

Storing Semi-structured Data on Disk Drives¹

MEDHA BHADKAMKAR, FERNANDO FARFAN, VAGELIS HRISTIDIS, and RAJU RANGASWAMI

Florida International University

Applications that manage semi-structured data are becoming increasingly commonplace. Current approaches for storing semi-structured data use existing storage machinery - they either map the data to relational databases, or use a combination of flat files and indexes. While employing these existing storage mechanisms provide readily available solutions, there is a need to more closely examine their suitability to this class of data. Particularly, retrofitting existing solutions for semi-structured data can result in a mismatch between the tree structure of the data and the access characteristics of the underlying storage device (disk drive). This study explores various possibilities in the design space of native storage solutions for semi-structured data by exploring alternative approaches that match application data access characteristics to those of the underlying disk drive. For evaluating the effectiveness of the proposed native techniques in relation to the existing solution, we experiment with XML data using the XPathMark benchmark. Extensive evaluation reveals the strengths and weaknesses of the proposed native data layout techniques. While the existing solutions work really well for *deep-focused* queries into a semi-structured document (those that result in retrieving entire subtrees), the proposed native solutions substantially outperform for the *non-deep-focused* queries, which we demonstrate are at least as important as the deep-focused. We believe that native data layout techniques offer a unique direction for improving the performance of semi-structured data stores for a variety of important workloads. However, given that the proposed native techniques require circumventing current storage stack abstractions, further investigation is warranted before they can be applied to general purpose storage systems.

Categories and Subject Descriptors: D.4.2 [OPERATING SYSTEMS]: Storage Management

General Terms: Architecture, Design, Algorithms, Performance

Additional Key Words and Phrases: Semi-structured data, Storage management, XML

1. INTRODUCTION

An increasing number of applications manage large amounts of semi-structured data. Common applications that use semi-structured data today include Bioinformatics sequence search and alignment [Delcher *et al.* 1999], genomic data analysis [Rokhsar 2007], multi-resolution video storage [Finkelstein *et al.* 1996], clinical data systems [CDA 2007], XML databases, and more [Papakonstantinou *et al.* 1995]. Given that a semi-structure such as a tree provides a more intuitive way of managing large amounts of data, the trend of storing data in such formats is likely to strengthen in the future.

Current approaches to store semi-structured data either map the data to an underlying relational database system (e.g., [Bohannon *et al.* 2002; Deutsch *et al.* 1999; Mergen and Heuser 2004; Ramanath *et al.*; Shanmugasundaram *et al.*]), use the

¹This work is sponsored in part by NSF grant IIS-0534530 and DoE grant ER25739. Author's addresses: M. Bhadkamkar, F. Farfan, V. Hristidis, and R. Rangaswami, School of Computing and Information Sciences, Florida International University, 11200 S.W. 8th Street, Miami FL 33199, email: medha@cis.fiu.edu, ffarfan@cis.fiu.edu, vagelis@cis.fiu.edu, raju@cis.fiu.edu.

abstraction provided by a general-purpose object storage manager [Carey *et al.* 1994], or use a combination of flat files and indices (e.g., XALAN [Xalan 2007], XT [XT 2007], Galax [Galax 2007], BLAST [Altschul *et al.* 1990], Timber [Jagadish *et al.* 2002] and Natix [Kanne and Moerkotte 2006]). Since these approaches retrofit existing storage mechanisms to work with semi-structured data, their scope is restricted to the underlying mechanisms, which are predominantly optimized for sequential accesses. Consequently, these approaches may result in a mismatch between the structure and navigational primitives of semi-structured data and the access characteristics of disk drives. In particular, semi-structured data have a *tree* (or *graph*) structure with tree-type operations. Relational databases, on the other hand, store structured tables that are optimized for row-based access, and flat files are unstructured, optimized for sequential access. Further complicating this mismatch, the underlying storage device, *i.e.* disk drives, store information in circular tracks that are accessed with mechanical seek and rotational overhead. Given the growing amount of semi-structured data, there is a need for re-examining the current storage and access machinery that support them.

In this paper, we explore strategies to optimize the storage 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 storage and access characteristics. In particular, we present algorithms that given the physical characteristics of a disk drive (number of tracks, sectors per track and rotational speed.), place semi-structured data on the disk drive in a way that facilitates navigation of the data by reducing access overheads. Such low-level control of data layout is made possible using information provided by standard disk profiling tools [Worthington *et al.* 1995; Talagala *et al.* 1999; Dimitrijevic *et al.* 2004].

The proposed technique first addresses the problem of grouping nodes of semi-structured data trees so that they can be mapped to disk blocks. We develop and experimentally evaluate our proposed grouping strategies and compare it with the Enhanced Kundu Misra (EKM) grouping strategy [Kanne and Moerkotte 2006]. Second, our proposed on-disk layout strategy for node groups optimizes common tree navigation operations such as parent-to-child and node-to-next-sibling traversals. Our on-disk layout strategies make use of semi-sequential disk access technique [Schindler *et al.* 2004] that allows the reduction and even elimination of rotational delay overhead during disk accesses.

Given that our approach requires circumventing the prevalent *logical block abstraction*, applying our layout strategy to a general purpose storage system is not straightforward.² Our goal in this paper is simply to expose the merits and demerits of this approach. Through experiments we show that our proposed approach is superior for a dedicated single-user storage system with standard caching and prefetching capabilities – for instance, a specialized system for analysis of biological data (suffix trees) [Bedathur and Haritsa 2006]. Based on this study, we believe that our approach provides a fresh perspective on the problem of storing semi-structured data that is worth the attention and research time of the community.

To evaluate the proposed native data layout techniques, we used XML as a case

²Prior research has made a similar argument in favor of fine-grained data layout by circumventing the logical block abstraction, for the case of tabular data [Schindler *et al.* 2004].

study. XML is becoming increasingly popular due to its ability to represent arbitrary semi-structured data. It is the de facto data representation format for many modern applications, including Geographic Information Systems Markup Language (GML) [GML 2008], Medical Markup Language (MML) [MML 2008], Health Level HL7 [HL7 2008], Clinical Document Architecture (CDA) [Dolin *et al.* 2006] used to represent Electronic Health Records (EHRs), Open Document Format (ODF) [ODS 2008; OOX 2008], and Scalable Vector Graphics (SVG) [SVG 2008] used to describe two-dimensional graphics and graphical applications. Despite the widespread use of XML, the challenge of optimizing access to XML data stores is a key challenge also identified in the latest report [Abiteboul *et al.* 2005] on the future directions on database research, published every few years by the database research community.

Table I. Query classification of popular XML benchmarks.

Benchmark	Workload	Document size	Total queries	# Non-deep-focused	# Deep-focused
TPoX	Financial app	2 - 25 KB	11	4	7
XMach-1	E-commerce app	2 - 100 KB	7	4	3
XMark	Auction Website	10MB - 10 GB	20	13	7
XPathMark	Education app	10MB - 10GB	54	20	34
XOO7	Web app	4MB - 1GB	23	4	19
XBench	Publications DB	1KB - 10 GB	17	11	6
MemBeR	Synthetic	11 MB	7	0	7
MBench	Synthetic	50MB - 50GB	37	37	0
Total			176	93	83

Recent surveys of popular XML benchmarks [Afanasiev and Marx 2006; Böhme and Rahm 2003; Nambiar *et al.* 2001] show that all queries to XML data can be classified into deep-focused and non deep-focused queries. In Table I, we summarize the key XML benchmarks available in the public domain. The Transaction Processing over XML (TPoX) benchmark [Nicola *et al.* 2007] evaluates the performance of XML stores, XML databases, indexes, etc. by generating a mix of XQueries for various financial transactions on the generated XML documents. XMach-1 [Böhme and Rahm 2001; 2003], XOO7 [Bressan *et al.*], XMark [Schmidt *et al.* 2002a] and XPathMark [Franceschet 2005] are typically used to evaluate query optimizations in XML. XMach-1 is based on an E-commerce website while XMark generates queries for an E-commerce website with information on bids, items, brokers and customers. XPathMark [Franceschet 2005] is an XPath based benchmark for XMark and generates an educational document that represents the English alphabet. The XBench [Yao *et al.* 2003] benchmark is an application oriented benchmark for XML databases. Finally, the MemBer [Afanasiev *et al.* 2005; Manolescu *et al.* 2006] and the Michigan Benchmark (MBench) [Runapongsa *et al.* 2003] are both micro-benchmarks that generate synthetic workloads wherein document structure can be finely controlled (varying their depth and fan-out) so as to be able to reproduce the access patterns of a variety of different real-world workloads.

This collection of well-accepted and standardized XML benchmarks demonstrate (*i*) that XML document sizes can be fairly large running sometimes into tens of gigabytes; this combined with the fact that XML parsers can consume as much as

5X the amount of main memory during parsing as the original size of the XML document [Nicola and John 2003] implies that secondary storage accesses must be optimized if at all possible, and (ii) that the non deep-focused queries, form at least half of the total queries suggested within these popular XML benchmarks ; this implies that optimizing accesses to the non-deep-focused query class is at least as important as optimizing for the deep-focused class. Further, in the event that a workload generates both classes of queries with similar frequency, the storage system could conceivably store data using both the traditional approach and tree-based approach with the caveat that this approach requires more consideration for write-dominant workloads that can incur an unacceptable amount of overhead for maintaining consistency.

For evaluating our native layout proposals, we employ XPath queries [XPath 2007] obtained from the XPathMark benchmark for the evaluation. We examine the relative performance of native layout against the *default* approach, which stores XML files sequentially. To do so, we augmented an existing XML parsing engine to implement the grouping techniques that we propose. To evaluate disk I/O performance, we use an instrumented DiskSim disk simulator [Bucy et al. 2003] and replayed the block access traces generated by XML query processing engines. Our evaluation also addresses I/O performance in the presence of query parallelism as would be typical for server environments. Summarizing, these experiments reveal that while the default sequential layout provides superior performance for the *deep-focused* class of XML queries (or access patterns retrieving entire subtrees of semi-structured data), the proposed native layout techniques outperform the default for all other query access patterns.

The rest of the paper is organized as follows. Section 2 presents the architecture of a native semi-structured storage system and the model used for semi-structured data and their access. In Section 3, we present native data-layout strategies for semi-structured data on disk drives. In Section 4, we present strategies for organizing and grouping nodes in the tree so that they can be mapped to disk blocks. In Section 5 we conduct a theoretical analysis of the performance impact of data layout. In Section 6, we evaluate the proposed approach for the case of XML data by comparing it against the default sequential layout. We survey related work in Section 7. We conclude and discuss future directions in Section 8.

2. SYSTEM ARCHITECTURE AND DATA MODEL

In this section, we propose an architecture for building a native semi-structured storage system which allows the use of our layout techniques with minimal changes to the current storage stack. We also present the semi-structured data and access model abstractions.

2.1 Modifying the Storage Stack

Modern disk drives provide a high-level logical block abstraction to the operating system, which does not export information about the physical data layout, performance characteristics, and internal operation of the disk drive. We propose a modified storage stack inside the operating system that will facilitate native data layout strategies by including mechanisms to effect low-level data layout.

The lowest levels of the current storage stack (shown in Figure 1(a)) form the

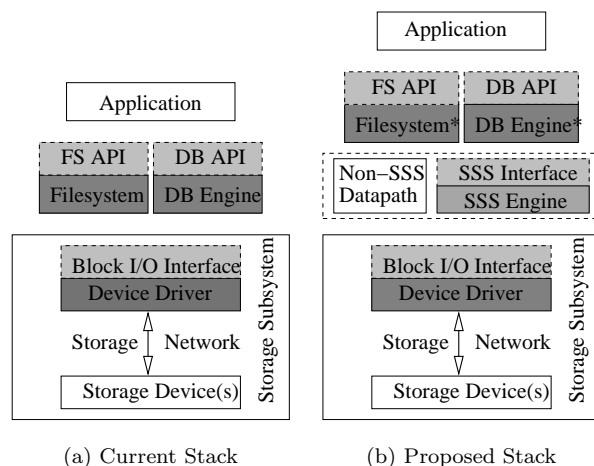


Fig. 1. Storage stack modification.

storage subsystem, which exports a logical block I/O interface. The dominant storage mechanisms, i.e., databases and file systems, form the middle layer that accesses data on the storage device(s) using the logical block interface while also providing high-level APIs for applications. These storage mechanisms are optimized for relational data and sequential files respectively.

