Florida International University Technical Report **TR-2005-04-01**

Efficient Native XML Storage

Medha Bhadkamkar **Vagelis Hristidis** **Raju Rangaswami**

School of Computer Science
Florida International University
Miami, FL 33199

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

Efficient Native XML Storage

Medha Bhadkamkar Vagelis Hristidis Raju Rangaswami
School of Computer Science
Florida International University
{medha,vagelis,raju}@cs.fiu.edu

ABSTRACT

XML has emerged as one of the popular data-representation formats for information storage and exchange. XML data today range from representing small files to encapsulating gigabytes of information. Large XML databases must be stored on mass storage devices for both persistence as well as cost-efficiency. For mass storage of data today, disk drives are the most cost-effective medium. Current approaches of mapping XML data to relational databases or simply using flat files incur a mismatch between the structure of XML data and the underlying storage device (disk drives). In this study, we investigate a new method to store XML data on disk drives that matches the characteristics of XML with those of disk drives. In particular, we present algorithms that, given an XML document and a disk drive, decide how to store the document on the drive, in a way that will later allow efficient execution of XML queries. We evaluate our proposed method using analytical modeling and by simulating the execution of benchmark XPath queries.

1. INTRODUCTION

The popularity of XML data has increased in recent years, and so did the efforts to develop XML processing systems (e.g., Galax[1], Timber[2], XALAN[3], and XT[4]). Although much work has been conducted on optimizing the evaluation of XML queries (e.g., [1, 13]), little work has tackled the problem of efficiently storing XML data. The problem of efficiently storing XML data on disk drives (the device of choice today for mass storage) has become critical as the amount of data stored in an XML database has increased to several gigabytes. Current approaches either map the XML data to an underlying relational database system [2, 5, 9, 23, 26, 27], or use the abstraction provided by a general-purpose object storage manager [4], or simply use flat files.

These storage schemes, however, ignore the specific characteristics of the XML data format as well as those of disk drives. In particular, XML has a *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 overheads. The performance of disk drives greatly depends on the I/O access pattern (orders of magnitude difference between sequential and random access times). To the best of our knowledge, there exists no data layout strategy that accounts for the structural mismatch between XML and disk drive storage.

[1]http://www.galaxquery.org.
[2]http://www.eecs.umich.edu/db/timber/.
[3]Xalan-Java is an XSLT processor for transforming XML documents (http://xml.apache.org/xalan-j).
[4]XT is an implementation of XSLT for Java (http://www.blnz.com/xt/index.html).

In this paper, we propose ways to optimize the storage (placement) and retrieval of XML data on disk drives by explicitly accounting for the mismatch between tree-structured XML data and disk drive characteristics. In particular, we present algorithms that given the physical characteristics of a disk drive (number of tracks, rotational speed, seek time, etc.), place XML data on the disk drive in a way that facilitates efficient execution of XML queries by reducing the disk access overhead. We exploit the idea of semi-sequential access [21] to place the XML data such that common navigation operations (parent to child and node to next sibling) are efficient, which in turn allows efficient querying.

To evaluate our work, which optimizes the navigation of XML documents, we consider XPath[5] queries, since XPath is the core navigation component of XQuery[6]. Notice that previous work on indexing of XML data [12, 14, 15] is orthogonal to our XML data placement approach, since we can leverage these indexes if available. To simplify presentation, we assume that no indexes are available for comparison purposes.

The baseline storage strategy that we compare our approach against is sequential layout (called *default* storage strategy from now on) of the XML file on the disk, as provided by general-purpose filesystems. Furthermore, we use the terms storage, layout and placement interchangeably. In this paper, we make the following contributions:

1. We present a new data layout strategy for XML data on disk drives, along with algorithms to implement it.

2. We create a model to analytically evaluate the performance of our strategy compared to the default approach. We explain how the characteristics of the hard disk (seek time, rotational speed) and of the XML document (height and width) affect the performance of our approach.

3. We compare the performance of benchmark [10] XPath query executions when the XML document is placed by our approach versus the default approach.

1.1 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 [2, 5, 9, 23, 26, 27] has been the most popular due to the success and maturity of the relational DBMSs. The latter approach [16, 17] is based on storing the XML document as a flat file and building separate indexes on top.

The only work that tackles the problem of storing XML data without using a DBMS or indexes is by Kanne and Moerkotte [3], in which 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

[5]http://www.w3.org/TR/xpath.
[6]http://www.w3.org/TR/xquery/.

number of blocks read to traverse the tree. However, this method ignores 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 like seek-time and rotational-delay. Finally, Atropos [21] is a system that exploits the physical properties of disk drives and uses semi-sequential accesses to store relational databases. Our work is different because we deal with XML data that has a tree (or graph)-like structure, which is more complex than relational tables. 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. Using a similar approach as ours, Semi-preemptible I/O [7] uses low-level drive information obtained from Diskbench [8] (described in Section 2) to make traditionally non-preemptible disk I/Os preemptible by issuing rotationally-optimal I/Os.

