7-28-2009

Improving Storage Performance Through Layout Optimizations

Medha Bhadkamkar
Florida International University, Medha.Bhadkamkar@fiu.edu

DOI: 10.25148/etd.FI09082405

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

IMPROVING STORAGE PERFORMANCE THROUGH LAYOUT OPTIMIZATIONS

A dissertation submitted in partial fulfillment of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Medha Bhadkamkar

2009

To: Dean Amir Mirmiran
College of Engineering and Computing

This dissertation, written by Medha Bhadkamkar, and entitled Improving storage performance through layout optimizations, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this dissertation and recommend that it be approved.

Evangelis Christidis

Giri Narasimhan

Jiri Schindler

Kaushik Dutta

Raju Rangaswami, Major Professor

Date of Defense: July 28, 2009

The dissertation of Medha Bhadkamkar is approved.

Dean Amir Mirmiran
College of Engineering and Computing

Dean George Walker
University Graduate School

Florida International University, 2009

DEDICATION

To my parents and my sister, who have always had faith in me.

ACKNOWLEDGMENTS

This dissertation is the result of many collaborations, help, support and motivation from many friends and colleagues.

No words can express my gratitude towards my adviser, Raju Rangaswami, who was always available and readily offered his help and support each time with every problem, research or personal, throughout my graduate career. This dissertation would not have been completed without his invaluable guidance and consistent motivation.

I am grateful to all my dissertation committee members, Vagelis Hristidis, Giri Narasimhan, Kaushik Dutta and Jiri Schindler for taking the time, effort and interest in my research. Their comments and suggestions have improved this dissertation considerably.

I would like to thank Jiri Schindler for also being my mentor and wishing the best for me. His suggestions not only on my dissertation but also on conducting computer science research have been invaluable.

I am thankful to Keith Smith for being an excellent mentor and for the time he always had for me. He motivated me to think independently and always showed his willingness to participate in discussions that shaped this thesis.

I am also indebted to my research partners Luis Useche, Jorge Guerra and Fernando Farfan for sharing their ideas and for all the brainstorming sessions. I would like to thank Luis and Jorge for contributing to the development, debugging and analysis of the results. I would also like to thank other members of the Database and Systems Lab, Ramakrishna Varadarajan, Sajib Kundu and Ricardo Koller for all the fun times we had.

I thank Nishad for all the love, laughter and the late night surprise meals around deadlines and for being with me throughout the course of my graduate study. And last be not the least, I am thankful for all my loving cats and dogs for their incessant love, for being good listeners and for waking me up every morning.

ABSTRACT OF THE DISSERTATION

IMPROVING STORAGE PERFORMANCE THROUGH LAYOUT OPTIMIZATIONS

by

Medha Bhadkamkar

Florida International University, 2009

Miami, Florida

Professor Raju Rangaswami, Major Professor

Disk drives are the bottleneck in the processing of large amounts of data used in almost all common applications. File systems attempt to reduce this by storing data sequentially on the disk drives, thereby reducing the access latencies. Although this strategy is useful when data is retrieved sequentially, the access patterns in real world workloads is not necessarily sequential and this mismatch results in storage I/O performance degradation. This thesis demonstrates that one way to improve the storage performance is to reorganize data on disk drives in the same way in which it is mostly accessed. We identify two classes of accesses: static, where access patterns do not change over the lifetime of the data and dynamic, where access patterns frequently change over short durations of time, and propose, implement and evaluate layout strategies for each of these. Our strategies are implemented in a way that they can be seamlessly integrated or removed from the system as desired. We evaluate our layout strategies for static policies using tree-structured XML data where accesses to the storage device are mostly of two kinds - parent-to-child or child-to-sibling. Our results show that for a specific class of deep-focused queries, the existing file system layout policy performs better by 5-54X. For the non-deep-focused queries, our native layout mechanism shows an improvement of 3-127X. To improve performance of the dynamic access patterns, we implement a self-optimizing storage system that performs rearranges popular block accesses on a dedicated partition based on the observed workload characteristics. Our evaluation shows an improvement of over 80% in the disk busy times over a range of workloads. These results show that applying the knowledge of data access patterns for allocation decisions can substantially improve the I/O performance.

Table of Contents

List of Tables

List of Figures

CHAPTER 1

INTRODUCTION

This chapter provides a synopsis of the thesis. It begins with the description of the storage I/O performance problem followed by an overview of our approach to alleviate it. It also states the contributions of this dissertation and provides a road-map to the rest of the thesis.

1.1 Problem Definition

This section describes why storage I/O performance is the bottleneck in present day computer systems.

I/O is the bottleneck

Almost all application workloads that are executed on servers or end-user systems require to store and retrieve data on a persistent, secondary storage device. Hard disk drives are the prevailing secondary storage media. However, accessing data from disk drives is many orders of magnitude slower than accessing data from the computers primary memory, RAM.

To compound this problem, there is a continual increase in the gap between CPU performance and disk drive performance. CPU performance increases at the rate of 60% per year while the increase in disk drive performance is limited to only 10% per year. Due to this gap, multiple requests from different applications queue at the storage controller waiting to be serviced. While the steady increase in main memory sizes attempts to bridge this gap, the impact is relatively small; Patterson et al. [**?**] have pointed out that disk drive capacities and workload data sizes tend to grow at a faster rate than main memory sizes. This leads to disk I/O performance bottleneck which becomes further restrictive with increased computational parallelism. Thus, there is an imperative need to optimize disk I/O operations which is only likely to become more important in the future as more available computing and storage capacity prompt running a larger number of more complex applications with increasingly larger working-set sizes, on servers as well as end-user systems.

Mismatch in access patterns and disk geometry

File systems are a fundamental component of computer systems – they control the storage and access to data. However, the basic file and directory layout techniques in file systems have not changed significantly since the early days of the Unix Fast File System [**?**]. Current file systems perform two key layout optimizations. First, they preserve the directory structure when mapping file system elements by allocating cylinder groups for storing directory sub-trees thereby attempting to reduce the seek latencies by co-locating related data. Second, for each file, they attempt to allocate sequential space on the disk drive to optimize for sequential file access. This is eliminates disk rotational overheads. This strategy works well when the workload access patterns are sequential. However, most workloads access data in a workload specific access patterns that are not necessarily sequential. Application disk access patterns are typically complex. Rather than accessing entire files sequentially, they may access multiple files partially and simultaneously, often in a specific sequence. Further, the data can be distributed over the entire disk space, especially for aged file systems, thereby increasing the seek overheads incurred in accessing it. This mismatch between the underlying storage and the workload access patterns result in increased disk latencies, which in turn degrade I/O performance. This behavior motivates the focus on accurately capturing and representing workload access patterns in as a key step towards optimized disk I/O performance.

Recent work has convincingly argued that the lack of intelligent use of data access patterns is a key shortcoming of current storage stack implementations [**?**, **?**, **?**, **?**]. These studies mostly advocate, reorganizing on-disk data layout based on observed access patterns. Recent work on improved prefetching [?, ?] or I/O scheduling algorithms [?] advocate complementary techniques. While these solutions can help reduce the number of I/O requests, good data layout can reduce or eliminate both seek as well as rotational-delay overheads.

In this dissertation we contend that disk I/O performance bottleneck can be reduced by decreasing the I/O response time by matching on-disk layout to the actual workload access pattern.

1.2 Approach

We derive popular block access patterns from workloads and perform block reorganizations such that on-disk layout favors these block access patterns and reduces the access time overheads. We broadly classify workload access patterns into two categories: *static*, where data is accessed in specific access patterns that does not vary with time, and *dynamic*, where access patterns tend to change over time. The following subsections explores each of these in more detail. We present the design and implementation of layout policies for each of these classes using the access patterns to make allocation decisions and evaluate our results using realistic workloads.

1.2.1 Static Access Patterns

Certain workloads exhibit deterministic access patterns that tend to remain static throughout the lifetime of the data stored on the disk. For instance, for tree-structured data such as XML documents, the storage device is typically accessed using a combination of two operations:

(i) parent node to child node access and

(ii) node-to-sibling access

Current approaches for storing such tree-structured data use existing storage mechanisms - 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 tree-structured data can result in a mismatch between the tree-structured data and the access characteristics of the underlying storage device (disk drive).

Tree-structured data is becoming increasingly popular due to its intuitive properties for storing and managing large amounts of data. It is used in numerous applications such as Bioinformatics sequence search and alignment [DKF+99], genomic data analysis [Rok01], multi-resolution video storage [FJS96], clinical data systems [CDA07], XML databases, and so on. Because of the popularity of the tree-structured data, the problem of optimizing access needs to be further explored.

This dissertation explores various possibilities in the design space of native storage solutions for tree-structured data by exploring alternative approaches that match application data access characteristics to those of the underlying disk drive. Because of their increasing popularity as a format for representing data in modern applications, we use tree-structured XML documents as our case study and use XPath queries to access the data in the XML documents to evaluate our layout strategies.

1.2.2 Dynamic Access Patterns

Many common workloads exhibit dynamic access patterns. For instance, a developer may access a certain set of files specific to a given project, which may no longer be used once she moves on to a different project.

Present day file systems employ a a mostly static layout of file system data with the defragmentation operation being the only dynamic aspect. These file systems implement static allocation policies mentioned above that do not take into account the dynamic characteristics of I/O workloads within its data management mechanisms. They are more optimized for sequential reads for entire files, but in the real world, file system accesses follow more complex patterns, which are not necessarily sequential accesses to co-located files on the storage system.

In this dissertation, we argue that a static layout of file system data leads to sub-optimal performance. More specifically, we contend that file systems can improve the system I/O performance by dynamically adapting file system layout based on how the file system is used by various applications.

Storage systems must self-optimize by adapting data layout to accommodate the dynamism in application access patterns without any manual intervention. They must be self-configuring and be able to determine the layout based on the observed workload patterns and must be self-managing where it should effectively adapt to the new layout.

In this dissertation, we propose a self-optimizing system that reorganizes blocks based on the access patterns to reduce the seek and rotational overheads. The block reorganizations occur online to avoid system downtime. Workload access patterns may change over time and to ensure these changes are effectively captured, the system does periodic reconfigurations. To provide a generic solution that can handle multiple, active file systems, our system is implemented at the block layer, independent of the file system above it. Self-optimizing storage systems are a significant deviation from current static-layout solutions and practical solutions must be able to fit into the existing storage stack architecture. Our system is minimally intrusive to ensure quick adoption and can co-exist with current optimizations.

1.3 Contributions

The main contributions of this dissertation are as follows:

- In this dissertation, we present the design, implementation and an extensive experimental evaluation of different layout strategies for workloads with either static or dynamic access patterns that reduce the access time latencies.

  While static access patterns remain deterministic, dynamic access patterns can constantly change over time and hence both these classes need to be handled differently. The static layout policies follow heuristics that optimize workload specific types of accesses, for instance parent-to-child and node-to-sibling for tree-structured data. The dynamic layout policies optimize the most popular block access patterns for the given workload for a specific time duration.

  The experimental evaluation measures the performance impact of the layout optimizations under under a variety of query access patterns and workloads for each of the proposed layout strategies. It also highlights the merits as well as the demerits of using the strategies and provides some insight on how they can be optimally used. Further, we also validate our strategies on different system configurations such as type of disks and the file systems.

- It identifies the ideal intelligence that can be used to implement the static and dynamic layout strategies and how it can be acquired inside the storage stack.

  Our static layout strategy uses information about the physical geometry of disk drives. Such low-level control of data layout is made possible using information provided by standard disk profiling tools [?, ?, ?]. For the self-optimizing, dynamic layout policy, we use different block, process and temporal attributes to obtain information about the access characteristics. This information is easily available at low profiling overheads and is sufficient to derive the popular workload block access patterns.

- It describes how the information obtained is used to improve storage performance with minimal changes to the existing storage stack.

  While the static access patterns can be derived from the queries used to access the data, the workload specific dynamic patterns can be obtained by analyzing traces collected while training the system. Further, our systems can be seamlessly integrated as another layer in the storage stack and can be easily disabled as necessary. The dynamic layout policy is implemented at the block layer which makes it independent of the file systems above it and of the storage devices below it.

- It explores the significant challenges and implementation issues involved in implementing such systems.

  We implemented our static layout strategy on a disk simulator and our dynamic layout policy as a module in the Linux kernel. We were faced with numerous challenges while integrating our changes with the existing system. This dissertation explores some of the most significant ones and also explains our approach in dealing with them.

1.4 Overview

The rest of the dissertation is organized as follows. Chapter 2 describes in detail native data-layout strategies for static workloads using tree-structured XML data as the case study. We implement the proposed approach using an instrumented DiskSim [?] disk-drive simulator, and evaluate it by comparing it against the default sequential layout. Chapter 3 provide the architecture and implementation details for a self-managing system called, BORG, that implements dynamic layout mechanism based on the observed workload characteristics. We implement this strategy in the Linux kernel and to evaluate the performance impact of our system we compare it with a vanilla system that implements the default data allocation policies of the file systems. Chapter 4 explains how workload characteristics can be used to make a choice of the best suited allocation mechanisms. Chapter 5 contains concluding remarks and future directions in which this work can be extended. Appendix A provides a brief description of disk drive storage mechanism.

CHAPTER 2

STATIC DATA ALLOCATION

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

2.1 Introduction

An increasing number of applications manage large amounts of tree-structured data. Common applications that use tree-structured data today include Bioinformatics sequence search and alignment [?], genomic data analysis [?], multi-resolution video storage [?], clinical data systems [?], XML databases, and more [?]. Given that a tree-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 tree-structured data either map the data to an underlying relational database system (e.g., [?, ?, ?, ?, ?]), use the abstraction provided by a general-purpose object storage manager [?], or use a combination of flat files and indices (e.g., XALAN [?], XT [?], Galax [?], BLAST [?], Timber [?] and Natix [?]). Since these approaches retrofit existing storage mechanisms to work with tree-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 tree-structured data and the access characteristics of disk drives. In particular, tree-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 tree-structured data, there is a need for re-examining the current storage and access machinery that support them.

In this Chapter, we explore strategies to optimize the storage and retrieval of tree-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 tree-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 [?, ?, ?].

The proposed technique first addresses the problem of grouping nodes of tree-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 [?]. 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 [?] 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. Prior 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 [?]. Our goal 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) [?]. Based on this study, we believe that our approach provides a fresh perspective on the problem of storing tree-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 study. XML is becoming increasingly popular due to its ability to represent arbitrary tree-structured data. It is the de facto data representation format for many modern applications, including Geographic Information Systems Markup Language (GML) [?], Medical Markup Language (MML) [?], Health Level HL7 [?], Clinical Document Architecture (CDA) [?] used to represent Electronic Health

Records (EHRs), Open Document Format (ODF) [?, ?], and Scalable Vector Graphics (SVG) [?] 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 [?] on the future directions on database research, published every few years by the database research community.

Table 2.1: 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 [?, ?, ?] show that all queries to XML data can be classified into deep-focused and non deep-focused queries. In Table 2.1, we summarize the key XML benchmarks available in the public domain. The Transaction Processing over XML (TPoX) benchmark [?] 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 [?, ?], XOO7 [?], XMark [?] and XPathMark [?] 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 [?] is an XPath based benchmark for XMark and generates an educational document that represents the English alphabet. The

XBench [?] benchmark is an application oriented benchmark for XML databases. Finally, the MemBer [?, ?] and the Michigan Benchmark (MBench) [?] 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 [?] 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 [?] 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 [?] 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 tree-structured data), the proposed native layout techniques outperform the default for all other query access patterns.

The rest of the Chapter is organized as follows. Section 2.2 presents the architecture of a native tree-structured storage system and the model used for tree-structured data and their access. In Section 2.3, we present native data-layout strategies for tree-structured data on disk drives. In Section 2.4, we present strategies for organizing and grouping nodes in the tree so that they can be mapped to disk blocks. In Section 2.5 we conduct a theoretical analysis of the performance impact of data layout. In Section 2.6, 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 2.7. We conclude and discuss future directions in Section 2.8.

2.2 System Architecture and Data Model

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

2.2.1 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

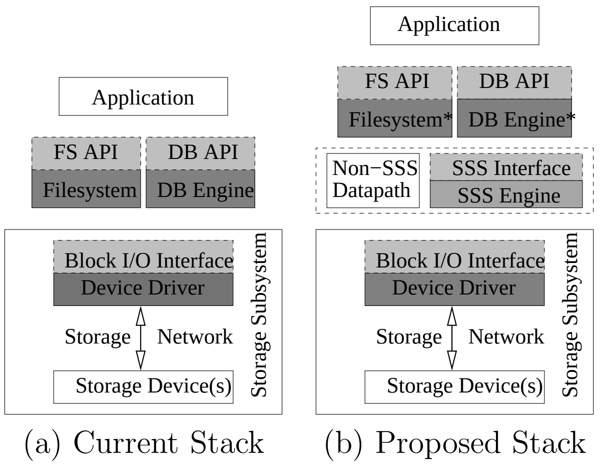


Figure 2.1: Storage stack modification.

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 2.1(a)) form the 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 2.1(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 tree-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 tree-structured data, or bypass it for non-tree-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 tree-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.

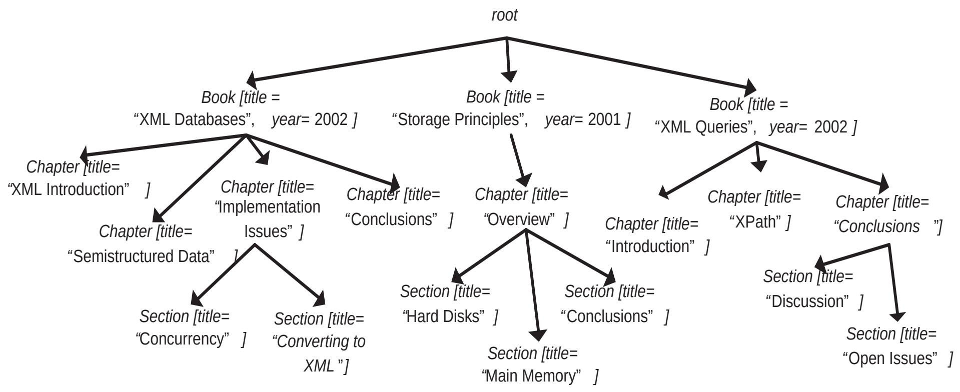

root
Book [title = "XML Databases", year= 2002]
Book [title = "Storage Principles", year= 2001]
Book [title = "XML Queries", year= 2002]
Chapter [title= "XML Introduction"]
Chapter [title= "Semistructured Data"]
Chapter [title= "Implementation Issues"]
Chapter [title= "Conclusions"]
Chapter [title= "Overview"]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

Figure 2.2: A sample tree-structured document.

2.2.2 Data and Access Model

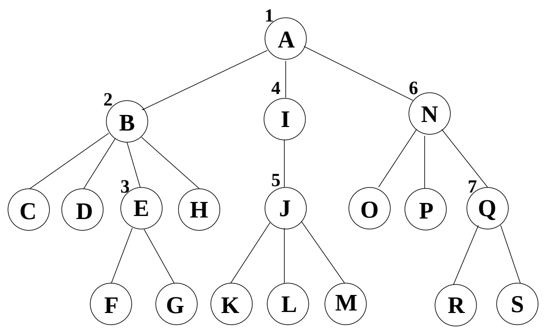


Figure 2.3: Tree structure for the XML document in Figure 2.2.

We view a tree-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.

Figure 2.2 shows an example of a tree-structured document (in this case an XML document) and Figure 2.3 shows the corresponding tree structure.

In the default layout strategy as is employed by current day file systems, the tree-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 [**?**] 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 [**?**] 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 [**?**]) 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 [?]

```
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
```

2.3 Semi-structured Data Layout

In this section, we present disk layout strategies for tree-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.

2.3.1 Basic Tree-structured Placement

A key limitation of the default storage method is that it is optimized only for accessing the tree-structured data tree in depth-first order since it places the data file sequentially on disk. For example, for the tree in Figure 2.3 (created by replacing the labels with node IDs in the tree-structured tree of Figure 2.2), 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 2.6. 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 2.4 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* [**?**]. 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.

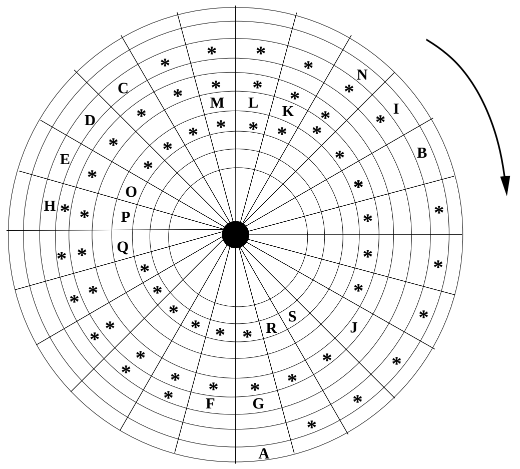


Figure 2.4: Basic tree-structured placement strategy.

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 2.3 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.

Figure 2.4 shows the layout of the tree of Figure 2.3 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 is rotationally skewed

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←(GetNextNode())
4:    t←(GetFirstFreeTrack())
5:    L← empty
6:    L←(Add(Children(n)))
7:    lbnFirstChild← (FindSemiSequential(n.lbn, T))
8:    Place(t,lbnFirstChild, L)
9: end while
```

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 2.3 are not numbered in the ordering and hence are not returned by `getNextNode()`, which is when the placement algorithm terminates.

2.3.2 Optimized Tree-structured Placement

The basic layout strategy, as is obvious in Figure 2.4, results in severe external fragmentation of disk space (internal fragmentation within a disk block is discussed in Section 2.4), 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 2.5, 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.

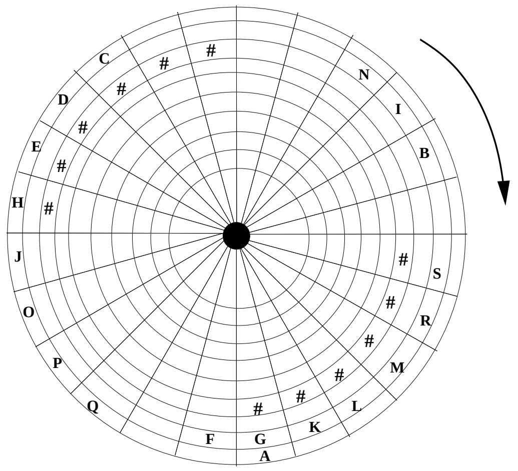


Figure 2.5: Optimized Strategy.

Figure 2.5 shows the layout of the tree in Figure 2.3 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 2.6) 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 re-

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
```

turns 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.

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.

2.3.3 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 tree-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 tree-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 semisequentially), 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 95^{th} 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* [**?**], 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 2.6) 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., [**?**, **?**, **?**]), 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 tree-structured data.

2.4 Supernode Trees

So far, we assumed that each node in the tree-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.

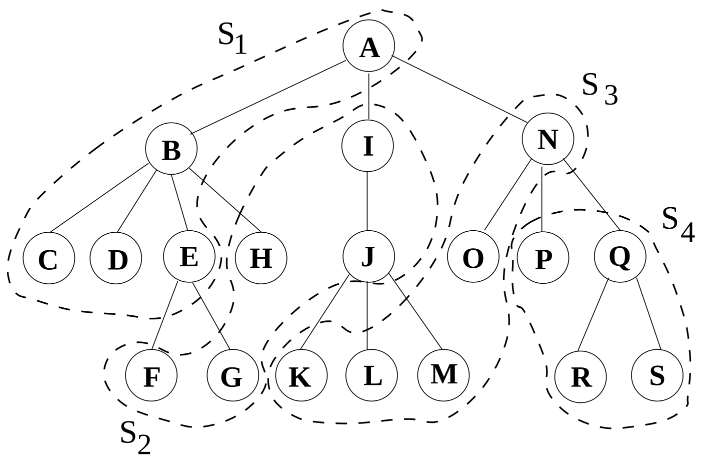
In this section, we first lay the foundation for grouping nodes in a tree-structured data tree T to form *supernodes* where each supernode occupies an entire disk block. Next, we describe how to organize the supernodes into a supernode tree structure T_S. The placement strategies of Section 2.3 are then applied on the supernode tree instead of the node tree.

