than relational tables. Second, we show that a naive application of the semi-sequential access paradigm to tree structures leads to large seek times and severe space fragmentation. Our optimized layout strategy reduces such overhead significantly. To the best of our knowledge, there is no existing work tackling the problem of laying out tree-structured data, accounting for low-level hard drive storage and operation semantics.

7 Conclusions and Future Work

Native layout of semi-structured data has been largely unexplored, except for the area of grouping nodes to supernodes. In this paper, we have taken a first step towards on-disk placement strategies for semi-structured data that explicitly accounts for the structural mismatch between semi-structured data and disk devices. We presented data placement strategies that improve the performance of common tree navigation operations. Evaluation performed using an instrumented DiskSim simulator [11] suggests reduction in average IO times of as much as 7X-34X for answering non-deep-focused class of XPath queries. On the other hand, the default placement typically performs better for deep-focused queries.

Based on our initial findings in this study, we conclude that native data layout strategies for semi-structured data hold promise for improving application performance. We are currently investigating the impact of using indexing techniques with our placement scheme. Ideally, the indexes and the data storage should be optimized for different navigation patterns to complement each other. We also intend to study the advantages of our strategy with different caching techniques and caches of varying sizes. Finally, we plan to extend our work to multi-query and multi-user environments, and work on handling updates within the tree-structured placement framework.

Acknowledgements

We would like to thank several anonymous referees who have helped improve this paper. This work was supported in part by the National Science Foundation under grant IIS-0534530 and the Department of Energy under grant DE-FG02-06ER25739.

References

[1] Xalan-Java. *http://xml.apache.org/xalan-j*.
[2] XT. *http://www.blnz.com/xt/index.html*.
[3] Xquery 1.0. *http://www.w3.org/TR/xquery/*, 2004.

[4] XML Path Language (XPath) Version 1.0. *http://www.w3.org/TR/xpath*, 2005.

[5] Galax. *http://www.galaxquery.org*, 2006.

[6] X-Hive Corp. X-Hive/DB. *http://www.x-hive.com*, 2006.

[7] S. Abiteboul and V. Vianu. Regular Path Queries with Constraints. In *PODS*, 1997.

[8] S. AG. Tamino Developer Community QuiP, a W3C XQuery Prototype. *http://developer.softwareag.com/tamino/quip*, 2004.

[9] K. Beyer, R. J. Cochrane, V. Josifovski, J. Kleewein, G. Lapis, G. Lohman, B. Lyle, F. Ozcan, H. Pirahesh, N. Seemann, T. Truong, B. V. der Linden, B. Vickery, and C. Zhang. System rx: one part relational, one part xml. In *SIGMOD*, 2005.

[10] M. Bhadkamkar, F. Farfan, V. Hristidis, and R. Rangaswami. Efficient Native Storage for Semi-structured Data (extended paper version). In *http://www.cis.fi u.edu/SSS/NativeXMLextended.pdf*, 2006.

[11] J. Bucy, G. Ganger, and Contributors. The DiskSim Simulation Environment Version 3.0 Reference Manual. *Carnegie Mellon University Technical Report CMU-CS-03-102*, January 2003.

[12] M. Carey, D. DeWitt, M. Franklin, N. Hall, M. McAuliffe, J. Naughton, D. Schuh, M. Solomon, C. K. Tan, O. Tsatalos, S. White, and M. Zwilling. Shoring up Persistent Applications. In *ACM SIGMOD*, 1994.

[13] A. Delcher, S. Kasif, R. Fleischmann, J. Peterson, O. White, and S. Salzberg. Alignment of Whole Genomes. *Nucleic Acids Research*, 27(11):2369–2376, 1999.

[14] A. Deutsch, M. F. Fernandez, and D. Suciu. Storing Semistructured Data with STORED. *ACM SIGMOD*, 1999.

[15] Z. Dimitrijevic, R. Rangaswami, E. Chang, D. Watson, and A. Acharya. Diskbench: User-level Disk Feature Extraction Tool. *UCSB Technical Report TR-2004-18*, 2004.

[16] A. Finkelstein, C. E. Jacobs, and D. H. Salesin. Multiresolution Video. *Proceedings of SIGGRAPH*, pages 281–290, August 1996.