2. BACKGROUND ON DISK DRIVES

In this section, we present a brief overview of how disk drives are significantly different on the inside from the *logical block* interface that they export. We also briefly present earlier work on profiling disk drives [8] that enables us to extract detailed information about the disk device, typically unavailable using the existing interface.

Accessing data on a disk drive consists of three time components: seek-time, during which the disk arm moves from the current cylinder to the target cylinder, rotational-delay, during which the disk waits for the target sector to rotate and appear below the disk head, and transfer-time, during which data is read from or written to the disk platter. The seek-time depends only on the distance between the current cylinder and the target cylinder, but is not a linear function. The rotational-delay depends on the RPM of the disk (which is fairly constant, varying less than 0.5%) and the angular distance of the target sector from the sector on which the disk head lands after the seek operation. The transfer-time of the disk depends on the RPM as well as the recording density of information on the disk zone[7].

Modern disk drives provide a logical block abstraction that makes them appear as a linear sequence of logical blocks to the operating system. This interface was introduced to serve two purposes: (a) free the operating system from dealing with low-level disk drive specific operation and management, and (b) allow disk manufacturers to innovate and optimize behind the interface. However, such an interface also disables the operating system and upper layers from obtaining accurate information about physical data layout, drive characteristics, operation semantics, and internal drive optimizations.

For XML as well as other data types which are accessed mostly non-sequentially, it is necessary to re-examine the existing interface. Of course changing or augmenting the existing block interface requires industry wide consensus and standardization effort which is practically infeasible within a short span of time. However, we can employ alternative techniques to get around the interface restriction [8, 20]. Diskbench [8], a disk profiling tool developed earlier, enables a richer access interface to disk drives by extracting accurate performance characteristics for disk drives. Profiled information includes:

[7]Multi-zone drives are the norm today, where each zone has different number of sectors-per-track. The interested reader is referred to [8] for a detailed discussion on zoning as well as other details of disk operation.

```
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 2: Standard XPath evaluation strategy.

rotational time, seek time, track and cylinder skew times, sizes of read cache and write buffer along with prefetching and buffering techniques, logical to physical block mappings, and access time prediction. With this profiled information, Diskbench allows us to control disk operations accurately and tailor it to specific data and application requirements, which in this case is tree-structured XML data.

3. XML TREE STORAGE

In this section, we first (Section 3.1) present the XML document abstraction and the XPath execution strategy used in this work, along with a set of assumptions we make on the operation of the hard disk. Then, Section 3.2 presents our basic placement strategy. Finally, Sections 3.2.1, 3.2.2 and 3.3 discuss variations and improvements of the basic strategy.

3.1 Data Model and Assumptions

XML data: We view an XML 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. We assume that there are no ID-IDREF edges[8] (which would make the tree a graph). Figure 1 shows an example of an XML document.

In this work we assume the existence of no indexes on the XML data. However, 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, because otherwise in the default storage method we would have to read the whole subtree of a node v to access the right sibling of v. Furthermore, we assume that there are no updates on the XML data. (Notice that the problem of updates is even harder for the default storage strategy.)

XPath: We use XPath queries to evaluate our storage strategy. We adopt the "standard" XPath evaluation method (Figure 3.1), described in [13], which is used (with slight modifications) to the best of our knowledge by popular XPath engines like XALAN and XT. Intuitively, the algorithm of Figure 3.1 processes an XPath query Q in a breadth first manner on the XML document, one step of Q ($Q.first$) at a time, and stores the intermediate results in a set S.

Disk drive assumptions: We make the following assumptions, which we plan to relax in the future (see Section 6). First, we assume that the disk drive has a single platter and surface on which data is recorded[9]. Second, we assume that each node of the XML tree requires a disk block of storage[10].

[8]Although our methodology supports ID-IDREF edges in principle, their navigation has not been optimized.

[9]There is a straightforward extensions to multiple surfaces and platters, although doing it in an optimal manner is an open research issue.

[10]Although this assumption seems unrealistic at face-value,

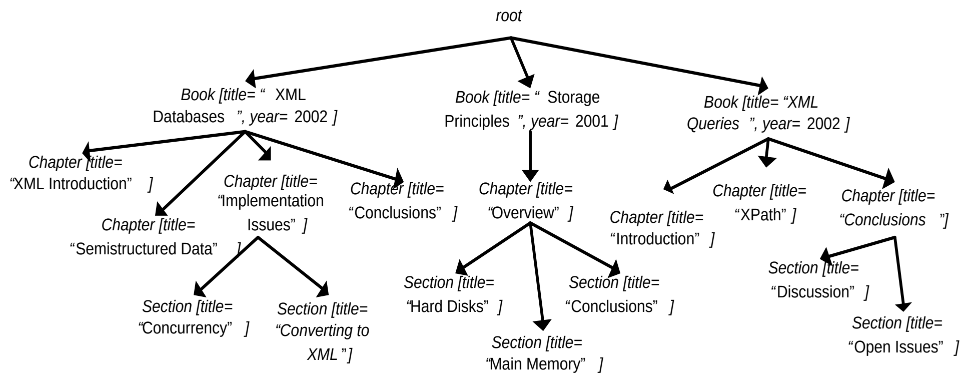
root
Book [title= " XML Databases ", year= 2002]
Book [title= " Storage Principles ", year= 2001]
Book [title= "XML Queries ", year= 2002]
Chapter [title= "XML Introduction"]
Chapter [title= "Semistructured Data"]
Chapter [title= "Implementation Issues"]
Chapter [title= "Conclusions"]
Chapter [title= "Overview"]
Chapter [title= "Introduction"]
Chapter [title= "XPath"]
Chapter [title= "Conclusions "]
Section [title= "Concurrency"]
Section [title= "Converting to XML "]
Section [title= "Hard Disks"]
Section [title= "Conclusions"]
Section [title= "Main Memory"]
Section [title= "Discussion"]
Section [title= "Open Issues"]

Figure 1: Sample XML document.

Third, we ignore OS-level I/O optimizations such as prefetching [18, 28] or retrieving "empty" pages to reduce I/O overhead [25]. These assumptions and optimizations, we believe, are orthogonal, and will apply equally to all the alternative placement strategies.

3.2 Tree-structured Placement

The limitation of the default storage method is that it is optimized only for accesses in depth-first order (notice that since the siblings in an XML tree are ordered, the left siblings are accessed first). For example, for the XML tree in Figure 3 (created by replacing the labels with node ids in the XML tree of Figure 1), 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.

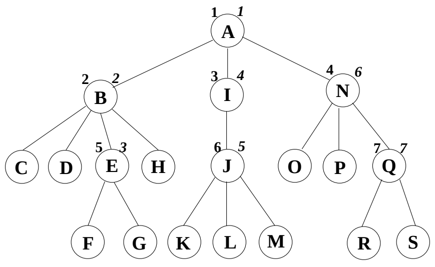

Figure 3: A Sample XML tree.

However, answering XPath 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. The default layout of the nodes would translate to random accesses (and therefore poor I/O performance) for both the above characteristics (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*. We consider breadth-first order to describe the tree-structured placement strategy, although other orders can be used instead as we discuss in Section 3.2.1. Once the ordering (numbering) of the nodes has been determined, they are placed on the disk starting from the outermost available track. In particular, we first place the root node v 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 [21]. This is accomplished by choosing a sector for u rotationally skewed from v such that the rotational delay between the sectors is equal to the seek time from the track of v to the track of u. Accessing any child of a parent node involves a semi-sequential access to reach the first child and a rotational-delay based on the child index.

strategies for node grouping such as those presented in [3] can be employed and are complementary to our work.

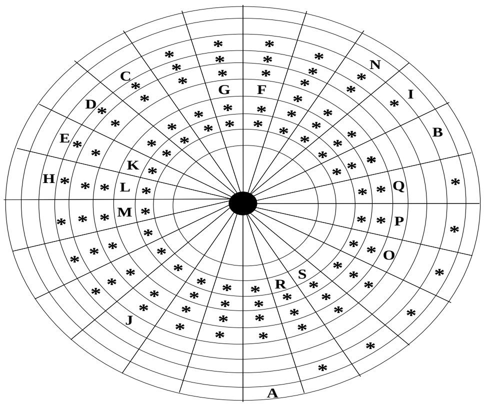

Figure 4: Tree-structured strategy for XML file layout.

Example 3.1 *Figure 4 shows the placement of the XML tree of Figure 3 on a hard disk (platter). For the sake of illustration, let us assume that tracks are numbered 0 to N, starting from the outermost track and that the platter rotates in the clockwise direction. Let us also 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 sector on which B is placed is rotationally skewed by 1 relative to A as a consequence of our assumption. Accessing B after A would require only seeking, which in this case would be 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 asterisk*ed *sectors in each track right before the first-child corresponds to the rotational skew between the parent node and the first-child.*

The tree-structured layout strategy, as is obvious in Figure 4, results in severe fragmentation of disk space. If the filesystem does not store other files to occupy the fragmented space, this strategy may be practically unusable. In Section 3.2.2, we present a variant of the tree-structured strategy that reduces fragmentation as well as random seek times drastically.

Figure 5 outlines the algorithm for tree-structured placement, along with the auxiliary methods used with their descriptions. Line 1 places the root node of the tree T on the outermost track. Lines 2-8 place on the next free track the children of the next node (that is, the node returned by `getNextNode()`, which is the root node in the first iteration). This is repeated until all the nodes are placed on the disk. Notice that the leaf nodes of T are not numbered and hence are not returned by `getNextNode()`.

Auxiliary Methods:

```
Node getNextNode() /* returns one node at a
                      time in ascending order */
Track getFirstFreeTrack()
placeInTrack(Track t,LBN lbnFirstChild,NodeList L)
LBN findSemiSequential(LBN parent, int cyl)
/* returns the logical block number (LBN) t,
such that t is cyl tracks away from parent and
the access from parent to t 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.   lbaFirstChild = findSemiSequential(n, t);
8.   placeIntrack(t,lbaFirstChild,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 = "XML Queries"]/Chapter[title = "XPath"] on the XML document of Figures 1 and 3. The following table shows the sequence of node accesses to answer this query using the evaluation strategy of Figure 3.1 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*

3.2.1 Node Ordering Variations

As mentioned earlier, there are multiple alternative strategies to determine the order in which nodes are chosen for placement. In the above example, we chose the breadth-first-ordering (*BFO*) of nodes. The BFO numbering (ignoring the leaf-nodes) for the XML tree in Figure 3, is illustrated as the left number above each node. Depth-first-ordering (DFO) is defined in the usual way as well and the numbering is shown (italicized) to the right of each node in Figure 3. As we show in Section 5, DFO results in drastically shorter average semi-sequential access times, due to the localization of the numberings for each subtree in DFO.

3.2.2 Handling Fragmentation and Random Seek

Using tree-structured placement, each track on the drive contains only the children of a single node. Therefore, this strategy, while reducing the access overhead for tree navigation operations, increases disk fragmentation. It also occupies much more tracks than the XML file-size warrants, as a result of which the random seek times within the tree are much larger than that for the default layout. We now present the *optimized tree-structured placement*, a variant of the original algorithm, that reduces both space fragmentation as well as random seek times drastically. The basic idea for the optimized layout 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 sector, but rather allows placing the first-child anywhere within a *rotationally-optimal track-region* (defined below).

To perform optimized placement, we use the fact that track-skew for adjacent tracks is usually between a quarter rotation to a sixth of a full rotation depending on the mechanical characteristics of the disk. Given the adjacent track-skew ts (ts can be obtained by profiling [8]), we can divide any track into $k = \lfloor 1/ts \rfloor$ equi-sized *track-regions.* Figure 6 shows the $k = 4$ track-regions separated by bold radial cuts for Track 0 (the outermost track).

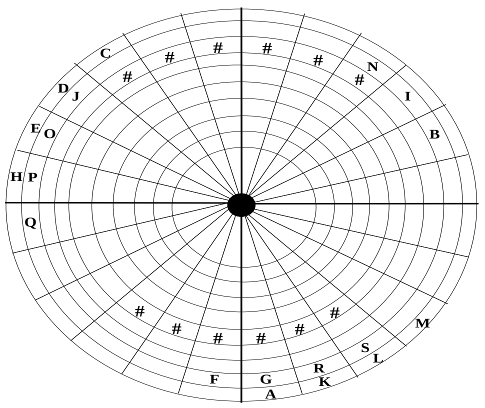

Figure 6: Optimized Strategy for XML file layout.

The optimized placement strategy is less restrictive than the basic tree-structured placement strategy in three ways: (1) it allows placing children on a *non-free track*, (2) it does not require the first-child to be placed at the rotationally-optimal sector, but rather allows placing the first-child anywhere within a rotationally-optimal track-region (defined below), and (3) it provides a choice of *candidate* rotationally-optimal track-regions on different tracks to place children nodes.

Given a parent node u, its *rotationally-optimal track-region* for a given track t is defined as the track-region starting from the sector where the disk head lands when seeking to track t starting from u. In Figure 6, two rotationally-optimal track-regions 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 placement algorithm chooses either the track-region closest to u or one that lies on the most fragmented (i.e., least occupied) track or a combination of these factors. The rationale behind choosing the least occupied track is that it will most likely result in providing the maximum number of candidate track-regions during future placement decisions.

Figure 6 shows the placement of the XML tree of 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 track-regions can be filled with children of various nodes, the space fragmentation is drastically reduced compared to the basic tree-structured placement. Figure 7 outlines the algorithm for optimized tree-structured placement.

Additional Auxiliary Methods:

```
<Track,LBN> findROTrackRegion(LBN parent)
/*returns the Track & Logical Block Number
(based on the offset in the rotationally-
optimal track-region) relative to parent. */
```

Optimized Tree-structured placement Algorithm:

```
1. placeInTrack(getFirstFreeTrack(),0,root(tree));
2. while (more nodes) {
3.   n = getNextNode();
4.   L = empty;
5.   L -> add(children(n));
6.   <t,lbaFirstChild> = findROTrackRegion(n.lba);
7.   placeInTrack(t,lbaFirstChild,L);}
```

Figure 7: Optimized tree-structured placement algorithm.