2.4.1 Grouping Nodes into Supernodes

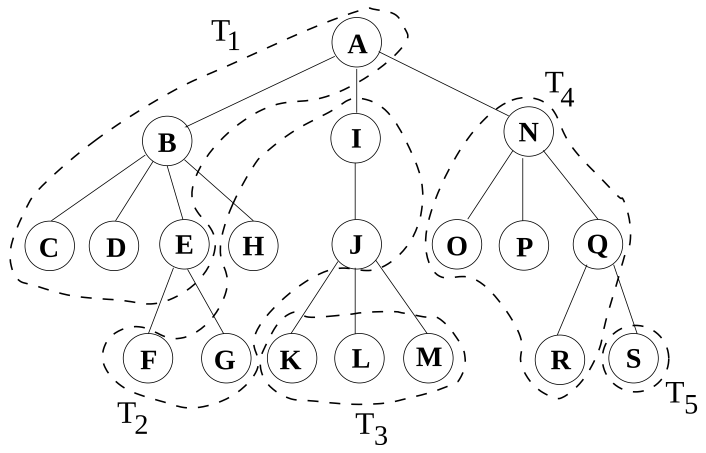
To reduce the internal fragmentation, it is desirable to group the maximum number of nodes into a supernode. It is also important to group adjacent nodes of T in the same supernode, so that navigating among these nodes requires only one disk access. If the size of a node is larger than the size of a disk block, it is stored using multiple supernodes, which are then stored in consecutive disk blocks. An 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 [**?**] to manage BLOBs.

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

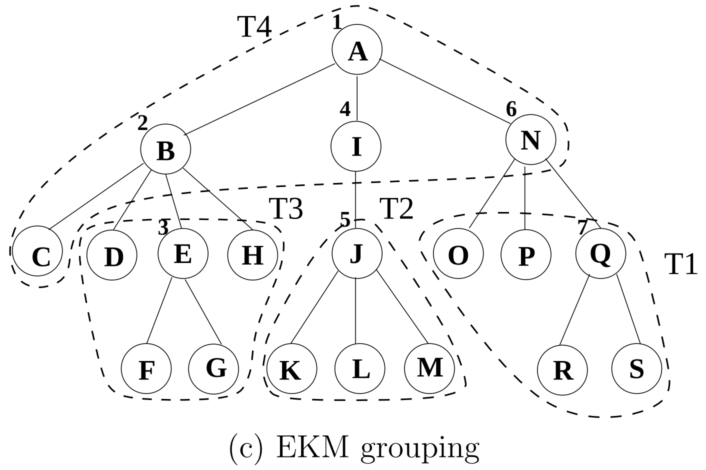
Sequential grouping. Nodes are added to a supernode starting from the root node using a depth-first (and left-to-right) traversal. The only difference is that a single node is not split nodes across disk blocks, unless the size of the node is greater



(a) Sequential grouping



(b) Tree-preserving grouping



(c) EKM grouping

Figure 2.6: Grouping strategies for creating supernodes.

than the size of a disk block. Figure 2.6(a) illustrates this grouping strategy for the tree presented earlier in Figure 2.3.

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 2.6(b) illustrates this grouping strategy for the tree of Figure 2.3.

Enhanced Kundu Misra grouping. We also implement a grouping technique developed independently at the same time by Kanne and Moerkotte [**?**] called the Enhanced Kundu Misra (EKM) grouping, an extension to the original Kundu-Misra grouping algorithm [**?**]. 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 tree into a binary tree representation, obtaining a layered partitioning that helps reducing the number of supernodes while preserving the connectedness. Figure 2.6(c) illustrates this grouping strategy for the tree of Figure 2.3.

2.4.2 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 2.3. 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 2.7(a) shows the formation of this list.

2. The *tree-preserving supernode tree*, which corresponds to the *tree-preserving* (with respect to grouping) *tree-structured* (with respect to placement algorithm) placement to be introduced in Section 2.6, 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 2.7(b) shows the formation of this tree.

3. The *sequential supernode tree*, which corresponds to the *sequential tree-structured placement algorithm* in Section 2.6, 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 2.7(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 2.7(d) shows the formation of this tree.

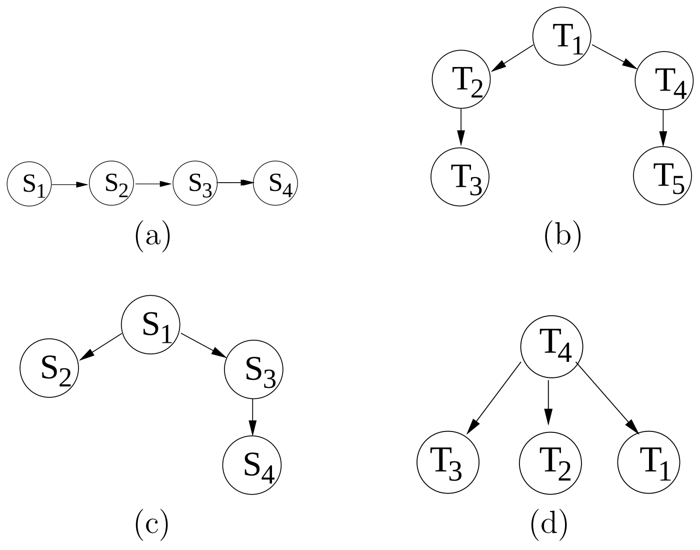


Figure 2.7: Supernode Trees: (a) Sequential supernode list. (b) Tree-preserving supernode tree. (c) Sequential supernode tree. (d) EKM supernode tree.

2.5 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 2.2 summarizes the description of each parameter used in this analysis.

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:

$$t_{rand} = seekTime\ (\frac{C}{3}) + \frac{1}{2}\ T_{rot} \tag{2.1}$$

Table 2.2: 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

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

$$\begin{aligned} seekTime\ (d) &= \alpha + \beta \cdot \sqrt{d}; if \quad d < \frac{C}{3} \\ &= \gamma + \delta \cdot d; otherwise \end{aligned} \tag{2.2}$$

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 2.7), 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 2.4). Thus, substituting these values in the above Equation 2.1, 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,

$$t_{seq} = 0 \tag{2.3}$$

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:

$$t_{semi-seq}(v) = seekTime\ (s(v)) \tag{2.4}$$

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

$$s(v) = \frac{f^{d-2}(d-2-f/(1-f)) + 2 + f/(1-f)}{2n'} \tag{2.5}$$

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

To understand this equation, lets 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 2.3 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

$$1 + f + \cdots + f^{d-k-1} = \frac{(1-f^{d-k})}{(1-f)} \tag{2.6}$$

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,

$$\frac{\left(\frac{f+(1-f^{d-k})}{(1-f)(1+\cdots+(f-1))}\right)}{f} = \frac{\left(\frac{f+(1-f^{d-k})}{(1-f)(f-1)f/2}\right)}{f} = \frac{(f^{d-k}+1)}{2} \tag{2.7}$$

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 2.5 is 1.83. Thus, the $seekTime(s(v))$ is $\alpha + \beta \cdot \sqrt{1.83} = 2.26$.

Equation 2.4 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,

$$t_{semi-seq}(v) = seekTime\ (s(v)) + \frac{1}{2k}T_{rot} \tag{2.8}$$

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 2.8, 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 2.2. 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 2.3 summarizes the access times in the default and the tree-structured storage for every a_i.

Table 2.3: 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 2.9 and 2.10 respectively.

$$T_{default} = \sum_{i=1}^{5} P_i \cdot t_{default}(a_i) \tag{2.9}$$

$$T_{tree} = \sum_{i=1}^{5} P_i \cdot t_{tree}(a_i) \tag{2.10}$$

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 2.6), if we did assume that a query exhibits all the access types shown in Table 2.3, 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 2.9 and 2.10 as:

$$T_{default} = \frac{1}{5} \cdot t_{seq} + \frac{1}{5} \cdot t_{rand} + \frac{1}{5} \cdot t_{rand} + \frac{1}{5} \cdot t_{seq} + \frac{1}{5} \cdot t_{rand}$$

$$= 3.594 \text{ ms, and}$$

$$T_{tree} = \frac{1}{5} \cdot t_{semi-seq} + \frac{1}{5} \cdot (t_{semi-seq} + \frac{f}{2}(T_{nt})) + \frac{1}{5} \cdot t_{seq} + \frac{1}{5} \cdot t_{seq} + \frac{1}{5} \cdot t_{rand}$$

$$= 2.344 \text{ ms}$$

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

2.6 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 [?] 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.

2.6.1 Data Set and Queries

We generated XML files (each file corresponds to an XML tree) of various sizes using the XMark generator [?] 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 [**?**]. There is ongoing work on improving memory efficiency of XML parsers [**?**] which promise to address this shortcoming in the near future. Earlier in Table 2.1, 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 2.2, 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 2.4, computing and storing the information about the supernode that would contain each XML node. We also implemented extensions to the DiskSim disk simulator [**?**] that allowed us to simulate the native layout strategy described in Section 2.3. We then used the supernode information to store them on disks simulated by DiskSim.

Table 2.4 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 the

Table 2.4: 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

root are leaves) [**?**]. 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 XPathMark benchmark [**?**], 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 2.5.

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 2.5. The former class describes the special class of XPath queries that navigate entire subtrees of the tree (queries $D_1, \ldots, D_9$ in Table 2.5). The latter class, non deep-focused queries $N_1, \ldots, N_9$ in Table 2.5, 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

Table 2.5: XPath queries for the deep-focused (D) and the non deep-focused (N) classes.

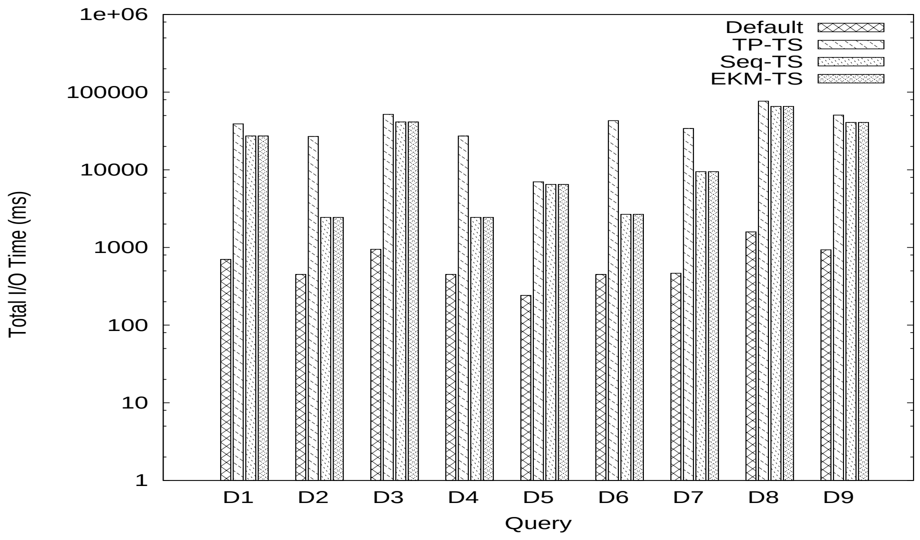
#	Query	#	Query
D1	*/site/closed_auctions/closed_auction/annotation/description/parlist/listitem/text/keyword*	N1	*/site/open_auctions/open_auction*
D2	*/site/people/person/watches*	N2	*/site/closed_auctions*
D3	*/site/open_auctions/open_auction/annotation/description/text/keyword*	N3	*/site/regions/australia*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

not the node-granularity navigation. Hence, queries like D_2, which do not access leaf nodes, are included in the first category since they access supernode leaves; the *watches* subtree is very small and fits in less than one supernode.

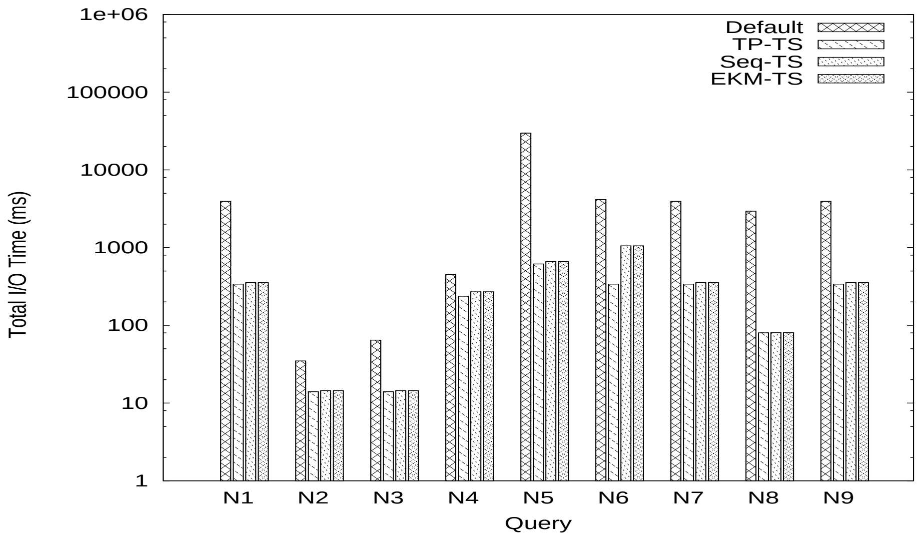
2.6.2 Tree Navigation Performance

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 2.4: *default* (Section 2.2.2), *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

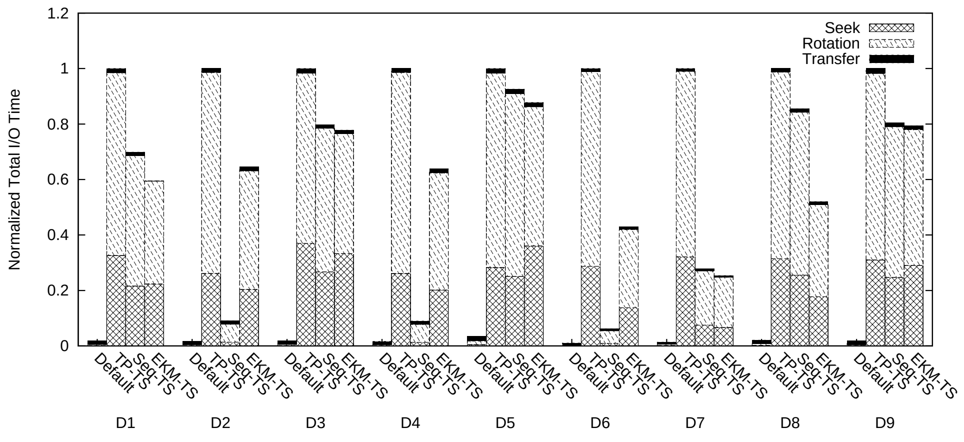


(a) Deep-focused queries



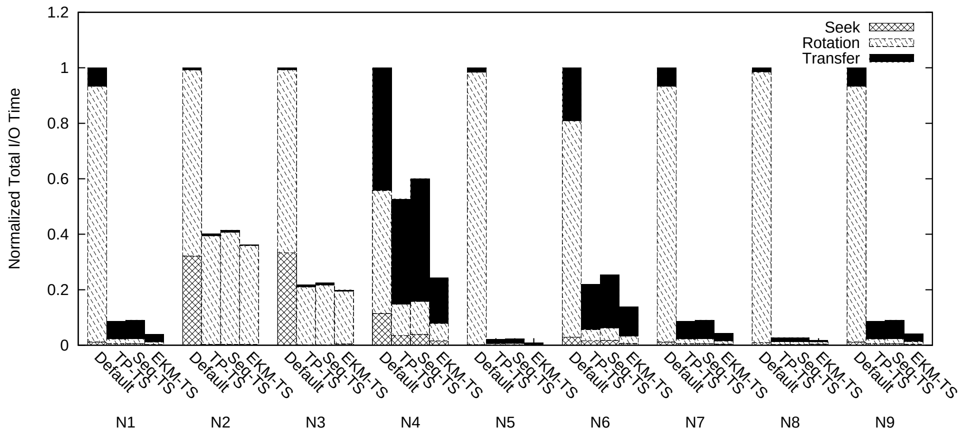
(b) Non-deep-focused queries

Figure 2.8: Total I/O times in logarithmic scale for various placement strategies.



Disk

(a) Deep-focused queries



Disk

(b) Non-deep-focused queries

Figure 2.9: Normalized total I/O times for various placement strategies.

Table 2.6: Navigational patterns for the two XPath query classes for $f = 0.5$. a_i's are defined in Table 2.2.

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

a reasonable assumption for XML trees, which are typically short even when their 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.

Total I/O time

Figure 2.8 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 2.9 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 2.8 and Figure 2.9, 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 2.6 shows the numbers of supernodes accesses for the five basic types of tree accesses, a_1 through a_5, defined in Table 2.3. 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 2.6. 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.

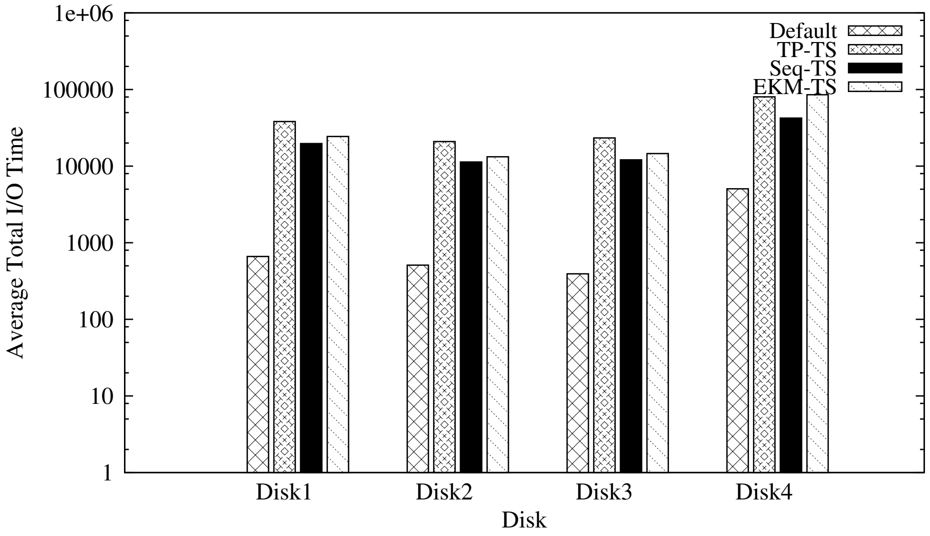
In Figure 2.9 (b), we see a somewhat unexpected outcome that the seek times reduce for queries N2 and N3 for TP-TS, Seq-TS and EKM placement. An answer can be found in the access patterns of these queries (Table 2.6). For N2 and N3, all accesses for the default placement are of type a_5, which are random 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 2.8. 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.

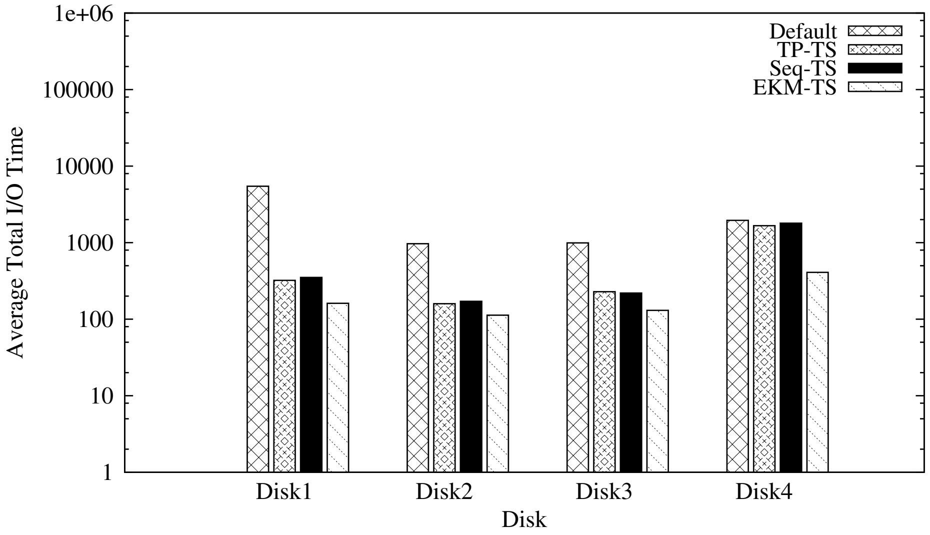
Sensitivity to drive characteristics

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 2.7, were the Seagate Barracuda, Seagate Cheetah 9LP, Seagate Cheetah 4LP, and the HP C3323A as representative of four performance classes of disk drives: *base, fast rotating and fast seeking, fast rotating,* and *slow rotating* respectively. A disk block is of size 512 bytes.

Figure 2.10 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



(a) Deep-focused queries

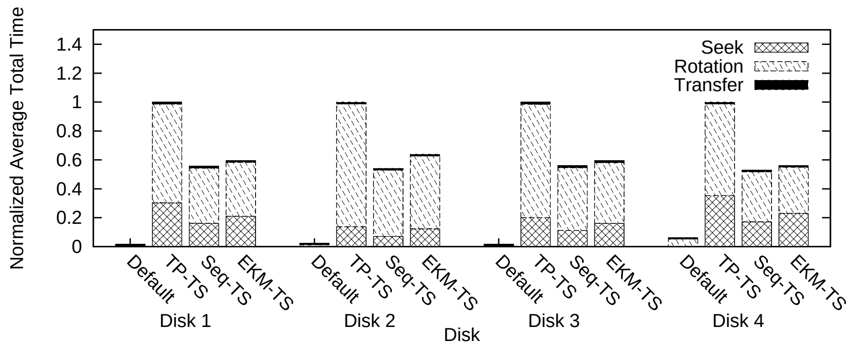


(b) Non-deep-focused queries

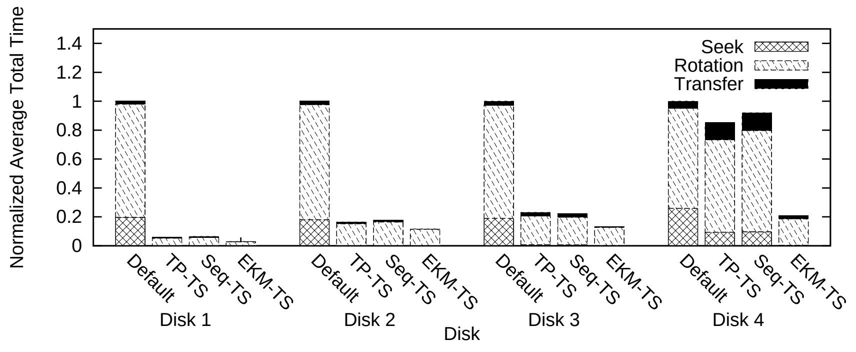
Figure 2.10: Sensitivity of query I/O times to changing disk drive characteristics (logarithmic scale).

Table 2.7: 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



(a) Deep-focused queries



(b) Non-deep-focused queries

Figure 2.11: Sensitivity of seek and rotational delay components of I/O access times to changing disk drive characteristics.

the various hard disk models. For the special class of deep-focused queries (Figure 2.10(a)), the default placement strategy performs better than the other strategies benefiting from optimized sub-tree retrievals. However, for all other queries (Figure 2.10(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 2.11 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 2.11(b)), the average rotational-delays are substantially reduced relative to the default layout.

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 2.3, this issue in its more general form (i.e., multi-process blocking I/O performance) has been addressed earlier with anticipatory I/O scheduling [**?**]. 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 2.5. The interleaved queries belonged to 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.

Table 2.8: 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	$D_2 + D_4 + D_6$	$N_5 + N_6 + N_8$

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 2.8 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.

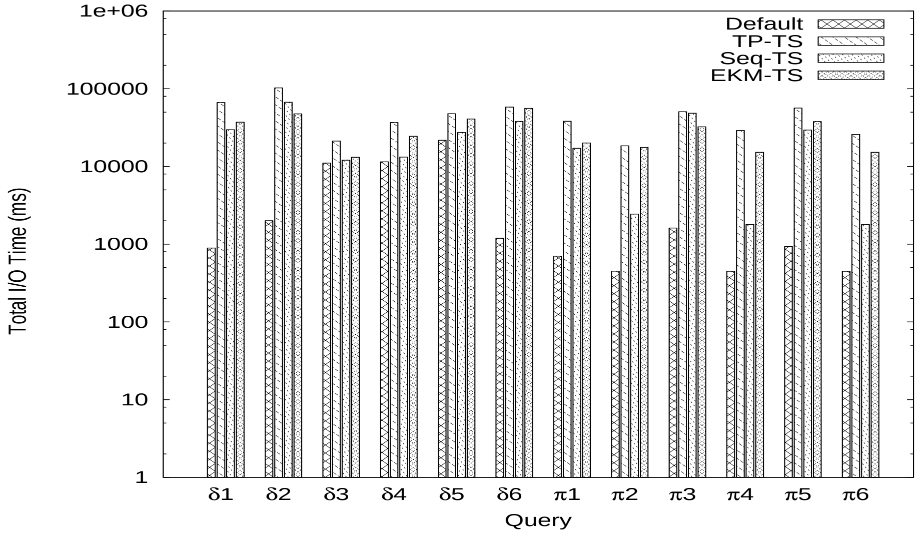
Figure 2.12 shows the total I/O time (in logarithmic scale) for the execution of interleaved deep-focused and non-deep-focused XPath queries. The results for the deep-focused queries from Figure 2.12 (a), show that like in single query execution, the default strategy performs better for multiple interleaved 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 2.12 (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.

