Efficient data organization and management on heterogeneous storage hierarchies

Minglong Shao

CMU-CS-07-170

May 2008

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Anastasia Ailamaki, Chair
Greg Ganger
Todd Mowry
Per-Åke (Paul) Larson (Microsoft Research)

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored by the National Science Foundation under grant numbers CCR-0205544 and IIS-0429334. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: data placement, data organization, data management, benchmark, optimization, buffer pool, database management system, storage, performance, multidimensional

To my family.

Abstract

Due to preferences for design and implementation simplicity, current data organization and management in database systems are based on simple assumptions about storage devices and workload characteristics. This has been the major design principle since the inception of database systems. While the device- and workload-oblivious approach worked well in the past, it falls short when considering today's demands for fast data processing on large-scale datasets that have various characteristics. The ignorance of rich and diverse features in both devices and workloads has posed unnecessary performance trade-offs in existing database systems.

This dissertation proposes efficient, flexible, and robust data organization and management for database systems by enhancing the interaction with workloads and hardware devices. It achieves the goal through three steps.

First, a microbenchmark suite is needed for quick and accurate evaluation. The proposed solution is DBmbench, a significantly reduced database microbenchmark suite which simulates OLTP and DSS workloads. DBmbench enables quick evaluation and provides performance forecasting for real large-scale benchmarks.

Second, *Clotho* investigates how to build a workload-concious buffer pool manager by utilizing query payload information. *Clotho* decouples the in-memory page layout from the storage organization by using a new query-specific layout called *CSM*. Due to its adaptive structure, *CSM* eliminates the long-standing performance trade-offs of *NSM* and *DSM*, thus achieving good performance for both DSS and OLTP applications, two predominant database workloads with conflict characteristics. *Clotho* demonstrates that simple workload information, such as query payloads, is of great value to improve performance without increasing complexity.

The third step looks at how to use hardware information to eliminate performance trade-offs in existing device-oblivious designs. *MultiMap* is first proposed as a new mapping algorithm to store multidimensional data onto disks without losing spatial locality. *MultiMap* exploits the new adjacency model of disks to build a multidimensional structure on top of the linear disk space. It outperforms existing mapping algorithms on various spatial queries. Later, *MultiMap* is expanded to organize intermediate results for hash join and external sorting where the I/O performance of different execution phases exhibits similar trade-offs as those in 2-D data accesses. Our prototype demonstrates an up to 2 times improvement over the existing implementation in memory limited executions. The above two projects complete *Clotho* by showing the benefits of exploiting detailed hardware features.

Acknowledgments

First, I would like to thank my adviser, Professor Anastasia Ailamaki, a passionate researcher and an inspiring mentor. Without her guidance, this dissertation would not have been possible. I cannot thank her enough for her encouragement and patience from the first day I started working with her. She first introduced me to the fascinating world of database systems and has been helping me with every aspect of my doctoral studies ever since. From her high-level insightful advice on the principles of system research, and advice about long-term goal setting as a professional woman, to her meticulous comments about writing slides and giving presentations, her teachings are of great value to my career and my life.

During my doctoral studies, I also have had the honor to work with three great professors: Greg Ganger, Babak Falsafi, and Christos Faloutsos. I am deeply grateful to them for their advice and encouragement. I would like to thank Professor Todd Mowry, and Dr. Per-Åke Larson for kindly agreeing to be on my thesis committee and for taking the time to read my dissertation and to give me comments.

I would like to thank my colleagues in the Fates project, Dr. Jiri Schindler and Dr. Steven Schlosser whom I also regard as my "minor mentors". They are the best colleagues one could ever have: knowledgeable, professional, cheerful, and considerate. I deeply appreciate their help throughout the entire project.

I would like to thank my colleagues in the database family: Shimin Chen, Stratos Papadomanolakis, Mengzhi Wang, Stavros Harizopoulos, and Vlad Shkapenyuk. It has been a great pleasure to work with them.

If I had not met so many wonderful friends, my studies at Carnegie Mellon University would have been only long and hard. Thanks to Ting Liu, Ke Yang, Ningning Hu, Qin Jin, Chang Liu, Rong Yan, Juchang Hua, Yanhua Hu, Yanjun Qi, Xiaofang Wang, Ippokratis Pandis, Kun Gao, Steven Okamoto, Jimeng Sun, Huiming Qu, Vahe Poladian, Monica Rogati, Fan Guo, Yanxi Shi, Shuheng Zhou, Tim Pan, Ying Zheng, and Zihan Ma for their friendship which I cherish heartily.

I feel lucky to be able to join the Computer Science Department at Carnegie Mellon University and to be a member of the database group and the Parallel Data Lab. I could not think of a better place to complete my Ph.D. work.

Last but not least, I would like to thank my parents, my brother, and Changhao Jiang for their love and support. This dissertation is dedicated to them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Data organization and management in database management systems study how to store data on devices efficiently regarding space utilization and/or the time to index, fetch, and process the information. The research area on this subject covers a wide variety of topics. For instance, a few topics that are closely related to this dissertation include low-level data structure designs, such as page layouts for indices and relational tables; algorithmic designs, such as strategies for space allocation and data sharing; architectural designs, such as the dividing of functionalities among different modules and the defining of interfaces for inter-modular communication.

Data organization and management play central roles in database management systems and have a direct impact on system functionalities and performance. Therefore, they have been the subject of numerous studies [4, 11, 15, 31, 40] since the emergence of relational databases. Research continues as the memory hierarchy remains the bottleneck [3] in database systems, and no single solution serves all needs in database systems where data structures, storage device characteristics, workload access patterns, and performance requirements vary significantly from time to time and from application to application. In this highly diverse environment, data organization and management that can adapt to the differences will be more resilient to changes, and will thus be able to maintain consistently good performance. To achieve this goal, desirable data organization strategies will need to promote close interactions between software and hardware. In other words, these strategies will need to show a deeper understanding of data structures and their access patterns, as well as an understanding of the characteristics of underlying storage devices and their technical trends.

This chapter is intended to promote the idea that a deeper understanding of storage devices is needed in data organization and management for database systems. This chapter also outlines the structure of the dissertation, highlights the contributions, and briefly summarizes the content of each of the following chapters. The thesis governing this document is the following: "*Database Management Systems can become more robust by eliminating performance trade-offs related to inflexible data layout in the memory and disk hierarchy. The key to improving performance is to adapt the data organization to workload characteristics, which is achieved (a) by intelligently using query payload information when managing the buffer pool, (b) by decoupling the in-memory layout from the storage organization, and (c) by exposing the semi-sequential access path available on modern disks to the storage manager.*"

As the thesis statement indicates, this dissertation centers on designing and implementing adaptable data management and organization for database systems by carefully exposing the information of query requests and detailed hardware characteristics. Specifically, it addresses the problem from the following aspects: (a) a simple yet representative database microbenchmark suite for quick and accurate evaluation; (b) adaptable page layouts for relational tables, and (c) algorithms for mapping logical geometric addresses to physical disk addresses for multidimensional data access. In the rest of this chapter, each section will provide the following: (a) a brief introduction to the background of the topic and how it relates to the thesis statement, (b) an explanation of the problems in existing approaches that motivated the current work, and (c) a summary of the high-level ideas and solutions contributed by this thesis.

1.1 Difficulties in database benchmarking at microarchitectural level

With the proliferation of database workloads on servers, database benchmarks have been widely used to evaluate and compare performance on server architecture. Existing prevalent database benchmarks, such as the benchmarks [76] proposed by Transaction Processing Performance Council (TPC), are designed to mimic different behavior of real-world workloads running on database systems, thus they focus on overall system functionality and performance. As a result, these benchmarks consist of complex query execution on very large datasets which can easily reach the scale of terabyte. The complexity in query execution and the size of datasets make existing database benchmarks inapplicable in microarchitecture and memory system research due to the following reasons.