3.3 Other Optimizations

An alternative strategy to reduce the space fragmentation is the following. We group adjacent nodes of T into supernodes (similar to [3]), such that the sum of the supernode sizes of the children of any supernode is approximately equal to the track-size. As a result, there is no (or minimal) fragmentation of disk space. However, since the internal structure of supernodes is complex, query executions may result in accessing much more data than required. Hence, I/O performance is likely to deteriorate compared to the basic/optimized strategies. Finally, the root node could be placed on the central track (rather than the outermost track) and the subtrees are distributed in either direction, to reduce the maximum seek distance during semi-sequential access. We will evaluate the above two optimizations in our future work.

4. QUANTITATIVE ANALYSIS

In this section, we present a quantitative model to analyze the access times for the default placement and for our tree-structured placement. Table 1 summarizes the description of each parameter used in this analysis.

$T_{default}$: 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 node of data

Table 1: Parameter Description

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:

$$t_{rand} = \gamma\ (\frac{1}{3}\ C) + \frac{1}{2}\ T_{rot} \tag{1}$$

where $\gamma()$ is a disk specific function computing the seek time given the number of tracks to jump.

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

$$t_{seq} = 0 \tag{2}$$

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:

$$t_{semi-seq}(v) = \gamma(s(v)) \tag{3}$$

where $s(v)$ is the number of tracks that are jumped and is computed in 4.1

Equation 3 assumes perfect semi-sequential time, which is achieved by the tree-structured algorithm (Figure 5). However, in the case of the optimized tree-structured algorithm (Figure 7), $t_{semi-seq}(v)$ depends on the number of regions, k.

Theorem 4.1 *In the optimized tree-structured placement,* $t_{semi-seq}(v) = \gamma(s(v)) + \frac{1}{2k}T_{rot}$.

Proof Sketch: 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.

Next, we discuss the time needed for each of the 5 basic access types of Table 1. 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 ($f/2$) 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), its 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 2 summarizes the access times in the default and the tree-structured storage for every a_i.

Access type a_i	$t_{default}(a_i)$	$t_{tree}(a_i)$
a_1	t_{seq}	$t_{semi-seq}$
a_2	t_{rand}	$t_{semi-seq} + \frac{f}{2}(T_{nt})$
a_3	t_{rand}	t_{seq}
a_4	t_{seq}	t_{seq}
a_5	t_{rand}	t_{rand}

Table 2: Average access times in default and tree-structured placement for each access type a_i.

The average access times in default and tree-structured storage are computed by Equations 4 and 5 respectively.

$$T_{default} = \sum_{i=1}^{5} P_i \cdot t_{default}(a_i) \tag{4}$$

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

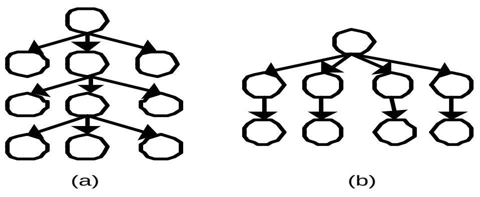

Figure 8: Best and worst case for both BFS and DFS.

The tree-structured placement is preferable when $T_{tree} < T_{default}$.

4.1 Semi-sequential Access Time Analysis

This section presents an analytical calculation of the semi-sequential access time $t_{semi-seq}$ for the tree-structured placement. In particular, we provide analytical formulas for the number of tracks, $s(v)$, between a node v and its first child, which is related to $t_{semi-seq}$ through the disk's seek curve, i.e., $t_{semi-seq}(v) = \gamma(s(v))$.

The calculation of $s(v)$ depends on the numbering scheme we use, that is, on the implementation of the $getNextNode()$. We focus on the two alternative numbering schemes described in Section 3.2.1: breadth-first (BFO) and depth first (DFO). Equation 6 shows the calculation of $s(v)$ for both BFO and DFO numbering.

$$s(v) = N(v) - N(parent(v)) \tag{6}$$

where $N(v)$ is the number of v, that is, after how many calls v is returned by $getNextNode()$, and $parent(v)$ returns the parent node of v. If v is the root node of T, then $s(v) = 0$.

The average $s(v)$ depends on the structure of T. The best case for both BFO and DFO is when only one node from any list of siblings has children (Figure 8 (a)). In this case $s(v) = 1$ for any v for both BFO and DFO. The other extreme (worst case) is when T is wide and short and specifically when the tree has height 3 (Figure 8 (b)). In this case, the average $s(v)$ is $(n' - 1)/2$, where n' is the number of internal nodes (non-leaf) of T.

Finally, we discuss the case of complete trees which have an average $s(v)$ between the two extremes. Theorems 4.2 and 4.3 show the average values for BFO and DFO respectively.