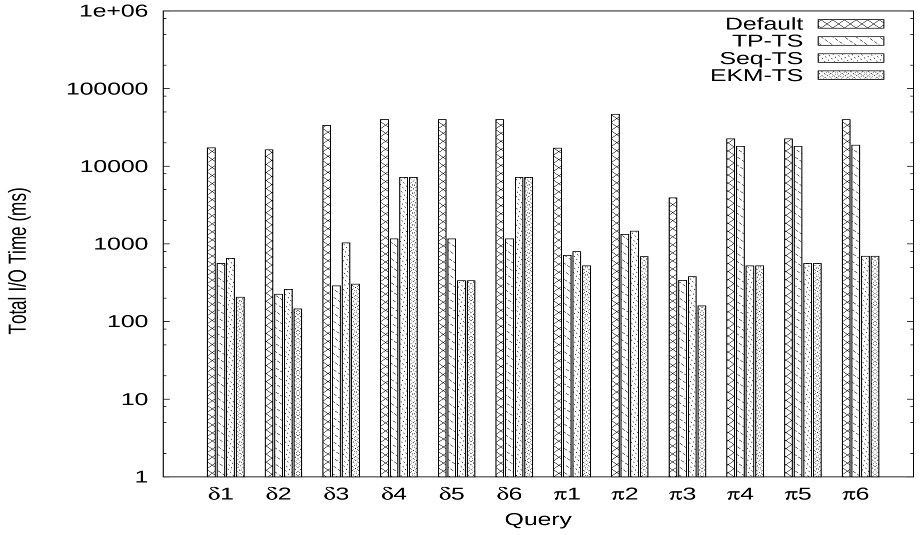
2.6.3 Fragmentation

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

Internal Fragmentation: Figure 2.13 (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 fragmen-

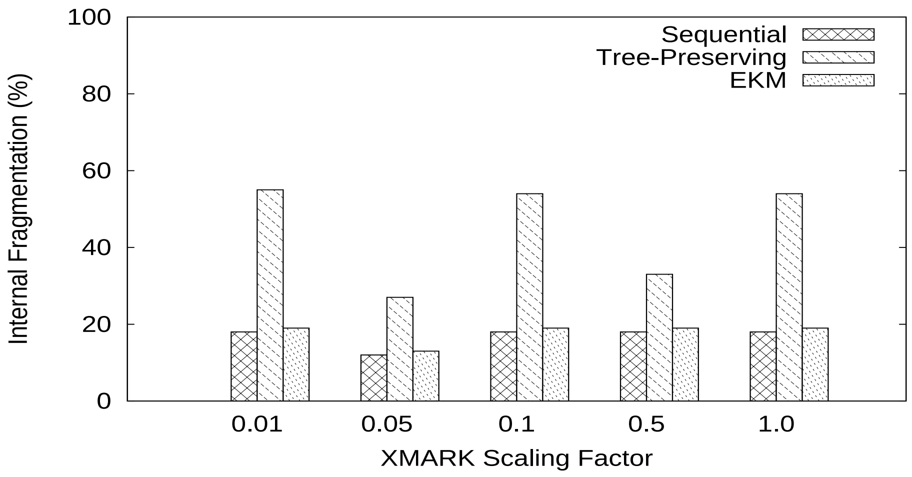


(a) Deep-focused queries

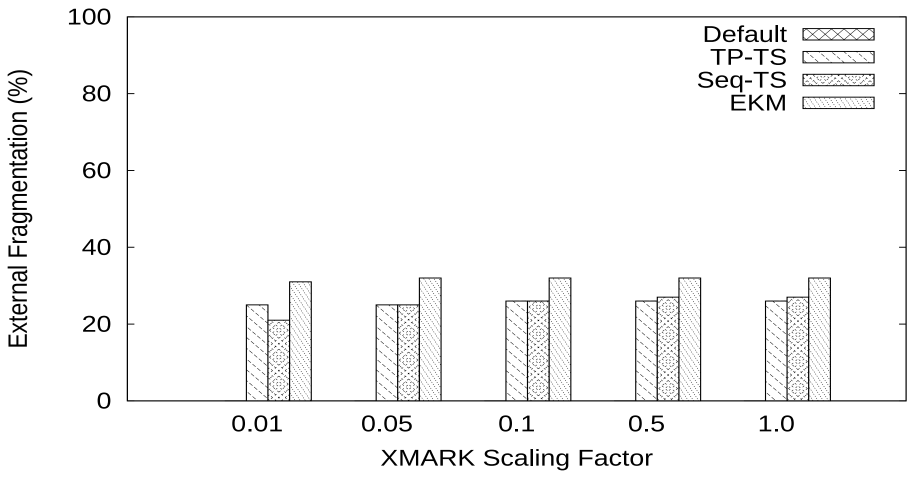


(b) Non-deep-focused queries

Figure 2.12: Total I/O times in logarithmic scale for interleaved XPath queries.



(a) Internal fragmentation



(b) External fragmentation

Figure 2.13: Internal and External Fragmentation.

tation 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 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 2.13 (b) shows the external fragmentation results for the data placement strategies. The default strategy incurs zero external fragmentation as it places the supernode list sequentially on the disk. 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.

2.7 Related Work

Storage of tree-structured data has received attention in the last few years because of its growing popularity. Most work has focused on storing tree-structured data in relational DBMSs or in flat files with indexes. The former approach (e.g.,[?, ?, ?, ?, ?, ?]) has been the most popular due to the success and maturity of the relational DBMSs. The latter approach (e.g., [?, ?]) is based on storing the data as a flat file and building separate indexes on top. These strategies do not use native layout of tree-structured data and are limited to the generic optimization strategies built into relational databases and file systems.

The problem of native storage of tree-structured data has been addressed in Natix [?, ?] and in System RX [?], 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 [?] uses schema information to make a storage plan for the tree-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 [?]. Systems such as [?, ?, ?] 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 tree-structured data into disk firmware.

Recent work by [?] 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 tree-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 tree-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 [?] 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.

2.8 Summary

With this work, we have taken a first step towards building native storage systems for tree-structured data, a problem which has been largely unexplored. We presented on-disk data layout techniques for tree-structured data that explicitly account for the structural mismatch between the tree-structured data and disk drives and reduce disk access overhead. These layout techniques are based on node-grouping algorithms for tree-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.

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 tree-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.

CHAPTER 3

DYNAMIC DATA ALLOCATION

This chapter presents the design, implementation, and evaluation of BORG, a self-optimizing storage system that performs automatic, dynamic block reorganization based on the observed I/O workload. BORG is motivated by three characteristics of I/O workloads: non-uniform access frequency distribution, temporal locality, and partial determinism in non-sequential accesses. To achieve its objective, BORG manages a small, dedicated partition on the disk drive, with the goal of servicing a majority of the I/O requests from within this partition with significantly reduced seek and rotational delays. BORG is transparent to the rest of the storage stack, including applications, file system(s), and I/O schedulers, thereby requiring no or minimal modification to storage stack implementations. We evaluated a Linux implementation of BORG using several real-world workloads, including individual user desktop environments, a web-server, a virtual machine monitor, and an SVN server on a single disk as well as different RAID configurations. These experiments comprehensively demonstrate BORG's effectiveness in improving I/O performance and its incurred resource overhead.

3.1 Introduction

Present day file systems, which control space allocation on the disk drive, employ static data layouts [?, ?, ?, ?, ?, ?]. Mostly, they aim to preserve the directory structure of the file system and optimize for sequential access to entire files. No file system today takes into account the dynamic characteristics of I/O workload within its data management mechanisms. In this dissertation, we contend that making allocation decisions based on the observed workload patterns can considerably improve the I/O performance of the storage system.

We conducted experiments to reconcile past observations about the nature of I/O workloads [**?**, **?**, **?**] in the context of current-day systems including end-user and server-class systems. Our key observations that motivate BORG are: *(i)* on-disk data exhibit a *non-uniform access frequency distribution*; the "frequently accessed" data is usually a small fraction of the total data stored when considering a coarse-granularity time-frame, *(ii)* considering a fine-granularity time-frame, the "on-disk working-set" of typical I/O workloads is dynamic; nevertheless, workloads exhibit *temporal locality* in the data that they access, and *(iii)* I/O workloads exhibit *partial determinism* in their disk access patterns; besides sequential accesses to portions of files, fragments of the block access sequence that lead to non-sequential disk accesses also repeat. We elaborate on these observations in § 3.2.

While the above observations mostly validate the prior studies, and may even appear largely intuitive, surprisingly, there is a lack of commodity storage systems that utilize these observations to reduce I/O times. We believe that such systems do not exist because *(i)* key design and implementation issues related to the feasibility of such systems have not been resolved, and *(ii)* the scope of effectiveness of such systems has not been determined.

We built BORG, an online *Block-reORGanizing* storage system to comprehensively address the above issues. BORG correlates disk blocks based on block access patterns to capture the I/O workload characteristics. It manages a dedicated, *BORG OPtimized Target (BOPT)* partition and dynamically copies working-set data blocks (possibly spread over the entire disk) in their relative access sequence contiguously within this partition, thus simultaneously reducing seek and rotational delays. In addition, it assimilates all *write requests* into the BOPT partition's write buffer. Since BORG operates in the background it presents little interference to foreground applications. Also, BORG provides strong block-layer data consistency to upper layers, by maintaining a persistent page-level *indirection map*.

We evaluated a Linux implementation of BORG for a variety of workloads including a development workstation, an SVN server, a web server, a virtual machine monitor, as well as several individual desktop applications for single disk as well as multiple disks. The evaluation shows both the benefits and shortcomings of BORG as well as its resource overheads. Particularly, BORG can degrade performance when a non-sequential read workload suddenly shifts its on-disk working-set. For most workloads, however, BORG decreased disk busy times in the range 6% to 50%, offering the greatest benefit in the case of non-sequential write-mostly workloads without tuning BORG parameters for optimality. A sensitivity study with various parameters of BORG demonstrates the importance of careful parameter choice which can lead to even greater improvements or degrade performance in the worst case; a self-configuring BORG is certainly a logical and feasible direction. Memory overheads of BORG are bound within 0.25% of BOPT, but CPU overheads are higher. Fortunately, most processing can be done in the background and there is ample room for improvement.

In this chapter, we: *(i)* study the characteristics of I/O workloads and show how the findings motivate BORG (§ 3.2) , *(ii)* motivate and present the detailed design and the first implementation of a disk data re-organizing system that adapts itself to changes in the I/O workload (§ 3.3 and § 3.4), *(iii)* present the challenges faced in building such a system and our solutions to it (§ 3.5), and *(iv)* evaluate the system to quantify its merits and weaknesses (§ 3.6).

3.2 Characteristics of I/O Workloads

In this section, we investigate the characteristics of modern I/O workloads, specifically elaborating on those that directly motivate BORG. We collected I/O traces, downstream of an active page cache, over a one-week period from four different machines.

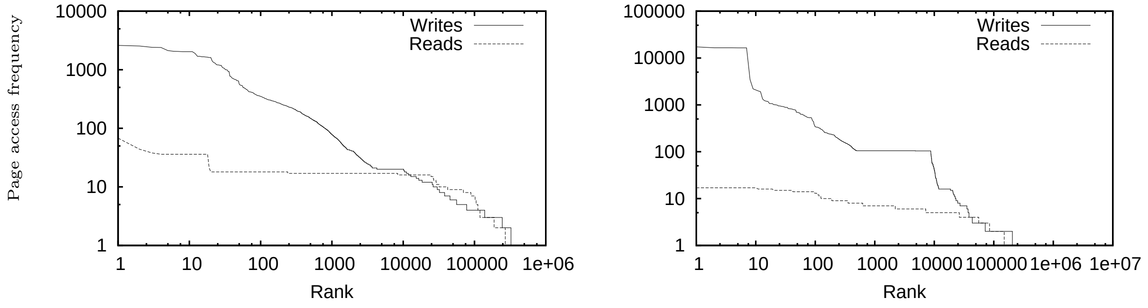
These machines have different I/O workloads, including *office* and *developer* desktop workloads, a version control *SVN (Subversion) server*, and a *web-server*. The office and developer workloads are single-user workloads. The former workload was composed mostly of web-browsing, graph plotting with gnuplot, and several open-office applications, while the latter consisted of extensive development using emacs, gcc, and gdb, document preparation using LaTeX, email, web-browsing, and updates of the operating system.

Workload type	**File System size [GB]**	**Memory size [GB]**	**Reads [GB]**		**Writes [GB]**		**File System accessed**	**Top 20% data access**
			Total	**Unique**	**Total**	**Unique**		
office	8.29	1.5	6.49	1.63	0.32	0.22	22.22 %	51.40 %
developer	45.59	2.0	3.82	2.57	10.46	3.96	14.32 %	60.27 %
SVN server	2.39	0.5	0.29	0.17	0.62	0.18	14.60 %	45.79 %
web server	169.54	0.5	21.07	7.32	2.24	0.33	4.51 %	59.50 %

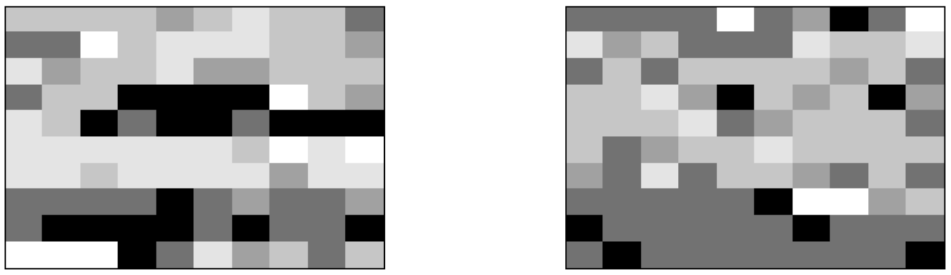
Table 3.1: Summary statistics of week-long traces obtained from four different systems.

The SVN server hosted document and project code-base repositories for our 6-person research group. Finally, the web-server workload mirrored the web-requests made to our department's production web-server on one of our lab machines and served 1.1 million web requests during the trace period.

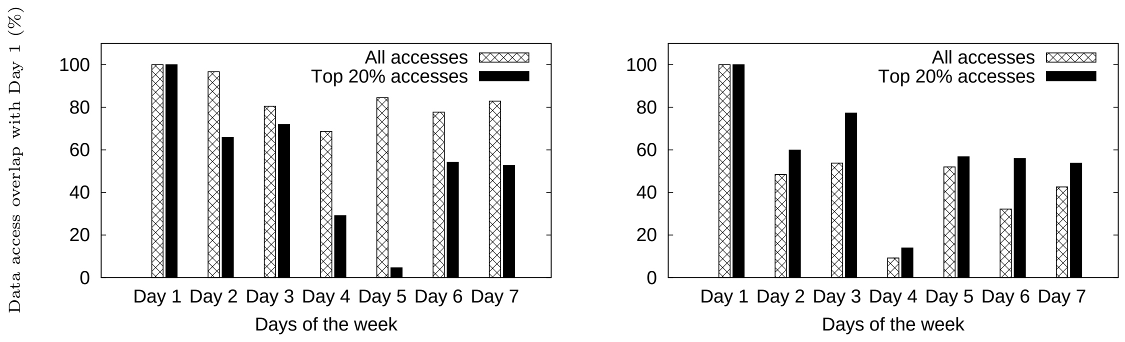
Key statistics for these workloads are summarized in Table 3.1. We define the *on-disk working-set* (henceforth also referred to simply as "working-set") of an I/O workload as the set of all unique blocks accessed in a given interval.



(a) Rank frequency



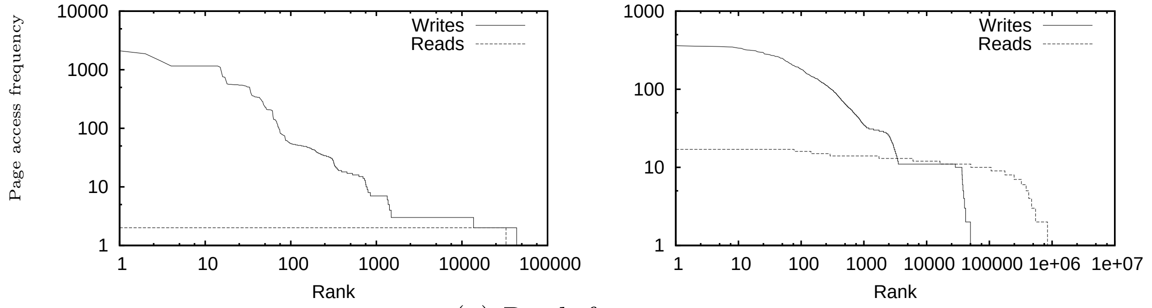
(b) Disk heat map



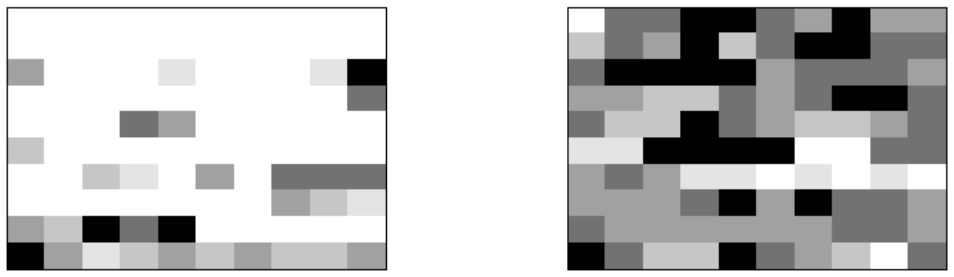
(b) Working set

office workload developer workload

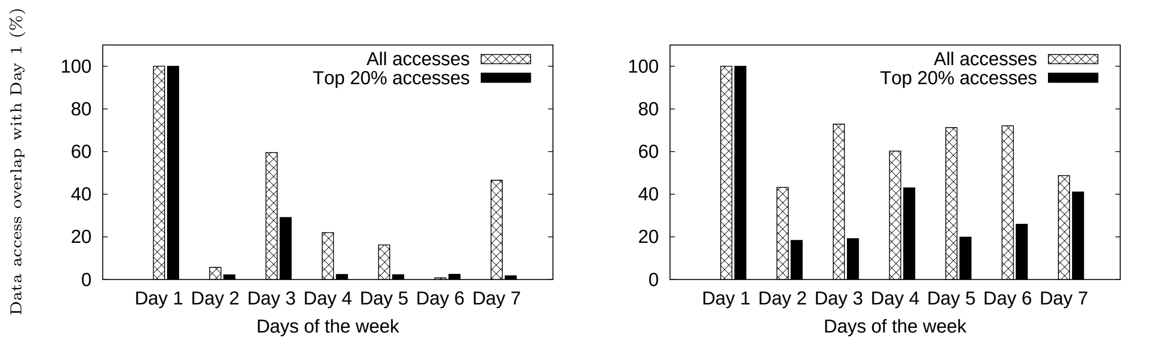
Figure 3.1: Rank-frequency, heatmap, and working-set plots for week-long traces for two different systems.



(a) Rank frequency



(b) Disk heat map



(b) Working set

SVN server web server

Figure 3.2: Rank-frequency, heatmap, and working-set plots for week-long traces for two server systems.

3.2.1 Non-uniform Access Frequency Distribution

Researchers have pointed out that file system data have non-uniform access frequency distribution [?, ?, ?]. This was confirmed in the traces that we collected where less than 4.5-22.3% of the file system data were accessed over the duration of an entire week (shown in Table 3.1). We observe that the office and web server workloads are read mostly, while the developer and SVN server are write mostly. Figures 3.1 and 3.2 (top row) shows page access rank-frequency plots for the workloads; file system pages were 4KB in size, composed of 8 contiguous blocks. A uniform trend to be observed across the various workloads is that the really high frequency accesses are due write to requests. However, and especially in the case of the read-mostly office and web server workloads, there are a large number of read requests that occur repeatedly. In either case (read or write), the access frequencies are highly skewed. Figures 3.1 and 3.2 (middle row) depicts disk *heatmaps* created by partitioning the disk into regions and measuring accesses to each region. The heatmaps depict frequency of accesses in various physical regions spread over the entire disk area, each cell representing a region. Six normalized, exponentially-increasing heat levels are used in each heatmap where darker cells represent higher frequency of accesses to the region. Disk regions are mapped to cells in row-major order.

Skewed data access frequency is further illustrated in Table 3.1 – the top 20% most frequently accessed blocks contributed to a substantially large ($\sim$45-66%) percentage of the total accesses across the workloads, which are within the ranges reported by Gómez and Santonja (Figure 2(a) in [?]) for the Cello traces they examined.

Based on the above observations, it is reasonable to expect that co-locating frequently accessed data in a small area of the disk would help reduce seek times when compared to the same data being spread throughout the entire disk area. Akyurek and Salem [?] have demonstrated the performance benefits of such an optimization via a simulation study. This observation also motivates reorganizing copies of popular blocks in BORG.

3.2.2 Temporal Locality

Temporal locality in I/O workloads is observed when the on-disk working-sets remain mostly static over short durations. Here, we refer to a locality of hours, days, or weeks, rather than seconds or minutes (typical of main memory accesses). For instance, a developer may work on a few projects over a period of a few weeks or months, typically resulting in her daily or weekly working sets being substantially smaller than her entire disk size. In servers, popularity of client requests result in temporal locality. A web server's top-level links tend to be accessed more frequently than content that is embedded much deeper in the web-site; an important new revision of a specific repository on an SVN server is likely to be accessed repeatedly over the initial weeks.

Figures 3.1 and 3.2 (bottom row) depict the changes in the per-day working-sets of the I/O workload. The two end-user I/O workloads and the web server workload exhibit large overlaps in the data accessed across successive days of the week-long trace with the first day of the trace. There is substantial overlap even among the top 20% most accessed data across successive days. Interestingly, these workloads do not necessarily exhibit a gradual decay in working-set overlap with day 1 as one might expect, indicating that popularity is consistent across multi-day periods. The SVN server exhibits anomalous behavior because of the commit and update operations. Periods of high *commit* activity degrade temporal locality as new data gets created, while periods of high *update* activity improve temporal locality.

These observations indicate that optimizing layout based on past I/O activity can improve future I/O performance for some workloads and motivates planning block reorganization based on past activity in BORG.

3.2.3 Partial Determinism

Workload type	**Partial determinism**
office	65.42 %
developer	61.56 %
SVN server	50.73 %
web server	15.55 %

Table 3.2: Partial Determinism in week-long traces obtained from four different systems.

Partial determinism in I/O workload occurs when certain non-sequential accesses in the block access sequence are found to repeat. A *non-sequential access* is defined by a sequence of two I/O operations that are addressed non-contiguous block addresses. It manifests in both end-user systems and servers. For instance, I/O during application start-up is largely deterministic, both in terms of the set of I/O requests and the sequence in which they are requested. Reading files related to a repeatable task such as setting up a project in an integrated development environment, compilation, linking, word-processing, etc. result in a deterministic I/O pattern. In a web-server, accessing a web-page involves accessing associated sub-pages, images, scripts, etc., in deterministic order. Partial determinism has been well-explored in previous literature for its use in prefetching and caching [**?**, **?**] and also more recently for disk layout [**?**, **?**].

In Table 3.2, we present the partial determinism for each workload calculated as the percentage of non-sequential accesses that repeat at least once during the week. The partial determinism percentages are high for the two end-user and the SVN server workloads. Further, for each of these workloads, there were a non-trivial amount of non-sequential accesses that repeated as many as 100 times. These findings suggest that there is ample scope for optimizing the repeated non-sequential access patterns.

3.3 Overview and Architecture

BORG is motivated by the simple question: *What storage system optimizations based on workload characteristics can allow applications to utilize the disk drive more efficiently than current systems do?* We conceived BORG with the goal of making the storage system aware of changing workload and pro-actively reducing on-disk mechanical delays. This section presents the rationale behind the design decisions in BORG and its system architecture.

3.3.1 BORG Design Decisions

A Disk-based Cache.