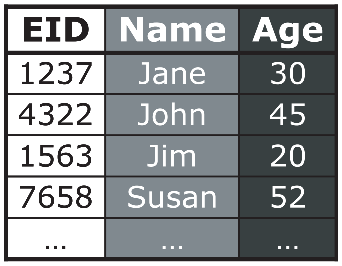
First, microarchitecture simulation tools used by computer architects are typically five or more orders of magnitude slower than real machines. Thus, it might take months or even longer to conduct an experiment with a full database benchmark. It is not acceptable at the early design stage where several design options need to be evaluated in a trial-and-error setting. Second, the execution of a full database benchmark is complicated. For instance, DSS workloads typically have complex query plans consisting of tens or hundreds operators. OLTP workloads involve concurrent transactions running queries on top of shared resources. The execution complexity makes it hard to do experiments in a controllable way and to pinpoint the bottlenecks. Third, the installation of conventional database benchmarks is time-consuming and error-prone. It takes an expert several days or weeks to set up an experiment environment where performance-sensitive parameters are assigned with the proper values.

The solution to the above problems is, naturally, scaled-down database benchmarks which has been adopted in previous research projects in various contexts [3, 8, 9, 12, 39, 54, 75]. These studies all employ ad hoc abbreviations of benchmarks with a hope that the reduced benchmarks possess the same characteristics as their full-size counterparts. Unfortunately, scaling down database benchmarks is tricky: changes in dataset sizes, query execution orders, and/or parameter values may cause severe and unpredictable deviation in characteristics.

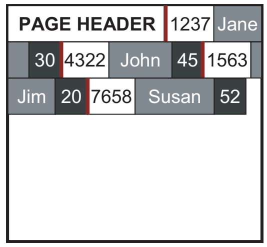
DBmbench, the first project of the thesis work, is a small and representative database microbenchmark suite designed with the goal to mimic the performance behavior of TPC-H and TPC-C. By identifying and executing the primary operations in the two benchmarks, DBmbench systematically scales down the dataset sizes and query complexity significantly and accurately captures the processor and memory performance behavior. DBmbench is also used in other projects of my thesis work to conduct sensitivity analysis on query selectivities and payloads. The details of DBmbench are discussed in Chapter 3 .

1.2 Problems of static page layouts for relational tables

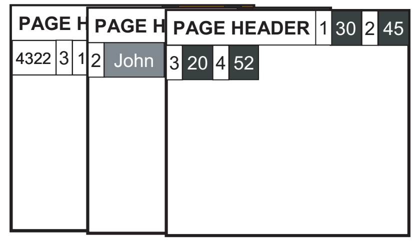
Conventional relational database systems provide a conceptual representation of data in relations or tables [17]. Each table is stored in fixed-sized pages of a typical size from 4 to 64 KB, which is also the minimal transferring unit between storage devices and main memory. Page layout, also known as *storage model*, describes the records that are contained within a page and how they are laid out. It usually has a header which stores some metadata, such as the number of records in the page and the size of the free space. Records are then stored according to the corresponding layout. Since the page layout determines what records and which attributes of a relation are stored in a single page, the storage model employed by a database system has far-reaching implications for the performance of a particular workload [3]. The most widely used page layouts in commercial database systems are the N-ary Storage Model (*NSM*) [52], which stores full records sequentially within a page, and the Decomposition Storage Model (*DSM*) [18], which partitions the table vertically into single-attribute subtables and stores them separately.



(a) **A table with 3 attributes.**



(b) **In *NSM*.**



(c) **In *DSM*.**

Figure 1.1: Diagrams of *NSM* and *DSM*. This graph shows how a simple table with three attributes is stored in the *NSM* and *DSM* layouts respectively.

Figure 1.1 shows a simple example of *NSM* and *DSM* with a three-attribute table.

Access patterns on database tables, by and large, fall into two categories. One is the full-record access frequent in Online Transaction Processing (OLTP) applications, where all (or almost all) attributes of records are usually requested. The full-record access is analogous to accessing a table along the row direction. The other is the partial-record access, represented by Decision Support System (DSS) applications, where very few attributes of records are processed. This can be viewed as accessing a table along the column direction. These two prevalent access patterns have exactly opposite characteristics. Existing page layouts optimized for them employ different design strategies, as the best page layout for one pattern usually performs poorly for the other.

Existing approaches face a performance trade-off by first predicting the application's dominant access pattern, and then by choosing a page layout optimized for that pattern at the expense of the other. Among the existing page layouts in the literature, *NSM* is preferable for OLTP workloads, whereas *DSM* and its variations [73, 86] are better choices for DSS workloads. Figure 1.2 illustrates the performance trade-offs of *NSM* and *DSM*. In this example, a simple table scan query is posed on a table with 15 attributes. We vary the number of attributes referenced by the query and measure the total query execution time. *DSM* outperforms *NSM* when the number of attributes referenced is small, whereas *NSM* is clearly the winner when more attributes are requested. Neither option can offer good performance across the spectrum.

Therefore, systems using a predetermined page layout based on prediction are not able to adapt to workload changes. If the actual access pattern deviates from the prediction, the performance plummets. In addition, workloads with mixed characteristics are becoming more common in the real world where two access patterns could appear in the same system and are equally important. In this case, solutions with one static page layout, *NSM* or *DSM*, fail. Several solutions have been proposed to address the performance trade-offs. Fractured Mirrors [53] tries

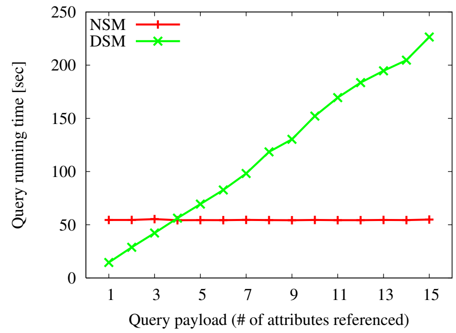


Figure 1.2: Performance trade-offs of *NSM* and *DSM*. In this example, we store a 15-attribute table in *NSM* and *DSM* respectively and measure the execution time of a simple table-scan query on this table as a function of the number of attributes referenced by the query.

to combine the advantages of *NSM* and *DSM* by storing tables in two layouts to accommodate the workload changes. Unfortunately, keeping two copies of data doubles the required storage space and complicate data management because two copies have to be maintained in synchrony to preserve data integrity. Focusing on the CPU cache performance, the *PAX* layout [4] provides another approach for unifying the two layouts by organizing records in a *DSM* format *within* a page. While *PAX* performs very well for both access patterns at the CPU cache level, it faces the same problem as *NSM* at the storage level. The Data Morphing technique [31] furthers the idea of *PAX* by reorganizing records within individual pages based on the workloads that change over time. It increases the flexibility of memory pages, but these fine-grained changes cannot address the trade-offs involved in accessing non-volatile storage. These solutions bring some degree of flexibility for mitigating the problems caused by static page layouts, but they are far from adequate in that they either pose extra complexity in data management or they only focus on one level of the memory hierarchy.

Clotho, as one part of the *Fates* database storage manager, solves the performance trade-off problem inherent to a particular page layout by decoupling the in-memory page layout from the storage organization. *Clotho* proposes a flexible page layout called *Clotho* Storage Model (*CSM*) that can change its structure and content in order to meet the different needs of various queries and to match the unique features of different storage devices. The benefit of the decoupling is twofold. First, pages in the buffer pool can be tailored to the specific needs of each query. The query-specific pages save memory space and I/O bandwidth. Second, data at each memory hierarchy level can be organized in a way that can fully exploit the device characteristics at that level. *Clotho*, together with other components in the Fates database storage project [59, 60, 66], also investigates the architectural issues of building a robust and adaptable storage manager for database systems.

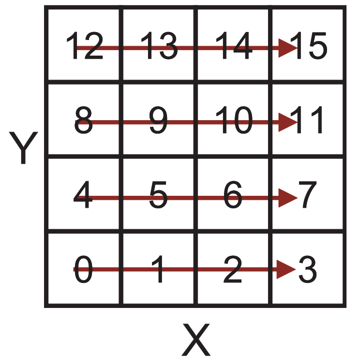
This thesis project implements *Clotho* on top of *Atropos* [60]. The experimental results show that *Clotho* with *CSM* page layout is able to eliminate the trade-offs that exist in *NSM* and *DSM*. It combines the best performance of the two page layouts at all levels of the memory hierarchy. A detailed description is presented in Chapter 4.