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 native data layout on a reserved contiguous area (partition) of the disk drive. Storage access modules (File system, 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. We chose not to build-in native support into an existing file system or existing DBMS, because we believe that the SSS engine as well as its interface can be made generic enough to work with any storage access module. Existing file and database systems can then be extended with native layout support for semi-structured data via the SSS engine. While the proposed approach call for significant changes to the operating system storage management, it is important to point out that applications retain their original interface to the operating system and remain transparent to the underlying mechanisms.

2.2 Data and Access Model

We view a semi-structured document 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 layout technique can also be applied to documents with cycles (e.g., ID-IDREF edges for XML documents); however, the navigation on such edges has not been optimized.

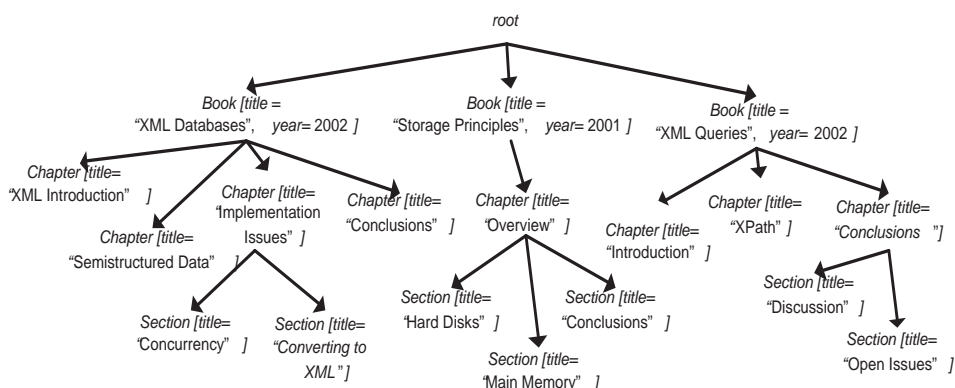


Fig. 2. A sample semi-structured document.

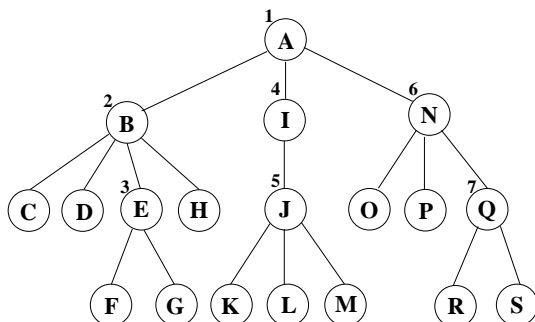


Fig. 3. Tree structure for the XML document in Figure 2.

Figure 2 shows an example of a semi-structured document (in this case an XML document) and Figure 3 shows the corresponding tree structure.

In the default layout strategy as is employed by current day file systems, the semi-structured data (say an XML document) is stored sequentially on the disk, which is equivalent to placing the tree in depth-first order. To ensure a fair comparison of our storage method to the default layout, a physical pointer is added from each node to its first child and its right sibling, thereby allowing to avoid reading the entire subtree of a node to access its right sibling. This optimization is used for the default strategy in all the experimental results we report.

For XML data, which we use as a case-study for evaluating our approach, XPath queries form the core navigation component of XML query processing systems. For evaluating XPath queries, we adopt the “standard” XPath evaluation strategy [Gottlob *et al.* 2002] shown in Figure 1. 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 [Bhadkamkar *et al.* 2006] we explain how optimizing XPath also leads to optimized XQuery.

Current implementations of XML parsers create an in-memory document tree structure that is populated (on-demand in some implementations [Noga *et al.* 2002]) by retrieving corresponding sections of the disk-resident XML document. XML

stores typically handle documents that are both smaller (i.e., tens of KB) as well as much larger size (several GB). Consequently, trivial solutions such as loading the entire XML document in memory prior to parsing are not deemed practical.

Algorithm 1: Standard XPath evaluation strategy [Gottlob *et al.* 2002]

```

1: procedure process-location-step(n0,Q)
2: /* n0 is the context node;
3: query Q is a list of location steps */
4: node set S := apply Q.first to node n0;
5: if Q.tail not empty then
6:   for each node n in S do
7:     process-location-step(n, Q.tail)
8:   end for
9: end if

```

3. SEMI-STRUCTURED DATA LAYOUT

In this section, we present disk layout strategies for semi-structured data. First, we introduce a basic tree-structured placement strategy, a simple strategy which illustrates the basic ideas of our approach. Next, we present an improved and optimized variant of the basic strategy, which addresses the shortcomings of the basic strategy. Finally, we discuss some practical challenges that must be addressed when implementing the proposed placement strategies.

3.1 Basic Tree-structured Placement

A key limitation of the default storage method is that it is optimized only for accessing the semi-structured data tree in depth-first order since it places the data file sequentially on disk. For example, for the tree in Figure 3 (created by replacing the labels with node IDs in the semi-structured tree of Figure 2), the nodes would be stored sequentially in alphabetical order. We refer to this henceforth as the default layout and use it for comparison purposes in Section 6. If this file is accessed in strictly depth-first order, such a placement scheme would be optimal. However, typical tree navigation during the answering of queries displays the following characteristics: (a) nodes are accessed along any path from the root to a leaf of the tree, and (b) siblings are often accessed together, without accessing their descendants. The default layout of the nodes would result in random accesses (and therefore poor I/O 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 layout strategy, *tree-structured placement*. To simplify the presentation of the algorithm we assume that each node in the tree occupies an entire disk block. This assumption is relaxed in Section 4 where we discuss in detail the grouping methods that can be employed to minimize internal fragmentation within disk blocks while maintaining the tree structure of the 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* [Schindler *et al.* 2004]. 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 at a rotationally-optimal point relative to their parent. Next, the grandchildren of the first child of the root are placed following a similar approach, and so on.

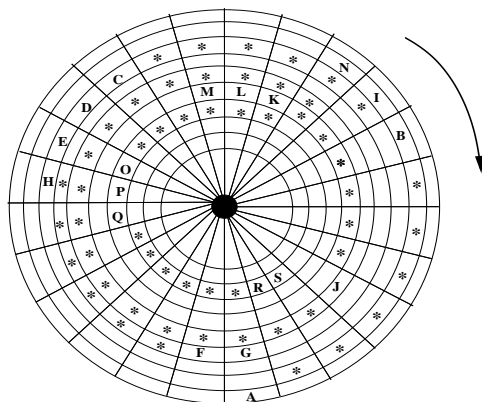


Fig. 4. **Basic tree-structured placement strategy.**

As described above, the basic tree structured layout chooses parent nodes to place their respective children in depth-first order (DFO). We also experimented with breadth-first-ordering (BFO) in choosing parents, but found DFO to consistently outperform in the experiments due to its significantly shorter seek times during parent-child traversals. Intuitively, this can be visualized in Figure 3 where we present the DFO numbering for parent nodes (above each node); notice the localization of the numbers within each subtree. The BFO ordering, on the other hand, scatters numbering over the entire tree, resulting in large seek times for parent-child traversals.

EXAMPLE 3.1. *Figure 4 shows the layout of the tree of Figure 3 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 \times quarter-rotation. The root node A is placed on the outermost track, track 0. Its first child B is placed on the first available free track closest to A , i.e., track 1. The block on which B is placed*

Algorithm 2: Basic Placement Algorithm

```

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

```

```

Require: Tree  $T$  to be placed
1: PlaceInTrack(getFirstFreeTrack(), 0, Root( $T$ ))
2: while more nodes do
3:    $n \leftarrow$  getNextNode()
4:    $t \leftarrow$  getFirstFreeTrack()
5:    $L \leftarrow$  empty
6:    $L \leftarrow$  Add(Children( $n$ ))
7:    $lbnFirstChild \leftarrow$  (findSemiSequential( $n.lbn$ ,  $T$ ))
8:   Place( $t, lbnFirstChild$ ,  $L$ )
9: end while

```

is rotationally skewed by a quarter-rotation relative to A as a consequence of our assumption. Accessing B after A would require only seeking to the next track. The remaining children of node A , i.e. I , and N , are placed sequentially next to the first child B . The asterisked blocks in each track immediately before the first-child represent the rotational skew between a parent and its first-child. The remaining nodes are placed following a similar approach to complete the placement of the tree.

Algorithm 2 outlines the procedure for tree-structured placement. Notice that the leaf nodes of the tree T shown in Figure 3 are not numbered in the ordering and hence are not returned by `getNextNode()`, which is when the placement algorithm terminates.

3.2 Optimized Tree-structured Placement

The basic 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 also increases the average seek time of I/O operations. We now describe an optimization of the basic tree-structured layout strategy that reduces external fragmentation as well as random seek times drastically.

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 is less restrictive than the basic tree-structured placement strategy in two specific 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 5, two rotationally-optimal track-regions ($N_{tr}=6$) for parent node ‘S’ are marked using the # symbol. To place the children nodes for node 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 tree-structured placement algorithm proceeds to place the tree similar to the basic placement algorithm.

In the above placement description, the choice of the rotationally-optimal 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 trade-off to be considered when choosing N_{tr} . In our experiments, we choose N_{tr} as a quarter of the track-size.

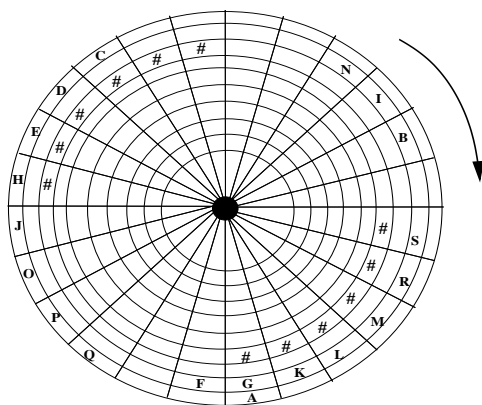


Fig. 5. **Optimized Strategy.**

Figure 5 shows the layout of the tree in Figure 3 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 a single track can contain the children of several nodes, the external fragmentation (shown in Section 6) is drastically reduced compared to the basic tree-structured placement.

The `PlaceInTrack` method in Algorithm 3 outlines the logic for optimized tree-structured placement. Line 1 places the root node of the tree T on the outermost track. Lines 2-7 place the children of the *next* node (which is the root node in the first iteration) on the rotationally-optimal track-region (returned by `FindRotTrackRegion`). The next node is returned by `getNextNode()`, which returns a non-leaf node of the XML tree based on the chosen ordering scheme. The above process is repeated until all the nodes are placed on the disk.

Algorithm 3: Optimized Placement Algorithm

```

Auxiliary Methods:
Track GetTrack(LBN)
/* returns the track for LBN */
LBN FreeTrackRegionStart(LBN, int, tracksToSkip)
/* Given a parent LBN, its number of children, and the number of tracks to skip, returns
the LBN for the first child if all children can be placed in the candidate tracks
rotationally-optimal track-region. Otherwise returns NULL. Candidate tracks are
the two tracks situated at parentTrack +/- tracksToSkip respectively. */

1: <Track,LBN> FindRotTrackRegion(LBN parent, int n)
2: tracksToSkip ← 1
3: parentTrack ← GetTrack(parent)
4: while true do
5:   if lbnFirstChild←FreeTrackRegionStart(parent,n,tracksToSkip) != NULL then
6:     return <GetTrack(lbnFirstChild),lbnFirstChild>
7:   end if
8:   tracksToSkip++
9: end while

Require: Tree  $T$  to be placed
1: PlaceInTrack(getFirstFreeTrack(),0,root(tree))
2: while more nodes do
3:   n←GetNextNode()
4:   L←empty
5:   L→add(children(n))
6:   <lbnFirstChild>←FindRotTrackRegion(n.lbn,L.size())
7:   Place(target,lbnFirstChild,L)
8: end while

```

Notice that the leaf nodes of T are not numbered in the ordering and hence are not returned by `getNextNode()`. The `findRotTrackRegion(LBN parent, int nchildren)` auxiliary method checks for availability of space in the rotationally optimal track-regions in tracks on either side of the parent's track, starting from the closest track. It returns the LBN for placing the first-child of the parent node. The remaining children are placed incrementally following the first child. The `direction` identifier specifies where the target track lies with respect to the parent. If the `direction` has a negative value, the target track is less than the parent track. Likewise, a positive value indicates that the target track is greater than the parent track.