The operating system uses main memory to cache frequently and recently accessed file system data to reduce the number of disk accesses incurred. In any given duration of time, the effectiveness of the cache is largely dependent on the on-disk working-set of the I/O workload, and can degrade when this working-set increases beyond the size of the page cache. Storage optimizations such as prefetching [?, ?, ?] and I/O scheduling [?, ?, ?, ?] help improve disk I/O performance in such situations. While good prefetching solutions can help reduce the number of I/O requests and amortize seek and rotational delay costs, improved I/O scheduling can help reduce head movement to the extent that the physical data layout and application access patterns permit.

Using a disk-based cache as an extension of the main memory cache offers three complementary advantages in comparison to main memory caching alone, prefetching, and I/O scheduling. First, it is more effective as a cache (than main memory) because it offers a less expensive (and thus larger) as well as reliable caching solution, thus allowing data to be cache-resident for long periods of time. Second, the size of the disk-based cache can easily be configured by the system administrator without changing any hardware. And finally, dynamically optimizing data layout based on access patterns within a disk-based cache provides the unique ability to make originally non-sequential data accesses more sequential.

A Block Layer Solution.

A self-optimizing storage solution can be built at any layer in the storage stack (shown in Figure 3.3). Block level attributes of disk I/O operations are not easily obtained at the VFS or the page cache layer. While file system layer solutions can benefit from semantic knowledge of blocks, they incur a significant disadvantage in being tied to a specific file system (and perhaps even version). Device driver encapsulations (interface at P4) are incapable of capturing upper layer attributes, such as process ID and request time-stamp due to I/O scheduler re-ordering and loss of process context. Interestingly, several studies [?, ?, ?, ?] use P4 as the profiling point. Since our design requires both the process- and block- level attributes as well as preservation of the temporal ordering of request issue events, we use `P3` as the profiling point.

We contend that the block layer (interface at P3) is ideal for introducing block reorganization for several reasons. First, key temporal, block- and process- level attributes about disk accesses are available. Second, operating at the block layer makes the solution independent of the file system layer above, allowing it the flexibility to support multiple heterogeneous file systems simultaneously. Finally, new abstractions due to virtualization trends (e.g., virtual block device abstraction) as well as network-attached storage environments (SAN and NAS) can be supported in a straightforward way. In the case of SAN, BORG can reside on the client where all context for I/O operations are readily available with the underlying assumption that the SAN device's logical block address space is optimized for sequential access. In the case of NAS, the BORG layer can reside within the NAS device where I/O context is readily available. Modifying the NAS interface to include process associations within file I/O requests can complete the profile information.

File System Oblivious

Making BORG independent of the file system has its trade-offs. While an implementation for a specific file system could show greater performance improvements, it would be difficult to implement it as a module that can seamlessly be inserted or removed on demand. Since BORG is independent of the file system above it, it can be used with any Linux file system without any changes to the kernel code. Further, BORG does not interfere with the data layout and autonomously manages it space concerns which is a critical requirement for modularity in system architecture and design.

Using an Independent BOPT partition.

The file system optimizes for sequential accesses to entire files, a common form of file access. This is characteristic of certain workloads, including logging, background maintenance, as well as streaming data applications. However, certain workloads, including application start-up, content indexing and web-page requests, exhibit a more non-sequential, but deterministic, access behavior. It is thus possible that the same set of data can be accessed sequentially by some applications and non-sequentially by others. Further, some deterministic non-sequential accesses may only be temporary phenomenon.

Based on this observation, Akyurek and Salem [?] have argued in favor of *copying* rather than *shuffling* [?, ?] of data. Copying retains original sequential layouts so a choice of location based on the observed access pattern may be possible. Reverting back to the original layout is straightforward. Similarly, rather than permanently disturbing the sequential layout of files, BORG operates on copies of blocks placed temporarily in an independent BOPT partition, optimizing for the current common case of access for each data block. This also allows BORG to ensure co-location of most accessed data to reduce seek, and a large contiguous space for reducing rotational delay.

3.3.2 BORG Architecture

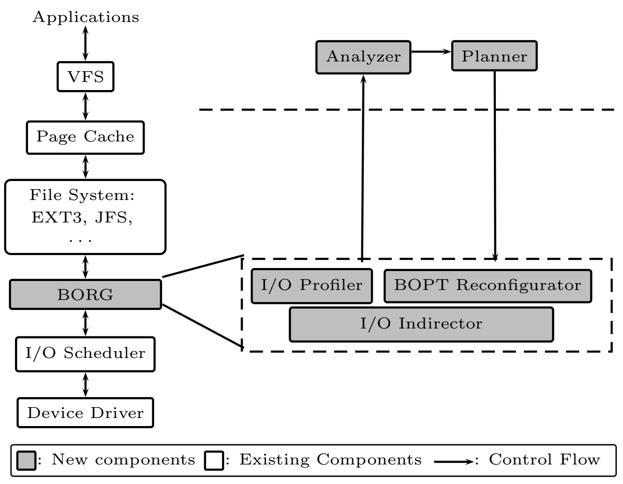


Figure 3.3: BORG System Architecture.

Abstractly, BORG follows a four-stage process:

1. *profiling* application block I/O accesses,

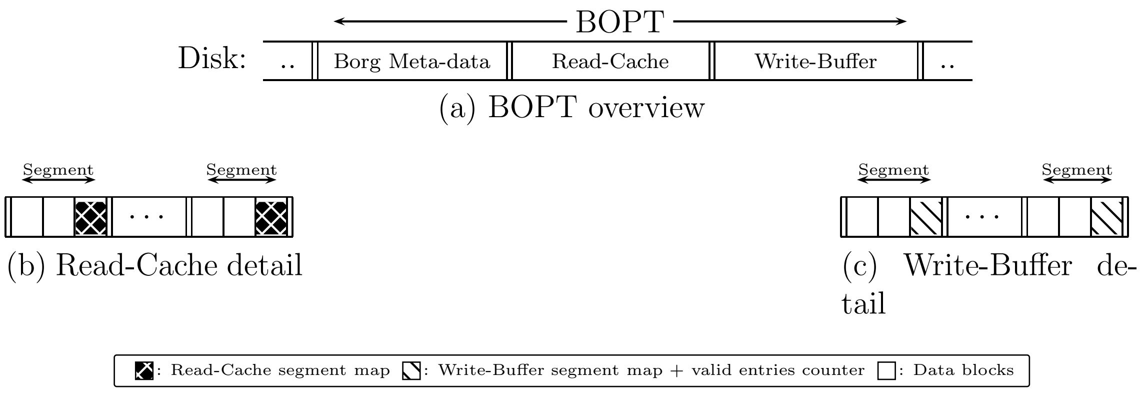
2. *analyzing* I/O accesses to derive access patterns,
3. *planning* a modification to the data layout, and
4. *executing* the plan to reconfigure the data layout.

In addition, an I/O indirection mechanism runs continuously, re-directing requests to the partition that it optimizes as required. Figure 3.3 presents the architecture of BORG in relation to the storage stack within the operating system. The modification to the existing storage stack is in the form of a new layer, which we term *BORG layer*, that implements three major components: the *I/O profiler*, the *BOPT reconfigurator* and the *I/O Indirector*. A secondary throttle-friendly user-space component implements the *analyzer* and the *planner* stages of BORG and performs computation and memory-intensive tasks. While profiling and indirection are both continuous processes, the other stages run periodically and in succession culminating in a reconfiguration operation.

For the I/O profiler, we use a low-overhead kernel tool called `blktrace` [?]. The analyzer reads the I/O trace collected by the profiler and derives data access patterns. Subsequently, the planner uses these data access patterns and generates a new reconfiguration plan for the BOPT partition, which it communicates to the BOPT reconfigurator component. The user-space analyzer and planner components run as a low-priority process, utilizing only otherwise free system resources. Under heavy system load, the only impact to BORG is that generating the new reconfiguration plan would be delayed.

The BORG OPtimized Target partition (BOPT) is a disk partition used exclusively by BORG. The BOPT reconfigurator is responsible for the periodic reconfiguration of the BOPT partition, according to the *layout plan* specified by the planner. The reconfigurator issues low-priority disk I/Os to accomplish its task, minimizing the interference to foreground disk accesses. Finally, the I/O indirector continuously directs I/O requests either to the FS partition or the BOPT partition, based on the specifics of the request and the contents of the BOPT.

3.3.3 BOPT Space Management



(a) BOPT overview

(b) Read-Cache detail

(c) Write-Buffer detail

Figure 3.4: Format of the BOPT partition. Each entry in the Write-Buffer and Read-Cache map tables is a 3-tuple of the form (FS LBA, BOPT LBA, valid bit).

The OPtimized Target partition (BOPT) as managed by BORG is shown in Figure 3.4. To reduce head movement, we suggest that the BOPT partition be created adjoining the *swap* partition if virtual memory is used. BORG partitions the BOPT into three fragments: *BORG Meta-data*, *Read-cache*, and *Write-buffer*. The Read-cache and Write-buffer are further sub-divided into fixed-length segments which store both data and (valid/invalid) map entries for the segment. The in-memory indirection map (elaborated in § 3.4.5) maintained by BORG is a union of all the segment map entries in the BOPT. The BOPT map entries are synchronously updated each time the in-memory map information changes. Additionally, the segment map in the write-buffer contains a "valid entries counter" to track space usage in the write buffer.

Table 3.3 depicts the BOPT meta-data fragment. It stores key persistent information that aid in the operation of BORG. The `BORG_REQUIRE` bit is *set* when the

Magic number	BORG BOPTpartition identifier.
`BORG_REQUIRE` bit	BOPT contains dirty data.
BOPT size	BOPT partition size.
Read-cache info	Offset and size of the Read-cache.
Write-buffer info	Offset and size of the Write-buffer.
Segment size	Fixed size of segments in the BOPT.

Table 3.3: Borg meta-data.

BOPT contains data that requires to be copied back to the FS. If set, the operating system initiates BORG at boot time to ensure consistent data accesses. The remaining meta-data information is used to correctly populate the in-memory indirection map structure during BORG initialization.

3.4 Detailed Design

In this section, we present the design details of BORG by elaborating on its individual components.

3.4.1 I/O Profiler

The *I/O profiler* is a data collection component that is responsible for comprehensively capturing all disk I/O activity. The I/O profiler generates an *I/O trace* that includes the temporal (timestamp of the request), process (process ID and executable) and the block-level (address range and read/write mode) attributes as shown in Figure 3.4.

We use the **Q** events reported by blktrace [**?**], which capture the I/O requests queued at the block layer. These include all requests as issued by the file system(s), including any journaling and/or page destageing mechanisms. We defer further details to the blktrace work [**?**].

[Timestamp]	**[PID]**	**[Exec.]**	**[StartLBA]**	**[Size]**	**[Mode]**
705423195774700	5745	screen	6914207	32	R
705423259644748	5745	utempter	24379775	8	R
705423379492524	5745	utempter	24787567	8	R
705423421266908	5753	bash	7498311	24	R
705423454005104	5745	utempter	24793415	8	R
705423493292648	5756	bash	34543375	64	R
705423565122668	5756	stty	34543439	16	R

Table 3.4: A sample I/O trace.

3.4.2 Analyzer

The *analyzer* is responsible for summarizing the disk I/O workload. It first splits the I/O trace obtained from the profiler into multiple I/O traces, one per process. Each process trace is used to build a directed *process access graph* $G_i(V_i, E_i)$, where vertices represent LBA ranges and edges a temporal dependency (correlation) between two LBA ranges. The weight on an edge between vertices (u, v) represents the frequency of accesses (reads or writes) from u to v. The *directed* and *weighted* graph representation is powerful enough to identify repeated sequences of multiple non-sequential requests.

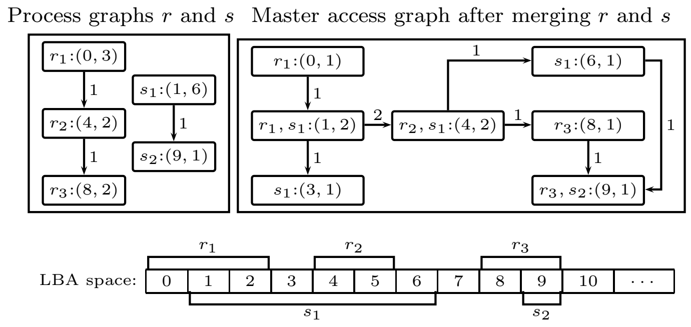


Figure 3.5: Building the master access graph.

Since multiple processes may access the same LBA, a single *master access graph* $G(V, E)$, that captures all available correlations into a single input for the reconfiguration planner is created (illustrated in Figure 3.5). The vertices in the graph are defined by (start LBA, size of request). Since vertices r_1 and s_1 have overlapping LBAs, r_1 is split into two vertices in the master access graph, one with size 1 and the other with the overlapping s_1 blocks, starting at LBA 1 with size 2. The complexity of the merge process increases if two vertices (either within the same graph or across graphs) have overlapping ranges. This is resolved by creating multiple vertices so that each LBA is represented in at most one range vertex. Figure 3.5 illustrates the merge process. The I/Os in this figure are tagged r_i and s_i corresponding to the process graphs r and s respectively. Since r_1 and s_1 have overlapping LBAs, r_1 is split into two vertices, one with size 1 and the other which contains the overlapping blocks, starting from LBA 1 with size 2, with s_1. The edge weights between r_1 and r_1,s_1 remain unchanged, since the LBA range r1, s1(1,2) is accessed only once. Likewise, all the overlapping vertices are split and merged and the master graph is obtained.

A simplistic algorithm to illustrate the splitting procedure is shown in Algorithm 4. *U* and *V* are the LBA's represented by the vertices *u, v* respectively and *i, j, k* are the new vertices that are created.

Algorithm 4: Split Vertices

```
Require: Graph $G$
 1: for every vertex $v$ in G do
 2:    for every other vertex $u$ in $G$ do
 3:       i = U $\cup$ V
 4:       j = U - V
 5:       k = V - U
 6:       for x in $u$ and $v$ do
 7:          Move the incoming edges to x to the vertex with the same starting LBN
 8:          Move the outgoing edges to x to the vertex with the same ending LBN
 9:          Where x was split, add an edge with weight = max(incoming, outgoing)
             weight for x
10:       end for
11:    end for
12: end for
```

An optimization we implement to reduce the complexity of the merge algorithm is to keep the vertices sorted by their initial LBA. The total time complexity for the analyzer stage is given by $O(n \times l)$, where n is the number of vertices and l is the size (in LBA) of the largest vertex in the graph. Once the merge operation is completed, the master access graph, G, is obtained.

3.4.3 Planner

The *planner* takes the master access graph, G, as input and determines a reconfiguration plan for the BOPT partition. It uses a greedy heuristic that starts by choosing for placement the most connected vertex, u, i.e., with the maximum sum of incoming and outgoing edges (Figure 3.6). Next it chooses the vertex v most connected (in one direction only, either incoming or outgoing) to u. If v lies on the outgoing edge of u, it is placed after u and if it lies on the incoming edge it is placed before. The next vertex to be placed is the one most connected to the *group* $u \cup v$. This process is repeated until either all the vertices in G are placed, or the read cache in the BOPT is fully occupied, or the edges connecting to the unplaced vertices in the master graph have weight below a chosen threshold. If the graph contains disconnected components, each of these are placed as separate groups. The time complexity for the planner is $O(n \times lg(m) + n^2)$ where n is the number of vertices and m is the number of edges; finding the most connected vertex takes $O(n \times lg(m))$ time and finding the next vertex takes $O(n)$ time .

Consider the master access graph in Figure 3.6. C is the most connected vertex and is chosen for placement first. Next, vertex B is placed $after$ vertex C since it is connected by an outgoing edge and has the highest weight of all the edges connected to C. Next, vertex G is placed $before$ vertex group $C \cup B$. The final sequence of

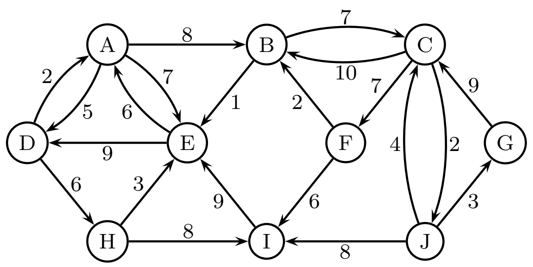


Figure 3.6: Placing the master access graph.

vertices from the lowest LBA to the highest is: $L = [F, H, J, A, G, C, B, E, D]$.

Based on this layout, the planner creates key information about the *type of copy* operations that must be performed by the reconfigurator.

A copy can be of three types: *(i)* the new `incoming` where blocks are copied from FS to BOPT, *(ii)* `relocate` where blocks are moved within the BOPT and *(iii)* the `outgoing` blocks, that are evicted from the BOPT. This information is obtained by comparing the current configuration of the BOPT partition, with the new layout plan. In addition, we classify the blocks that retain their position in the BOPT as `static`.

The above tags are used by the reconfigurator to minimize the data movement between the OPT and the filesystem partitions as well as the data movement within OPT. For instance, if a block is tagged as a `small relocate`, it means that it remains in OPT but has to be moved to another location. Thus it can directly be copied from its previous location to the new location in OPT. This saves the additional overhead of copying it back to its original location in the filesystem and then back to OPT.

3.4.4 BOPT Reconfigurator

The *BOPT reconfigurator* implements the plan created by the planner component by performing the actual data movement to realize the new configuration of the BOPT. This task is complicated primarily because of consistency and overhead concerns. Overhead is partially addressed by issuing low-priority I/O requests for data layout reconfiguration, making the use of a priority scheduler a prerequisite. BORG ensures block data consistency between the FS and BOPT copies of data blocks by maintaining a persistent indirection map, termed the `borg_map`, that continuously tracks the most up-to-date location of a data block. This map is updated each time a block location changes.

The reconfigurator copies blocks in three stages; *outgoing*, where it copies all the dirty blocks that are no longer in the new plan back to the original file system (FS) location, *relocate*, where it copies blocks that have to be relocated within the BOPT, and *incoming* where it copies all the new blocks that have to be copied from the FS to the BOPT. A single data movement operation and the corresponding update on `borg_map` entry can be considered 'atomic' since any application *write* request to the *source* LBA during data movement is put on hold until after the movement is complete and the `borg_map` entry is updated. This ensures that an up-to-date version of data is always maintained by the file system.

The reconfiguration process is depicted in Figure 3.7. The planner uses the graph G (obtained from the analyzer) and the current Current `borg_map` to create the new `plan`. The FS block C, identified as `Incoming` is currently being reconfigured. The reconfigurator reads its information from the plan, copies the block from the FS location C to the BOPT location C' and creates an entry in `borg_map`.

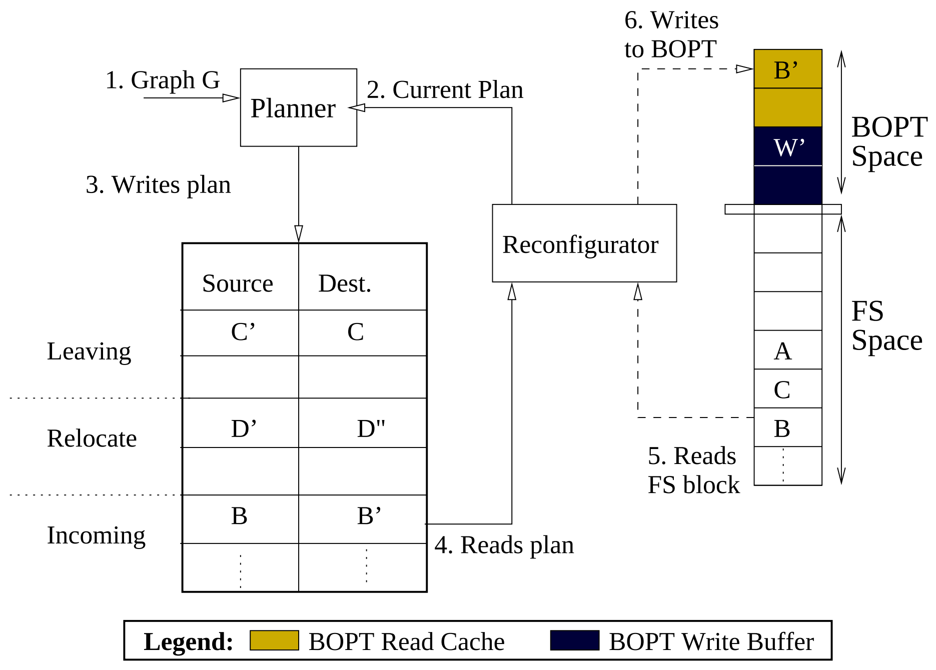


Figure 3.7: Data Reconfiguration process.

3.4.5 I/O Indirector

The *I/O indirector* operates continuously, redirecting file system I/O requests as required. An I/O request may be composed of an arbitrary number of pages. Each page request is handled separately based on (i) number of blocks that can be satisfied from the BOPT as per the `borg_map` entry, (ii) type of operation (read or write) and (iii) presence of a free page in the BOPT.

Algorithm 5 presents the formal algorithm used by the I/O indirector.

For each I/O request larger than one page, the indirector splits it into multiple per-page requests (Line 1). Each page request is looked up in the `borg_map` to check if it is mapped to the OPT.

If a mapping exists for all the pages of the I/O request in the `borg_map`, the request is indirected to the BOPT(Lines 3-7). If no mapping exists, and the request is a read request, it is issued unchanged to the file system (Line 12). If only

Algorithm 5: Indirect I/O Requests

```
Require: I/O Request: $req$, Indirection map: $map$
 1: $children \leftarrow split\_req(req)$ {Returns a set composed by chunks of 8 blocks (1
    page) afer splitting $req$}
 2: $mapped, not\_mapped \leftarrow classify\_children(children)$
 3: if $not\_mapped = \varnothing$ then {All children are mapped}
 4:   for all $r \in mapped$ do
 5:     $r \leftarrow remap(r, map)$ {Remap the blocks in $r$ to their location in OPT
        according to the map table.}
 6:     $issue(r)$
 7:   end for
 8: else if $mapped = \varnothing$ then {No child is mapped}
 9:   if type($req$)==write AND $space\_in\_opt(req, map)$ then
10:     $req \leftarrow remap(req, map)$
11:
12:   $issue(req)$
13: else {Partial translation}
14:   for all $r \in mapped$ do
15:     $r \leftarrow remap(r, map)$
16:     if type($r$)== write then
17:       $borg\_map\_set\_dirty(r)$
18:
19:     $issue(r)$
20:   end for
21:   for all $r \in not\_mapped$ do
22:     if type($r$)== write AND $space\_in\_opt(r, map)$ then
23:       $r \leftarrow remap(r, map)$
24:       $borg\_map\_set\_dirty(r)$
25:
26:     $issue(r)$
27:   end for
```

some pages of a read I/O request are mapped and the mapped entries are clean, the entire I/O is indirected to the file system; this optimization reduces the seek overhead incurred to serve the request partially from the BOPT and the rest from the FS (Line 19). For a write request, when no mapping exists for any of the pages, the blocks are written to a *write-buffer* portion of the BOPT reserved for assimilating write requests (if space permits) along with an additional request for updating corresponding mapping entries in the `borg_map` (Lines 8-12) . For partially-mapped writes, the mapped blocks are indirected to their BOPT locations; the unmapped pages are also absorbed in the write-buffer, space permitting, otherwise these are issued to the FS (Lines 14-26).

Figure 3.8 provides an illustration.

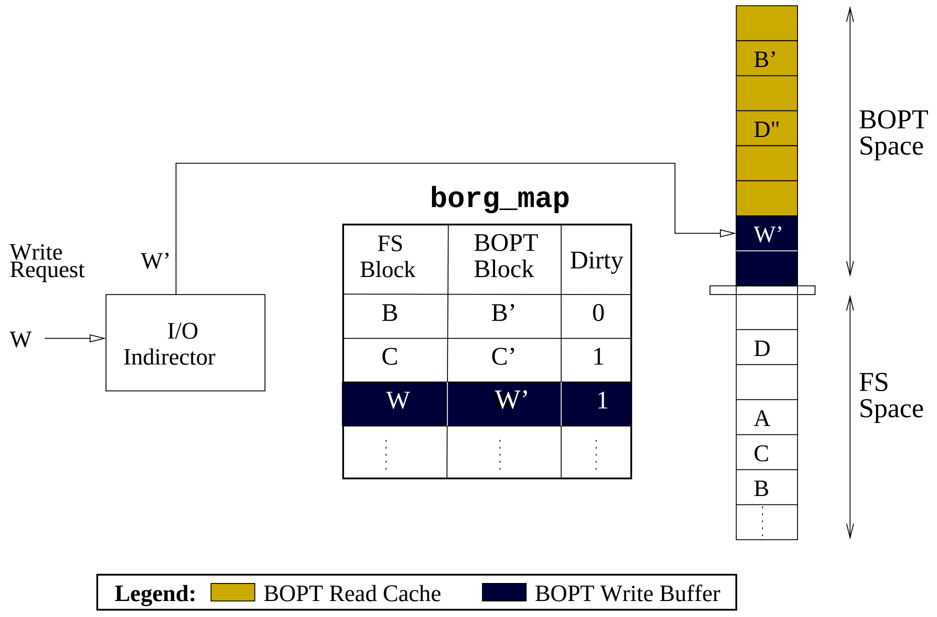


Figure 3.8: I/O Indirection.