Theorem 4.2 *For BFO, when T is a complete tree with height d and degree f, the average $s(v)$ is $\frac{\sum_{i=1}^{n'}(i-round(i/f))}{n'}$, where $n' = (1 - f^{d-1})/(1 - f)$ is the number of internal nodes.*

Proof Sketch: $i - round(i/f)$ is the BFO numbering difference between a child and its parent.

Theorem 4.3 *For DFO, when T is a complete tree with height d and degree f, the average $s(v)$ is $\frac{f^{d-2}(d-2-f/(1-f))+2+f/(1-f)}{2n'}$, where $n' = (1 - f^{d-1})/(1 - f)$ is the number of internal nodes.*

Proof Sketch: Notice that we assume than root is at depth 1 and the leaves at depth d. First notice that if there is two edges u1-v1 and u2-v2 where u1, u2 are on the same level and v1 (v2) is the l-th child of u1 (u2), then $DFO(v1) - DFO(u1) = DFO(v2) - DFO(u2)$, that is, the jumps from v1 to his childen and from v2 to his children are the same.

Second, we calculate the average $s(v)$ for the nodes v of level $k+1$. To do so we need to find the size of the subtree rooted at v. It is $1+f+\cdots+f^{d-k-1} = (1-f^{d-k})/(1-f)$. 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, $(f+(1-f^{d-k})/(1-f)(1+\cdots+(f-1)))/f = (f+(1-f^{d-k})/(1-f)(f-1)f/2)/f = (f+(f^{d-k}-1)f/2)/f = 1+(f^{d-k}-1)/2 = (f^{d-k}+1)/2$. Hence, for level k it is $(f^{d-k-1}+1)/2$.

Query#	P_1	P_2	P_3	P_4	P_5
1	0.21	0.00	0.17	0.48	0.14
2	0.28	0.00	0.19	0.36	0.17
3	0.11	0.00	0.21	0.44	0.24
4	0.11	0.00	0.21	0.44	0.24
5	0.34	0.00	0.20	0.30	0.16

Table 3: Access probabilities for XPathMark queries.

5. EXPERIMENTS

This section evaluates the suitability of our approach for placing XML data. To do so, we have developed an analytical cost model (Appendix B) to compare the performance of tree-structured placement with the default placement strategy (placing an XML file sequentially on disk) when the following parameters change: the structure of the XML tree, the node numbering (DFO vs. BFO), and the distribution of the basic tree navigation operations (e.g., parent to first child). We use this cost model to derive our results in this section. However, our analysis here is only a yardstick and by no means a complete evaluation. A thorough evaluation would involve experimentation using both trace-driven simulations as well as an actual extension of the filesystem that places XML files as an on-disk tree structure. Section 6 outlines some initial steps that we have taken in these directions.

XPath benchmark queries: The analysis of Section 4 derives the disk access time for answering an XPath query, given the access probabilities $P_1, \ldots, P_5$ (P_1 is parent to first child, P_2 is parent to non-first child, P_3 is non-leaf node to right sibling, P_4 is leaf node to right sibling, P_5 is the rest accesses). To get a sense of the distribution of these probabilities we calculated them using the XML documents and XPath queries defined in the XPathMark benchmark [10], and the XPath evaluation algorithm described in Figure 3.1. Table 3 shows these probabilities for the first 5 queries for the first document of the benchmark (the document is generated using XMark [24]).

Figure 9 compares the average access times for the default, the optimized BFO tree-structured and the optimized DFO tree-structured placements for the five benchmark queries. The size of the XML tree (T) used is $n = 1,000,000$ nodes and the average fanout of nodes is 10. The disk fragmentation due to the tree-structured placement strategy is assumed to be 50%. In practice, we expect this number to be much smaller. Notice that for all placement methods the same total number of node accesses are required. Hence, the differences in average access times reflect the differences in the total execution times as well. The DFO tree-structured placement always performs better than both the BFO tree-structured and the default placement. The intuition why DFO is better than BFO is that when storing a specific subtree of the XML tree T, the average seek-distance from a parent node to its first-child ($s(v)$), does not depend on the rest of T, whereas for BFO, $s(v)$ always increases when going deeper in T. The other observation from Figure 9 is that the tree-structured placement has the highest performance advantage for Queries 3 and 4 since they have the highest P_3/P_1 ratio. P_3 is the strong point for tree-structured placement since the access is

depth	*worst*	*best*	*compl − DFO*	*compl − BFO*
3	250000	1	1	500
4	undef	1	1.48	5000
7	undef	1	2.73	50000
10	undef	1	3.48	87350
20	undef	1	4.98	131100

Table 4: Average seek-distance in cylinders for semi-sequential accesses for various XML trees.

sequential instead of random-access for default placement. On the other hand, P_1 is the strong point for default placement since it's access is sequential instead of semi-sequential in tree-structured placement. A more detailed analysis is available in Section 4.