3.3 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 accommodate the semi-structured data is available. This assumption is realistic for both file systems and database systems since they typically allocate a large contiguous disk partition and can reserve a fraction of this space for storing semi-structured data.

Second, after a tree node is read from the disk drive, a non-negligible CPU think time is typically required before the next I/O request is issued. We address this issue as follows. If the next request is for a sibling node (stored sequentially in our approach), then on-disk pre-fetching mechanisms ensure that this node is

pre-fetched 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 think time and hence no semi-sequential access would be possible. To address this, we skew the first child by an additional rotational delay equivalent to 95th percentile of a sample from the think time distribution. 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 I/Os are likely to degrade sequential or semi-sequential accesses to random ones. This problem is prominent even in traditional relational database and filesystem accesses. Techniques at the disk scheduling layer such as *anticipatory scheduling* [Iyer and Druschel 2001], which group together requests from a single process and minimize the effects of multiple interleaved I/O request streams, address this issue well. We evaluate the impact of query parallelism (in Section 6) with anticipatory I/O scheduling to demonstrate the effectiveness of native layout strategies in the simulated environment.

Finally, existing storage interfaces are restrictive which makes it non-trivial to obtain profiling information or control data layout. While the need for more expressive storage interfaces has been brought up repeatedly in the storage research community (e.g., [Ganger 2001; Keeton *et al.* 1998; Riedel *et al.* 1998]), for the time-being, we can circumvent this restriction by employing disk profiling and control tools. Profiled information includes: rotational time, seek time, track and cylinder skew times, sizes of read cache and write buffer along with pre-fetching and buffering techniques, logical to physical block mappings, and access time prediction. This profiled information enable fine-grained control for disk drives, tailored specifically for semi-structured data.

4. SUPERNODE TREES

So far, we assumed that each node in the semi-structured data tree occupies an entire disk block. This assumption, however, is not realistic; in practice, the tree nodes are of variable size, ranging from a fraction of a disk block to multiple disk blocks.

In this section, we first lay the foundation for grouping nodes in a semi-structured data tree T to form *supernodes* where 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.

4.1 Grouping Nodes into Supernodes

To reduce the internal fragmentation, it is desirable to group the maximum number of nodes into a supernode. It is also important to group adjacent nodes of T in the same supernode, so that navigating among these nodes requires only one disk access. If the size of a node is larger than the size of a disk block, it is stored using

multiple supernodes, which are then stored in consecutive disk blocks.³

To elucidate the following grouping techniques, we assume that all nodes have the same size, and one supernode can contain at most five nodes.

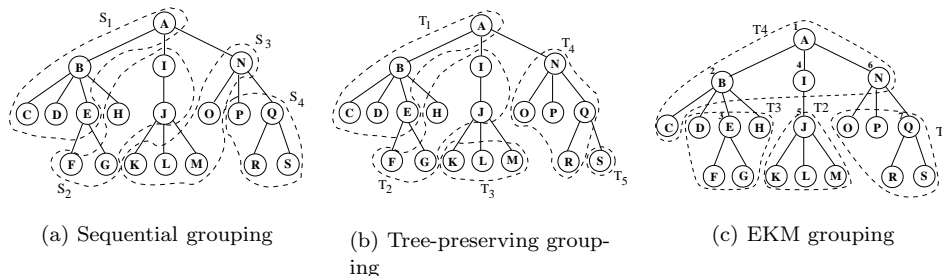


Fig. 6. Grouping strategies for creating supernodes.

Sequential grouping. Nodes are added to a supernode starting from the root node using a depth-first (and left-to-right) traversal. The only difference is that a single node is not split across disk blocks, unless the size of the node is greater than the size of a disk block. Figure 6(a) illustrates this grouping strategy for the tree presented earlier in Figure 3.

Tree-preserving grouping. The tree-preserving grouping proceeds as in the sequential grouping except it ensures that cycles of supernodes do not form in the grouped tree. At each step, before adding a node v to a supernode S , the following additional conditions are checked:

- (i) the parent node of v is in S , or
- (ii) the parent node of v is in the parent supernode of S .

If any of these conditions hold, then we add v to S . If neither holds, then by adding v to S a cycle of supernodes in the original tree T would be created. To avoid that, we close S and add v to a new supernode. This strategy aims at preserving the tree-structure of the original tree T in the supernode tree. Figure 6(b) illustrates this grouping strategy for the tree of Figure 3.

Enhanced Kundu Misra grouping. We also implement a grouping technique developed independently at the same time by Kanne and Moerkotte [Kanne and Moerkotte 2006] called the Enhanced Kundu Misra (EKM) grouping, an extension to the original Kundu-Misra grouping algorithm [Kundu and Misra 1977]. The EKM strategy operates in a bottom-up fashion and aims at reducing the number of node groups while preserving the original tree structure, thereby increasing navigations between nodes within the same group. It operates by converting the n-ary

³An alternative strategy to avoid breaking the tree-structure of the rest nodes would be to store a pointer to a Binary Large Object (BLOB) and use an object storage manager [Carey *et al.* 1994] to manage BLOBs.

tree into a binary tree representation, obtaining a layered partitioning that helps reducing the number of supernodes while preserving the connectedness. Figure 6(c) illustrates this grouping strategy for the tree of Figure 3.

4.2 Building Supernode Trees

The organization of the supernodes into a supernode tree, T_S , 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:

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. Figure 7(a) shows the formation of this list.
2. The *tree-preserving supernode tree*, which corresponds to the *tree-preserving⁴ tree-structured⁵ placement* to be introduced in Section 6, 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. Figure 7(b) shows the formation of this tree.
3. The *sequential supernode tree*, which corresponds to the *sequential tree-structured placement algorithm* in Section 6, 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. Figure 7(c) shows the formation of this tree.
4. The *EKM supernode tree* builds a tree on the EKM supernodes. Again no cycles exist due to the nature of EKM grouping. Figure 7(d) shows the formation of this tree.

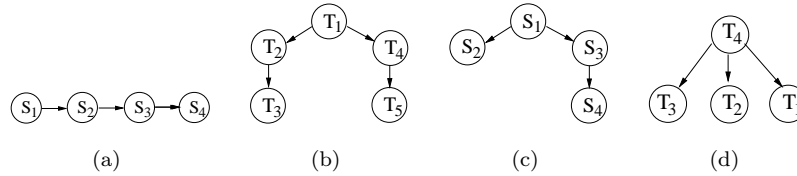


Fig. 7. **Supernode Trees:** (a) Sequential supernode list. (b) Tree-preserving supernode tree. (c) Sequential supernode tree. (d) EKM supernode tree.

⁴with respect to grouping

⁵with respect to placement algorithm

5. THEORETICAL ANALYSIS

In this section, we present a quantitative model to analyze the access times for the default and the optimized tree-structured placement strategies. Table II summarizes the description of each parameter used in this analysis.

Table II. **Parameter Description**

$T_{default}$: Average access time in default placement
T_{tree} : Average access time in tree-structured placement
t_{seq} : Average access time for sequential access
t_{rand} : Average access time for random access
$t_{semi-seq}$: Average access time for semi-sequential access
a_1 : Access is from parent to first child
a_2 : Access is from a parent node to non-first child
a_3 : Access is from a non-leaf node to its right sibling
a_4 : Access is from a leaf node to its right sibling
a_5 : All other accesses (that is, $P_5 = (1 - (\sum_{i=1}^4 P_i))$)
P_i : Probability that access a_i occurs; $1 \leq i \leq 5$
$t_{default}(a_i)$: Average time for a_i in default placement
$t_{tree}(a_i)$: Average time for a_i in tree-structured placement
C : Number of Cylinders
T_{rot} : Rotational Period
T_{nt} : Time taken to transfer one block of data

First we compute the random, sequential and semi-sequential access times. The average random access time t_{rand} , is a function of the average seek time and rotational delay and is given by:

$$t_{rand} = seekTime\left(\frac{C}{3}\right) + \frac{1}{2} T_{rot} \quad (1)$$

where $seekTime$ is a disk specific function computing the seek time given the number of tracks to seek [Ruemmler and Wilkes 1994] and is given by:

$$\begin{aligned} seekTime(d) &= \alpha + \beta \cdot \sqrt{d}; \text{ if } d < \frac{C}{3} \\ &= \gamma + \delta \cdot d; \text{ otherwise} \end{aligned} \quad (2)$$

where d is the seek distance in cylinders, C is the total cylinder count, and α, β, γ and δ are disk specific parameters.

For the barracuda disk, chosen as the base disk configuration in the experiments (and also further described in Table VII), the rotational latency is given by $T_{rot} = 8.33$ ms and $\alpha = 1.83, \beta = 0.17, \gamma = 2.85$ and $\delta = 0.0035$. For an XML document of size 50MB occupies 129188 blocks or 325 cylinders after grouping with the tree-preserving grouping strategy (Table IV). Thus, substituting these values in the above Equation 1, the random access time for the area occupied by this document is given by $t_{rand} = 5.99$ ms.

The average sequential access time t_{seq} from one block to the next is a very small value, approaching zero. Hence,

$$t_{seq} = 0 \quad (3)$$

For the tree-structured placement, the access between a parent and its first child is semi-sequential, and from a node to its right sibling is sequential. The average time for semi-sequential access $t_{semi-seq}$ given by:

$$t_{semi-seq}(v) = seekTime(s(v)) \quad (4)$$

where $s(v)$ is the number of tracks to be sought during a semi-sequential access. When T is a complete tree with height d and degree f , the average $s(v)$ is given by:

$$s(v) = \frac{f^{d-2}(d-2-f/(1-f)) + 2 + f/(1-f)}{2n'} \quad (5)$$

where n' is the number of internal nodes given by $n' = \frac{(1-f^{d-1})}{(1-f)}$

To understand this equation, let's assume that the root is at depth 1 and the leaves at depth d . If there are two edges $u_1 - v_1$ and $u_2 - v_2$ where u_1 and u_2 are on the same level and v_1 and v_2 are their l^{th} respectively, then $DFO(v_1) - DFO(u_1) = DFO(v_2) - DFO(u_2)$. Thus, the distance in tracks from v_1 to its child u_1 and from v_2 to u_2 are the same. In the above relation, $DFO(x)$ is the corresponding number in the DFO ordering. The numbers above the internal nodes in the tree shown in Figure 3 illustrate the DFO ordering.

To calculate the average $s(v)$ for the nodes v of level $k+1$, we need to find the size of the subtree rooted at v which is

$$1 + f + \dots + f^{d-k-1} = \frac{(1-f^{d-k})}{(1-f)} \quad (6)$$

The average of $s(v)$ for the nodes v of level $k+1$ is the average $s(v)$ of any set of siblings at level $k+1$. That is,

$$\frac{\left(\frac{f+(1-f^{d-k})}{(1-f)(1+\dots+(f-1))}\right)}{f} = \frac{\left(\frac{f+(1-f^{d-k})}{(1-f)(f-1)f/2}\right)}{f} = \frac{(f^{d-k} + 1)}{2} \quad (7)$$

Hence, for level k it is $\frac{(f^{d-k-1}+1)}{2}$.

For an average fanout of 10 and a depth of 5 in an XML tree, $s(v)$ from Equation 5 is 1.83. Thus, the $seekTime(s(v))$ is $\alpha + \beta \cdot \sqrt{1.83} = 2.26$.

Equation 4 assumes perfect semi-sequential time, which is achieved by the tree-structured algorithm (Algorithm 2). However, in the case of the optimized tree-structured algorithm (Algorithm 3), $t_{semi-seq}(v)$ depends on the number of track-regions per-track, k . Hence,

$$t_{semi-seq}(v) = seekTime(s(v)) + \frac{1}{2k}T_{rot} \quad (8)$$

Since the first-child is placed anywhere within a rotationally-optimal track-region rather than rotationally optimal sector, accessing the first child may involve anywhere between 0 to $\frac{1}{k}T_{rot}$ rotational delay after the seek operation. This additional rotational delay during the semi-sequential access is $\frac{1}{2k}T_{rot}$ on an average. When a track is divided in 8 track regions, $k=8$ and for the barracuda disk, $s(v)$ is calculated above and is 1.83 ms. Substituting these values in Equation 8, the average

semi-sequential time is given by $t_{semi-seq}(v) = 2.79$ ms, a significant reduction of 53.4 % from an average random access time of 5.99 ms.