3.4.6 Kernel Data Structures

The persistent data structure `borg_map` is implemented as a radix tree such that given an FS LBA, the BOPT LBA can be retrieved efficiently and vice-versa. It also maintains the *dirty* information for the BOPT LBAs. For every page of 4KB, BORG stores 4 bytes each for the forward and the reverse mapping and one dirty bit. If all the pages in the BOPT of size S GB are occupied, the worst case memory requirement is $2 \times S$ MB (S MB for forward and reverse mapping each), and $\frac{S}{2^5}$ MB for the dirty information. Thus, in the worst case, `borg_map` requires memory of 0.25% of the size of the BOPT partition, an acceptable requirement for kernel-space memory.

3.5 Implementation Issues

In this section, we discuss the particularly challenging aspects of the BORG implementation that help address data consistency and overhead.

3.5.1 Persistent Indirection Map

Since BORG replicates popular data in the BOPT space, the system must ensure that reads are always up-to-date versions of data, including after a clean shutdown or a system crash. BORG implements a persistent `borg_map`, which is distributed within read-cache and write-buffer segments of the BOPT. Map entries on-disk are updated (along with their in-memory version) each time the BOPT partition is reconfigured or when a new map entry is added to accommodate a new write absorbed into the BOPT. Upon writes to an existing BOPT mapped block, its indirection entry in the in-memory copy of the reconfiguration map is marked as dirty, once the I/O is completed. To minimize overhead for BOPT writes, we chose not to maintain dirty information in the on-disk copy. Upon reboot after an unclean shut down, all entries in the persistent map are marked as dirty and future IOs to these blocks are directed to the BOPT. This guarantees that even on a system failure, the most up-to-date data is accessed upon reboot.

3.5.2 Indirection during Reconfiguration

During reconfiguration, data is copied from *source* to *destination* locations. I/O requests issued by applications during reconfiguration of associated data give rise to race conditions. While it is perhaps simpler to postpone servicing the application I/O request until the reconfiguration operation for the associated data is completed (to ensure data consistency), this (substantial) delay affecting the application can be avoided. We designed special policies to handle *read* and *write* operations issued by the application during reconfiguration. If the application issued a *read* request to a page being moved due to reconfiguration from *source* to *destination*, the indirector dispatches the read to the *source* page location. For a *write* request, the request is issued to the *destination* location, the page movement is discontinued, and the `borg_map` is updated. These policies help to alleviate the overhead the reconfiguration causes to the user level applications by minimizing I/O wait time for foreground I/O operations.

3.5.3 Optimizing Reconfiguration

Consider a set L of n LBAs, $L_1, \cdots, L_n$, sequentially located in the BOPT space. L forms a *chain* if $\forall L_i \in L$, where $L_i \neq L_n$, L_i has to be relocated to location $L_i{+}1$ and L_n is an outgoing block. If L_n, has to be relocated to L_1 within the BOPT, L forms a *cycle*. Information about chains and cycles, that occur exclusively for the relocated blocks, can be used to further optimize data movement during the reconfiguration operation. If a cycle exists, it is broken by copying the last block L_n back to the FS (if dirty) and then deleting the plan entry for that block; an additional plan entry is then created to mark this as `incoming` block to L_o. Next, all remaining blocks belonging to the same chain/cycle are copied to their new locations in the BOPT. To do so, the reconfigurator issues all reads to the source locations in parallel; once all reads have been completed, it issues all the writes in parallel, in each case allowing the I/O scheduler to optimize the request schedule.

3.5.4 Request Splitting

BORG maintains metadata at the granularity of a *page* (rather than *block*) to reduce metadata memory requirement (by 8X for Linux file systems). Consequently, the indirector must carefully handle I/O requests whose sizes are not multiples of the page-size and/or which are not page-aligned to the beginning of the target partition. We address this issue via I/O request splitting and page-wise indirection, techniques borrowed from our earlier work on EXCES [**?**], a block-layer extension that manages a persistent cache for reducing disk power consumption.

3.5.5 Module Unload

BORG is dynamically included in the I/O stack by substituting the `make_request` function of the device targeted for performance optimization. While module insertion is straightforward, module removal/unload must ensure that all the data from the BOPT has been copied back to their original locations in the file system and handle foreground I/O correctly. Once again, BORG uses techniques from EX-

CES [?] and flushes dirty BOPT blocks to their original locations in the file system upon removal. To address race conditions caused when an application issues an I/O request to a page that is being flushed to disk, BORG stalls (via `sleep`) the foreground I/O operation until the specific page(s) being flushed are written to the disk.

3.6 Evaluation

In this section, we compare the performance of BORG with a *vanilla* system in which all the blocks are located in their original FS space under various workloads to answer the following questions.

(i) How well does BORG perform?

We use the total disk busy time (i.e., excluding all idle periods) as the primary metric of performance. Due to BORG's optimizations, apart from the potentially improved head positioning times, the degree of merging of requests may also be increased when compared with the vanilla configuration, thus changing the request pattern itself. Thus, the more common I/O response time metric is an ill-suited choice. The total disk busy time (henceforth simply referred to as disk busy time) is also robust against the trace-replay speedups we employ in some of our experiments.

(ii) Why is BORG effective?

We would like to know if BORG performance gains are because of the sequentiality or the proximity of data (or both) in the BOPT. We use two metrics, *average seek distance* and *non-sequential accesses percentage* for this purpose. The latter is measured as $\frac{\#\ \text{Seeks}}{\#\ \text{BlocksRead}}$. Since non-sequential accesses are at least an order of magnitude less efficient than sequential accesses, even a small reduction in this metric may lead to substantial performance benefit.

Further, to evaluate the importance of capturing access patterns in BORG, we compare BORG against a strawman *hot-block caching* mechanism that uses a greedy heuristic and places the most popular blocks in the order of the highest number of accesses without taking into account access patterns.

(iii) When is BORG not effective?

BORG can degrade the system performance for certain workloads. We evaluate BORG for varying workloads to determine in which cases it could perform worse than the vanilla system.

(iv) How much CPU resource overhead does BORG incur?

While the upper bound on memory overhead was examined in § 3.4.6, the CPU resources consumed by BORG should also be within acceptable limits. We use the execution times for various stages of BORG as an approximate measure of CPU resource utilization.

(v) How is BORG affected by its parameters?

We perform a sensitivity analysis of BORG to its parameters - reconfiguration interval, BOPT size, and BOPT write buffer fraction - to evaluate their impact on performance.

(vi) How does BORG perform with RAID systems?

To observe the impact of BORG on multi-disk systems, we evaluate BORG performance on different RAID configurations. We use the average per-disk busy times as our metric for the RAID experiments.

Experimental Setup. All experiments were performed on machines running the Linux 2.6.22 kernels. We used host machines, `O1` through `O5`, with differing hardware configurations and disk drives (Table 3.5). We used `reiserfs` for `O1` and `O3`, and

Host	**Make**	**Model**	**RAM (MB)**	**Capacity (GB)**		
				Total	**FS**	**BOPT**
`01`	WD	2500AAKS	1024	250	46	1
`02`	WD	360GD	1024	39	24	2
`03`	Maxtor	6L020L1	1024	20	15	2
`04`	WD	2500AAKS	1024	250	180	8
`05`	Maxtor	6L020J1	1536	20	8	1

Table 3.5: Experimental test-bed details.

`ext3` for the rest. No additional hardware was required to implement BORG.

We conducted four different sets of experiments. The first set uses week-long traces of a developer's system and a Subversion control server (SVN). The second experiment is an actual deployment of a web server that mirrors our CS department's web server. The third experiment evaluates BORG performance in a virtual machine environment. The fourth experiment evaluates the performance improvement due to BORG for application start-up events.

In each experiment, we performed 4 reconfigurations equally spaced in time; this gave us a reasonable number of phases for detailed analysis. By not choosing more favorable times such as idle disk periods based on well-known diurnal workload cycles, we would only over-estimate the overhead of BORG during reconfiguration. We further discuss the selection of this parameter in § 3.6.5 and § 5.2.2. Finally, we use the notation R_i and N_j in various graphs to denote reconfiguration phase i and non-reconfiguration phase j respectively.

3.6.1 Trace Replay

To evaluate BORG under realistic workloads, we conducted trace replay experiments using SVN server and developer workloads described in Table 3.1. For the traces and the replay, we used *blktrace* and *btreplay* respectively [**?**]. We used an

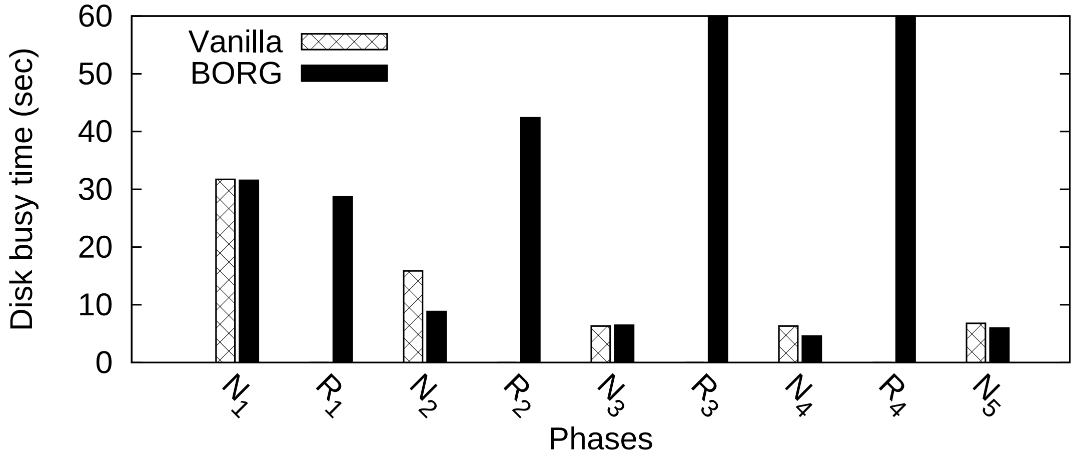


Figure 3.9: Disk busy times in various phases of the SVN server trace replay.

acceleration factor of 168X that reduces the experimentation time from one week to a manageable one hour after verifying that the resultant block access sequence was unaffected. The trace-playback acceleration factor was reverted to 1X during each reconfiguration operation to accurately estimate reconfiguration overhead. Since we only measure disk busy times, the comparison between normal and reconfigurations phases remains valid despite the varying acceleration factors. To understand the impact of BORG on user perceived times we also measured the total processing times for per-request I/O response time for each of the phases. This metric reflects the time for the complete life cycle of an I/O request including the time needed to put a request in the I/O scheduler queue, to dispatch the request to the storage device and to service the request from the device.

SVN Server

For the SVN server trace replay, we used the host `O2` (Table 3.5). The write buffer size was set to 20% of the BOPT size. Figure 3.9 shows the disk busy times during different phases of the experiment. R_i and N_j correspond to reconfiguration phase

i and non-reconfiguration phase j respectively. R_3 and R_4 are beyond the y-axis range with values of 272 and 564 seconds respectively. In all the reconfiguration phases the busy time with BORG is notably higher than the vanilla case. This is due to substantial head movement during reconfiguration for relocating blocks. The longest reconfiguration phase lasted approximately 10 minutes. R_3 and R_4 have substantially higher busy time than the previous two reconfigurations. After trace analysis, we found that while the amount of data movement was similar across the four reconfiguration instances, in the latter two phases, the I/O scheduler merge ratio and the sequential disk accesses dropped dramatically; this can be attributed to the blocks relocated within the BOPT being spread out more than in the previous reconfigurations. However, As is evident by the vanilla busy times, the foreground activity during these intervals are negligible and thus the increased reconfiguration durations have little impact to foreground I/O.

In all the non-reconfiguration phases, each of which lasted 1.75 days approximately, BORG offers better performance for foreground I/O than the vanilla configuration. In the best case (range N_2), BORG decreases the disk busy time by approximately 45%. This is a surprising result, since as per Figure 3.2, the working-set for this workload undergoes rapid shifts. The explanation lies in the fact that the SVN server is a write-intensive workload and the BOPT write-buffer is successful in sequentializing a rapidly changing, possibly non-sequential, write workload. Analysis of the block level traces revealed that with BORG, the non-sequential access percentage reduced from 1.70% to 1.15%, and the average seek distance reduced from 704 to 201 cylinders during the non-reconfiguration phases.

Table 3.6 shows the total processing times,that reflects the impact BORG has on user-perceived times, for the I/O requests in their different phases. As before, the reconfiguration phases show higher response time with BORG since these requests

	SVN		Developer	
Phases	`vanilla`	BORG	`vanilla`	BORG
N1	269	268	89	87
R1	2	5810	1	662
N2	95	54	71	60
R2	11	3257	1	766
N3	369	367	8	6
R3	220	8070	2	891
N4	319	223	5	3
R4	426	7776	1	917
N5	349	317	8	8

Table 3.6: Per request I/O response times (in msecs) for different phases of SVN and Developer workloads.

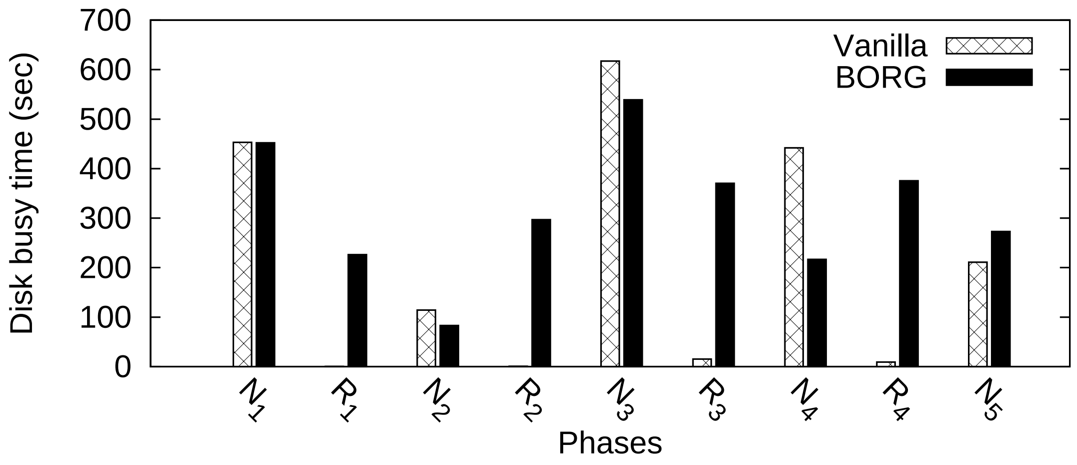


Figure 3.10: Disk busy time in various phases of the developer trace replay.

are issued asynchronously and result in higher queueing times.

Developer

For the developer trace replay, we used the host `O1` (Table 3.5) with the BOPT write buffer set to 40% of the BOPT size. Figure 3.10 shows the disk busy time for this experiment in various phases. R_i and N_j correspond to reconfiguration phase i and non-reconfiguration phase j respectively. Table 3.6 shows the total I/O time. With this workload, the longest measured reconfiguration phases were R_3 and R_4 which lasted approximately 7 minutes each. We observe reduced disk busy times (13% to 50% reductions) across the non-reconfiguration periods, except for N_5 which shows an increase of 25%. Overall, the developer workload is a write-mostly workload and thus, largely conducive to BORG optimizations. Analysis of the block level traces revealed that overall, the non-sequential access percentage reduced from 3.93% to 3.30%, and the average seek distance reduced from 1203 to 782 cylinders when using BORG.

3.6.2 Web Server

To evaluate BORG in a production server environment, we made a copy of the our Computer Science department web server on the `O4` machine (see Table 3.5), and replayed all the web requests for a week. During this week a total of 1137234 requests to 256017 distinct files were serviced. We set BORG to reconfigure four times during this period, using an BOPT of 8GB ($< 5\%$ of the 180GB web server file system). To measure the influence of the I/O history, we conducted two sets of experiments. In the first experiment, we used all the traces gathered from the beginning of the experiment as input to the reconfigurator (*cumulative*). For the second, we only used the portion of the trace corresponding to the period since the last reconfiguration (*partial*).

Figure 3.11 shows the improvements in disk busy time across various non-reconfiguration and reconfiguration phases during the experiment. For both the cumulative and partial experiments, BORG reduces disk busy time in all non-reconfiguration phases with reductions ranging from 14% to 35% for cumulative and 5% to 39% for the partial configuration, except N_5 for cumulative which reported a 6% increase for cumulative due to drastic change in the last interval's workload.

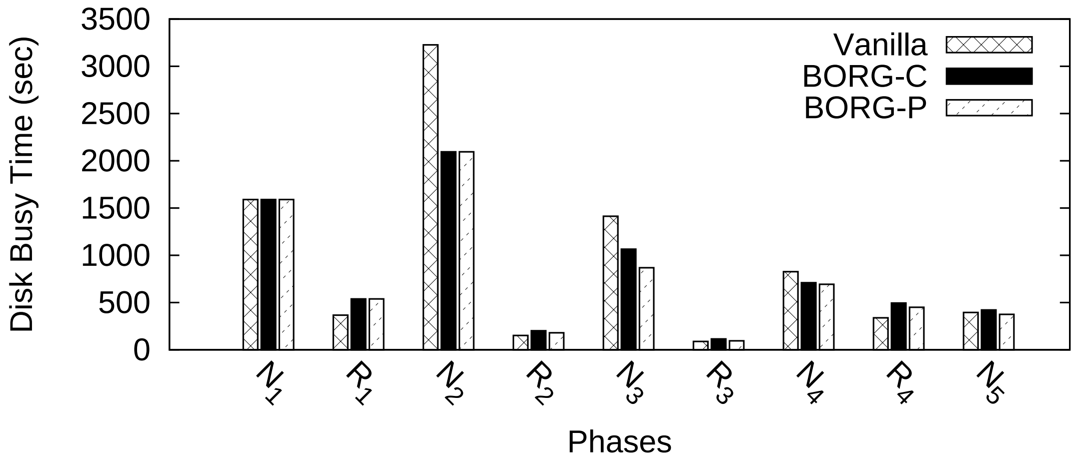


Figure 3.11: Disk busy time for the week long web log replay. Borg-C *and* Borg-P *correspond to using cumulative and partial traces respectively.*

Disk busy times in reconfiguration phases are typically higher due to the overhead of copying data to the BOPT. Nevertheless, BORG was able to obtain overall reductions of 14% and 18% for cumulative and partial configuration. It is interesting to note that short term training yielded better results in this case, perhaps due to greater influence of short term locality.

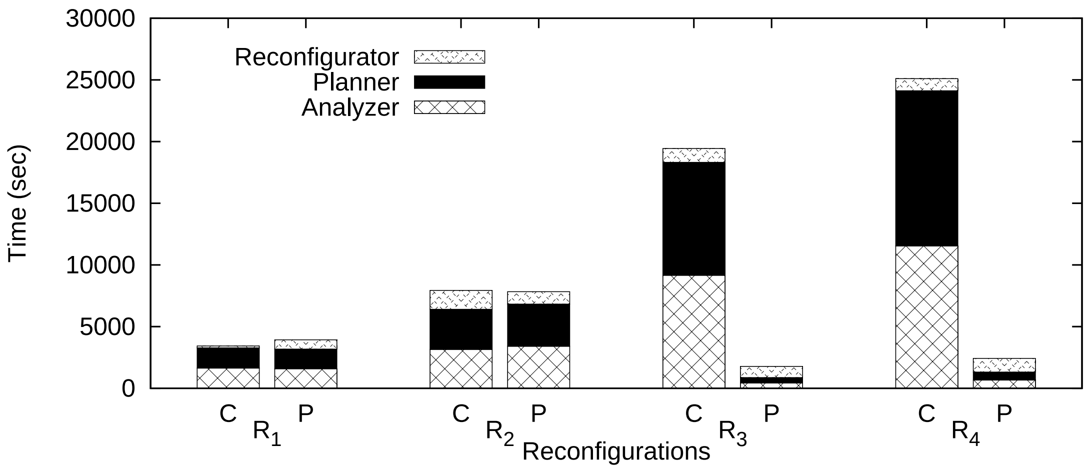


Figure 3.12: BORG overhead. *Bars* C *and* P *represent the cumulative and partial traces experiments respectively.* R_i *indicates the ith reconfiguration.*

Next we examine operational overhead of BORG. Figure 3.12 shows the amount of time spent in each phase of the reconfiguration. With cumulative traces, the time required for the analyzer and planner phases increases linearly. While the planner and analyzer stages can run as low-priority tasks in the background, we must point out that the current implementation of BORG analyzer and planner stages are highly unoptimized and there is substantial room for improvement. We discuss possible improvements for both subsystems in §5.2.2. With partial traces, the time increases until the second reconfiguration, but then decreases and stays almost constant for the following ones, indicating a gradually stabilizing working-set.

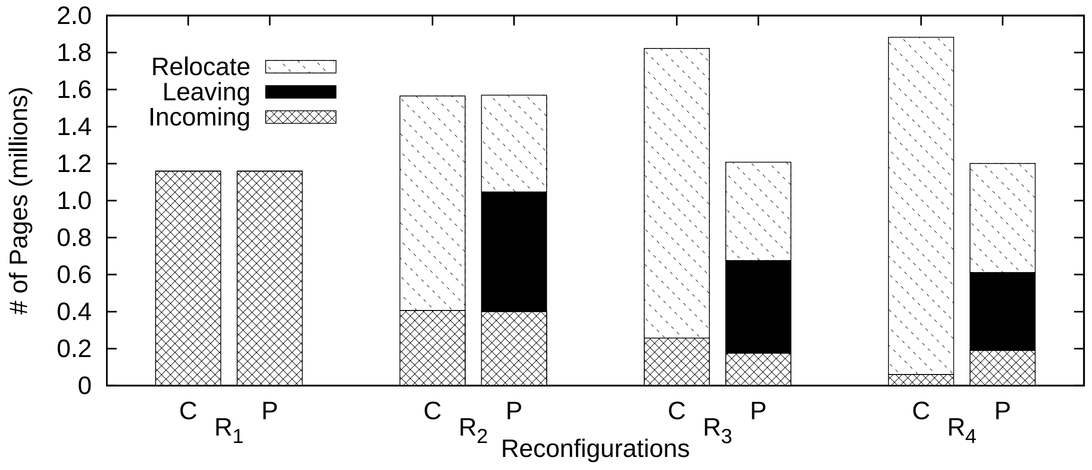


Figure 3.13: Differences in the reconfiguration plans.

To explain this further, we examined the reconfiguration plan divided by the type of operation (refer to § 3.4.4), presented in Figure 3.13. We note that the size of the plan consistently increases when using cumulative traces and most of the movements correspond to page relocates, which are page movements within the BOPT itself. The story is quite different for partial traces, where we see pages not accessed in the past interval leaving the BOPT, resulting in a smaller working set in the BOPT and thereby reducing the amount of work done by the analyzer, planner, and reconfigurator stages.

3.6.3 Virtual Machines

BORG has the potential to significantly improve the performance of virtualized environments, by co-locating multiple virtual machine (VM) localities spread across a physical volume. We evaluated the impact on the per-VM boot time and the overall performance of virtual machines by deploying BORG in a Xen [?] virtual machine monitor. We created four VMs, each with 64MB memory and 4GB physical partition on the host `05` (refer to Table 3.5). For evaluating boot-time improvement, we trained BORG with the boot-time events of all the virtual machines. BORG showed an almost 3X average improvement in VM boot-times - 167 seconds with vanilla and 65 seconds with BORG.