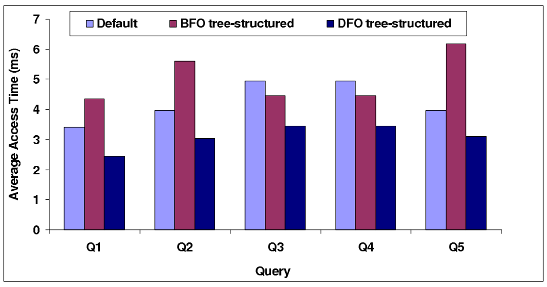

Figure 9: Average access time for XML queries.

Effect of tree-structure on access time: In this experiment, we fix the number of nodes in the XML tree T to $n = 1,000,000$. We then plot the average seek-distance, $s(v)$, for a semi-sequential access (averaged over all possible parent-to-first-child accesses in the XML tree T) for XML trees with varying depth (and consequently varying fanout). In particular, we consider the following types of trees: (a) the best case tree (See Section 4), (b) the worst case tree, (c) a complete tree where BFO numbering is used, and (d) a complete tree with DFO placement strategy. The results, which are shown in Table 4, clearly show the superiority of the DFO numbering.

Effect of drive characteristics on access time: Since the proposed data placement scheme is directly related to the drive characteristics, we simulate a range of drive technologies, similarly to [19], by varying rotational period, seek-time characteristics, and data transfer-rate. We use the 7200 RPM Western Digital WD400JD [29] as the base disk, which has $C = 16,383$ cylinders. We also assume the existence of a fast-seeking drive, which cuts down seek times by a factor of two, while a slow-seeking one increases it by a factor of 1.5. Our fast-rotating and slow-rotating drives spin at 10,000 RPM and 5,400 RPM respectively. These numbers are presented in Table 5.

Configuration	T_{rot}	**Seek [ms]**			**R**
	[ms]	*Track switch*	*Avg. Seek*	*Full stroke*	$[\frac{MB}{s}]$
1. Base	8.3	2.0	8.9	21	100
2. Fast seek	8.3	1.0	4.45	10.5	100
3. Slow seek	8.3	3.0	13.35	31.5	100
4. Fast rotate	6.0	2.0	8.9	21	139
5. Slow rotate	11.1	2.0	8.9	21	75
6. Fast seek+rot	6.0	1.0	4.45	10.5	139
7. Slow seek+rot	11.1	3.0	13.35	31.5	75

Table 5: Characteristics of disk configurations used in the experiments.

Figure 10 shows the average access times for the disk configurations of Table 5, for the default, the BFO tree-structured and the DFO tree-structured placement strategies. Again, we assume that tree-structured placement strategies incur 50% disk fragmentation. We consider the access probabilities of the first query of Table 3. First, we notice that with 50% fragmentation, BFO access times are worse than the default (i.e. sequential layout) strategy, while DFO performs consistently better for all disk configurations. For the base configuration (i.e., Disk1), the improvement in average access time with DFO is approximately 30%. For Disk2 and Disk5, rotational-delays are relatively higher than seek-times when compared to the base configuration since these are fast-seeking and slow-rotating drives respectively. As a result, we notice that the tree-structured DFO strategy which cuts down on rotational-delays, performs even better than for the base configuration.

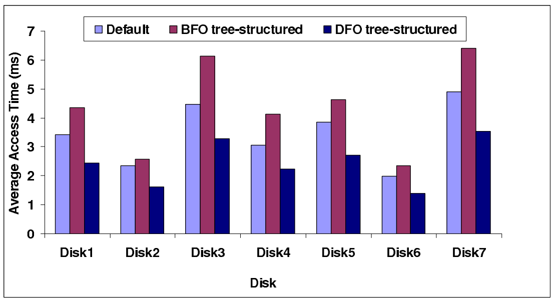

Figure 10: Average access time for various disk drives.

Effect of fragmentation on sequential access time: Finally, the average access time for the default, BFO tree-structured and DFO tree-structured strategies are plotted for varying disk fragmentation values (from 10% to 50%) as shown in Figure 11. In practice we expect disk fragmentation due to tree-structured placement to lie between 5% and 25%. However, this is only an intuitive estimate for now. We notice that with increased disk fragmentation, there is significant degradation in BFO access times and only a slight increase for DFO. At 10% fragmentation, DFO access times are more than 35% better than the default sequential layout of the XML file. This implies that the DFO strategy scales well with fragmentation and is a practical approach overall.