Next, we discuss the time needed for each of the five basic access types of Table II. When the first child is accessed from its parent (a_1), a sequential access occurs in the default placement, whereas a semi-sequential access occurs in the tree-structured placement. When a non-first child is read from its parent (a_2), it is a random access in the default placement, whereas for the tree-structured placement, it is the sum of the semi-sequential time and the average sibling index ($f/2$, where f is the tree fanout) times T_{nt} (time required to transfer data from one node). When the access is from a non-leaf node to its right sibling (a_3) it is a random access in the default placement, and a sequential access in the tree-structured placement. When from a leaf-node we access its right sibling (a_4), it is a sequential access in either placement strategy. In all other cases (a_5), such as when moving up the tree, for both placements a random access will be performed. Table III summarizes the access times in the default and the tree-structured storage for every a_i .

Table III. Average access times in default and tree-structured placement for each access type a_i .

Access type a_i	Description	$t_{default}(a_i)$	$t_{tree}(a_i)$
a_1	Parent to first child	t_{seq}	$t_{semi-seq}$
a_2	Parent to non-first child	t_{rand}	$t_{semi-seq} + \frac{f}{2}(T_{nt})$
a_3	Non-leaf node to right sibling	t_{rand}	t_{seq}
a_4	Leaf node to right sibling	t_{seq}	t_{seq}
a_5	All other accesses	t_{rand}	t_{rand}

The average access times in default and tree-structured storage are computed by Equations 9 and 10 respectively.

$$T_{default} = \sum_{i=1}^5 P_i \cdot t_{default}(a_i) \quad (9)$$

$$T_{tree} = \sum_{i=1}^5 P_i \cdot t_{tree}(a_i) \quad (10)$$

Tree-structured placement is better when $T_{tree} < T_{default}$.

While this is not realistic (and necessarily subjective to the query as demonstrated extensively later in Table VI), if we did assume that a query exhibits all the access types shown in Table III, with each access type occurring equally frequently, the average I/O times for the default and the tree placement can be obtained by substituting their values in Equations 9 and 10 as:

$$\begin{aligned} T_{default} &= \frac{1}{5} \cdot t_{seq} + \frac{1}{5} \cdot t_{rand} + \frac{1}{5} \cdot t_{rand} + \frac{1}{5} \cdot t_{seq} + \frac{1}{5} \cdot t_{rand} \\ &= 3.594 \text{ ms, and} \end{aligned}$$

$$\begin{aligned} T_{tree} &= \frac{1}{5} \cdot t_{semi-seq} + \frac{1}{5} \cdot (t_{semi-seq} + \frac{f}{2}(T_{nt})) + \frac{1}{5} \cdot t_{seq} + \frac{1}{5} \cdot t_{seq} + \frac{1}{5} \cdot t_{rand} \\ &= 2.344 \text{ ms} \end{aligned}$$

where the transfer time $T_{nt} = 0.03$ ms.

6. EVALUATION CASE STUDY: XML

In this section, we experimentally evaluate the grouping and native layout strategies for placing the XML data on disk drives.

We used the DiskSim [Bucy *et al.* 2003] disk simulator for our evaluations, instrumenting it to provide the additional interface: `<LBN> findSemiSequential(LBN parent, int cyl, int track)` which given a parent LBN, returns an LBN X on `<cyl,track>`, such that access from the parent LBN to X is semi-sequential. The optimized-tree placement in Algorithm 3 uses this interface to find semi-sequential LBA for subsequent nodes in the tree that has to be placed on the disk. The optimized tree-structured and the default placement algorithms were implemented in C and integrated with the instrumented DiskSim code. The grouping algorithms were implemented as a separate module.

6.1 Data Set and Queries

We generated XML files (each file corresponds to an XML tree) of various sizes using the XMark generator [Schmidt *et al.* 2002b] with different scaling factors from $f = 0.01$ to $f = 1.00$, corresponding to file sizes ranging from 1MB to 100MB. The limit of 100MB for the maximum file size is due to the memory constraints in currently available open-source XML parsing engine implementations. These engines create the navigation tree data structures for the entire tree in memory during parsing, while at the same time consuming as much memory as five times the original document size [Nicola and John 2003]. There is ongoing work on improving memory efficiency of XML parsers [Farfan *et al.* 2007] which promise to address this shortcoming in the near future. Earlier in Table I, we presented the document sizes used by several popular benchmarks typically used to evaluate XML query optimizations, storage, indexing and so on. As mentioned earlier in Section 2, trivial solutions that load the entire document in memory are not practical for large (several gigabyte sized) XML documents. Although the XML documents we experiment with are small relative to the size of the disk, these serve as examples to illustrate the *relative* effectiveness of native layout when compared to the existing approaches. It should additionally be noted that the on-disk buffer is small (1-8MB) for the disks we use, substantially smaller relative to the size of the documents, and is not in any significant way capable of influencing the I/O access patterns apart from on-disk readahead.

We implemented the three grouping strategies - *sequential*, *tree-preserving*, and *EKM* - described in Section 4, computing and storing the information about the supernode that would contain each XML node. We also implemented extensions to the DiskSim disk simulator [Bucy *et al.* 2003] that allowed us to simulate the native layout strategy described in Section 3. We then used the supernode information to store them on disks simulated by DiskSim.

Table IV provides information about the XML trees used and the corresponding supernode trees formed. The number of supernodes in the sequential grouping is the lowest since it groups the nodes to form supernodes without any restrictions. EKM does a bottom-up grouping of the tree and reduces the number of resulting supernodes by reducing the problem of finding supernodes for arbitrary trees to the simpler problem of finding supernodes for flat trees (trees in which all nodes but

Table IV. XML Tree and Supernode Tree Parameters

XMark	Tree (KB)	#Nodes	Avg Bytes per node	# Supernodes			Avg Bytes/Supernode		
				TP	Seq.	EKM	TP	Seq.	EKM
0.01	1667	17132	25.21	2576	2119	2148	343.8	418	412.3
0.05	8270	59641	25.8	12834	10625	10703	373.2	450.8	447.5
0.1	16765	167865	25.85	25991	21435	21628	345.3	418.7	414.9
0.5	83726	832911	26.09	129188	106592	114775	345.3	418.5	414.6
1	168755	1666315	26.07	259575	214326	216140	345.3	418.2	414.7

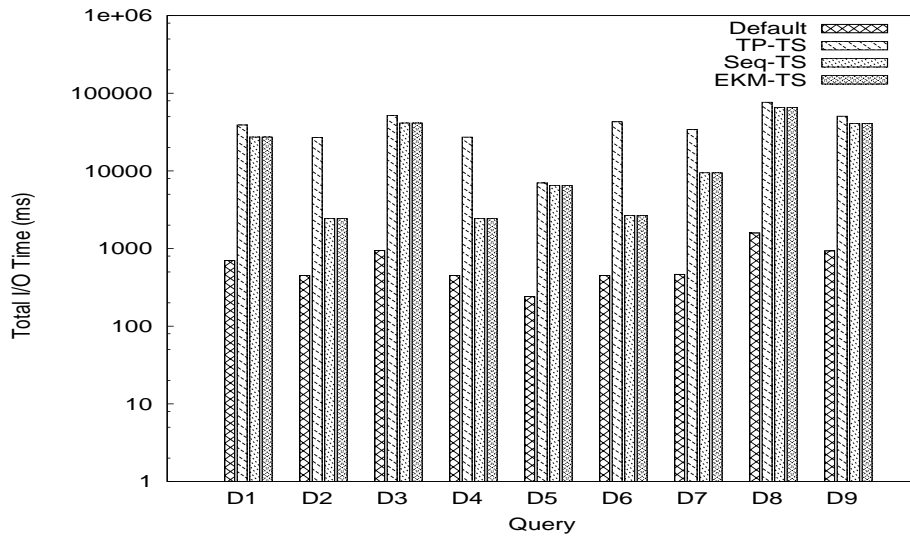
Table V. XPath queries for the deep-focused (D) and the non deep-focused (N) classes.

#	Query	#	Query
D1	<code>/site/closed_auctions/closed_auction/annotation/description/parlist/listitem/text/keyword</code>	N1	<code>/site/open_auctions/open_auction</code>
D2	<code>/site/people/person/watches</code>	N2	<code>/site/closed_auctions</code>
D3	<code>/site/open_auctions/open_auction/annotation/description/text/keyword</code>	N3	<code>/site/regions/australia</code>
D4	<code>/site/people/person/address/country</code>	N4	<code>/site/closed_auctions/closed_auction</code>
D5	<code>/site/regions/australia/item/description/text/emph</code>	N5	<code>/site/regions/*/item</code>
D6	<code>/site/people/person/*/business</code>	N6	<code>/site/*/australia</code>
D7	<code>/site/closed_auctions/closed_auction/*/description</code>	N7	<code>/site/open_auctions/open_auction[@id='open_auction0']/bidder</code>
D8	<code>/site/regions/*/item/description/text</code>	N8	<code>/site/regions/asia/item[@id='item4']/mailbox/mail/from</code>
D9	<code>/site/closed_auctions//itemref</code>	N9	<code>/site/open_auctions/open_auction[@id='open_auction0']/keyword</code>

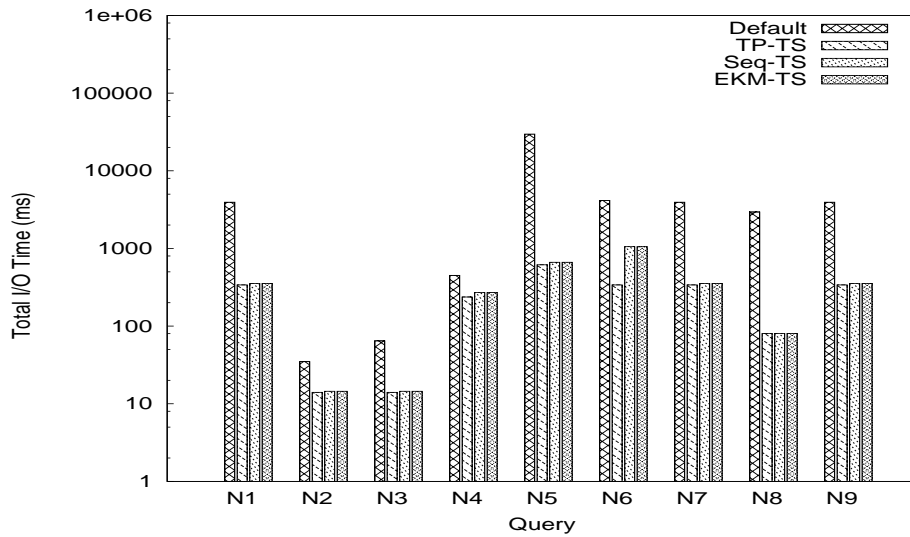
the root are leaves) [Kanne and Moerkotte 2006]. Tree-preserving grouping avoids cycles by placing restrictions on the nodes being added to the supernode. This in turn reduces the number of nodes per supernode and subsequently increases the number of supernodes. The average nodes/supernode is six for the tree-preserving grouping and is 8 for Sequential and EKM grouping.

For the query workload, we adopted performance-sensitive queries from the XPath-Mark benchmark [Franceschet 2004], but omitted the ones that check for features supported by XPath (e.g., *Q18: /comment()*). To compute reliable results we added more queries with similar properties of depth, number of conditions and selectivity. The query workload is summarized in Table V.