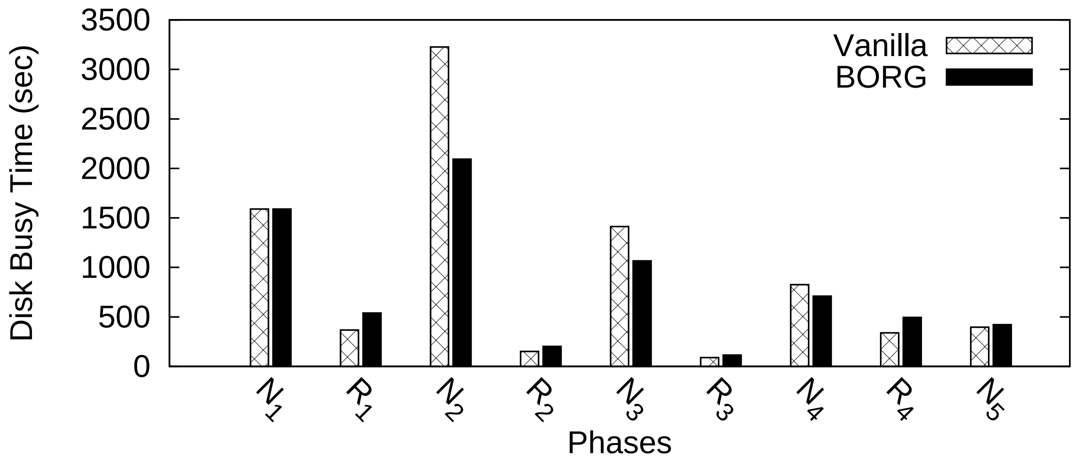


Figure 3.14: BORG with a VMM.

To measure normal execution performance improvement for the VMs, we ran the Postmark benchmark which emulates an e-mail server and creates and updates small files. We set the number of files to be 2000 in 500 directories and performed 200,000 transactions. We reconfigured BORG after every 20% of the benchmark was executed with the training set including I/O operations from the start of the execution of the benchmark. The results for the I/O performance are shown in

App	**Start-up time**		**Rand. I/O %**		**Avg seek (#cyl)**	
	V	B	V	B	V	B
`firefox`	3.71	2.32	2.7	1.2	132	37
`oowriter`	5.30	2.74	3.8	0.2	193	20
`xemacs`	7.26	2.72	2.1	0.3	87	9
`acroread`	6.20	2.65	4.6	0.1	39	9
`eclipse`	4.12	1.52	2.5	0.3	198	29
`gimp`	3.62	3.66	2.5	2.1	102	63
`ooimpress`	5.18	1.97	2.7	0.3	61	39

Table 3.7: Application start-up time improvement. V: vanilla, B: BORG.

Figure 3.14. As before, the reconfiguration phases see a increased disk busy times with BORG. For the normal operation, as the training set increases, the disk busy times with BORG starts decreasing. Overall, there is an average decrease of 6% in busy time during the non-reconfiguration phases. However, this improvement is not consistent; performance degrades substantially even during normal operation in the early stages of the benchmark. The *loss of process context* inside the VMM is a key problem that tends to convert sequentially laid out files into non-sequential upon reconfiguration. We believe that making BORG aware of process context inside the VMM [?] can substantially improve the BOPT layout, resulting in much greater performance benefit.

3.6.4 Application Start-up

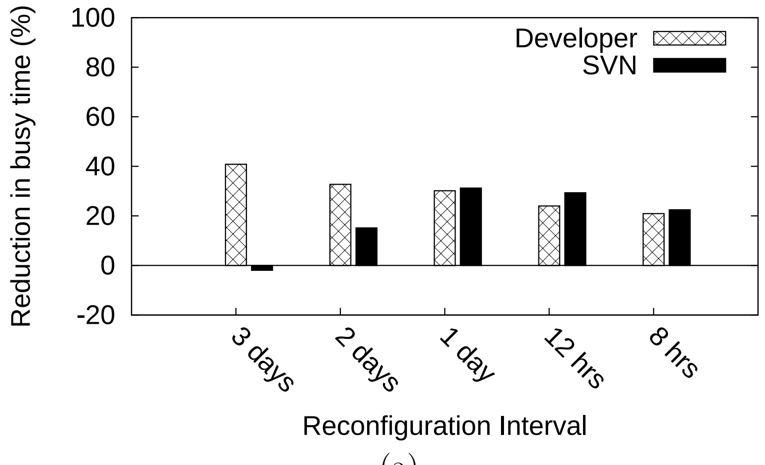
We evaluated the impact of BORG on I/O-bound start-up phase for common desktop applications using host `O3`. We first trained the system for a duration of approximately four hours, during which we invoked a subset of the applications listed in Table 3.7 (but specifically excluding `gimp` and `ooimpress`) multiple times for performing common office tasks. We invalidated the page cache periodically to artificially dilate time and simulate system reboots. Table 3.7 shows the difference in application start-up times, the percentage of sequential accesses and average seek overhead.

For the applications that were used in training, it can be observed that there is a noticeable improvement in the I/O time with BORG - at least 43% for `oowriter` and up to 67% for `eclipse`. Further, it is interesting to observe that although the percentage of sequential I/Os decreases for `oowriter` and `acroread` with BORG, there is an overall improvement in I/O performance, possibly due to a reduction in the rotational overhead . There is barely any difference in the performance for untrained application `gimp`. However, although `ooimpress` was not used in the training, its start-up user-time shows an improvement of 62% in the average I/O time; this can be attributed to large shared libraries also used by the `oowriter` which was included in training.

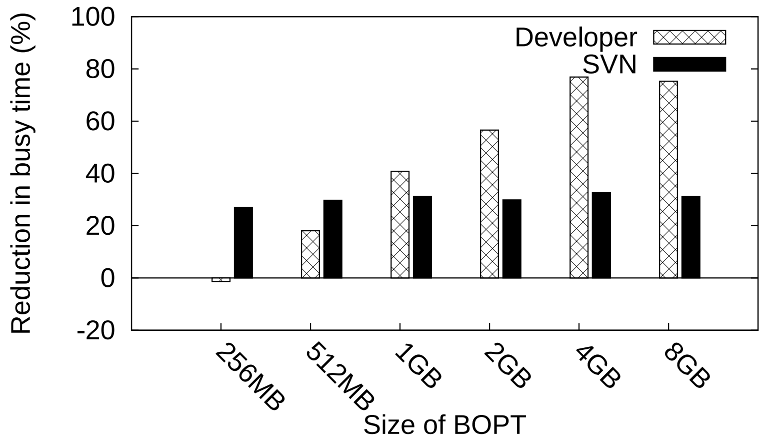
3.6.5 Sensitivity Analysis

To gain maximum performance improvement with BORG its configurable parameters – the reconfiguration interval, the BOPT size, and the BOPT write buffer fraction — must be carefully tuned for a given workload.

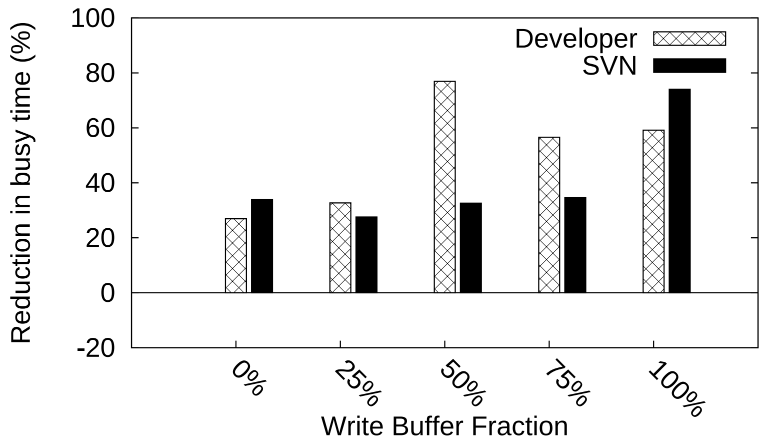
To better understand the effects of these parameters, we replayed the developer and the SVN workload traces on host `O1` varying each of these parameters over a range of values. In all the experiments, the trace replay begins at the same starting point, that is after a *base reconfiguration*, which uses the first six hours of the trace as the training period. We measure the relative efficiency of disk I/O using BORG averaged across the non-reconfiguration intervals by reporting the *improvement in disk busy time throughput* (referred to henceforth as "throughput improvement") when compared to a vanilla system.



(a)



(b)



(c)

Figure 3.15: A sensitivity analysis of BORG performance to its configurable parameters.

Reconfiguration Interval

Figure 3.15 (top) shows the percentage improvement over the vanilla system. The reconfiguration interval is varied from 8 hours (18 reconfigurations) to 3 days (1 reconfiguration). To bootstrap the sensitivity analysis, the BOPT size is fixed to 1GB, with 50% reserved for write buffering in this experiment. For the developer workload, as the reconfiguration interval increases the throughput increases, the training set becomes larger, and BORG can more effectively capture the working-set. For the SVN workload, the performance decreases for higher intervals. This is because the SVN working-set changes quite frequently (elaboration in § 3.2 and Figure 3.2).

BOPT size

We use the best-case reconfiguration intervals of 3 days for the developer and a day for the SVN workload from the previous experiment. We vary the BOPT size from 256MB to 8GB, of which the write buffer is always chosen as 50% of the BOPT size. Figure 3.15 (center) shows that as the BOPT size increases, BORG's performance with the developer workload increases since the developer workload has a larger working set. When most of the blocks in the working set can be accommodated in the BOPT, the performance improvement stabilizes. Since the working set size for the SVN workload is relatively smaller, the performance improvement is almost same for the BOPT sizes >256MB.

Write Buffer Variation

From our previous results, we pick an interval of 3 days and 1 day and BOPT size of 2GB and 4GB for the developer and the SVN workloads respectively. We vary the write buffer from 0-100%. Figure 3.15 (bottom) shows that for the developer workload, not having a write buffer results in the lowest throughput. There is a steady increase in performance, peaking at 50% write buffer. Thereafter, it starts falling since read performance begins to degrade due to lesser available read cache. For the write-intensive SVN workload, the performance increases with increase in the write buffer size, since all the writes can be co-located in the BOPT partition.

Configuring BORG parameters

The above experiments indicate that configuring parameters incorrectly can lead to sub-optimal performance improvements with BORG. Fortunately, iterative algorithms can be easily employed to identify better parameter combinations in a straightforward way. Exploring such iterative algorithms more formally is one aspect of our future work.

3.6.6 Alternate layout strategy - Hot block caching

We compare BORG against a `hot block caching` system that copies the most popular blocks in a given workload in the BOPT, ignoring their access patterns. The analyzer and planner components are modified, while the profiler, reconfigurator and indirector components remain the same. The analyzer reads the input trace from the profiler and creates a sorted list of LBAs in the order of frequency of block accesses. The planner reads this list and allocates sequential LBAs for the most popular blocks in the BOPT's read cache. As before, the reconfigurator performs the actual data movement.

We evaluate this layout under two workloads - the SVN server workload and the developer workload. We use `O2` as the host machine with the write buffer set to 20% of the BOPT size in case of the SVN server workload and to 40% of the BOPT size

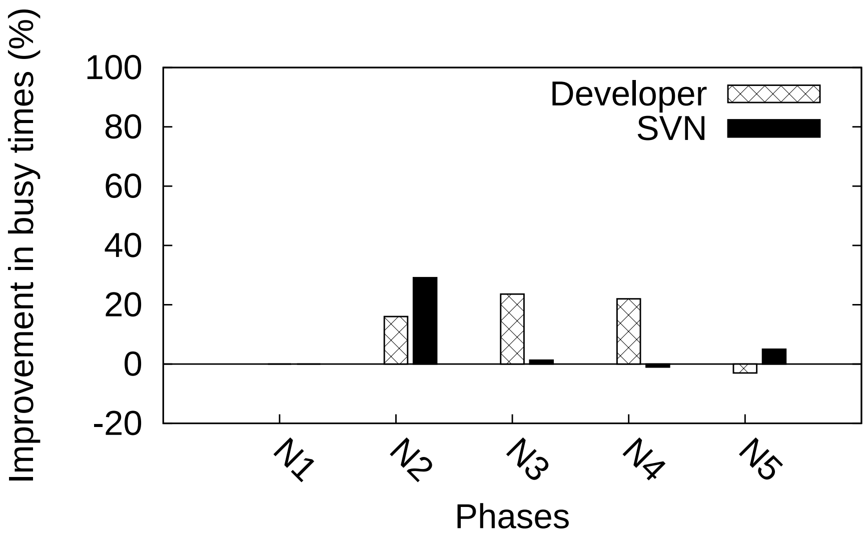


Figure 3.16: Percentage improvement in disk busy time with BORG during various phases of the developer and the SVN workloads.

for the developer workload, based on results obtained in the previous experiments. As before, the number of reconfigurations were set to 4.

Figure 3.16 shows the resulting improvement with BORG over the *hot block caching* system. As can be observed, in most cases, the original BORG layout does significantly better for both the reconfiguration and the non-reconfiguration phases. For the non-reconfiguration phases, BORG outperforms *hot block caching* by up to 37% for the SVN and 24% for the developer workloads. This is because in BORG, in addition to the advantage of reduced seek times due to colocation of data, using the popular access patterns also improves the sequentiality. This is corroborated from the analysis of the traces where the sequentiality in the `hot blocks caching` method is lower by 3.3% for the SVN workload and 4.09% for the developer workload, relative to BORG. It can thus be inferred that the performance improvement in BORG is not only because of the co-location of the frequently accessed blocks, but also because of the increased sequentiality obtained by allocating blocks contained in popular access patterns on contiguous LBAs in the BOPT.

It should be noted that for the phase N_4 the percentage improvement with BORG reduces by 6%.

3.6.7 BORG on RAID

We evaluated BORG with the three most popular RAID configurations used today:

- RAID-0, where data is evenly striped across two or more disks,
- RAID-1, which mirrors data on two or more disks, and
- RAID 5, where data is striped and parity for the stripe is distributed across three or more disks.

We used the SVN and the developer workloads to quantify the disk busy times. We also did a sensitivity analysis to understand the behaviour of BORG when varying BORG parameters.

Experimental Setup: We used software RAID for each of the configurations. All our RAID systems were setup using WD 360 disks of size 39 GB. The BOPT partition was set to 2 GB for the developer workload and to 1 GB for the SVN workload, based on previous sensitivity tests for single disk evaluation. The write buffer was set to 50% of the BOPT for the developer and the SVN server.

Since we used multiple disks for these experiments, we used the average per disk busy time as our metric which was calculated as the average of the disk busy times for all the disks constituting the RAID configuration. For instance, for a RAID-0 system with 2 disks, `sda` and `sdb`, the average per disk busy time is the average of the busytimes of `sda` and `sdb`, i.e. $\frac{busytime_{sda}+busytime_{sdb}}{2}$

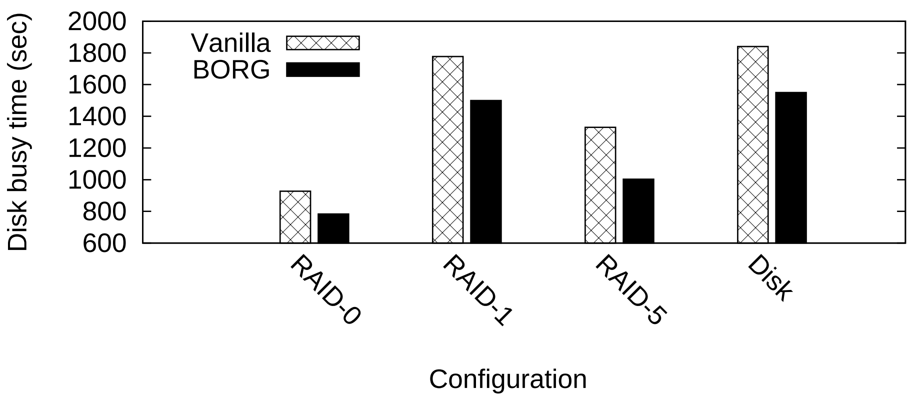
3.6.8 Performance impact

To evaluate BORG on RAID systems, we did a trace replay using the *btreplay* tool for the developer and the SVN workloads for each of the RAID configurations. We performed 4 reconfigurations. The traces were played at 168X during the non-reconfiguration periods and were reduced back to 1X during the reconfiguration phases. For this experiment, the minimum number of disks required for each of the RAID configuration were used, that is two for RAID-0 and RAID-1 and 3 for RAID-5.

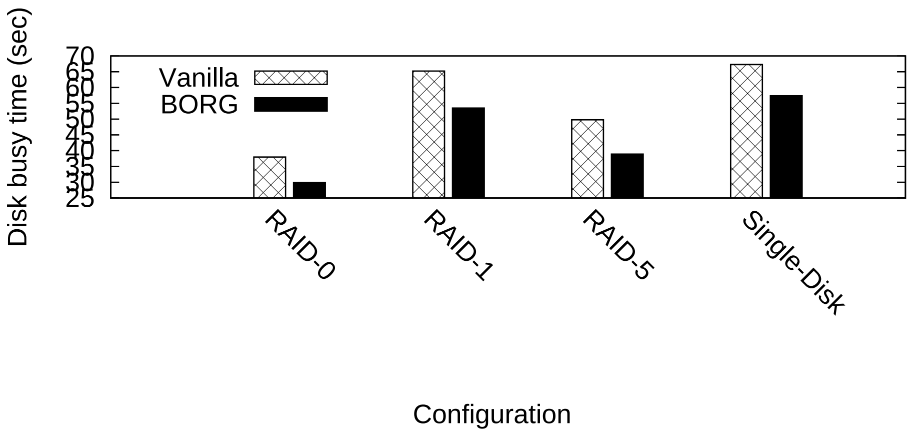
Figures 3.17 (a) and (b) shows the average per disk busy times for the BORG non-reconfiguration phases for the RAID systems for the Developer and the SVN workloads respectively, The first three clusters show the results for RAID-0, RAID-1 and RAID-5 systems respectively. The fourth cluster shows the results for the single disk experiments and is included here for the sake of comparision.

As can be seen, BORG shows considerable improvement in performance for all the RAID configurations. Since data is striped across the disks, BORG performs best with RAID-0. RAID-1 mirrors data on a seperate disk and hence for the two-disk configuration, the average busy times for both the devices are similar to those of the single disk system. Data is striped across the disks in RAID-5, and hence its shows an improvement in performance relative to the single disk systems. However, it also incurs additional I/O overheads to maintain the parity information and hence it does not perform as good as the RAID-0 system.

Figure 3.18 shows similar trends for the reconfiguration phases during the reconfiguration phases, where the busy time is highest for RAID-1 systems and lowest for RAID-0 systems.

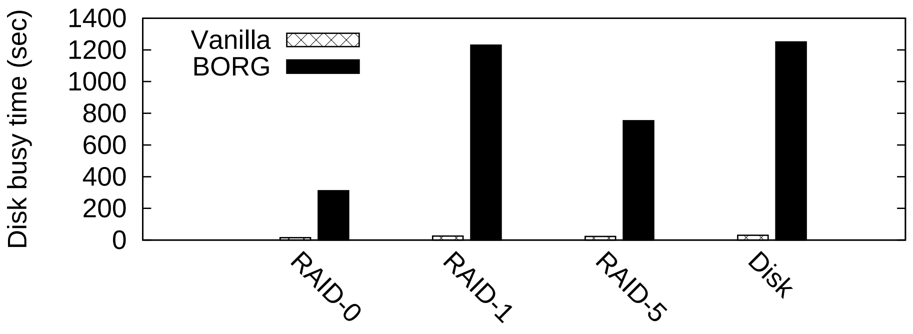


(a) Developer Workload

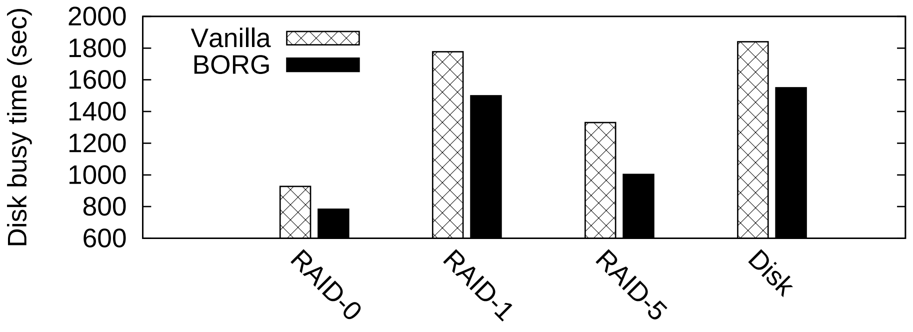


(b) SVN workload

Figure 3.17: Disk busy times for Non-reconfiguration phases on different RAID configurations



(a) Developer Workload



(b) SVN workload

Figure 3.18: Disk busy times for Reconfiguration phases on different RAID configurations

3.6.9 Sensitivity Analysis

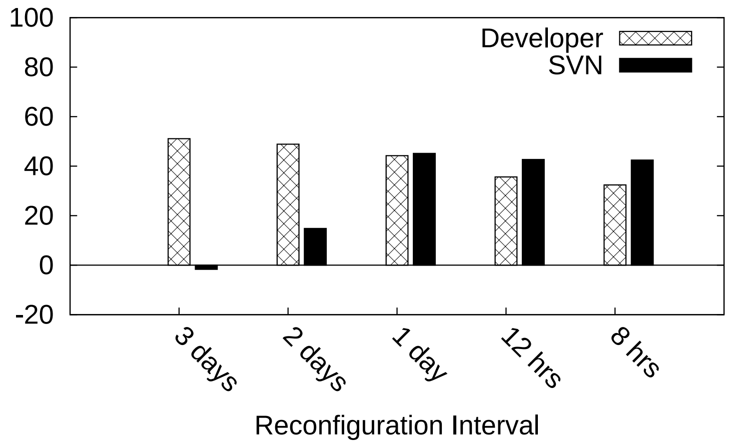
We did a sensitivity study of the BORG parameters i.e. reconfiguration interval size of the BOPT and the size of the write buffer on the different RAID configurations. We replayed the SVN and the developer workload traces on the RAID configurations for this study.

BORG parameters

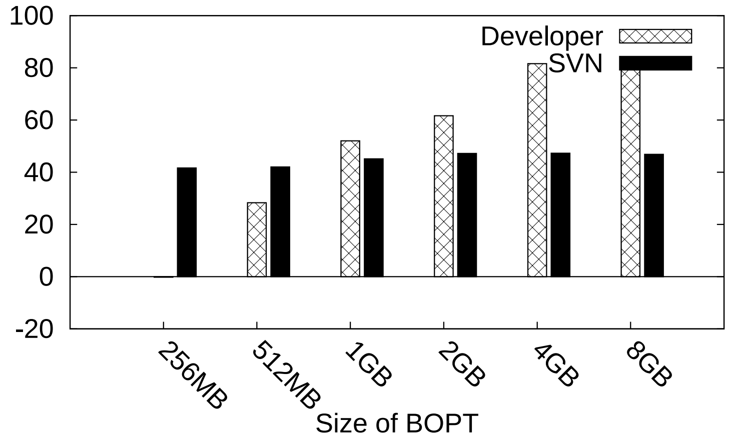
For all the three RAID systems, RAID-0, RAID-1 and RAID-5, we followed the same sequence of steps as in Section 3.6.5. In the first experiment, we set the BOPT size to 1 GB and the write buffer to 25% and varied the reconfiguration interval. Next, we choose the interval with the highest performance gain for each of the workloads and varied the BOPT size. Lastly, we choose the best interval and BOPT size and varied the BOPT write buffer.

The results for this experiment for a RAID-0 configurations are shown in Figure 3.19. As is evident, the results follow the same trend as in case of the single disk sensitivity study. Similar results were obtained for the RAID-1 and the RAID-5 configurations.

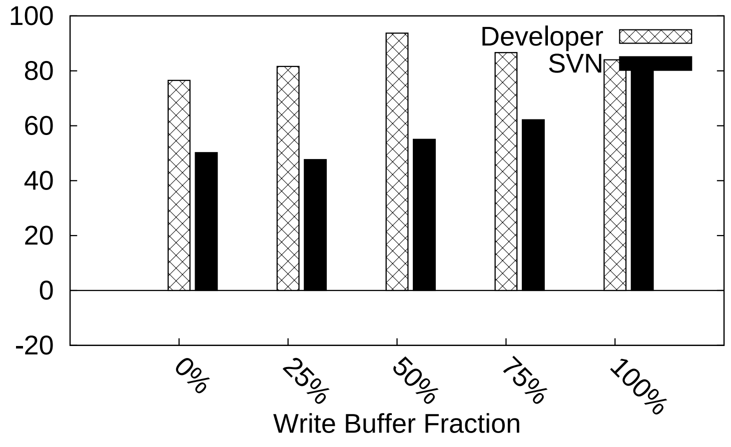
It can be observed that RAID systems shows higher performance gains than single disk systems. It is our surmise that this is because of the greater bandwidth available in case of RAID systems. The increased sequentiality in BORG due to the allocation of popular block access patterns to contiguous LBAs can better use the higher bandwidths to deliver greater performance improvements.