1.3 Trade-offs of storing multidimensional data on disks

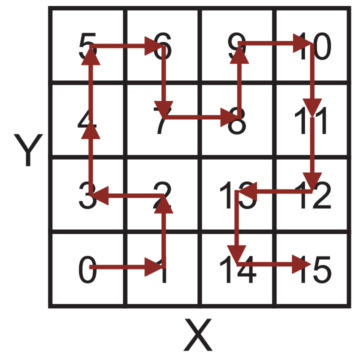
Multidimensional datasets are widely used to represent geographic information, present multimedia datasets, and to model multidimensional objects in both scientific computing (such as the 3-D model of earth in earthquake simulation) and business applications (such as the data cubes in OLAP applications). Their growing popularity has brought the problem of efficient storage and retrieval of multidimensional data to the fore. The major challenge is to find a mapping algorithm that maps logical geometric addresses, such as the coordinates of data, to locations on disks, usually identified by logical block numbers (*LBN*). The goal of mapping algorithms is to preserve spatial locality as much as possible on disks so that nearby objects in the original geometric space are stored in close-by disk blocks. The property of preserving spatial locality is often called *clustering* in the literature. Since neighboring data are usually accessed together, algorithms with better clustering will have better I/O performance because fetching nearby blocks is more efficient than fetching remote blocks.

The existing mapping algorithms have not been very successful at preserving spatial locality. The fundamental obstacle is the conflict between multidimensional data and the traditional *linear abstraction* of storage devices offered by standard interfaces such as SCSI. Under the abstraction, storage devices are simplified as a sequence of blocks identified by LBNs. Therefore, to organize multidimensional data on linear disks, all existing mapping algorithms have to put an order on the dataset. Generally speaking, there are two ways to do that: (a) serialize all data based on a pre-selected dimension (often called major order), or (b) use space-filling curves [42].

A simple implementation that serializes data along a pre-selected dimension, called *Naive*, traverses a dataset as following. For example, in a two-dimensional (2-D) space (X, Y), data are ordered first along the X axis, then the Y axis, as Figure 1.3(a) shows. This mapping scheme perfectly preserves the spatial locality on the X axis because successive points on the X axis are stored physically onto contiguous disk blocks. However, it completely ignores the spatial locality of the Y axis. The result is the optimal sequential access along the X axis and the worst random-like access for Y. Sarawagi et al. [57] optimize the naive mapping by first dividing the original space into multidimensional tiles, called *chunks*, according to the predicted access patterns. Together with other techniques, such as storing redundant copies sorted along different



(a) ***Naive* mapping.**



(b) ***Hilbert* curve mapping.**

Figure 1.3: Two mapping algorithms based on linearization. The graph illustrates how *Naive* and *Hilbert* mapping algorithms map the cells in a 4x4 2-D grid to a linear space. The numbers in the cells indicate the order of the cells, i.e., the cells will be stored on the disk sequentially according to this order.

dimensions, this chunk-based solution is up to an order of magnitude faster over the unoptimized method. However, their work adopts the linear abstraction, so it still faces the same problem as the *Naive* approach of losing spatial locality on non-major orders.

Alternatively, one can traverse a multidimensional space by following a space-filling curve. Space-filling curves are continuous, non-intersecting curves that pass ("*cover*") all points in a space. The fractal dimensions [64] of space-filling curves are usually greater than 1, which makes them "dense" and "closer" to a multidimensional object. Therefore, space-filling curves are preferable to the *Naive* linear mapping. Some well-known space-filling curves are Gray-coded Curve [23], Hilbert Curve [32], and Z-ordering Curve [47]. Figure 1.3(b) shows the mapping based on Hilbert Curve on a 2-D grid. Moon et al. [42] analyzed the clustering properties of Hilbert Curve and indicate that it has better clustering than *Z-ordering* Curve, and hence better I/O performance in terms of average per-node access time. Unlike the approaches which linearize along a single major order, space-filling curves do not favor any dimension, but rather balance their performance along all dimensions, which results in better performance for range queries. Unfortunately, space-filling curve-based solutions still face the problem of losing spatial locality. The performance improvement also comes at the high cost of losing the sequential bandwidth on all dimensions. Finally, space-filling curve-based mapping algorithms are not practical for dimensions higher than three. In fact, some space-filling curves can not even be generalized to higher dimensions [42].

In the rest of the dissertation, *Hilbert* denotes the linear mapping algorithm that uses Hilbert Curve. Figure 1.4 illustrates the trade-offs of the *Naive* and *Hilbert* mapping algorithms on a 3-D dataset. In this example, the dataset has three dimensions, X, Y, and Z, where X is selected as

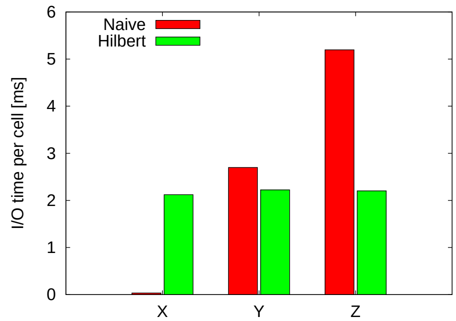


Figure 1.4: Performance trade-offs of *Naive* and *Hilbert* mapping. This graph shows the average I/O response time to fetch contiguous points along each dimension, X, Y, and Z, using different mapping algorithms.

the major order in the *Naive* mapping. The experiment accesses contiguous points along each dimension and measures the average I/O response time. The trade-offs are easy to see. The *Naive* mapping optimizes the performance on the major order X while severely compromising the performance on the other two dimensions, Y and Z. The *Hilbert* mapping balances the performance on all dimensions at the cost of losing the sequential bandwidth: the response time on all dimensions in the *Hilbert* mapping is almost two orders of magnitude longer than the response time on the X dimension in the *Naive* mapping.

Naturally, a solution that eliminates the trade-offs would be preferred. However, under the assumption of the linear disk abstraction, the loss of spatial locality of multidimensional datasets, as well as the performance trade-offs, are inevitable. Any new algorithms that seek to solve this problem need to resolve the discord of dimensionality. *MultiMap*, an algorithm used to map data to disks, solves the problem by building a multidimensional view for disks based on a novel adjacency model. *MultiMap* allows applications to store data without any serialization. Therefore, it is able to preserve spatial locality of multidimensional data and eliminate the performance trade-offs. *MultiMap* is an example of how detailed knowledge about storage devices can aid in solving some long-standing problems in database systems. Details about *MultiMap* are discussed in Chapter 5.

1.4 Opposite access patterns during query execution

Clotho and *MultiMap* are able to address the performance trade-offs caused by "spatial" conflict I/O access patterns, the opposite access patterns that originate from the data structures (e.g. relational tables and multidimensional data structures). There also exist "temporal" conflict I/O access patterns in database systems: the opposite I/O access patterns in different phases during query execution. The idea is to organize the intermediate results in a smart way so that access to the intermediate results in all execution phases are efficient. This work focuses on two major database operators, (a) hash join and (b) external sorting.

Hash join, as an efficient algorithm to implement equijoins, is commonly used in commercial database systems. In its simplest form, the algorithm first builds a hash table on the smaller (*build*) relation, and then probes the hash table using tuples from the larger (*probe*) relation to find matches. In the real world, database systems adopt variations of a more practical algorithm, called the *GRACE* hash join algorithm [36], to avoid excessive disk accesses due to a lack of memory resources. *GRACE* begins by partitioning the two joining relations into smaller sub-relations (also called "partitions") using a certain hash function on the join attributes such that each sub-relation of the build relation and its hash table can fit into the main memory. It then joins each pair of build and probe sub-relations separately as in the simplest algorithm. The two phases of the *GRACE* algorithm are called the "partition phase" and the "probe phase" respectively.