To contrast the relative advantages of using our native strategies with those of the default sequential layout, we classify XPath queries into two categories: *deep-focused queries* and *non deep-focused queries*. A subset of each class is shown in Table V. The former class describes the special class of XPath queries that navigate entire subtrees of the tree (queries D_1, \dots, D_9 in Table V). The latter class, non deep-focused queries N_1, \dots, N_9 in Table V, represents all queries that do not belong to the former class. As we shall demonstrate, the default layout primarily addresses the class of deep-focused queries and is sub-optimal for all other queries. Notice that only the supernode-granularity navigation matters for overall I/O performance, and not the node-granularity navigation. Hence, queries like D_2 , which do not access



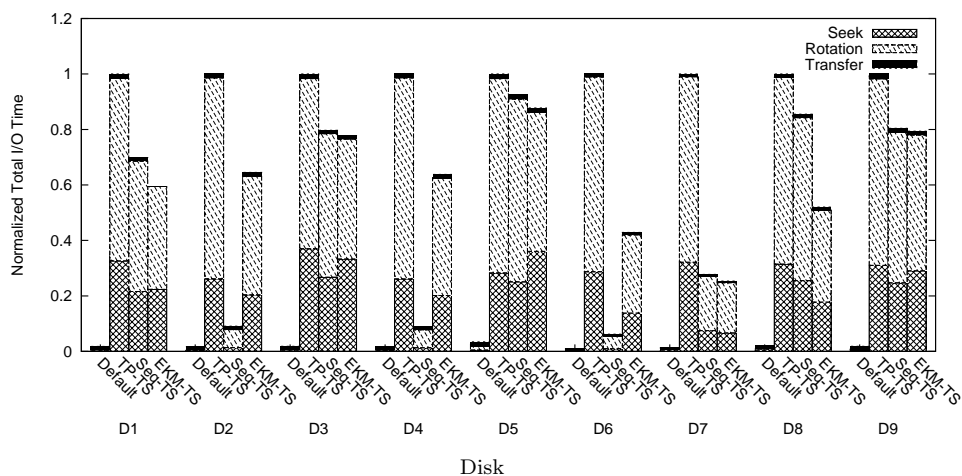
(a) Deep-focused queries



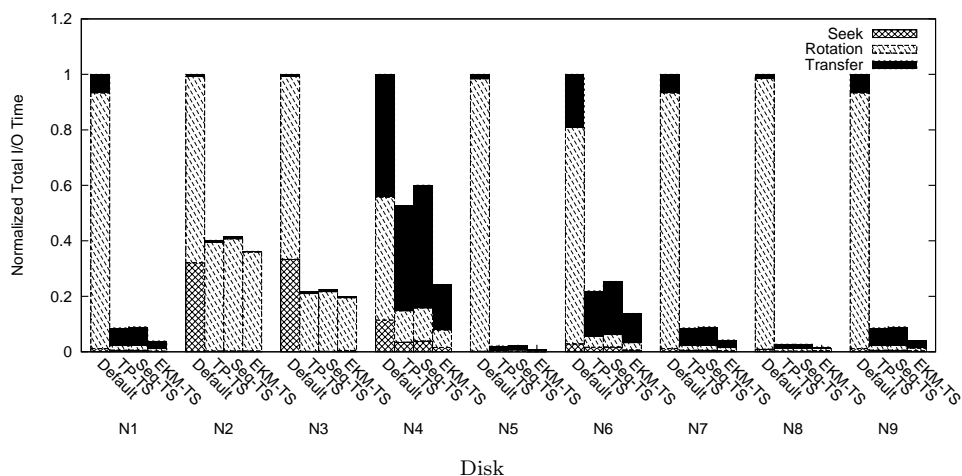
(b) Non-deep-focused queries

Fig. 8. Total I/O times in logarithmic scale for various placement strategies.

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.



(a) Deep-focused queries



(b) Non-deep-focused queries

Fig. 9. Normalized total I/O times for various placement strategies.

6.2 Tree Navigation Performance

We conducted experiments that compare the I/O times for answering XML queries for four different layout strategies, corresponding to the supernode tree organizations of Section 4: *default* (Section 2.2), *tree-preserving tree-structured* (TP-TS), *sequential tree-structured* (Seq-TS), and *EKM tree-structured* (EKM-TS) layout strategy.

To consider caching effects in our experiments, we assumed that all nodes along the path from the root to a single leaf node would be cached in main memory, either in the operating system VFS or a custom application level cache. This is a reasonable assumption for XML trees, which are typically short even when their

Table VI. Navigational patterns for the two XPath query classes for $f = 0.5$. a_i 's are defined in Table II.

Default Placement											
Query	a1	a2	a3	a4	a5	Query	a1	a2	a3	a4	a5
D1	9046	0	0	0	1982	N1	1098	0	0	0	4775
D2	7211	0	0	0	55	N2	0	0	0	0	5
D3	12744	0	0	0	1895	N3	0	0	0	0	10
D4	7211	0	0	0	55	N4	1387	0	0	0	3053
D5	1823	0	0	0	759	N5	1322	0	0	0	9323
D6	7315	0	0	0	4	N6	9324	0	0	0	8418
D7	2765	0	0	0	2814	N7	1098	0	0	0	4775
D8	11937	0	0	0	9654	N8	121	0	0	0	870
D9	16166	0	0	0	5	N9	1098	0	0	0	4775
TP-TS Placement											
Query	a1	a2	a3	a4	a5	Query	a1	a2	a3	a4	a5
D1	4438	1182	1799	1114	5117	N1	1	1	71	5513	1
D2	3250	3	333	1801	3251	N2	0	1	0	4	0
D3	6171	1729	2428	902	7897	N3	0	1	0	9	0
D4	3287	3	333	1764	3288	N4	0	2	42	3762	0
D5	659	319	507	169	976	N5	0	6	42	10065	5
D6	5218	1	371	3	5049	N6	4	2	485	14647	4
D7	1344	2665	42	71	3758	N7	1	1	71	5513	1
D8	4071	4831	1360	2164	8896	N8	0	2	2	937	1
D9	8213	1	4657	4	7199	N9	1	1	71	5513	1
Seq-TS Placement											
Query	a1	a2	a3	a4	a5	Query	a1	a2	a3	a4	a5
D1	6856	1073	1768	219	1112	N1	1074	859	5	24	3911
D2	6714	47	47	0	458	N2	0	1	0	0	4
D3	9582	458	2347	123	2129	N3	0	1	0	0	9
D4	6714	47	47	0	458	N4	1347	777	2	7	2307
D5	1149	175	487	33	738	N5	1305	2576	0	103	6661
D6	6765	1	95	0	458	N6	8771	1719	83	47	7122
D7	2620	1098	2	44	1815	N7	1074	859	5	24	3911
D8	9193	3364	1385	715	6934	N8	120	227	0	6	638
D9	10564	1	4602	0	1004	N9	1074	859	5	24	3911
EKM-TS Placement											
Query	a1	a2	a3	a4	a5	Query	a1	a2	a3	a4	a5
D1	2126	4153	1795	1319	5521	N1	0	2	88	2305	1
D2	2040	1117	3342	1983	3156	N2	0	1	0	0	0
D3	3259	5042	3838	731	7981	N3	0	1	0	4	0
D4	2040	1117	3342	1983	3156	N4	0	2	151	1495	0
D5	445	1106	395	287	1414	N5	0	6	89	3588	5
D6	2242	1129	3347	1924	3306	N6	0	6	3584	6174	4
D7	803	2000	151	1237	2801	N7	1	2	88	2304	2
D8	2730	9672	913	3323	12399	N8	0	2	12	327	1
D9	3180	2581	6116	0	4029	N9	1	2	88	2304	1

total size is large, due to large fan-out. Consequently, we ignore repeated accesses to nodes (such as parent, ancestor nodes) during the depth first traversal of the XML tree. 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 layout strategy.

6.2.1 *Total I/O time.* Figure 8 shows (in logarithmic scale) the I/O times for each query, for the two classes of queries, deep-focused (D_i) and non deep-focused (N_i), for an XMark file with scaling factor $f = 0.5$. We executed five simulation runs for each column shown in the graph. For the first run, the start LBA for the placement of the root node was 0. For all the subsequent runs, it varied with increments of 250 ($>$ track size). Thus, the start LBA was varied over the range 0 – 1250. The confidence interval, for a confidence level of 95%, for all the five runs

was found to be $< \pm 10.96$. The results shown in the graph are for the start LBA 0.

For the deep-focused class of queries, the default placement strategy performs consistently better than the others, since it can retrieve entire subtrees more efficiently. 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 sub-tree accesses are few.

Figure 9 shows the relative performance with the normalized total I/O time to reduce the impact of the large variance across queries. Each value is scaled relative to the maximum value for the experiment. To better demonstrate the relative distribution of seek, rotational delay, and transfer time components, the total normalized I/O time is further split to show these I/O access time components. It can be seen that the average rotational delays for the tree-structured placement strategies (in the case of non-deep-focused queries) are substantially lower relative to the default strategy. However, this is not the case for the deep-focused class where the default strategy outperforms in all respects.

To better understand and explain the graphs of Figure 8 and Figure 9, we counted the different types of accesses in the supernode tree (each access translates to a disk I/O operation) for answering the XPath queries for both the deep-focused and non deep-focused classes. Table VI shows the numbers of supernodes accesses for the five basic types of tree accesses, a_1 through a_5 , defined in Table III. As an example, observe that for the TP-TS placement, Query D1 requires 4438 a_1 accesses, the parent-to-first-child type accesses.

We can make some general observations from Table VI. First, the default placement causes all the accesses to be either of type a_1 or a_5 , since only parent-to-first-child sequential accesses are possible for this layout. Second, the deep-focused queries are dominated by a_1 and a_5 type accesses, while the non-deep-focused queries are dominated by a_3 and a_4 accesses (except in the case of default placement). This enables the non-deep-focused queries to exploit native layout, since all the accesses to siblings are sequential, as opposed to the large number of random accesses the deep-focused queries require. Observe further that the EKM and TP-TS placement strategies increase the number of accesses from parent to non-first child, thus utilizing the semi-sequential and sequential access optimization to a larger extent. For the deep-focused queries, on the other hand, the default placement performs the best both because the number of sequential accesses for this placement is the highest and number of random accesses is lowest (in most cases) among all placement techniques.

In Figure 9 (b), we see a somewhat unexpected outcome that the seek times reduce for queries N2 and N3 for TP-TS, Seq-TS and EKM placement. An answer can be found in the access patterns of these queries (Table VI). For N2 and N3, all accesses for the default placement are of type a_5 , which are random accesses, where as for the TP-TS and EKM placement, they are either semi-sequential or sequential accesses, leading to the observed difference in seek overhead. Further, the Seq-TS has a slightly lower performance relative to these two because of the increase in the number of random accesses for this placement. Note that although

the number of random accesses in Seq-TS is relatively higher, it is still lower than the default placement and hence it performs better than the default placement.

The above discussion serves to reinforce the arguments we made earlier when discussing Figure 8. In summary, the EKM-TS placement strategy performs better overall due to its lower internal fragmentation and tree-structure preservation property; it results in I/O times which are 3X-127X better than the default strategy. Between the remaining strategies, TP-TS performs better on an average, since it better preserves the original tree-structure.

6.2.2 Sensitivity to drive characteristics. To evaluate the effect of drive characteristics, we conducted a sensitivity study of I/O access time for representative disk-drive models. The drive models chosen, shown in Table VII, 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.

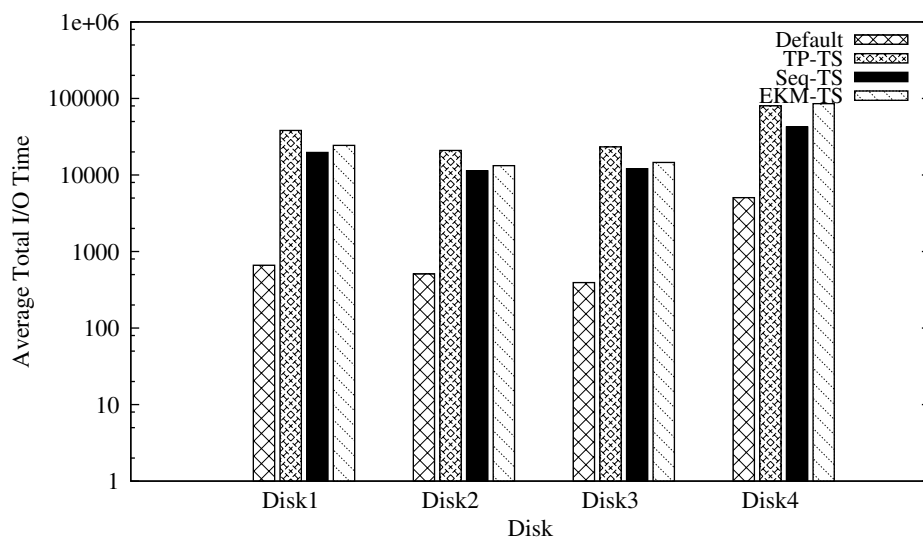
Table VII. Characteristics of experimented disk drives.