(a)



(b)



(c)

Figure 3.19: A sensitivity analysis of BORG performance, on a RAID-0 system, to its configurable parameters.

3.7 Related Work

Significant research has been done in dynamically adapting data layout on disk drives to improve performance. Our work on BORG is primarily motivated by several recent work, including IBM's autonomic computing initiative [?], CMU's self-$\star$ work [?], ALIS [?], and FS2 [?], and the early works of Vongsathorn [?], Ruemmler [?], and Akyurek [?] on adaptive disk data layouts.

3.7.1 Block level approaches

Early work [?] on optimized data layout argued for placing the frequently accessed data in the center of the disk. Vongsathorn *et al.* [?] and Ruemmler and Wilkes [?] propose using *Cylinder Shuffling.* Ruemmler and Wilkes specifically demonstrated via simulation studies that reducing the shuffling quantum and performing relatively infrequent shuffling (once a week) led to greater improvement in I/O performance. In Akyurek and Salem's work [?], the frequently accessed blocks are copied to a reserved space in the center of the disk. The authors demonstrated via simulation studies the advantages of *copying* over *shuffling* and the importance of reorganization at the block (rather than cylinder) level. These early *data clustering* approaches emphasized on process- and access-pattern- agnostic *block counts* to perform the data reorganization. Early work on mining disk access patterns at the block-level addressed different end goals. Kuenning *et al.* [?] and Griffoen *et al.* [?] suggest inferring frequently accessed data by more complex mining techniques for caching in mobile systems and reducing file system latency respectively.

Researchers have also investigate self-optimizing RAID systems. Wilkes *et al.* proposed HP AutoRAID [?], a controller-based solution, that transparently adapts to workload changes by using a two-level storage hierarchy; the upper level provides data redundancy for popular data while the lower level provides RAID 5 parity protection for inactive data. Work on eager writing [?] and distorted mirrors [?] address mirrored/striped RAID configurations primarily for database OLTP workload (which are characterized by little locality or sequentiality) that choose to write to a free sector closest to the head position on one more disk drives. Yu *et al.* [?] propose a different approach for trading disk capacity for performance in a RAID system, by storing several *rotational replicas* of each block and using a rotationally latency sensitive disk scheduler. While we are yet to explore BORG's use in multi-disk systems, the optimizations used in BORG are different and mostly complementary to the above proposals, whereby BORG attempts to capture longer-term on-disk working-sets within a dedicated volume.

Hu *et al.*'s work on Disk Caching Disk citeHY95 uses an additional logging disk (or disk partition) to perform writes sequentially and subsequently, destage to their original locations. Write buffering in BORG is slightly different in that writes to data already in the BOPT partition are written in place. The DCD work does not optimize for data read operations; BORG optimizes reads as well so head movement is substantially restricted.

Among recent work on block reorganization, Salmon *et al.* [?] describe a generic two-tiered architecture that provides the framework for combining multiple heuristics for data reorganization. BORG could possibly be re-factored to provide such hints within this larger framework. C-Miner [?] uses advanced data mining techniques to mine correlations between block I/O requests. They find the frequent sequences from a set of short sequences which in turn infer the correlations between blocks. Some of these techniques can be utilized in BORG to infer complex disk access patterns. The Intel Application Launch Accelerator [?] reorganizes blocks used during application start-up to be more sequential, but does not provide a generic solution to improve overall disk I/O performance of the system. For throughput improvement, Schindler *et al.* have proposed free-block scheduling and track-aligned extents [?, ?, ?, ?] which use intelligent I/O scheduling (rather than block reorganization); these are complementary techniques that can be used in conjunction with BORG.

Among block level approaches, our work is closest to ALIS [?], wherein frequently accessed blocks as well as block sequences are placed sequentially on a dedicated, *reorganized area* on the disk, similar to the BOPT partition used in BORG. There are differences in design and implementation. First, BORG incurs reduced space, maintenance, and metadata overhead since it maintains at most one copy of each data block. The multiple replicas in ALIS can become stale (and therefore unusable) quickly in write-intensive workloads. Further, unlike BORG, ALIS does not optimize write traffic. Finally, the evaluation of ALIS techniques is performed using a disk simulator with trace playback. On the other hand, we implement and evaluate an actual system, thereby having the opportunity to address a greater detail of practical system implementation issues.

3.7.2 File level approaches

In one of the early file oriented approaches, Staelin *et al.* [?] proposed monitoring file accesses and moving frequently accessed files (entirely) to the center of the disk. Log-structured file systems (LFS [?]) offer superior performance for workloads with large number of small writes by batching disk writes to the end of a disk-sequential *log*. BORG writes all data to the BOPT partition to achieve a similar effect, but also attempts to co-locate a majority of read operations with the writes. Matthews *et al.* [?] proposed an optimization to LFS by incorporating data layout reorganization to improve read performance. Their use of *block access graphs* is similar to the *process access graphs* used in BORG. Their LFS-specific solution moves blocks within the LFS partition storing exactly one copy of each block. Since BORG stores two copies, it can optimize for sequential and application-driven deterministic, non-sequential accesses simultaneously. Further, while their approach partitions the graph for independent layout within each FS partition, BORG merges graphs and uses a single BOPT partition to aggressively optimize seek overhead even when multiple partitions are accessed simultaneously by applications.

Researchers have used data-specific layout mechanisms as opposed to the generic and automatic techniques of BORG. Ganger and Kaashoek [?] have advocated collocating inodes and file blocks for small files in C-FFS. In a converse approach, PLACE [?], a gray-box technique, exposes the underlying layout structure to applications, so they can perform custom data placement to improve the I/O performance. Sivathanu *et al.* [?] propose semantically-smart disk systems (SDS) that infer file system semantic associations for blocks. Performance improvement is a secondary goal wherein they propose aligning files with track boundaries. File access patterns can be used to predict future file accesses. Amer *et al.* [?] and Yeh *et al.* [?] use such predictions for purposes of file caching and energy conservation respectively. Similarly, in [?], file predictions are used to conserve energy in mobile computers. Windows XP [?] uses the defragmenter for co-locating temporally correlated file data for speeding up application start-up events. This file system specific solution does not address the issues of shared data between applications. Similar to the Intel approach, it does not target generic I/O optimization for arbitrary workload.

Among file level approaches, BORG is closest to the FS2 [?]. FS2 proposes replication of frequently accessed blocks based on disk access patterns in file system free space. This strategy, unfortunately, also restricts the degree of seek and rotational-delay optimization due to the distribution of free space. Since FS2 may create multiple copies of a block simultaneously, staleness, and consequently, space and I/O bandwidth wastage, become important concerns (similar to those in ALIS); BORG maintains at most one extra copy of each block and its strength is in being a non-intrusive, storage-stack friendly, and file system independent (portable) solution.

3.8 Summary

We presented BORG, a self-optimizing layer in the storage stack that automatically reorganizes disk data layout to adapt to the workload's disk access patterns. BORG was designed to optimize both read and write traffic dynamically by making reads and writes more sequential and restricting majority of head movement within a small optimized disk partition. A Linux implementation of BORG was evaluated and shown to offer performance gains in the average case for varied workloads including office and developer class end-user systems, a web server, an SVN server, and a virtual machine monitor. Disk busy time reductions with BORG across these workloads during non-reconfiguration intervals range from 6% (for the VM workload) to 50% (for the developer server workload), with even greater improvements possible with careful parameter selection within BORG.

BORG performs occasionally worse than a vanilla system, specifically when a read-mostly workload drastically shifts its working set. BORG is able to easily address changing working-sets with a (possibly non-sequential) write workload, since it has the ability to absorb and sequentialize writes inside the BOPT. A sensitivity analysis revealed the importance of choosing the right configuration parameters for reconfiguration interval, BOPT size, and the write-buffer fraction. In summary, we believe that BORG offers a novel and practical approach to building self-optimizing storage systems that can offer large I/O performance improvements in commodity environments.

CHAPTER 4
DISCUSSION

We investigate static and dynamic layout mechanisms comprehensively in Chapters 2 and 3. Although we implement and evaluate these strategies separately to expose their impact on performance, we believe that a combination of these strategies can also be used to improve performance. The choice of the best strategy is contingent on the workload characteristics. An obvious question that arises is how does a system administrator, the target user of the layout optimizations presented in this thesis, determine the layout strategy that provides the most performance benefits for her workload. The choice of the layout strategy depends on the workload characteristics of the system. Obtaining these characteristics is a simplified task that involves analyzing workload traces. This Chapter describes some of the workload characteristics that can be used to govern layout policies by the system administrator and provides some insights into how these can be used.

4.1 Workload characteristics

Workload characteristics can be extracted through low overhead profiling of the block level I/Os using easily available profiling tool such as as `blktrace` [?]. While workloads exhibit numerous complex properties that can be obtained using sophisticated tools or scripts, we identify some of the generic properties that can be easily obtained and analyzed to make intelligent allocation choices.

It should be noted that the allocation policy of a system can be revised if the workload characteristics change and performance degradation is noticed. Continuous or periodic profiling can identify shifts in workload characteristics or performance degradation and can potentially be used to change the allocation policies or to disable them altogether and use legacy allocation schemes implemented by file systems. We identify three categories of workload characteristics that can be used by a system administrator to make the correct choice of the layout mechanisms best suited to the workload.

- Temporal attributes - This category contains attributes that reflect when the data is accessed from the disk drive. It is composed of attributes such as temporal locality, explained in Section 3.2, that indicates how often the working sets change and burstiness of requests, that indicates if I/O requests arrive in batches and if the workload contains any idle time.
- Spatial attributes - Attributes that demonstrate how the data that is being accessed is stored on the storage system fall in this category. I/O sequentiality, partial determinism and non-uniform access frequency distribution constitute this category. Sequentiality of the I/O requests indicate if data is stored on contiguous locations on the disk drive. Partial determinism, described in Section 3.2, identifies the non-sequential accesses in the I/O. Non-uniform access frequency distribution, also described in Section 3.2, shows if the disk contains regions of high frequency access and how these regions are spread over the entire disk. The proposed layout mechanisms attempt to improve the spatial locality of accesses and consequently reduce the seek and rotational delays.
- Operational attributes - This indicates the types of requests such as reads or writes that dominate the workload for durations when the workload access patterns remain static.

The system administrator can infer the choice of the layout strategy based on the knowledge of the above characteristics. A more accurate choice can be made if all the information is available. Since no tool currently exists that automates the

	Temporal Attributes		**Spatial Attributes**			**Operational Attributes**	
	Temporal Locality	Bursts	Seq	Partial Determinism	Non-unif. Freq. Dist.	Reads	Writes
Static layout	-	-	↓	↑	↓	↑	↓
Dynamic layout	↓	↑	↓	↑	↑	-	-
Dynamic layout for static accesses	↑	↑	↓	↑	↑	-	-
Static + Dynamic	-	↑	↓	↑	-	-	-
Sequential	-	-	↑	↓	↓	-	-

Table 4.1: Choosing the befitting layout strategy.

retrieval of workload access characteristics and predicting the appropriate layout mechanism to be used, devising such a tool is a challenging future venture.

4.2 Choosing the right layout

This section provides some insight in to how the above mentioned characteristics can be used to elect a suitable layout mechanism. Table 4.1 summarizes the properties that affect the choice of the proposed layout strategies. The up or down arrow indicates higher or lower degree of the specific characteristic in the given workload respectively.

4.2.1 Static allocation mechanisms

Static layouts are better suited for workloads whose accesses are spread over the entire disk area and the access patterns do not change as long as the application is used. Queries to XML documents, database queries and operations such as search or alignment over genomic strings are examples of such workloads. In general, these layouts can be used for workloads that exhibit the following trends.

- Show a uniform access frequency distribution, where the data accessed is evenly spread across the entire disk area.
- Temporal locality indicates that the access patterns remain static over the lifetime of the application.
- Partial determinism indicates that non-contiguous data access patterns exist.

Since the proposed static layouts mechanisms do not optimize for updates to the data that alter the structure of the tree, it is necessary for the workloads to be read dominant to be able to use them. The tree-placement layout can handle updates if they overwrite the tree node data without altering the tree structure.

4.2.2 Dynamic allocation mechanisms

Dynamic layouts can be used for workloads where access patterns change over a period. A programmer workload, where the set of files being accessed are mostly specific to the project under development is an example of such a workload. A system administrator can choose dynamic mechanisms if workload analysis shows the following properties.

- Accesses to data on disk drives indicate a non-uniform distribution of data on the disk drive.
- Temporal locality indicates changes in working sets during the lifetime of the application hours, days or weeks.
- Partial determinism indicates that non-contiguous data access patterns exist.
- I/O requests arrive in bursts which indicates there is some system idle time during which the data can be reorganized.

The proposed self-optimizing, dynamic layout mechanism, BORG, can be used effectively by tuning the read cache or write buffer sizes based on the ratio of reads and writes in the workload. Also, reconfiguration phases can be triggered in idle times, especially for workloads where the requests arrive in bursts, which is typical of most common workloads today.

4.2.3 Dynamic layouts for static workloads

Dynamic layout strategies can also improve I/O performance for static workloads in certain cases. For instance, BORG can be used to optimize accesses to tree-structured XML data if the access patterns reflect the following. We recommend using dynamic layout mechanisms for static workloads for the following cases.

- A non-uniform access frequency distribution where there are distinct hot regions on the disk.
- Temporal locality shows access patterns for an application do not change.
- Partial determinism indicates that non-contiguous data access patterns exist.

BORG can be effectively used to optimize performance for XML data if the blocks accessed by popular queries can be contained in the BOPT partition.

4.2.4 Combining static and dynamic mechanisms

A system administrator can also choose to use a both of static and dynamic layouts for combined benefits, based on the system requirements. For instance, BORG can be used to handle updates, that modify the XML tree structure, in the tree-placement layouts by using the entire BOPT partition as a write buffer. A node can be deleted from the original file system location. Nodes in the file system locations can be overwritten as long as the tree structure is maintained. Nodes that alter the tree structure can be written to the BOPT partition. The tree placement algorithm can then be executed again to write the updates to the tree back to the file system locations during system idle times. Since the size of the XML documents is typically large, the overheads for re-execution of the tree placement algorithm can be high and the system should have sufficient idle times to execute all the additional I/O requests generated during data reorganization. This strategy may not be feasible for workloads that do not show burstiness or that frequently update data.

4.2.5 Legacy allocation mechanism

Static or dynamic layout strategies can be used if the percentage of sequentiality shows room for improvement and if the workloads exhibits some partial determinism. However, workloads may also access data sequentially where the partial determinism for randomly placed blocks is negligible and the degree of non-uniform access frequency distribution is low where the entire disk is uniformly accessed. Examples of such access patterns include sequential streaming of multimedia data such as video frames or audio files. Also, based on our results, BORG degrades performance for read intensive workloads whose working sets change very frequently. Such workloads may benefit from the existing file system layout strategies that favor sequential accesses.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

This Chapter presents concluding remarks for the work presented in this dissertation and directions in which this work can be extended.

5.1 Concluding Remarks

Processor speeds, network bandwidth and disk and memory capacities show steep improvements every generation and this leads to an increase in application data. However, disk performance improvements occur at a much slower rate mainly because of the challenges in reducing mechanical delays. This results in storage systems being the bottleneck for most modern applications.

The results presented in this dissertation for both static and dynamic placement strategies, suggest a promising approach to alleviate this bottleneck. It is evident that using the same layout mechanisms across workloads with varying access characteristics restricts increases the I/O access time latencies. However, I/O performance can be substantially improved by using the knowledge of workload specific access patterns to (i) reduce rotational latencies by matching workload specific access patterns to underlying disk geometry and (ii) reduce seek latencies by co-locating frequently accessed data on disk drives.

The design and implementation of the strategies presented in this dissertation was an iterative process that underwent numerous changes and refinements. These improvements were the result of our increased understanding of the I/O subsystem, implementation issues or even errors or bugs in the code. The following subsections briefly touch some aspects of the development process for both static and dynamic placement strategies and summarize the lessons learned from the same.

5.1.1 Static Layout Optimization

Development Process: The research presented here started with the design and implementation of the static layout policies. To evaluate the performance of our strategy, we created an analytical cost model to better understand how the characteristics of the hard disk (seek time, rotational speed) and of the tree-structured XML document (height and width) affect the performance of disk access sequences typical to the application. Preliminary results showed up to 35% reduction in access time with our tree-placement policies. Next, we created a prototype, where we modified DiskSim to provide additional interfaces for tree layout, and extensively evaluated the placement algorithms with common query patterns.

Lesson Learned: 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 tree-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. The results obtained with the static layout optimizations encouraged us to pursue similar optimizations for workloads that do not exhibit any specific access pattern.

5.1.2 Dynamic Layout optimization

Development Process: BORG evolved over a period of more than two years. File systems perform a critical task of providing a simple file abstraction to complex storage devices where, each file or directory element is mapped to a set of LBAs.

This mapping is dictated by the data layout technique employed by the file system and influences the performance of the system. Based on this premise, BORG was initiated as a self-optimizing file system rather than a block layer solution. A preliminary analysis of workload access characteristics portended promising improvements in disk access times by copying a percentage of frequently accessed data on a small contiguous, dedicated partition. However, an actual realization of a self-optimizing file system is a lengthy and arduous task and instead of constraining performance improvement due to BORG to a single file system, we decided to let make BORG independent of the file system by implementing it as a module at the operating systems block layer. This provided more flexibility to the user, where the user could avail the advantages of the file system of their choice based on their requirements and use BORG whenever desired. For greater performance improvement, we also added a write buffer in the later stages to absorb all the new writes in the BOPT and restrict the movement of the disk head to lower the access latencies.

Lesson Learned: Our results for optimizations for dynamic workloads show noticeable improvement in disk busy times for most workloads. However, fast-changing read workloads may either not show any improvement, since the blocks in the BOPT are hardly accessed, or in some cases degrade performance when the disk-head alternates between the BOPT and the file system partition and consequently incurs large seek overheads. For all other workloads, by using the right choice of the configurable parameters, BORG can be used at its maximum potential

We faced numerous challenges during its development of BORG some of which were described in 3.5. Our experience made us realize that we spent most of the time resolving bugs and handling other roadblocks in the Linux kernel space development which are likely to be faced by other developers who build similar self-managing extensions at the block layer. This lead to the inception of the Active Block Layer

Extensions (ABLE) project [?], an infrastructure that simplifies the development and management of extensions within the storage stack. ABLE also facilitates interoperation of multiple extensions in the system.

5.2 Future Research Directions

This work demonstrates the potential of using different layout policies based on the observed workload characteristics for both the static and dynamic access patterns. The functioning of the proposed systems can be further improved by incorporating the following optimizations.

5.2.1 Static Layout Optimization

A straightforward extension to using our approach would be to maintain two copies of the data, one using the sequential layout and the other using the native layout. Queries can be answered from one or the other depending on their class, thus ensuring that deep-focused queries are also optimally answered.

As an important future direction, new indexing techniques for tree-structured data can be explores that complement our native data layout strategies to further improve performance. The work presented here focuses on exploring the effects of the data placement and therefore we only consider no-index query execution plans to make the comparison clear. Further, existing file system implementations can be extended to support efficient access to such data . Finally, we plan to extend an existing file system to support efficient access to such data. Past efforts on providing additional application control of the storage system [?] indicate that a similar approach for tree storage is feasible.

Another interesting direction would be to study the advantages of our strategy with different caching techniques and caches of varying sizes. This work can also be extended to multi-query and multi-user environments, and work on handling updates within the tree-structured placement framework.

Finally, although, this work focuses on XML data, we believe that the tree-structured placement strategies presented here are general enough to address placement problems in other domains, such as directory-file tree-structures present in general-purpose file systems, multi-resolution video data, and Bioinformatics Suffix-Tree structures. The broader impact of this work surely needs further investigation. One direction is to analyze different workload patterns, such as database queries or suffix arrays in bio-informatics genomic strings, classify them based on their access characteristics and make the storage interface more expressive so that it can utilize information about the application access patterns and use appropriate techniques to efficiently use the underlying storage devices and improve along a specific dimension such as performance, reliability or security.

5.2.2 Dynamic Layout optimization

While our experiences with BORG have been largely positive, there are several directions in which the current version can be either improved or extended. We now discuss some of the significant directions that can serve as subjects of future investigation.

Analyzer and Planner optimization. The current versions of the analyzer (§ 3.4.2) and the planner (§ 3.4.3) components of BORG do not use the results of past executions and therefore incur higher overheads for every subsequent reconfiguration when using cumulative traces for training. Each of these components can be substantially optimized by making them more intelligent. The analyzer can build the master access graph incrementally rather than from scratch; likewise, the planner can incrementally create the new plan for BOPT reconfiguration during each iteration.

Alternate BOPT layout strategies. The current version of BORG uses a simple BOPT layout strategy starting from the most-connected vertex – the vertex with the highest sum of its edge-weights – in the master access graph, and then choosing the vertex most connected to it, and so on. Alternate layout strategies can be envisioned that potentially yield greater benefit. For instance, the placement can begin with the nodes connected to the highest weight edge, and then resorting to the same incremental addition of vertices. Alternatively, a distributed layout algorithm can be designed which uses many starting points for building the layout.

Configuring BORG parameters. BORG can be used to its maximum potential by correct configuration of its parameters, i.e. the reconfiguration interval, the size of the BOPT and the size of the write buffer. Fortunately, simple iterative algorithms can be quite effective in identifying the right parameter combination; a formal investigation of such an approach is an avenue for future work.

Timely reconfiguration. The current reconfiguration trigger in BORG is based on a fixed interval. However, opportune times for performing reconfiguration are during periods of no or low foreground I/O activity, especially for workloads that exhibit obvious idle or peak periods of activity. More sophisticated triggers can use alternate metrics to identify "unwanted" or "much needed" reconfiguration, such as the BOPT hit rate or the percentage of sequential accesses pre- and post- indirection to evaluate the effectiveness of the current BOPT layout. The above techniques can help substantially reduce the impact of reconfiguration to foreground I/O and increase the effectiveness of each reconfiguration operation.

Avoiding performance degradation. BORG can degrade performance for certain workloads, for instance, a read-intensive workload that has a very large or unstable working-set (§ 3.6.2). Future versions of BORG can be made intelligent to measure the impact of reconfiguration on such workloads by comparing the percentage sequentiality and the spatial locality for the accesses before (vanilla) and after (BORG) the indirection operation. If these metrics degrade post-BORG, BORG can be disabled. Such a mechanism will allow system performance to degrade gracefully in the event that the workload is not conducive to benefit from block reorganization.

BIBLIOGRAPHY

[AAB$^+$05] Serge Abiteboul, Rakesh Agrawal, Phil Bernstein, Mike Carey, Stefano Ceri, Bruce Croft, David DeWitt, Mike Franklin, Hector Garcia Molina, Dieter Gawlick, Jim Gray, Laura Haas, Alon Halevy, Joe Hellerstein, Yannis Ioannidis, Martin Kersten, Michael Pazzani, Mike Lesk, David Maier, Jeff Naughton, Hans Schek, Timos Sellis, Avi Silberschatz, Mike Stonebraker, Rick Snodgrass, Jeff Ullman, Gerhard Weikum, Jennifer Widom, and Stan Zdonik. The lowell database research self-assessment. *Commun. ACM*, 48(5):111–118, 2005.

[AGM$^+$90] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *J Mol Biol*, 215(3):403–410, October 1990.

[ALPB02] "A. Amer, D. Long, J. Paris, and R. Burns". File Access Prediction with Adjustable Accuracy. *International Performance Conference on Computers and Communication*, 2002.

[AM06] Loredana Afanasiev and Maarten Marx. An analysis of the current xquery benchmarks. In *ExpDB*, pages 9–20, 2006.

[AMM05] Loredana Afanasiev, Ioana Manolescu, and Philippe Michiels. Member: A micro-benchmark repository for xquery. In Stéphane Bressan, Stefano Ceri, Ela Hunt, Zachary G. Ives, Zohra Bellahsene, Michael Rys, and Rainer Unland, editors, *Database and XML Technologies, Third International XML Database Symposium, XSym 2005, Trondheim, Norway, August 28-29, 2005, Proceedings*, volume 3671 of *Lecture Notes in Computer Science*, pages 144–161. Springer, 2005.

[AS95] Sedat Akyurek and Kenneth Salem. Adaptive Block Rearrangement. *Computer Systems*, 13(2):89–121, 1995.

[Axb07] Jens Axboe. blktrace user guide, February 2007.