In the partition phase, the algorithm writes out tuples into different partitions based on the hash value of the join attributes. In practice, accesses to different partitions are interleaved, which results in a random access pattern. In contrast, during the probe phase, the algorithm reads and processes each partition one after another. The optimization of I/O performance in these two phases conflicts with each other; sequential access in one phase will inevitably cause random access in the other phase. A popular practice is to optimize the probe phase. In this manner, partitions are stored sequentially, so that fetching one partition incurs efficient sequential access, whereas the interleaved writing to all partitions is done in a random fashion.

Similarly, conflicting I/O access patterns are also found in the external sorting algorithms. External sorting algorithms [37] are used to sort massive amounts of data that do not fit into main memory. Generally speaking, external sorting algorithms have two phases. The first phase partitions the data into smaller chunks using different strategies. The second phase processes these chunks and outputs the final sorted file. Based on the different strategies in the first phase, external sorting can be roughly classified into two groups [79]: (a) distribution-based sorting and (b) merge-based sorting. Both face the sequential versus random accesses in the two phases. Details are discussed in Chapter 6.

This dissertation proposes a solution that exploits the newly proposed adjacency model in order to eliminate the expensive random I/O access. This is achieved by organizing partitions (in the hash join algorithm) and chunks (in the external sorting algorithm) along the semi-sequential access paths. Details are discussed in Chapter 6. This work is another example of how a deeper understanding of hardware features can help to organize data more efficiently and hence, to improve performance.

1.5 Thesis road map and structure

To solve the problems described in the previous sections, this dissertation first proposes DBmbench as a significantly reduced database microbenchmark suite which simulates OLTP and DSS workloads. DBmbench enables quick evaluation on new designs and provides forecasting for performance of real large scale benchmarks. After that, I design and develop *Clotho*, a new buffer pool manager for database management systems. *Clotho* decouples the in-memory page layout from the storage organization by using a new query-specific layout called *CSM*. *CSM*, as a flexible page layout, combines the best performance of *NSM* and *DSM*, achieving good performance for both DSS and OLTP workloads. The layout for two-dimensional tables in *Clotho* inspires the idea of mapping data in high dimensional spaces to disks which leads to the next project: *MultiMap*. *MultiMap* is a new mapping algorithm that stores multidimensional data onto disks without losing spacial locality. *MultiMap* exploits the new adjacency model for disks to build a multidimensional structure on top of the linear disk space. It outperforms existing multidimensional mapping schemes on various spatial queries. I later utilize the multidimensional structure to improve the performance of two database operators that have opposite I/O access patterns in their different execution phases.

The rest of the thesis is organized as follows. Chapter 2 explains the basic concepts of adjacent blocks and semi-sequential access paths as the background knowledge. Chapter 3 introduces DBmbench. Chapter 4 presents the *Fates* database storage architecture with a focus on *Clotho*. Chapter 5 presents *MultiMap*, a new locality-preserving mapping algorithm to store multidimensional datasets. Chapter 6 continues to explore the opportunities brought by the new disk model in two major query operators, hash join and external sorting. Finally, Chapter 7 concludes my thesis work.

Chapter 2

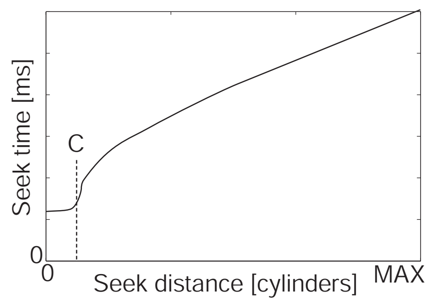
Background: adjacency model for modern disks

For completeness, this chapter reviews the adjacency model on which *Fates* and *MultiMap* are built. This chapter provides the background information that is necessary for understanding this dissertation. Detailed explanations and evaluations of disk technologies across a range of disks from a variety of vendors are provided by Schlosser et al. [62].

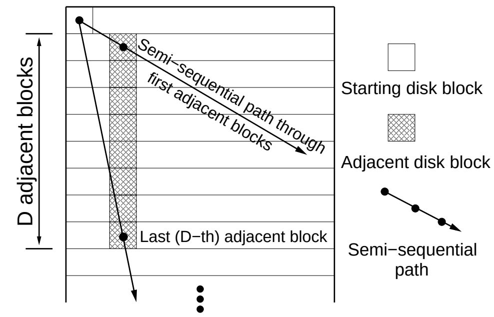
2.1 The traditional model for disks

It is well understood that a disk access consists of two phases: first, the disk head moves to the destination track; second, the head waits until the rotating disk platter brings the target block right under the head. After this, the head starts moving the data to or from the media. The time spent on the mechanical movements of the disk head and the platter is often called *positioning cost*. Accordingly, there are two components associated with the two phases respectively: seek time and rotational latency [55].

Compared to the actual time spent reading/writing data, the positioning cost—or the overhead of one disk access— is very high. For small chunks of data, the overhead could be more than 90% of the total time [58]. Therefore, the most efficient way to access data is the one that pays the positioning cost only once (i.e., at the beginning of the access). After the initial positioning cost, the disk head just reads/writes data continuously. Since the traditional abstraction for disks, such as SCSI, is a sequence of fixed-size blocks identified by ascending block numbers, the above access path can be easily expressed by accessing sequential disk blocks. For other access patterns involving non-contiguous blocks, the linear abstraction falls short of giving any insight on the relationships among those blocks, thus simply categorizing any non-sequential accesses



(a) Conceptual seek profile of modern disks.



(b) Adjacent blocks and semi-sequential access.

Figure 2.1: Conceptual seek profile of modern disk drives and illustration of adjacent blocks.

as "random accesses." In practice, most of the random accesses incur the overhead of seek time and rotational latency. With the above model, when it is not possible to store data in sequential disk blocks, applications will blindly use non-contiguous blocks that happen to be available at that time. This often leads to bad I/O performance.

The recent development of disk technologies sparks research on new disk models that can disclose more features to upper applications without burdening programmers. This chapter describes the one proposed by Schlosser et al. [62], called the Adjacency model. This new model exposes the second most efficient access paths, referred to as semi-sequential access paths, which are later utilized to store multidimensional datasets.

The new adjacency model introduces two primary concepts: *adjacent blocks* and *semi-sequential access*. The rest of this chapter explains them in detail. As described by Schlosser et al. [62], the necessary disk parameters can be exposed to applications in an abstract, disk-generic manner.

2.2 Adjacent disk blocks

The concept of adjacent blocks is based on two characteristics of modern disks as shown in Figure 2.1 [62]:

1. Short seeks of up to some cylinder distance, C, are dominated by the time to settle the head on a destination track;
2. Firmware features that are internal to the disk can identify and, thus, access blocks that require no rotational latency after a seek.

Figure 2.1(a) shows a conceptual view of seek time as a function of cylinder distance for modern disks. For very short distances of up to C cylinders, seek time is near constant and

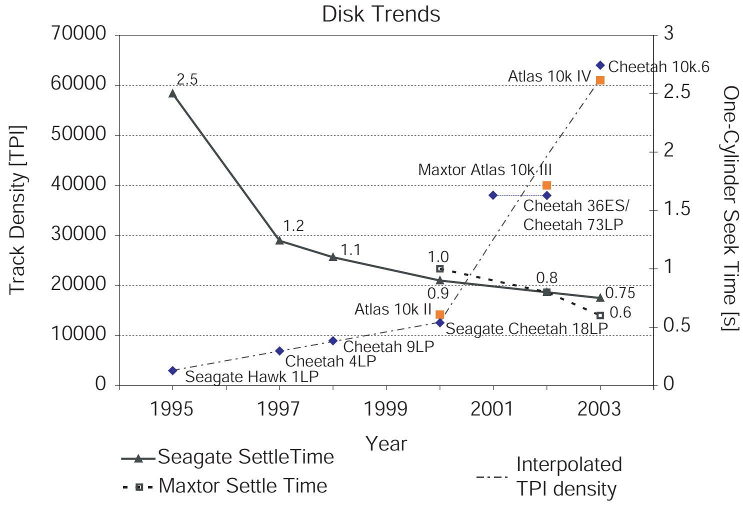


Figure 2.2: Disk trends for 10,000 RPM disks. Seagate introduced the Cheetah disk family in 1997 and Quantum/Maxtor introduced its line of Atlas 10k disks in 1999. Notice the dramatic increase in track density, measured in Tracks Per Inch (TPI), since 2000. The most recent disks introduced in 2004 Cheetah 10k.7 and Atlas 10k V (not shown in the graph) have densities of 105,000 and 102,000 TPI respectively, and settle times ≈0.1 ms shorter than their previous counterparts.

dominated by the time it takes for the disk head to settle on the destination track, referred to as *settle time*. If each of these cylinders is composed of R tracks, up to $D = R \times C$ tracks can be accessed from a starting track for equal cost.