Disk model	Disk type	Size [GB]	RPM	Stroke [ms]	Transfer [MBps]	Track Size [sectors]	Cylinders
Barracuda	Base	2	7200	16.679	10-15	119-186	5172
Cheetah 9LP	Fast disk	9.1	10045	10.627	19-28.9	167-254	6962
Cheetah 4LP	Fast rotate	4.5	10033	16.107	15-22.1	131-195	6581
HP C3323A	Slow rotate	1	5400	18.11	4.0-6.6	72-120	2982

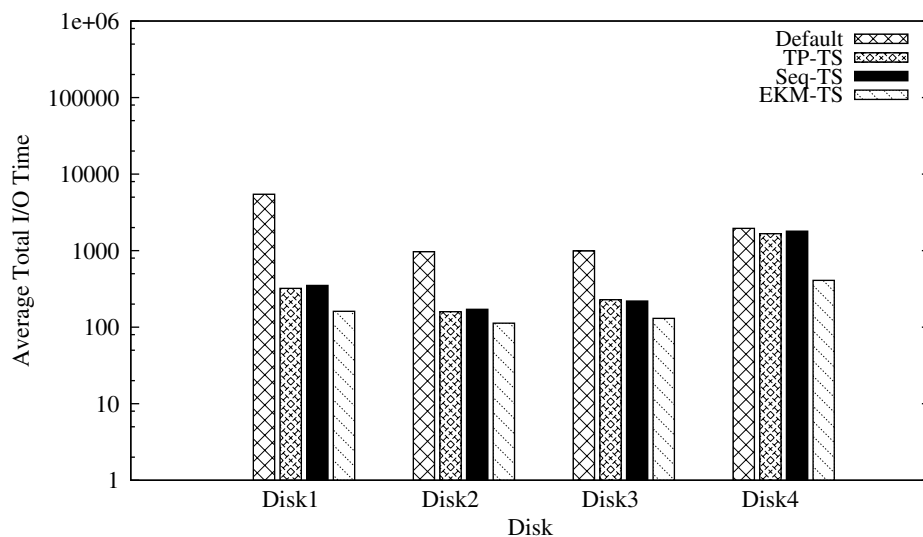
Figure 10 shows the average (across queries in a query-class) total I/O times (in logarithmic scale) for the two query classes for an XMark file with $f = 0.5$ with the various hard disk models. For the special class of deep-focused queries (Figure 10(a)), the default placement strategy performs better than the other strategies benefiting from optimized sub-tree retrievals. However, for all other queries (Figure 10(b)), the tree-structured placement strategies perform better for all disk models, offering as much as 7X-34X reduction in average I/O time for answering queries. This underscores the importance of native layout strategies for XML data.

We break down the gains further in Figure 11 into the relative reduction in seek and rotational delay components for each of the drives by normalizing the I/O times at each disk drive using the maximum value as reference.. Notice for the non-deep-focused query class (Figure 11(b)), the average rotational-delays are substantially reduced relative to the default layout.

6.2.3 Effect of Query Interleaving. One concern with a native layout targeted to a optimize a specific access pattern is the impact of multi-processing in the system. For instance, a server is likely to execute multiple XPath queries simultaneously; optimizing individual query executions may not necessary translate to overall performance improvement when the corresponding I/O request sequences are interleaved. As elaborated in Section 3, this issue in its more general form (i.e.,



(a) Deep-focused queries



(b) Non-deep-focused queries

Fig. 10. Sensitivity of query I/O times to changing disk drive characteristics (logarithmic scale).

multi-process blocking I/O performance) has been addressed earlier with anticipatory I/O scheduling [Iyer and Druschel 2001]. Consequently, we expect that XML servers would be configured with I/O schedulers that include an anticipation core.

To evaluate the performance of our grouping and placement techniques under multiple simultaneous XPath queries, we interleaved a subset of deep-focused and non-deep-focused queries stated in Table V. The interleaved queries belonged to

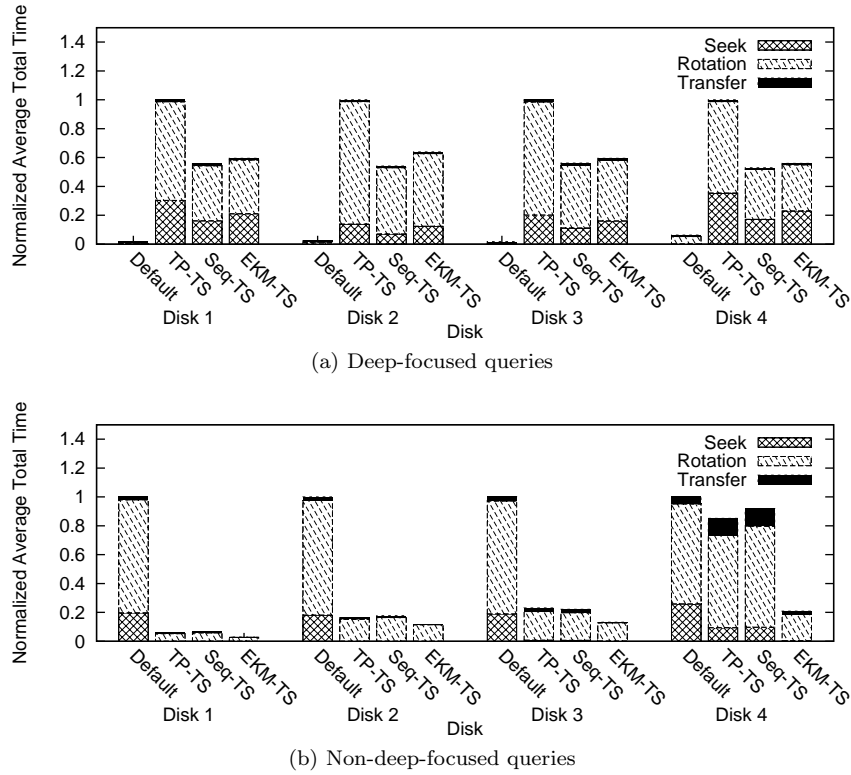


Fig. 11. Sensitivity of seek and rotational delay components of I/O access times to changing disk drive characteristics.

either the disjoint set of queries which accessed disparate portions of the tree or intersecting queries whose access paths overlapped. The ordering of the I/Os after interleaving were based on anticipatory scheduling. We simulate the behavior of the anticipatory I/O scheduler assuming that each query is serviced within an independent thread and issues synchronous I/O requests. The behavior of the non-work-conserving anticipatory scheduler would result in optimizing the schedule of successive I/O operations resulting from the same query, in spite of them being issued synchronously, as long as other queries in the system access disjoint portions of the XML tree. When there is an overlap of subtrees between two queries, their I/Os must interleave.

For the choice of queries, we selected both *disjoint queries*, which traverse different subtrees of the document, as well as *intersecting queries*, that access common subtrees, which navigate common sub-trees of the document. Table VIII shows the selected queries that were interleaved in each of these categories, where δ_i refers to disjoint queries and π_i represents intersecting queries.

Figure 12 shows the total I/O time (in logarithmic scale) for the execution of interleaved deep-focused and non-deep-focused XPath queries. The results for the deep-focused queries from Figure 12 (a), show that like in single query execution,

Table VIII. Query Interleaving for Multi-User Simulations.

Disjoint Queries	Deep-focused Queries	Non-deep-focused Queries
δ_1	$D_1 + D_4$	$N_1 + N_4$
δ_2	$D_4 + D_8$	$N_4 + N_8$
δ_3	$D_5 + D_7$	$N_5 + N_7$
δ_4	$D_1 + D_4 + D_5$	$N_1 + N_4 + N_5$
δ_5	$D_4 + D_5 + D_7$	$N_4 + N_5 + N_7$
δ_6	$D_4 + D_5 + D_9$	$N_4 + N_5 + N_9$
Intersecting Queries	Deep-focused Queries	Non-deep-focused Queries
π_1	$D_1 + D_7$	$N_1 + N_6$
π_2	$D_2 + D_4$	$N_5 + N_6$
π_3	$D_5 + D_8$	$N_7 + N_9$
π_4	$D_4 + D_6$	$N_1 + N_6 + N_7$
π_5	$D_1 + D_7 + D_9$	$N_6 + N_7 + N_9$
π_6	$D_2 + D_4 + D_6$	$N_5 + N_6 + N_8$

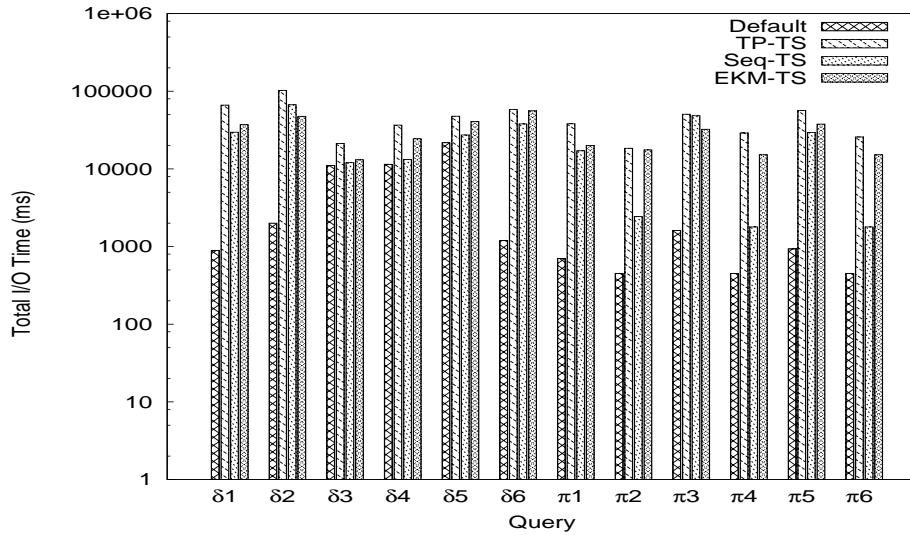
the default strategy performs better for multiple interleaved queries than the other strategies.

Similarly, the behavior with the the non-deep-focused interleaved queries mostly mimic their single query counterparts. The native layout strategies provide much better execution times for both the disjoint and intersecting queries, as shown in Figure 12 (b). Moreover, the EKM-TS performs better the most consistently across the interleaved query executions. The breadth-first grouping approach of this placement strategy causes the I/Os corresponding to the upper levels of the XML tree to be read in parallel. For lower tree levels, the anticipatory scheduler which ensures that the I/O sequences generated by the individual query threads are grouped successfully. Finally, the default placement performs consistently worse for the disjoint queries, since the I/O sequences generated by individual query threads are executed almost sequentially.

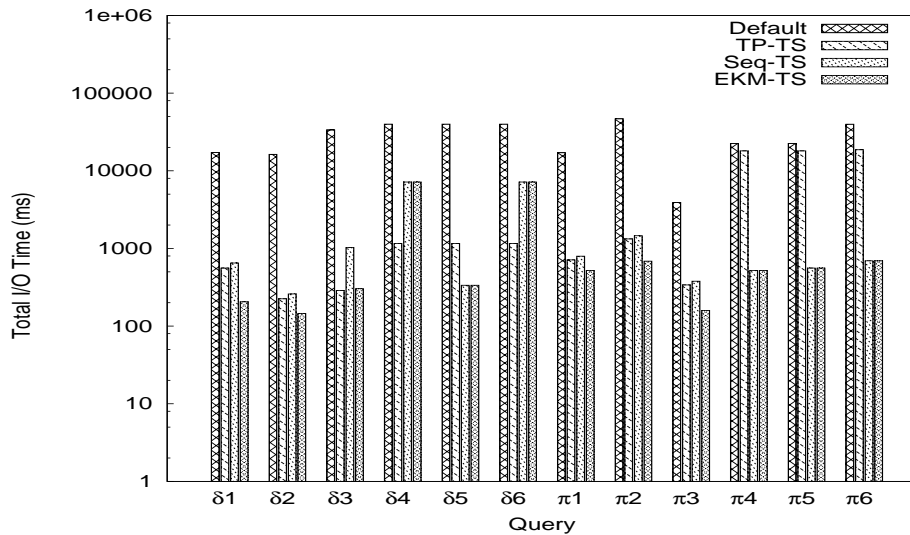
6.3 Fragmentation

We now measure the internal and external fragmentation incurred by the grouping and placement algorithms respectively.