[17] M. Franceschet. XPathMark: An XPath Benchmark For XMark. *University of Amsterdam Technical Report PP-2004-04*, 2004.

[18] G. R. Ganger. Blurring the Line Between OSes and Storage Devices. *Carnegie Mellon University Technical Report CMU-CS-01-166*, December 2001.

[19] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A Cost-Effective, High-Bandwidth Storage Architecture. *Proceedings of the ACM ASPLOS*, October 1998.

[20] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *VLDB*, 1997.

[21] G. Gottlob and C. Koch. Monadic Datalog and the Expressive Power of Languages for Web Information Extraction. *J. ACM*, 51(1):74–113, 2004.

[22] G. Gottlob, C. Koch, and R. Pichler. Efficient Algorithms for Processing XPath Queries. In *VLDB*, 2002.

[23] T. Grust. Accelerating XPath Location Steps. In *ACM SIGMOD*, 2002.

[24] L. Huston, R. Sukthankar, R. Wickremesinghe, M. Satyanarayanan, G. R. Ganger, E. Riedel, and A. Ailamaki. Diamond: A Storage Architecture for Early Discard in Interactive Search. *Proceedings of the USENIX Conference on File and Storage Technologies*, March 2004.

[25] S. Iyer and P. Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous i/o. In *Symposium on Operating Systems Principles*, pages 117–130, 2001.

[26] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. S. Lakshmanan, A. Nierman, S. Paparizos, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. TIMBER: A Native XML Database. *The VLDB Journal*, 11(4):274–291, 2002.

[27] C. Kanne and G. Moerkotte. Efficient Storage of XML Data . *Universitaet Mannheim Technical Report*, 1999.

[28] C.-C. Kanne, M. Brantner, and G. Moerkotte. Cost-Sensitive Reordering of Navigational Primitives. *SIGMOD*, 2005.

[29] C.-C. Kanne and G. Moerkotte. A Linear Time Algorithm for Optimal Tree Sibling Partitioning and Approximation Algorithms in Natix. In *VLDB*, 2006.

[30] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth. Covering Indexes for Branching Path Queries. *SIGMOD*, 2002.

[31] Q. Li and B. Moon. Indexing and Querying XML Data for Regular Path Expressions. *VLDB Journal*, 2001.

[32] M. McKusick, W. Joy, S. Leffler, and R. Fabry. A Fast File System for UNIX*. *ACM Transactions on Computer Systems 2*, 3:181–197, August 1984.

[33] Natix. http://www.dataexmachina.de/. 2006.

[34] J. Naughton, D. DeWitt, and e. a. David Maier. The Niagara Internet Query System. *http://www.cs.wisc.edu/niagara*, 2000.

[35] J. Nugent, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Controlling your PLACE in the File System with Gray-box Techniques. *Proceedings of the USENIX Annual Technical Conference*, pages 311–324, June 2003.

[36] A. Rajasekar, M. Wan, and R. Moore. MySRB & SRB - Components of a Data Grid. *Proceedings of High Performance Distributed Computing (HPDC-11)*, July 2002.

[37] J. Schindler, S. W. Schlosser, M. Shao, A. Ailamaki, and G. R. Ganger. Atropos: A Disk Array Volume Manager for Orchestrated Use of Disks. *Proceedings of the USENIX Conference on File and Storage Technologies*, March 2004.

[38] S. W. Schlosser, J. Schindler, S. Papadomanolakis, M. Shao, A. Ailamaki, C. Faloutsos, and G. R. Ganger. On Multidimensional Data and Modern Disks. *Proceedings of the 4th USENIX Conference on File and Storage Technology*, December 2005.

[39] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management. *VLDB*, 2002.

[40] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. 1999.

[41] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-Smart Disk Systems. *Proceedings of the USENIX Symposium on File and Storage Technologies*, pages 73–88, March 2003.

[42] W. Wang, H. Wang, H. Lu, H. Jiang, X. Lin, and J. Li. Efficient Processing Of Xml Path Queries Using The Disk-Based F & B Index. *Vldb*, 2005.