While settle time has always been a factor in positioning disk heads, the dramatic increase in areal density over the last decade has brought it to the fore, as shown in Figure 2.2. At lower track densities (e.g., in disks introduced before the year 2000), only a single cylinder can be reached within the settle period. However, with the large increase in track density since the year 2000, up to C can now be reached.

The growth of track density has been one of the strongest trends in disk drive technology over the past decade, while settle time has decreased very little [5], as shown in Figure 2.2 for two families of enterprise-class 10,000 RPM disks from two manufacturers. With such trends, more cylinders can be accessed as track density continues to grow while settle time has improved very little. The Maxtor Atlas 10k III disk from the previous example has $C = 17$, and up to 8 surfaces for a total capacity of 73 GB. Thus, it has $D = 136$ adjacent blocks, according to the definition.

While each of these D tracks contains many disk blocks, there is one block on each track that can be accessed immediately after the head settles on the destination track, with no additional rotational latency. These blocks can be viewed as being *adjacent* to the starting block. Accessing any of these adjacent blocks takes just the settle time, the minimum time to access a block on

Conservatism	D	Settle time
0°	136	1.10 ms
10°	180	1.25 ms
20°	380	1.45 ms

Table 2.1: Adding extra conservatism to the base skew of $51°$ for the Atlas 10k III disk.

another track.

Figure 2.1(b) illustrates the adjacent blocks on a disk. For a given starting block, there are D adjacent disk blocks, one in each of the D adjacent tracks. All adjacent blocks have the same offset from the starting block because the offset is determined by the number of degrees the disk platters rotate within the settle time. For example, if W denotes the offset, with a settle time of 1 ms and a rotational period of 6 ms (i.e., for a 10,000 RPM disk), the offset will be $W = (1/6 \times 360°) = 60°$.

As settle time is not deterministic (i.e., due to external vibrations, thermal expansion, etc.), it is useful to add some extra conservatism to W to avoid rotational misses and to avoid suffering full revolution delay. Adding conservatism increases the number of adjacent tracks, D, that can be accessed within the settle time at the cost of added rotational latency, as shown in Table 2.1 for the Atlas 10k III disk.

2.2.1 Semi-sequential access

Accessing successive adjacent disk blocks enables *semi-sequential* disk access [25, 60], which is the second most efficient disk access method after pure sequential access. Figure 2.1(b) shows two potential semi-sequential paths from a starting disk block. Traversing the first semi-sequential path accesses the first adjacent disk block of the starting block, and then the first adjacent block of each successive destination block. Traversing the second path accesses the successive *last* or *Dth* adjacent blocks. Either path achieves equal bandwidth, despite the fact that the second path accesses successive blocks that are physically further away from the starting block. Recall that the first, second, or (up to) Dth adjacent block can be accessed for equal cost.

2.3 Quantifying access efficiency

A key feature of adjacent blocks is that, by definition, they can be accessed immediately after the disk head settles. To quantify the benefits of such access, suppose an application is accessing d non-contiguous blocks that map within D tracks. Without explicit knowledge of adjacency, accessing each pair of such blocks will incur, on average, rotational latency of half a revolution,

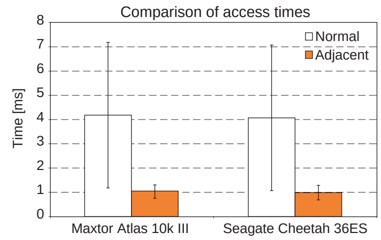


Figure 2.3: Quantifying access times.This graph compares the access times to blocks located within C cylinders. For both disks, the average rotational latency is 3 ms. For the Atlas 10k III disk, C = 17 and seek time within C ranges from 0.8 ms to 1.2 ms. For the Cheetah 36ES disk, C=12 and seek time ranges from 0.7 ms to 1.2 ms.

in addition to the seek time equivalent to the settle time. If these blocks are specifically chosen to be adjacent, then the rotational latency is eliminated and the access to the d blocks is much more efficient.

A system that takes advantage of accesses to adjacent blocks outperforms traditional systems. As shown in Figure 2.3, such a system, labeled Adjacent, outperforms a traditional system, labeled Normal, by a factor of 4, thanks to the elimination of all rotational latency when accessing blocks within C cylinders. Additionally, the access time for the Normal case varies considerably due to variable rotational latency, whereas the access time variability is much smaller for the Adjacent case; this is entirely due to the difference in seek time within the C cylinders, as depicted by the error bars.

2.4 Hiding low-level details from software

While the adjacency model advocates exposing adjacent LBNs to applications, it is unrealistic to burden application developers with such low-level details as settle time, physical skews, and data layout in implementing the algorithm described above. The application need not know the reasons for why disk blocks are adjacent, it just needs to be able to identify them through a `GETADJACENT` call to a software library or logical volume manager interface that encapsulates the required low-level parameters. This software layer can exist on the host as a device driver or within a disk array. It would either receive the necessary parameters from the disk manufacturer or extract them from the disk drives when the logical volume is initially created.

For a long time, the linear abstraction of disks has imposed limitations on database systems in storing relational tables and other datasets with more than one dimension. The new adjacency model for disks and the semi-sequential access that hides the rotational latency bring new opportunities to data layout designs in the database community. Database should enhance the interaction with storage devices to understand the device-specific features and to exploit them. Through carefully-designed interfaces, the improved interaction can improve performance without sacrificing the design and implementation simplicity.

This thesis will demonstrate how the semi-sequential access path is later utilized by *Atropos* to store relational tables and by *MultiMap* to store multidimensional datasets without losing spatial localities. It is also used to organize intermediate results in hash join and external sorting (Chapter 6).

Chapter 3

DBmbench

Make everything as simple as possible, but not simpler.
— Albert Einstein

This chapter presents DBmbench, a small yet representative database benchmark for computer microarchitecture. With the proliferation of database workloads on servers, much recent research on server architecture has focused on database system benchmarks. The TPC benchmarks for the two most common server workloads, OLTP and DSS, have been used extensively in the database community to evaluate the database system functionality and performance. Unfortunately, these benchmarks fall short of being effective in microarchitecture and memory system research due to several key shortcomings. First, setting up the experimental environment and tuning these benchmarks to match the workload behavior of interest involves extremely complex procedures. Second, the benchmarks themselves are complex and preclude accurate correlation of microarchitecture- and memory-level bottlenecks to dominant workload characteristics. Finally, industrial-grade configurations of such benchmarks are too large and preclude their use in detailed but slow microarchitectural simulation studies of future servers. In this paper, we first present an analysis of the dominant behavior in DSS and OLTP workloads, and highlight their key processor and memory performance characteristics. We then introduce a systematic scaling framework to scale down the TPC benchmarks. Finally, we propose the DBmbench, consisting of two substantially scaled-down benchmarks: μTPC-H and μTPC-C that accurately ($> 95\%$) capture the processor and memory performance behavior of DSS and OLTP workloads.

3.1 Introduction

Database workloads — such as Decision Support Systems (*DSS*) and Online Transaction Processing (*OLTP*) — are emerging as an important class of applications in the server computing market. Nevertheless, recent research [3, 9, 35] indicates that these workloads perform poorly on modern high-performance microprocessors. These studies show that database workloads have drastically different processor and memory performance characteristics as compared to conventional desktop and engineering workloads [71] that have been the primary focus of microarchitecture research in recent years. As a result, researchers from both the computer architecture and database communities are increasingly interested in careful performance evaluation of database workloads on modern hardware platforms [3, 6, 8, 9, 12, 20, 35, 39, 54, 74, 75].