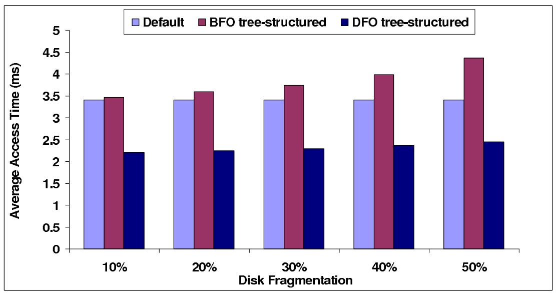

Figure 11: Average access time vs. Fragmentation.

6. CONCLUSIONS AND FUTURE WORK

We have presented a new on-disk placement strategy for tree-structured XML data. The new placement strategy explicitly accounts for the structural mismatch between XML data and disk devices. We first presented a basic placement strategy that improves the performance of common XML tree-navigation operations, but suffers from severe space fragmentation and large random seek-times. We proposed the optimized tree-structured layout to reduce space fragmentation as well as random seek-time. Preliminary experimental evaluation suggests that the optimized tree-structured placement strategy reduces the average access time by as much as 35% compared to the default (sequential) placement. We work on simulating our placement strategies with DiskSim [11]. In the future we also plan to extend an existing filesystem to support efficient access to tree-structured data. Our experience from past efforts on providing additional application control of the storage system [6] indicates that a similar approach for tree storage is feasible.

Furthermore, we plan to adapt the single-platter-surface placement strategies to multi-platter disks as well as to a RAID environment. The strategies presented here can also be adapted to new storage media like MEMS-based storage [22]. In addition, we plan to study the effect of disk prefetching in our placement strategy. Finally, we plan to work on handling updates within the tree-structured placement framework. The slight fragmentation incurred by our placement strategies can aid us in this pursuit.

7. REFERENCES

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

[2] P. Bohannon, J. Freire, P. Roy, and J. Siméon. From XML Schema to Relations: A Cost-based Approach to XML Storage. *ICDE*, 2002.

[3] G. M. C. Kanne. Efficient Storage of XML Data . *Universitaet Mannheim Technical Report*, 1999.

[4] 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.

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

[6] Z. Dimitrijevic and R. Rangaswami. Quality of Service Support for Real-time Storage Systems. *Proceedings of International IPSI Conference*, October 2003.

[7] Z. Dimitrijevic, R. Rangaswami, and E. Chang. Design and Implementation of Semi-preemptible IO. *Proceedings of Usenix FAST*, March 2003.

[8] Z. Dimitrijevic, R. Rangaswami, E. Chang, D. Watson, and A. Acharya. Diskbench: User-level disk feature extraction tool. *UCSB Technical Report.*, April 2004.

[9] D. Florescu and D. Kossmann. Storing and Querying XML Data using an RDBMS. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.

[10] M. Franceschet. XPathMark: An XPath Benchmark for XMark. In *http://staff.science.uva.nl/ francesc/xpathmark/index.html.*

[11] G. R. Ganger, B. L. Worthington, and Y. N. Patt. The DiskSim Simulation Environment Version 2.0 Reference Manual. *Reference Manual*, December 1999.

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

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

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

[15] A. Halverson, J. Burger, L. Galanis, A. Kini, R. Krishnamurthy, A. Rao, F. Tian, S. Viglas, Y. Wang, J. Naughton, and D. DeWitt. Efficient Algorithms for Processing XPath Queries. In *VLDB*, 2003.

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

[17] J. McHugh, J. Widom, S. Abiteboul, Q. Luo, and A. Rajaraman. Indexing Semistructured Data. *Stanford Technical Report*, 1999.

[18] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed Prefetching and Caching. In *Proc. of the 15th ACM Symp. on Operating System Principles*, December 1995.

[19] F. I. Popovici, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Robust, Portable I/O Scheduling with the Disk Mimic. *Proceedings of the USENIX Annual Technical Conference*, pages 297–310, June 2003.

[20] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *Computer*, 2:17–28, 1994.

[21] 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.

[22] S. W. Schlosser, J. L. Griffin, D. Nagle, and G. R. Ganger. Designing Computer Systems with MEMS-based Storage. In *Architectural Support for Programming Languages and Operating Systems*, pages 1–12, 2000.

[23] A. Schmidt, M. L. Kersten, M. Windhouwer, and F. Waas. Efficient Relational Storage and Retrieval of XML Documents. *WebDB*, 2001.

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

[25] B. Seeger, P.-A. Larson, and R. McFadyen. Reading a Set of Disk Pages. *In Proceedings of VLDB*, 1993.

[26] 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.

[27] I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and Querying Ordered XML using a Relational Database System. In *ACM SIGMOD*, 2002.

[28] P. Varman and R. Verma. Tight Bounds for Prefetching and Buffer Management Algorithms for Parallel I/O Systems. In *Foundations of Software Technology and Theoretical Computer Science*, Decemeber 1998.

[29] Western Digital Technologies Inc. WD Caviar SE Datasheet. *http://www.wdc.com/en/library/sata/2879-001081.pdf*, 2005.