Internal Fragmentation: Figure 13 (a) 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. Supernodes are not occupied completely if its the remaining space 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 I/O access time than by I/O capacity, trading capacity for improving access is acceptable. The internal fragmentation with EKM is very close to that for



(a) Deep-focused queries

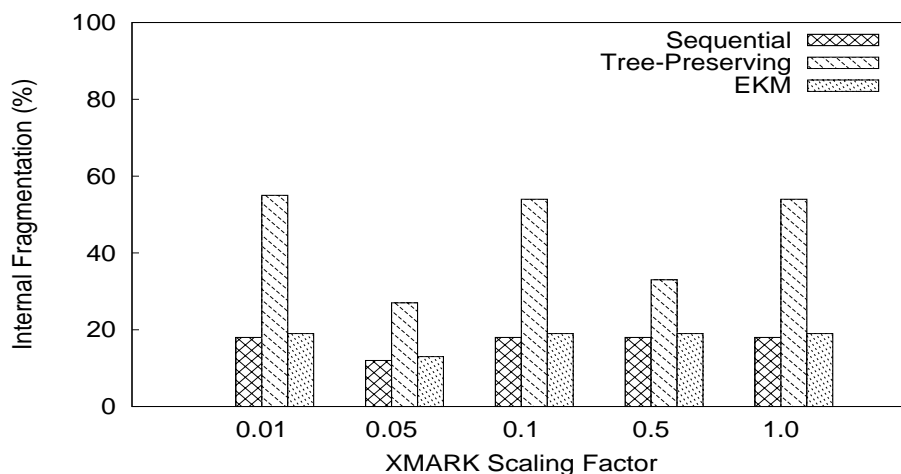


(b) Non-deep-focused queries

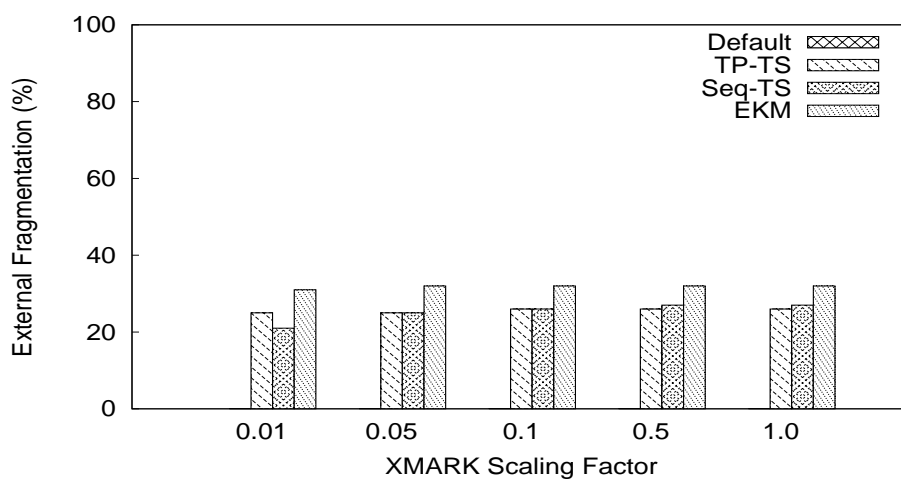
Fig. 12. Total I/O times in logarithmic scale for interleaved XPath queries.

sequential grouping. The EKM algorithm has the flexibility that allows selecting any of a node's many subtrees as partition, 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.

External Fragmentation: Figure 13 (b) shows the external fragmentation re-



(a) Internal fragmentation



(b) External fragmentation

Fig. 13. Internal and External Fragmentation.

sults for the data placement strategies. The default strategy incurs zero external fragmentation as it places the supernode list sequentially on the disk. TP-TS and Seq-TS incur external fragmentation of less than 28%, while that of the EKM-TS is higher at around 32%. However, we once again contend that these numbers are acceptable, following the arguments mentioned above. EKM-TS incurs the highest external fragmentation, 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, unlike 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. Furthermore, for a native storage solution that is well integrated into the existing file or database system, it is relatively easy to utilize fragmented free space.

7. RELATED WORK

Storage of semi-structured data has received attention in the last few years because of its growing popularity. Most work has focused on storing semi-structured data in relational DBMSs or in flat files with indexes. The former approach (e.g., [Barbosa *et al.* 2001; Du *et al.*; Shanmugasundaram *et al.*; Nekrestyanov *et al.* 2000; Deutsch *et al.* 1999; McHugh *et al.* 1997]) has been the most popular due to the success and maturity of the relational DBMSs. The latter approach (e.g., [Kaushik *et al.* 2002; Li and Moon 2001]) is based on storing the data as a flat file and building separate indexes on top. These strategies do not use native layout of semi-structured data and are limited to the generic optimization strategies built into relational databases and file systems.

The problem of native storage of semi-structured data has been addressed in Natix [Kanne and Moerkotte 1999; Kanne *et al.* 2005] and in System RX [Beyer *et al.* 2005], where the tree-structured data is split into pages and each page is stored in a disk block, thereby reducing the number of read accesses while traversing the tree. OrientStore [Meng *et al.* 2003] uses schema information to make a storage plan for the semi-structured data. The above studies however view a disk drive as a list of pages and do not take into account the physical characteristics of its operation whereas 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.

Given the restrictive block IO interface, the clear case for a more expressive interface has been made before [Ganger 2001]. Systems such as [Gibson *et al.*; Huston *et al.* 2004; Sivathanu *et al.* 2003] use intelligence from upper layers of the storage stack inside storage devices to improve overall IO performance. Our work, if deployed, can use such systems, to incorporate storage techniques for semi-structured data into disk firmware.

Recent work by [Schlosser *et al.* 2005] uses the idea of semi-sequential access for efficient storage of multi-dimensional data. This work is significantly different from our work in that unlike semi-structured data, multi-dimensional data is structured with access patterns along data dimensions and can afford efficient layout based on fixed attribute cardinality. Also, with semi-structured data, grouping multiple data elements to be stored on a disk block is non-trivial due to the variable size of the data elements.

Atropos [Schindler *et al.* 2004] exploits the physical properties of disk drives and uses semi-sequential accesses to store relational databases. Our work targets XML data that has a tree structure, quite different from the relational tables. We also show that a naive application of the semi-sequential access paradigm to XML 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 XML data, accounting for low-level hard drive storage and operation semantics.

8. CONCLUSIONS

In this paper, we have taken a first step towards building native storage systems for semi-structured data, a problem which has been largely unexplored. We presented on-disk data layout techniques for semi-structured data that explicitly account for the structural mismatch between the semi-structured data and disk drives and reduce disk access overhead. These layout techniques are based on node-grouping algorithms for semi-structured data that reduce the number of disk I/O operations required when accessing the data. We have suggested directions for addressing the challenges that would arise in integrating the proposed layout techniques in existing storage systems.

8.1 Summary of Experimental Findings and Lessons Learned

We conducted an evaluation of the native layout techniques using XML as a case-study. All experiments were performed on XPathMark benchmark queries with an instrumented DiskSim simulator. Our experiments revealed that:

- For the specific class of *deep-focused* queries, which result in access patterns retrieving entire sub-trees, the existing file system layout mechanism (i.e., sequential layout of the tree in depth-first-order) offers significantly better performance than native layout (5X-54X across the query set). For such queries, we believe that sequential layout is the right choice.
- For all other query classes, which we group as *non-deep-focused*, native layout taking into account tree navigation primitives, offers as much as 3X-127X performance improvement across the range of XPathMark queries that we experimented with, representing a large improvement. A sensitivity study across a range of disk models, representing drives of varying performance, suggest that average I/O performance improvement across the non-deep-focused query set of 7X-34X.
- Of the various native layout techniques we considered, the EKM-TS provided consistently better performance, barring a few cases. The above findings were largely preserved when we experimented with multiple simultaneous query executions with the anticipatory I/O scheduler. This scheduler naturally carries forward the benefits of native layout into the I/O schedule.
- Native layout strategies, however, can result in substantial fragmentation of disk space. Our initial estimates reveal total fragmentation (internal+external) of as much as 50% for the best-performing EKM-TS layout technique. This fragmented space can be reclaimed with clever file system or database system implementations to store non semi-structured data. Even if that were not feasible, we believe an additional 50% of space overhead for several magnitudes of I/O bandwidth increase could be acceptable in many settings.

Our findings in this study serve to more closely examine and evaluate layout techniques based on the nature and distribution of queries (i.e., access patterns). Further, based on our findings in this study, it can be inferred that a single layout technique is unlikely to be optimal for navigating semi-structured data; the optimality of any layout technique closely depends on the nature of the workload. A prudent choice of the underlying data layout strategy can drastically improve I/O

access times if knowledge of the access patterns (e.g., query workload) is available beforehand.

REFERENCES

- ABITEBOUL, S., AGRAWAL, R., BERNSTEIN, P., CAREY, M., CERI, S., CROFT, B., DEWITT, D., FRANKLIN, M., MOLINA, H. G., GAWLICK, D., GRAY, J., HAAS, L., HALEVY, A., HELLERSTEIN, J., IOANNIDIS, Y., KERSTEN, M., PAZZANI, M., LESK, M., MAIER, D., NAUGHTON, J., SCHEK, H., SELLIS, T., SILBERSCHATZ, A., STONEBRAKER, M., SNODGRASS, R., ULLMAN, J., WEIKUM, G., WIDOM, J., AND ZDONIK, S. 2005. The lowell database research self-assessment. *Commun. ACM* 48, 5, 111–118.
- AFANASIEV, L., MANOLESCU, I., AND MICHELS, P. 2005. Member: A micro-benchmark repository for xquery. In *Database and XML Technologies, Third International XML Database Symposium, XSym 2005, Trondheim, Norway, August 28-29, 2005, Proceedings*, S. Bressan, S. Ceri, E. Hunt, Z. G. Ives, Z. Bellahsene, M. Rys, and R. Unland, Eds. Lecture Notes in Computer Science, vol. 3671. Springer, 144–161.
- AFANASIEV, L. AND MARX, M. 2006. An analysis of the current xquery benchmarks. In *ExpDB*. 9–20.
- ALTSCHUL, S. F., GISH, W., MILLER, W., MYERS, E. W., AND LIPMAN, D. J. 1990. Basic local alignment search tool. *J Mol Biol* 215, 3 (October), 403–410.
- BARBOSA, D., BARTA, A., MENDELZON, A. O., MIHAILA, G. A., RIZZOLO, F., AND RODRIGUEZ-GUIANOLLI, P. 2001. Tox - the toronto XML engine. In *Workshop on Information Integration on the Web*. 66–73.
- BEDATHUR, S. AND HARITSA, J. 2006. Search-optimized suffix-tree storage for biological applications. In *12th IEEE International Conference on High Performance Computing (HiPC)*, D. A. Bader, M. Parashar, S. Varadarajan, and V. K. Prasanna, Eds. Lecture Notes in Computer Science, vol. 3769. IEEE, Springer, Goa, India, 29–39.
- BEYER, K., COCHRANE, R. J., JOSIFOVSKI, V., KLEWEIN, J., LAPIS, G., LOHMAN, G., LYLE, B., OZCAN, F., PIRAHESH, H., SEEMANN, N., TRUONG, T., DER LINDEN, B. V., VICKERY, B., AND ZHANG, C. 2005. System rx: One part relational, one part xml. In *SIGMOD*.
- BHADKAMKAR, M., FARFAN, F., HRISTIDIS, V., AND RANGASWAMI, R. 2006. Efficient Native Storage for Semi-structured Data (extended paper version). In <http://www.cis.fiu.edu/SSS/NativeXMLextended.pdf>.
- BOHANNON, P., FREIRE, J., ROY, P., AND SIMÉON, J. 2002. From XML Schema to Relations: A Cost-based Approach to XML Storage. *ICDE*.
- BÖHME, T. AND RAHM, E. 2001. Xmach-1: A benchmark for xml data management. In *Datenbanksysteme in Büro, Technik und Wissenschaft (BTW), 9. GI-Fachtagung*. Springer-Verlag, London, UK, 264–273.
- BÖHME, T. AND RAHM, E. 2003. Multi-user evaluation of xml data management systems with xmach-1. In *Proceedings of the VLDB 2002 Workshop EEXTT and CAiSE 2002 Workshop DTWeb on Efficiency and Effectiveness of XML Tools and Techniques and Data Integration over the Web-Revised Papers*. Springer-Verlag, London, UK, 148–158.
- BRESSAN, S., DOBBIE, G., LACROIX, Z., LEE, M. L., LI, Y. G., NAMBIAR, U., AND WADHWA, B. XOO7: Applying OO7 benchmark to XML query processing tool. 167–174.
- BUCY, J., GANGER, G., AND CONTRIBUTORS. 2003. The DiskSim Simulation Environment Version 3.0 Reference Manual. *Carnegie Mellon University Technical Report CMU-CS-03-102*.
- CAREY, M., DEWITT, D., FRANKLIN, M., HALL, N., MCAULIFFE, M., NAUGHTON, J., SCHUH, D., SOLOMON, M., TAN, C. K., TSATALOS, O., WHITE, S., AND ZWILLING, M. 1994. Shoring up Persistent Applications. In *ACM SIGMOD*.
- CDA. 2007. HL7 Clinical Document Architecture, Release 2.0. In <http://lists.hl7.org/read/attachment/61225/1/CDA-doc-20version.pdf>. 2007.
- DELCHER, A., KASIF, S., FLEISCHMANN, R., PETERSON, J., WHITE, O., AND SALZBERG, S. 1999. Alignment of Whole Genomes. *Nucleic Acids Research* 27, 11, 2369–2376.
- DEUTSCH, A., FERNANDEZ, M. F., AND SUCIU, D. 1999. Storing Semistructured Data with STORED. *ACM SIGMOD*.