To design microprocessors on which database workloads perform well, computer architects need benchmarks that accurately represent these workloads. There are a number of requirements that suitable benchmarks should satisfy. First, modern wide-issue out-of-order superscalar processors include a spectrum of mechanisms to extract parallelism and enhance instruction execution throughput. As such, the benchmarks must faithfully mimic the performance of the workloads at the microarchitecture-level to allow for designers to pinpoint the exact hardware bottlenecks. Second, microarchitecture simulation tools [70] are also typically five or more orders of magnitude slower than real hardware [69, 80]. To allow for practical experimentation turnaround, architects need benchmarks that are scaled down variations of the workloads [30] and have minimal execution time. Third, the benchmark behavior should be deterministic when across scaled datasets and varying system configurations to allow for conclusive experimentation. Finally, the benchmark sources or executables should either be readily available [71] or at most require installation and setup skills characteristic of a typical computer system researcher and designer.

Unfortunately, conventional DSS and OLTP database benchmarks, *TPC-H* and *TPC-C* [28], fall far short of satisfying these requirements. The TPC benchmarks have been primarily designed to test functionality and evaluate overall performance of database systems on real hardware. These benchmarks have orders of magnitude larger execution times than needed for use in simulation. To allow for practical experimentation turnaround, most prior studies [3, 8, 9, 12, 39, 54, 75] employ ad hoc abbreviations of the benchmarks (scaled down datasets and/or a subset of the original queries) without justification. Many of these studies tacitly assume that microarchitecture-level performance behavior is preserved.

Moreover, the TPC benchmarks' behavior at the microarchitecture-level may be non-deterministic when scaled. The benchmarks include complex sequences of database operations that may be reordered by the database system depending the nature of the sequence, the database system configuration and the dataset size, thereby substantially varying the benchmark behavior. Recent research by Hankins et al. [30], rigorously analyzes microarchitecture-level performance metrics of scaled datasets for *TPC-C* workloads and concludes that performance metrics cease to match when the dataset is scaled below 12GB. Unfortunately, such dataset sizes are still too large to allow for practical simulation turnaround.

Finally, the TPC benchmark kits for most state-of-the-art database systems are not readily available. Modern database systems typically include over one hundred configuration and installation parameters. Writing and tuning the benchmarks according to the specifications [76] on a given database system to represent a workload of interest may require over six months of experimentation even by a trained database system manager [34] and requires skills beyond those at hand for a computer system designer.

In this chapter, we present *DBmbench*, a benchmark suite representing DSS and OLTP workloads tailored to fit the requirements for microarchitecture research. The DBmbench is based on the key observation that the executions of database workloads are primarily dominated by a few intrinsic database system operations — e.g., a sequential scan or a join algorithm. By identifying these operations, microarchitecture-level behavior of the workloads can be mimicked by benchmarks that simply trigger the execution of these operations in the database system. We present the DBmbench benchmarks in the form of simple database queries, readily executable on database systems, and substantially reducing execution complexity as compared to the TPC benchmarks. Moreover, by isolating operation execution in stand-alone benchmarks, the datasets can be scaled down to only hundreds of megabytes while resulting in deterministic behavior precluding any optimizations in operation ordering by the database system.

Using hardware counters on an Intel Pentium III platform running IBM DB2, we show that the DBmbench benchmarks can match a key set of microarchitecture-level performance behavior, such as cycles-per-instruction (CPI), branch prediction accuracy, and miss rates in the cache hierarchy, of professionally tuned TPC benchmarks for DB2 to within 95% (for virtually all metrics). As compared to the TPC benchmarks, the DBmbench DSS and OLTP benchmarks: (1) reduce the number of queries from 22 and 5 to 2 and 1 simple queries respectively, (2) allow for scaling dataset sizes down to 100MB, and (3) reduce the overall number of instructions executed by orders of magnitude.

The remainder of this chapter is organized as follows: Section 3.2 presents a brief survey of recent database workload characterization studies and the research on microbenchmarks. Section 3.3 describes a framework to scale down database benchmarks and the design of DBmbench. Section 3.4 discusses the experimental setup and the metrics used to characterize behavior at the micro-architecture level. Sections 3.5 evaluates the scaling framework and the DBmbench. Section 3.6 concludes the chapter and outlines future work.

3.2 Related work

Database workloads evaluation at the architectural level is a prerequisite toward improving the suboptimal performance of database applications on today's processors. It identifies performance bottlenecks in software and hardware and points out the direction of future efforts. Several workloads characterization efforts [3, 6, 8, 9, 12, 39, 54, 75] explore the characteristics of OLTP and/or DSS on various hardware platforms using either a small-scale database or a subset of a standard workload or both. Three studies [9, 54, 75] emphasize the scale-down issues and demonstrate that the modified benchmarks they use do not affect the results. However, they still lack detailed analysis based on sufficient experiments on database systems with different scales. Most recently, Diep et al. [20] report how varying the configuration parameters affects the behavior of an OLTP workload. They propose a parameter vector consisting of number of processors, disks, warehouses, and concurrent clients to represent an OLTP configuration. They then formulate empirical relationships of the configurations and show how these configurations change the critical workload behavior. Hankins et al [30] continue this work by first proposing two metrics, average instructions per transaction (*IPX*) and average cycles per instruction (*CPI*) to characterize OLTP behavior. Then they conduct an extensive, empirical examination of an *Oracle* based commercial OLTP workload on a wide range of the proposed metrics. Their results show that the IPX and CPI behavior follows predictable trends which can be characterized by linear or piece-wise linear approximations.

There are a number of recent proposals for microbenchmarking database systems. The first processor/memory behavior comparison of sequential-scan and random-access patterns across four database systems [3] uses an in-memory TPC-like microbenchmark. The microbenchmark used consists of a sequential scan simulating a DSS workload and a non-clustered index scan approximating random memory accesses of an OLTP workload. Although the microbenchmark suite is sufficiently similar to the behavior of TPC benchmarks for the purposes of the study, a comprehensive analysis varying benchmark configuration parameters is beyond the scope of that paper. Another study [34] evaluates the behavior of a similar microbenchmark. Their microbenchmark simulates two sequential scan queries (Q1 and Q6) from the TPC-H suite, whereas for TPC-C, it devises read-only queries that generate random memory/disk access to simulate the access pattern of OLTP applications. Computation complexity affects the representativeness of the proposed micro-DSS benchmark, while the degree of database multiprogramming affects the micro-OLTP benchmark.

In this work, we build on the previous work as follows. First, we address the scaling problem from a database's point of view in addition to the traditional microarchitecture-approaches. We examine how query complexity, as one important dimension of the scaling framework, can be reduced while preserving their key hardware level characteristics. Second, we use a wealth of metrics that are important to obtain a complete picture of the workload behavior. Third, we build microbenchmarks for both DSS and OLTP workload.

3.3 Scaling down benchmarks

This section outlines a framework to scale down benchmarks. We identify three dimensions along which we can abbreviate benchmarks and discuss the issues involved when scaling database benchmarks workload along the dimensions. Then, we present the design of DBmbench.

Decision-support system (DSS) workloads are typically characterized by long, complex queries (often 1MB of SQL code) running on large datasets at low concurrency levels. DSS queries are characterized from sequential access patterns (through table scans or clustered index scans). By contrast, on-line transaction processing (OLTP) workloads consist of short read-write query statements grouped in atomic units called *transactions* [28]. OLTP workloads have high concurrency levels, and the users run many transactions at the same time. The queries in the transactions typically use non-clustered indices and access few records, therefore OLTP workloads are characterized by concurrent random accesses.

The prevalent DSS benchmark is TPC-H [28]. TPC-H consists of eight tables, twenty-two read-only queries ($Q1$–$Q22$) and two batch update statements, which simulate the activities of a wholesale supplier. For OLTP, the TPC-C benchmark portrays a wholesale supplier and several geographically distributed sale districts and associated warehouses [76]. It is comprised of nine tables and five different types of transactions. TPC-H is usually executed in a single-query-at-a-time fashion while TPC-C models multiple clients running concurrently.