[BBM$^+$01] Denilson Barbosa, Attila Barta, Alberto O. Mendelzon, George A. Mihaila, Flavio Rizzolo, and Patricia Rodriguez-Guianolli. Tox - the toronto XML engine. In *Workshop on Information Integration on the Web*, pages 66–73, 2001.

[BCJ$^+$05] Kevin Beyer, Roberta J. Cochrane, Vanja Josifovski, Jim Kleewein, George Lapis, Guy Lohman, Bob Lyle, Fatma Ozcan, Hamid Pirahesh, Normen Seemann, Tuong Truong, Bert Van der Linden, Brian Vickery, and Chun Zhang. System rx: One part relational, one part xml. In *SIGMOD*, 2005.

[BDF$^+$03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proc. of the ACM SOSP*, October 2003.

[BDL$^+$] Stephane Bressan, Gillian Dobbie, Zoe Lacroix, Mong Li Lee, Ying Guang Li, Ullas Nambiar, and Bimlesh Wadhwa. XOO7: Applying OO7 benchmark to XML query processing tool. pages 167–174.

[BFHR06] Medha Bhadkamkar, Fernando Farfan, Vagelis Hristidis, and Raju Rangaswami. Efficient Native Storage for Semi-structured Data (extended paper version). In *http://www.cis.fiu.edu/SSS/NativeXMLextended.pdf*, 2006.

[BFRS02] Philip Bohannon, Juliana Freire, Prasan Roy, and Jérôme Siméon. From XML Schema to Relations: A Cost-based Approach to XML Storage. *ICDE*, 2002.

[BGC03] John Bucy, Gregory Ganger, and Contributors. The DiskSim Simulation Environment Version 3.0 Reference Manual. *Carnegie Mellon University Technical Report CMU-CS-03-102*, January 2003.

[BH06] Srikanta Bedathur and Jayant Haritsa. Search-optimized suffix-tree storage for biological applications. In David A. Bader, Manish Parashar, Sridhar Varadarajan, and Viktor K. Prasanna, editors, *12th IEEE International Conference on High Performance Computing (HiPC)*, volume 3769 of *Lecture Notes in Computer Science*, pages 29–39, Goa, India, October 2006. IEEE, Springer.

[BR01] Timo Böhme and Erhard Rahm. Xmach-1: A benchmark for xml data management. In *Datenbanksysteme in Büro, Technik und Wissenschaft (BTW), 9. GI-Fachtagung,*, pages 264–273, London, UK, 2001. Springer-Verlag.

[BR03] Timo Böhme and Erhard Rahm. 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 Integra-*

tion over the Web-Revised Papers, pages 148–158, London, UK, 2003. Springer-Verlag.

[CDA07] CDA. HL7 Clinical Document Architecture, Release 2.0. In *http://lists.hl7.org/read/attachment/61225/1/CDA-doc 20version.pdf. 2007*, 2007.

[CDF$^+$94] 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.

[CK99] G. Moerkotte C. Kanne. Efficient Storage of XML Data . *Universitaet Mannheim Technical Report*, 1999.

[Cus94] Helen Custer. Inside the Windows NT File System. *Microsoft Press*, August 1994.

[DAB$^+$06] Dolin, Alschuler, Boyer, Beebe, Behlen, Biron, and Shabo Shvo. HL7 Clinical Document Architecture Release 2. *J Am Med Inform Assoc.*, 13(1), Jan-Feb 2006.

[DAYF] Fang Du, Sihem Amer-Yahia, and Juliana Freire. ShreX: Managing XML Documents in Relational Databases.

[DFS99] Alin Deutsch, Mary F. Fernandez, and Dan Suciu. Storing Semistructured Data with STORED. *ACM SIGMOD*, 1999.

[DKF$^+$99] A.L. Delcher, S. Kasif, R.D. Fleischmann, J. Peterson, O. White, and S.L. Salzberg. Alignment of Whole Genomes. *Nucleic Acids Research*, 27(11):2369–2376, 1999.

[DR03] Zoran Dimitrijevic and Raju Rangaswami. Quality of Service Support for Real-time Storage Systems. *Proc. of International IPSI Conference*, October 2003.

[DRC$^+$04] Zoran Dimitrijevic, Raju Rangaswami, Edward Chang, David Watson, and Anurag Acharya. Diskbench: User-level Disk Feature Extraction Tool. *UCSB Technical Report TR-2004-18.*, April 2004.

[FHR07] Fernando Farfan, Vagelis Hristidis, and Raju Rangaswami. Beyond lazy xml parsing. In *Proceedings of International Conference on Database and Expert Systems Applications (DEXA)*, September 2007.

[FJS96] Adam Finkelstein, Charles E. Jacobs, and David H. Salesin. Multiresolution Video. *Proc. of SIGGRAPH*, pages 281–290, August 1996.

[Fra04] Massimo Franceschet. Xpathmark: An Xpath Benchmark For Xmark. *University of Amsterdam Technical Report PP-2004-04*, 2004.

[Fra05] Massimo Franceschet. *XPathMark: An XPath Benchmark for the XMark Generated Data*. 2005.

[GA94] James Griffoen and Randy Appleton. Reducing File System Latency using a Predictive Apporach. *Proc. of the Summer USENIX Conference*, pages 197–207, June 1994.

[gal] Galax. *http://www.galaxquery.org*.

[Gan01] Gregory R. Ganger. Blurring the Line Between OSes and Storage Devices. *Carnegie Mellon University Technical Report CMU-CS-01-166*, December 2001.

[GK97] Gregory R. Ganger and M. Frans Kaashoek. Embedded inodes and explicit grouping: Exploiting disk bandwidth for small files. *Proc. of the USENIX Technical Conference*, 1997.

[GKP02] G. Gottlob, C. Koch, and R. Pichler. Efficient Algorithms for Processing XPath Queries. In *VLDB*, 2002.

[GML08] GML. Geography markup language. *http://opengis.net/gml/*, 2008.

[GNA$^+$] Garth A. Gibson, Dave F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A Cost-Effective, High-Bandwidth Storage Architecture. *Proceedings of the ACM ASPLOS*, Oct 98.

[GS02] M.E. Gómez and V. Santonja. Characterizing Temporal Locality in I/O Workload. *Proc. of the International Symposium on Performance Evaluation of Computer and Telecommunication Systems*, 2002.

[GSK03] Gregory R. Ganger, John D. Strunk, and Andrew J. Klosterman. Self-* storage: Brick-based storage with automated administration. *Carnegie Mellon University Technical Report, CMU-CS-03-178*, August 2003.

[GUB$^+$08] Jorge Guerra, Luis Useche, Medha Bhadkamkar, Ricardo Koller, and Raju Rangaswami. The case for active block layer extensions. *SIGOPS Oper. Syst. Rev.*, 42(6):3–9, 2008.

[HD96] M. Holton and R. Das. XFS: A Next Generation Journalled 64-bit filesystem with Guaranteed Rate IO. *SGI Technical Report*, 1996.

[HHS05] Hai Huang, Wanda Hung, and Kang G. Shin. FS2: Dynamic Data Replication in Free Disk Space for Improving Disk Performance and Energy Consumption. *Proceedings of ACM SOSP*, pages 263–276, October 2005.

[HL708] HL7. Health level seven xml. *http://www.hl7.org/special/Committees/xml/xml.htm*, 2008.

[HSW$^+$04] Larry Huston, Rahul Sukthankar, Rajiv Wickremesinghe, M. Satyanarayanan, Gregory R. Ganger, Erik Riedel, and Anastassia Ailamaki. Diamond: A Storage Architecture for Early Discard in Interactive Search. *Proc. of the USENIX Conference on File and Storage Technologies*, March 2004.

[HSY05] Windsor W. Hsu, Alan Jay Smith, and Honesty C. Young. The automatic improvement of locality in storage systems. *ACM Transactions on Computer Systems*, 23(4):424–473, Nov 2005.

[ID01a] Sitaram Iyer and Peter Druschel. Anticipatory Scheduling: A Disk Scheduling Framework to Overcome Deceptive Idleness in Synchronous I/O. *Proc. of the ACM SOSP*, Sept 2001.

[ID01b] Sitaram Iyer and Peter Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous i/o. In *Symposium on Operating Systems Principles*, pages 117–130, 2001.

[Int98] Intel Corporation. Intel application launch accelerator. *http://support.intel.com/support/chipsets/iaa/*, 1998.

[JADAD06] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Antfarm: Tracking processes in a virtual machine environment. *Proc. of the USENIX Technical Conference*, May 2006.

[JAKC$^+$02] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. S. Lakshmanan, A. Nierman, S. Paparizos, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. TIMBER: A Native XML Database. *The VLDB Journal*, 11(4):274–291, 2002.

[Jim05] Jim Gray. Greetings! From a File System User. *Opening Keynote at the USENIX Conference on File and Storage Technologies*, December 2005.

[KBBA] Dave Kleikam, Dave Blaschke, Steve Best, and Barry Arndt. JFS for Linux. *http://jfs.sourceforge.net/.*

[KBM05] Carl-Christian Kanne, Matthias Brantner, and Guido Moerkotte. Cost-Sensitive Reordering of Navigational Primitives. *SIGMOD*, 2005.

[KBNK02] Raghav Kaushik, Philip Bohannon, Jeffrey F Naughton, and Henry F Korth. Covering Indexes for Branching Path Queries. *SIGMOD*, 2002.

[KC03] Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. *IEEE Computer*, 36(1):41–50, January 2003.

[KM77] Sukhamay Kundu and Jaydev Misra. A Linear Tree Partition Algorithm. In *SIAM J. Comput.*, pages 6(1):151–154, March 1977.

[KM06] Carl-Christian Kanne and Guido Moerkotte. A Linear Time Algorithm for Optimal Tree Sibling Partitioning and Approximation Algorithms in Natix. In *VLDB*, 2006.

[KP97] Geoffrey H. Kuenning and Gerald J. Popek. Automated hoarding for mobile computers. *Proc. of the ACM SOSP*, October 1997.

[KPH98] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. A Case for Intelligent Disks (IDISKS). *SIGMOD Record*, 27(3):42–52, September 1998.

[LCSZ04a] Z. Li, Z. Chen, S. Srinivasan, and Y. Zhou. C-Miner: Mining Block Correlations in Storage Systems. *Proc. of the USENIX FAST*, April 2004.

[LCSZ04b] Zhenmin Li, Zhifeng Chen, Sudarshan M. Srinivasan, and Yuanyuan Zhou. C-Miner: Mining Block Correlations in Storage Systems. *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 173–186, March 2004.

[LM01] Q. Li and B. Moon. Indexing and Querying XML Data for Regular Path Expressions. *VLDB Journal*, 2001.

[LS05] Chuanpeng Li and Kai Shen. Managing Prefetch Memory for Data-Intensive Online Servers. *Proc. of the USENIX FAST*, December 2005.

[LSG02] Christopher R. Lumb, Jiri Schindler, and Gregory R. Ganger. Freeblock Scheduling Outside of Disk Firmware. *Usenix Conference on File and Storage Technologies*, January 2002.

[LSGN00] Christopher R. Lumb, Jiri Schindler, Gregory R. Ganger, and David F. Nagle. Towards Higher Disk Head Utilization: Extracting Free Bandwith From Busy Disk Drives. *Proc. of the OSDI*, 2000.

[MAG$^+$97] Jason McHugh, Serge Abiteboul, Roy Goldman, Dallan Quass, and Jennifer Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3):54–66, 1997.

[MH04] Sergio L. S. Mergen and Carlos A. Heuser. Matching of XML Schemas and Relational Schemas. In *SBBD*, 2004.

[Mic06] Microsoft Corporation. Fast System Startup for PCs Running Windows XP. *Windows Platform Design Notes*, December 2006.

[MJLF84] M. McKusick, W. Joy, S. Leffler, and R. Fabry. A Fast File System for UNIX*. *ACM Transactions on Computer Systems 2*, 3:181–197, August 1984.

[MLLA03] Xiaofeng Meng, Daofeng Luo, Mong-Li Lee, and Jing An. Orientstore: A schema based native xml storage system. In *VLDB*, pages 1057–1060, 2003.

[MML08] MML. Medical Markup Language. *http://www.ncbi.nlm.nih.gov/*, 2008.

[MMM06] Ioana Manolescu, Cédric Miachon, and Philippe Michiels. Towards micro-benchmarking xquery. In *ExpDB*, pages 28–39, 2006.

[MRC+97] Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello, Randolph Y. Wang, and Thomas E. Anderson. Improving the Performance of Log-Structured File Systems with Adaptive Methods. *Proc. of the ACM SOSP*, 1997.

[NADAD03] James Nugent, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Controlling your PLACE in the File System with Gray-box Techniques. *Proc. of the USENIX Technical Conference*, June 2003.

[Nam] Namesys, Inc. The ReiserFS File System. *http://www.namesys.com/*.

[NJ03] Matthias Nicola and Jasmi John. Xml parsing: A threat to database performance. In *Proceedings of the ACM International Conference on Information and Knowledge Management (CIKM)*, pages 175–178, 2003.

[NKS07] Matthias Nicola, Irina Kogan, and Berni Schiefer. An xml transaction processing benchmark. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 937–948, New York, NY, USA, 2007. ACM.

[NLB+01] U. Nambiar, Z. Lacroix, S. Bressan, M. Lee, and Y. Li. Xml benchmarks put to the test, 2001.

[NNP00] Igor Nekrestyanov, Boris Novikov, and Ekaterina Pavlova. An analysis of alternative methods for storing semistructured data in relations. In *ADBIS-DASFAA*, pages 354–361, 2000.

[NSL02] Markus L. Noga, Steffen Schott, and Welf Lowe. Lazy xml processing. In *Proceedings of the ACM Symposium on Document Engineering*, pages 88–94, 2002.

[ODS08] ODS. Open Document Specification v1.0. *http://www.oasis-open.org/committees/download.php/12572/OpenDocument-v1.0-os.pdf*, 2008.

[OOX08] OOX. Openoffice xml file format v1.0, 2008.

[PADAD03] Florentina I. Popovici, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Robust, Portable I/O Scheduling with the Disk Mimic. *Proc. of the USENIX Technical Conference*, June 2003.

[PGG$^+$95] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed Prefetching and Caching. In *Proc. of the 15th ACM SOSP*, December 1995.

[PGMW95] Yannis Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. Object Exchange Across Heterogeneous Information Sources. In *ICDE '95: Proc. of the Eleventh International Conference on Data Engineering*, 1995.

[PS05] Athanasios E. Papathanasiou and Michael L. Scott. Aggressive Prefetching: An Idea Whose Time Has Come. *Proc. of the Workshop on HotOS*, June 2005.

[RFGN00] Erik Riedel, Christos Faloutsos, Gregory R. Ganger, and David F. Nagle. Data mining on an OLTP system (nearly) for free. *Proc. of the ACM SIGMOD*, May 2000.

[RFHR] M. Ramanath, J. Freire, J. Haritsa, and P. Roy. Searching for efficient XML to relational mappings.

[RGF98] Erik Riedel, Garth Gibson, and Christos Faloutsos. Active Storage For Large-Scale Data Mining and Multimedia. *Proc. of the VLDB*, August 1998.

[RO91] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. *Proc. of the ACM SOSP*, October 1991.

[Rok01] Daniel Rokhsar. Computational Analysis of Genomic Sequence Data. *http://www.nersc.gov/news/annua_reports/annrep01/sh_BER_06.html*, 2001.

[RP03] Lars Reuther and Martin Pohlack. Rotational-position-aware real-time disk scheduling using a dynamic active subset (DAS). *Proc. of the IEEE RTSS*, December 2003.

[RPJ$^+$03] K. Runapongsa, J. Patel, H. Jagadish, Y. Chen, and S. Al-Khalifa. The michigan benchmark: Towards xml query performance diagnostics, 2003.

[RW91] C. Ruemmler and J. Wilkes. Disk Shuffling. *Technical Report HPL-CSP-91-30, Hewlett-Packard Laboratories*, October 1991.

[RW93] Chris Ruemmler and John Wilkes. UNIX disk access patterns. *Proc. of the Winter USENIX Conference*, 1993.

[RW94] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *Computer*, 27(3):17–28, 1994.

[SCO90] M Seltzer, P Chen, and J Ousterhout. Disk Scheduling Revisited. *Proc. of the Winter USENIX Technical Conference*, 1990.

[SGLG02] Jiri Schindler, John Linwood Griffin, Christopher R. Lumb, and Gregory R. Ganger. Track-aligned Extents: Matching Access Patterns to Disk Drive Characteristics. *Proc. of USENIX FAST*, 2002.

[SGM91] Carl Staelin and Hector Garcia-Molina. Smart Filesystems. In *USENIX Winter Conference*, 1991.

[SO91] Jon A. Solworth and Cyril U. Orji. Distorted Mirrors. *Proc. of PDIS*, 1991.

[SPP$^+$03] Muthian Sivathanu, Vijayan Prabhakaran, Florentina I. Popovici, Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Semantically-Smart Disk Systems. *Proc. of the USENIX FAST*, March 2003.

[SS97] Margo Seltzer and Christopher Small. Self-Monitoring and Self-Adapting Operating Systems. *Proc. of the Workshop on HotOS*, May 1997.

[SSP$^+$05] Steven W. Schlosser, Jiri Schindler, Stratos Papadomanolakis, Minglong Shao, Anastassia Ailamaki, Christos Faloutsos, and Gregory R. Ganger. On Multidimensional Data and Modern Disks. *Proceedings of the 4th USENIX Conference on File and Storage Technology*, December 2005.

[SSS$^+$04] Jiri Schindler, Steven W. Schlosser, Minglong Shao, Anastassia Ailamaki, and Gregory R. Ganger. Atropos: A Disk Array Volume Manager for Orchestrated Use of Disks. *Proc. of the USENIX Conference on File and Storage Technologies*, March 2004.

[STSG03] B. Salmon, E. Thereska, C. Soules, and G. Ganger. A Two-tiered Software Architecture for Automated Tuning of Disk Layouts. *Workshop on Algorithms and Architectures for Self-Managing Systems*, 2003.

[STZ$^+$] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *VLDB 1999*.

[SVG08] SVG. Scalable vector graphics. *http://www.w3.org/Graphics/SVG/*, 2008.

[SWK$^+$02a] A. Schmidt, F. Waas, M. Kersten, M. Carey, I. Manolescu, and R. Busse. Xmark: A benchmark for xml data management, 2002.

[SWK$^+$02b] Albrecht Schmidt, Florian Waas, Martin L. Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. XMark: A Benchmark for XML Data Management. *VLDB*, 2002.

[TADP99] Nisha Talagala, Remzi H. Arpaci-Dusseau, and David Patterson. Microbenchmark-based Extraction of Local and Global Disk Characteristics. *UC Berkeley Technical Report*, 1999.

[Twe98] S. C. Tweedie. Journaling the Linux ext2fs File System. *The Fourth Annual Linux Expo*, May 1998.

[UGB$^+$08] Luis Useche, Jorge Guerra, Medha Bhadkamkar, Mauricio Alarcon, and Raju Rangaswami. EXCES: External caching in energy saving storage systems. *IEEE HPCA*, 2008.

[VC90] Paul Vongsathorn and Scott D. Carson. A System for Adaptive Disk Rearrangement. *Softw. Pract. Exper.*, 20(3):225–242, 1990.

[WGPW95] B. Worthington, G. Ganger, Y. Patt, and J. Wilkes. Online Extraction of SCSI Disk Drive Parameters. *Proc. of ACM Sigmetrics Conference*, pages 146–156, 1995.

[WGSS95] John Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID Hierarchical Storage System. *Proc. of the ACM SOSP*, 1995.

[Won80] C. K. Wong. Minimizing Expected Head Movement in One-Dimensional and Two-Dimensional Mass Storage Systems. *ACM Computing Surveys*, 12(2):167–178, 1980.

[xal] Xalan-Java. *http://xml.apache.org/xalan-j.*

[XPa05] XML Path Language (XPath) Version 1.0. *http://www.w3.org/TR/xpath*, 2005.

[xt] XT. *http://www.blnz.com/xt/index.html.*

[YGC$^+$00] X. Yu, B. Gum, Y. Chen, R. Y. Wang, K. Li, A. Krishnamurthy, and T. E. Anderson. Trading Capacity for Performance in a Disk Array. *Proceedings of Operating Systems Design and Implementation*, October 2000.

[YLB01a] Tsozen Yeh, Darrell Long, and Scott Brandt. Caching Files with a Program-based Last N Successors. *Workshop on Caching, Coherency and Consistency (WC3 '01)*, June 17, 2001.

[YLB01b] Tsozen Yeh, Darrell Long, and Scott Brandt. Conserving Battery Energy through Making Fewer Incorrect File Predictions. *IEEE Workshop on Power Management for Real-Time and Embedded Systems at the IEEE Real-Time Technology and Applications Symposium*, pages 30–36, May 29, 2001.

[YOK03] Benjamin B. Yao, M. Tamer Özsu, and John Keenleyside. Xbench - a family of benchmarks for xml dbmss. 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*, pages 162–164, London, UK, 2003. Springer-Verlag.

[ZYKW02] C. Zhang, X. Yu, A. Krishnamurthy, and R. Y. Wang. Configuring and Scheduling an Eager-Writing Disk Array for a Transaction Processing Workload. *Proc. of USENIX FAST*, January 2002.

APPENDIX A

Disk-drive Background

Data is stored in blocks on disk-drives where each block is identified by its cylinder-head-sector number (CHS addressing) on the drive by the drive controller. However, modern disk drives provide a 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. In this Logical Block Addressing (LBA) every sector is identified as a fixed-size block in a linear address space. The device driver receives the block I/Os as LBA definitions which is passes to the drive controller which in turn translates this to the device specific CHS address.

Latencies: 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 block 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 a non-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 block from the block 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 target disk zone.

Accesses: Accesses to disk blocks can be of two types. *Sequential*, where contiguous disk blocks are accessed and *Random* where non-contiguous disk blocks are accessed. Sequential accesses are the most efficient since they do not incur any seek or rotational delays. Random accesses are known to be two orders of magnitude slower than sequential accesses [**?**].

VITA

MEDHA BHADKAMKAR

EDUCATION

Ph.D	Florida International University School of Computer and Information Sciences 2009
M.S	Florida International University School of Computer and Information Sciences 2006
B.E	University of Pune, India Electronics Engineering 2000

RESEARCH EXPERIENCE

TEACHING EXPERIENCE

PUBLICATIONS AND PRESENTATIONS

- Storing Semi-structured data on disk drives Medha Bhadkamkar, Fernando Farfan, Vagelis Hristidis, and Raju Rangaswami ACM Transactions on Storage, Vol. 5, Issue 2, May, 2009.
- BORG: Block-reORGanization for Self-Optimizing Storage Systems Medha Bhadkamkar, Jorge Guerra, Luis Useche, Sam Burnett, Jason Liptak, Raju Rangaswami, and Vagelis Hristidis Proceedings of the File and Storage Technologies, FAST, Februrary 2009.
- The case for Active Block Layer Extensions Jorge Guerra, Luis Useche, Medha Bhadkamkar, Ricardo Koller, and Raju Rangaswami Workshop on Storage and I/O Virtualization, Performance, Energy, Evaluation and Dependability, SPEED, February 2008.
- EXCES: EXternal Caching in Energy Saving Storage Systems Luis Useche, Jorge Guerra, Medha Bhadkamkar, Mauricio Alarcon, and Raju Rangaswami Proceedings of IEEE International Symposium on High-Performance Computer Architecture, HPCA, February, 2008.
- Storing Trees on Disk Drives Medha Bhadkamkar, Fernando Farfan, Vagelis Hristidis, and Raju Rangaswami Proceedings of the File and Storage Technologies WiP, December 2005.

- Feasibility, Efficiency, and Effectiveness of Self-Optimizing Storage Systems Medha Bhadkamkar, Sam Burnett, Jason Liptak, Raju Rangaswami, and Vagelis Hristidis Florida International University Technical Report TR-2007-01-01, January 2007.
- A Case for Self-Optimizing File Systems Medha Bhadkamkar, Sam Burnett, Jason Liptak, Raju Rangaswami, and Vagelis Hristidis Florida International University Technical Report TR-2006-09-03, September 2006.
- Efficient Native Storage Systems for Semi-structured Data Medha Bhadkamkar, Fernando Farfan, Vagelis Hristidis, and Raju Rangaswami Florida International University Technical Report TR-2006-09-01, September 2006.
- Efficient Native XML Storage Medha Bhadkamkar, Vagelis Hristidis, and Raju Rangaswami Florida International University Technical Report TR-2005-04-01, April 2005.