- DIMITRIJEVIC, Z., RANGASWAMI, R., CHANG, E., WATSON, D., AND ACHARYA, A. 2004. Diskbench: User-level Disk Feature Extraction Tool. *UCSB Technical Report TR-2004-18*.
- DOLIN, ALSCHULER, BOYER, BEEBE, BEHLEN, BIRON, AND SHVO, S. 2006. HL7 Clinical Document Architecture Release 2. *J Am Med Inform Assoc.* 13, 1 (Jan-Feb).
- DU, F., AMER-YAHIA, S., AND FREIRE, J. ShreX: Managing XML Documents in Relational Databases.
- FARFAN, F., HRISTIDIS, V., AND RANGASWAMI, R. 2007. Beyond lazy xml parsing. In *Proceedings of International Conference on Database and Expert Systems Applications (DEXA)*.
- FINKELSTEIN, A., JACOBS, C. E., AND SALESIN, D. H. 1996. Multiresolution Video. *Proceedings of SIGGRAPH*, 281–290.
- FRANCESCHET, M. 2004. XPathMark: An XPath Benchmark For XMark. *University of Amsterdam Technical Report PP-2004-04*.
- FRANCESCHET, M. 2005. *XPathMark: An XPath Benchmark for the XMark Generated Data*.
- GALAX. 2007. Galax. <http://www.galaxquery.org>.
- GANGER, G. R. 2001. Blurring the Line Between OSes and Storage Devices. *Carnegie Mellon University Technical Report CMU-CS-01-166*.
- GIBSON, G. A., NAGLE, D. F., AMIRI, K., BUTLER, J., CHANG, F. W., GOBIOFF, H., HARDIN, C., RIEDEL, E., ROCHBERG, D., AND ZELENKA, J. A Cost-Effective, High-Bandwidth Storage Architecture. *Proceedings of the ACM ASPLOS*.
- GML. 2008. Geography markup language. <http://opengis.net/gml/>.
- GOTTLÖB, G., KOCH, C., AND PICHLER, R. 2002. Efficient Algorithms for Processing XPath Queries. In *VLDB*.
- HL7. 2008. Health level seven xml. <http://www.hl7.org/special/Committees/xml/xml.htm>.
- HUSTON, L., SUKTHANKAR, R., WICKREMESINGHE, R., SATYANARAYANAN, M., GANGER, G. R., RIEDEL, E., AND AILAMAKI, A. 2004. Diamond: A Storage Architecture for Early Discard in Interactive Search. *Proceedings of the USENIX Conference on File and Storage Technologies*.
- IYER, S. AND DRUSCHEL, P. 2001. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous i/o. In *Symposium on Operating Systems Principles*. 117–130.
- JAGADISH, H. V., AL-KHALIFA, S., CHAPMAN, A., LAKSHMANAN, L. V. S., NIERMAN, A., PAPPARIZOS, S., PATEL, J. M., SRIVASTAVA, D., WIWATWATTANA, N., WU, Y., AND YU, C. 2002. TIMBER: A Native XML Database. *The VLDB Journal* 11, 4, 274–291.
- KANNE, C. AND MOERKOTTE, G. 1999. Efficient Storage of XML Data. *Universitaet Mannheim Technical Report*.
- KANNE, C.-C., BRANTNER, M., AND MOERKOTTE, G. 2005. Cost-Sensitive Reordering of Navigational Primitives. *SIGMOD*.
- KANNE, C.-C. AND MOERKOTTE, G. 2006. A Linear Time Algorithm for Optimal Tree Sibling Partitioning and Approximation Algorithms in Natix. In *VLDB*.
- KAUSHIK, R., BOHANNON, P., NAUGHTON, J. F., AND KORTH, H. F. 2002. Covering Indexes for Branching Path Queries. *SIGMOD*.
- KEETON, K., PATTERSON, D. A., AND HELLERSTEIN, J. M. 1998. A Case for Intelligent Disks (IDISKS). *SIGMOD Record* 27, 3 (September), 42–52.
- KUNDU, S. AND MISRA, J. 1977. A Linear Tree Partition Algorithm. In *SIAM J. Comput.* 6(1):151–154.
- LI, Q. AND MOON, B. 2001. Indexing and Querying XML Data for Regular Path Expressions. *VLDB Journal*.
- MANOLESCU, I., MIACHON, C., AND MICHIELS, P. 2006. Towards micro-benchmarking xquery. In *ExpDB*. 28–39.
- McHUGH, J., ABITEBOUL, S., GOLDMAN, R., QUASS, D., AND WIDOM, J. 1997. Lore: A database management system for semistructured data. *SIGMOD Record* 26, 3, 54–66.
- MENG, X., LUO, D., LEE, M.-L., AND AN, J. 2003. Orientstore: A schema based native xml storage system. In *VLDB*. 1057–1060.

- MERGEN, S. L. S. AND HEUSER, C. A. 2004. Matching of XML Schemas and Relational Schemas. In *SBBD*.
- MML. 2008. Medical Markup Language. <http://www.ncbi.nlm.nih.gov/>.
- NAMBIAR, U., LACROIX, Z., BRESSAN, S., LEE, M., AND LI, Y. 2001. Xml benchmarks put to the test.
- NEKRESTYANOV, I., NOVIKOV, B., AND PAVLOVA, E. 2000. An analysis of alternative methods for storing semistructured data in relations. In *ADBIS-DASFAA*. 354–361.
- NICOLA, M. AND JOHN, J. 2003. Xml parsing: A threat to database performance. In *Proceedings of the ACM International Conference on Information and Knowledge Management (CIKM)*. 175–178.
- NICOLA, M., KOGAN, I., AND SCHIEFER, B. 2007. An xml transaction processing benchmark. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. ACM, New York, NY, USA, 937–948.
- NOGA, M. L., SCHOTT, S., AND LOWE, W. 2002. Lazy xml processing. In *Proceedings of the ACM Symposium on Document Engineering*. 88–94.
- ODS. 2008. Open Document Specification v1.0. <http://www.oasis-open.org/committees/download.php/12572/OpenDocument-v1.0-os.pdf>.
- OOX. 2008. Openoffice xml file format v1.0.
- PAPAKONSTANTINOY, Y., GARCIA-MOLINA, H., AND WIDOM, J. 1995. Object Exchange Across Heterogeneous Information Sources. In *ICDE '95: Proceedings of the Eleventh International Conference on Data Engineering*.
- RAMANATH, M., FREIRE, J., HARITSA, J., AND ROY, P. Searching for efficient XML to relational mappings.
- RIEDEL, E., GIBSON, G., AND FALOUTSOS, C. 1998. Active Storage For Large-Scale Data Mining and Multimedia. *Proceedings of the VLDB*.
- ROKHSAR, D. 2007. Computational Analysis of Genomic Sequence Data. http://www.nersc.gov/news/annual_reports/annrep01/sh_BER_06.html.
- RUEMLER, C. AND WILKES, J. 1994. An introduction to disk drive modeling. *Computer* 27, 3, 17–28.
- RUNAPONGSA, K., PATEL, J., JAGADISH, H., CHEN, Y., AND AL-KHALIFA, S. 2003. The michigan benchmark: Towards xml query performance diagnostics.
- SCHINDLER, J., SCHLOSSER, S. W., SHAO, M., AILAMAKI, A., AND GANGER, G. R. 2004. Atropos: A Disk Array Volume Manager for Orchestrated Use of Disks. *Proceedings of the USENIX Conference on File and Storage Technologies*.
- SCHLOSSER, S. W., SCHINDLER, J., PAPADOMANOLAKIS, S., SHAO, M., AILAMAKI, A., FALOUTSOS, C., AND GANGER, G. R. 2005. On Multidimensional Data and Modern Disks. *Proceedings of the 4th USENIX Conference on File and Storage Technology*.
- SCHMIDT, A., WAAS, F., KERSTEN, M., CAREY, M., MANOLESCU, I., AND BUSSE, R. 2002a. Xmark: A benchmark for xml data management.
- SCHMIDT, A., WAAS, F., KERSTEN, M. L., CAREY, M. J., MANOLESCU, I., AND BUSSE, R. 2002b. XMark: A Benchmark for XML Data Management. *VLDB*.
- SHANMUGASUNDARAM, J., TUFTE, K., ZHANG, C., HE, G., DEWITT, D. J., AND NAUGHTON, J. F. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *VLDB 1999*.
- SIVATHANU, M., PRABHAKARAN, V., POPOVICI, F. I., DENEHY, T. E., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. 2003. Semantically-Smart Disk Systems. *Proceedings of the USENIX Symposium on File and Storage Technologies*, 73–88.
- SVG. 2008. Scalable vector graphics. <http://www.w3.org/Graphics/SVG/>.
- TALAGALA, N., ARPACI-DUSSEAU, R. H., AND PATTERSON, D. 1999. Microbenchmark-based Extraction of Local and Global Disk Characteristics. *UC Berkeley Technical Report*.
- WORTHINGTON, B., GANGER, G., PATT, Y., AND WILKES, J. 1995. Online Extraction of SCSI Disk Drive Parameters. *Proceedings of ACM Sigmetrics Conference*, 146–156.
- XALAN. 2007. Xalan-Java. <http://xml.apache.org/xalan-j>.
- ACM Transactions on Storage, Vol. V, No. N, Month 20YY.

- XPATH. 2007. XML Path Language (XPath) Version 1.0. <http://www.w3.org/TR/xpath>.
- XT. 2007. XT. <http://www.blz.com/xt/index.html>.
- YAO, B. B., ÖZSU, M. T., AND KEENLEYSIDE, J. 2003. Xbench - a family of benchmarks for xml dbms. In *Proceedings of the VLDB 2002 Workshop EEXTT and CAiSE 2002 Workshop DTWeb on Efficiency and Effectiveness of XML Tools and Techniques and Data Integration over the Web-Revised Papers*. Springer-Verlag, London, UK, 162–164.