3.3.1 A scaling framework

A database benchmark is typically composed of a dataset and a workload (set of queries or transaction) to run on the dataset. Inspired by the differences between DSS and OLTP outlined in Section 3.3, we scale down a full benchmark along three orthogonal dimensions, shown in Figure 3.1: workload complexity, dataset size, and level of concurrency.

In order to scale down a benchmark's workload complexity, one approach is to choose a subset of the original queries [9, 54, 75]. Another approach is to reduce the query complexity by removing parts of the query or reducing the number of items in the SELECT clause. Both methods effectively reduce query complexity at the cost of sacrificing representativeness; choosing

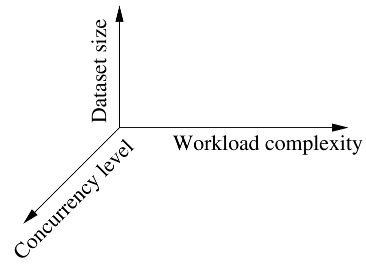


Figure 3.1: Benchmark-scaling dimensions.

a subset of queries may exclude important queries that significantly affect behavior, while the complexity reduction method may inadvertently result in dramatic changes to the query plans and thus modify the benchmark's behavior.

Scaling down along the dataset size dimension is fairly straightforward, because benchmark specifications typically provide rules or software to scale down datasets. The main concern when scaling down along this dimension is to preserve the performance characteristics of the workload, as reducing the database size is likely to alter the query plans (and consequently the instruction mix) and cause performance bottlenecks to shift. Similarly, scaling the level of concurrency is straightforward, because benchmarks include in their specifications the how many users should run per data unit. It is important to abide by the scaling rules in the specifications, to maintain the data and usage properties.

3.3.2 Framework application to DSS and OLTP benchmarks

From the perspective of benchmark evaluation, DSS queries are mostly read-only and usually access a large portion of the dataset. While there are also batch updates, read-only operations are the critical part in a DSS workload. Queries are executed one-at-a-time, and the execution process for each query is predictable and reproducible. Furthermore, while DSS queries vary enormously in functionality, they typically spend most of their time executing basic query operations such as sequential scan and/or join.

When examining the optimizer's suggested plans for TPC-H queries, we find that 50% queries are dominated by table scans (over 95% of their execution time is estimated to be due to table scans) whereas 25% of the queries spend more than 95% of the time executing nested-loop joins. The remaining 25% of the queries executed table scans for about 75% on average and nested-loop joins for about 25% on average. Therefore, we can represent scan-bound and join-bound queries by executing the two dominant operators.

Considering the complexity and depth of a TPC-H query plan, this result may seem counter-

<table>
<tr><th>Table T1</th><th>Table T2</th></tr>
<tr><td>CREATE TABLE T1 (
    a1 INTEGER NOT NULL,
    a2 INTEGER NOT NULL,
    a3 INTEGER NOT NULL,
    <padding>,
    FOREIGN KEY (a1) references T2
);</td><td>CREATE TABLE T2 (
    a1 INTEGER NOT NULL PRIMARY KEY,
    a2 INTEGER NOT NULL,
    a3 INTEGER NOT NULL,
    <padding>
);</td></tr>
</table>

Table 3.1: DBmbench database: table definitions.

intuitive; however, the major part of the filtering is done at the lowest levels of the operator tree, and the result size is reduced dramatically as execution continues to the upper levels of the tree. In conclusion, DSS workloads can be scaled down by (1) constructing representative queries that execute the dominant operators; (2) using small datasets that fit in the research testbed. The concurrency level is already low in DSS.

OLTP workloads are characterized by a large number of concurrent and continuous update-intensive transactions that generate random-like memory access patterns. Queries in OLTP workloads are simple and only touch a small fraction of the dataset. OLTP execution is quite different from that of DSS, in that it involves a stream of concurrent transactions including numerous simple queries and insert/update statements. Scaling down OLTP benchmarks involves decreasing the number of concurrent clients and reducing the dataset sizes. To accurately mimic the workload's scattered dataset access pattern, the concurrent clients should execute one or more queries with random access to memory.

3.3.3 DBmbench design

DBmbench is a microbenchmark suite that can emulate DSS and OLTP workloads at the computer architectural level. DBmbench includes two tables and three simple queries. The design principles are (1) keeping table schemas and queries as simple as possible; (2) focusing on the dominant operations in DSS and OLTP.

DBmbench tables. DBmbench uses two tables, T1 and T2, as shown in Table 3.1. T1 and T2 have three fields each, $a1$, $a2$, and $a3$, which will be used by the DBmbench queries. "`padding`" stands for a group of fields that are not used by any of the queries. We use the values of these fields as "`padding`" to make records 100 Byte long, which approximates the average record length of TPC-H and TPC-C. The type of these fields makes no difference in the performance, and by varying its size we can experiment with different record sizes without affecting the benchmark's queries.

μSS query	μNJ query	μIDX query

`SELECT distinct (a3)`
`FROM T1`
`WHERE Lo < a2 < Hi`
`ORDER BY a3`
	`SELECT avg (T1.a3)`
	`FROM T1, T2`
	`WHERE T1.a1=T2.a1 AND Lo < T1.a2 < Hi`
		`SELECT avg (a3)`
		`FROM T1`
		`WHERE Lo < a2 < Hi`

Table 3.2: DBmbench workload: queries.

The values of field $a1$ is in the range of 1 and $150,000$. The field of $a2$ takes values randomly within the range of 1 to $20,000$ and $a3$ values are uniformly distributed between 1 and 50. Since $a1$ is the primary key of T2, T2 has $150,000$ records which are ordered by $a1$. For each record in T2, a random number of rows within $[1 \ldots 7]$ are generated in T1. The distributions and values in these tables are a properly scaled-down subset of the data distributions and values in the *lineitem* and *orders* table of TPC-H. Whether to create indices on T1 and T2 depends on the type of workloads DBmbench is trying to mimic. Details are in the next section.

DBmbench queries. Based on the discussion in Section 3.3.2, the design of the DSS microbenchmark mainly focuses on simplifying query complexity. Moreover, as discussed previously, scan and join operators typically dominate DSS query execution time. Therefore, we propose two queries for the DSS microbenchmark, referred to as μTPC-H, as follows: sequential scan query with sort (μSS) and join query (μNJ). The first two columns of Table 3.2 show the SQL statements for these two queries.

The μSS query is a sequential scan over table T1. We will use it to simulate the DSS queries whose dominant operators are sequential scans. The two parameters in the predicate, L_o and H_i, are used to obtain different selectivities. The order-by clause sorts the query results by the values in the a3 field, and is added for two reasons. First, sort is an important operator in DSS queries, and the order-by clause increases the query complexity effectively to overcome common shortcomings in existing microbenchmarks [3, 34]. Second, the clause will not alter the sequential scan access method, which is instrumental in determining the basic performance characteristics. The μSS query is run against T1 without any indices.

Previous microbenchmarks use aggregation functions in the projection list to minimize the server/client communication overhead [3, 34]. To prevent the optimizer from omitting the sort operator, μSS uses "distinct" instead of the aggregate. "Distinct" eliminates duplicates from the answer and achieves the same methodological advantage as the aggregate, because the number of distinct values in $a3$ is small (less than or equal to 50), and does not interfere with the performance characteristics. Our experiment results corroborate these hypotheses.

Although previously proposed microbenchmark suites [34] often omit the join operator, it is actually an important component in DSS queries and has very different behavior from table scan [3]. To mimic the DSS workload behavior accurately, we consider the join operator and propose the μNJ query to simulate the DSS queries dominated by the join operator. The predicate "$L_o < T1.a2 < H_i$" adds an adjustable selectivity to the join query so that we can control the number of qualifying records by changing the values of L_o and H_i. An index on $T2.a1$ is created for the μNJ query to reduce the execution time.

The OLTP microbenchmark, which we call μTPC-C, consists of one non-clustered index scan query (μIDX), shown in the third column of Table 3.2. Accordingly, an index on $T1.a2$ is created as the non-clustered index. The μIDX query is similar to the μSS query in μTPC-H. The key difference is that, when evaluating the predicate in the "where" clause, the table scan through the non-clustered index generates a TPC-C-like random access pattern. The proposed μIDX query is a read-only query which only partly reflects the type of actions in TPC-C. The transactions also include a significant number of write statements (updates, insertions, and deletions). In our experiments, however, we found that adding updates to the DBmbench had no effect in the representativeness of the benchmark. The reason is that, like queries, updates use the same indices to locate data, and the random accesses on the tables through index search is the dominant behavior in TPC-C. Therefore, the μIDX query is enough to represent the benchmark. We scale down the dataset to the equivalent of one warehouse (100MB) and the number of concurrent users to ten (as directed by the TPC-C specification).

3.4 Experimental methodology

In this section, we present the experimental environment and methodology we use in the work. Industrial-strength large-scale database servers are often configured with fully optimized high-performance storage devices so that the execution process is typically CPU- rather than I/O-bound. A query's processor execution and memory access characteristics in such settings dominate overall performance [9]. As such, we ignore I/O activity in this work and focus on microarchitecture-level performance.

We conducted our experiments on a 4-way 733 MHz Intel Pentium III server. Pentium III is a 3-way out-of-order superscalar processor with 16 KB level-one instruction and data caches, and a unified 2 MB level-two cache. The server has 4 GB of main memory and four SCSI disks of 35 GB capacity. To measure microarchitecture-level performance, we use the hardware counters featured in the processors to count events or measure operation latencies. [1] We use Intel's

[1] We have also verified that the microarchitecture-level event counts between the TPC benchmarks and DBmbench match on a Pentium 4 platform. However, we are not aware of an execution time breakdown model for the platform to match the stall time components, and therefore we omit these results in the interest of brevity.

EMON tool to operate the counters and perform measurements. The counted events include the total number of retired instructions, the number of cache misses at each level, mispredicted branch instructions, and CPU cycles, etc.

We use IBM DB2 UDB V.7.2 with Fix Package 11 [19] on Linux (kernel version 2.4.18) as the underlying database management system, and run TPC-H and TPC-C benchmarks. As in prior work [8, 9, 12, 34, 75], we focus on the read-only queries which are the major components of the TPC-H workload, but our results can easily be extended to include the batch updates. For our experiments, we used a slightly modified version of the TPC-C kit provided by IBM which has been optimized for DB2. Prior work [3] suggests that commercial DBMS exhibit similar microarchitecture-level performance behavior when running database benchmarks. Therefore, expect the results in this work to be applicable to other database servers.

For TPC-H, we record statistics for the entire execution of all the queries. We measure work units in order to minimize the effect of startup overhead. Each work unit consists of multiple queries of the same type but with different values of the substitute parameters (i.e., selectivity remains the same, but qualifying records vary). We run each work unit multiple times, and measure events per run. The measurement is repeated several times to eliminate the random factors during the measurement. The reported results have less than 5% discrepancy across different runs.

For TPC-C, we count a pair of events during a five-second fixed time interval. We measure events multiple times and in different order each time. For all experiments, we ensure that the standard deviation is always lower than 5% and compute an average over the per-event collected measurements.

When scaling dataset sizes, we also change the system configuration parameters to ensure the setup is valid. Database systems include a myriad of software-configured parameters. In the interest of brevity and to allow for practical experimental turnaround time, in this work we focus on the buffer pool size as the key database system parameter to vary. As database applications are heavily memory-bound, the buffer pool size: (1) is expected to have the most fundamental effect on processor/memory performance, and (2) often determines the values of other memory-related database system parameters. For TPC-C, where the number of concurrent users is intuitively important for the system performance, we also vary the degree of concurrency. While we have studied other parameters (such as degree of parallelism), we did not find any insightful results based on them.

When measuring performance, we are primarily interested in the following characteristics: (1) query execution time breakdown, (2) memory stall time breakdown in terms of cycles lost at various cache levels and TLBs, (3) data and instruction cache misses ratios at each level (4) branch misprediction ratio.

To break down query execution time, we borrow the model proposed and validated in [3] for the the Pentium III family of processors. In this model, query execution time is divided into cycles devoted to useful computation and stall cycles due to various microarchitecture-level mechanisms. The stalls are further decomposed into different categories. Hence, the total execution time T_Q can be expressed by the following equation:

$$T_Q = T_C + T_M + T_B + T_R - T_{OVL}$$

T_C is the actual computation time; T_M is wasted cycles due to misses in the cache hierarchy; T_B refers to stalls due to the branch prediction unit including branch misprediction penalty and BTB miss penalty; T_R is the stalls due to structural hazards in the pipeline due to lack of functional units or physical rename registers; T_{OVL} indicates the cycles saved by the overlap of the stall time because of the out-of-order execution engine.

T_M is further broken down into six components:

$$T_M = T_{L1D} + T_{L1I} + T_{L2D} + T_{L2I} + T_{DTLB} + T_{ITLB}$$

These are stalls caused by L1 cache misses (data and instruction), L2 cache misses (data and instruction), and TLB misses respectively.

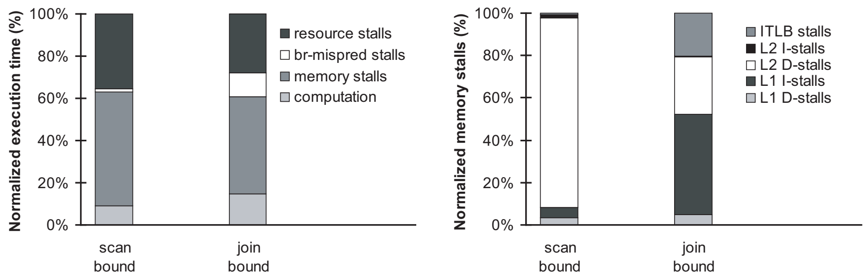
3.5 Evaluation

In this section, we compare and contrast the microarchitecture-level performance behavior of the TPC and DBmbench benchmarks. We first present results for the DSS benchmarks followed by results for the OLTP benchmarks.

3.5.1 Analyzing the DSS benchmarks

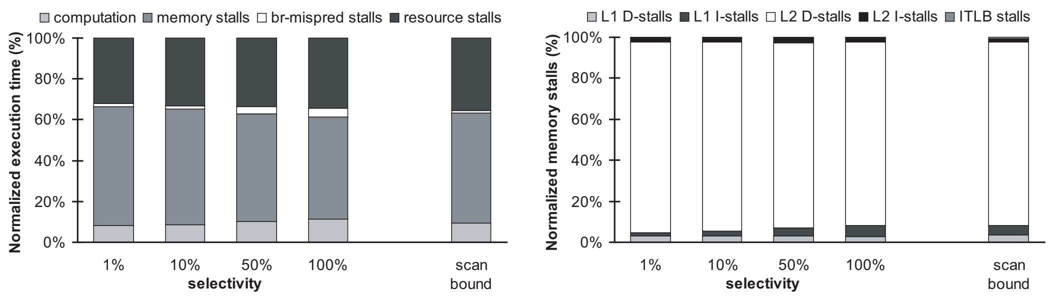
When taking a close look at the query plans provided by the optimizer, we corroborate our intuition from 3.3.2 that one of the two “scan” or “join” operators account for more than 95% of the total execution time in each of the TPC-H queries. We also find that these two operators remain dominant across database system configurations and dataset sizes. Therefore, we classify the TPC-H queries into two major groups: “scan bound” query and “join bound” query. We evaluate the microarchitecture-level performance these groups on a 10GB dataset.

Figure 3.2(a) shows the representative execution time breakdowns of the two groups. Each



(a) **TPC-H execution time breakdown.** (b) **TPC-H memory stall breakdown.**

Figure 3.2: TPC-H time breakdowns.Representative time breakdowns for the "scan bound" and "join bound" groups, which spend their execution time mainly on sequential scan and join operators respectively.



(a) **Execution time breakdown comparison.** (b) **Memory stall breakdown comparison.**

Figure 3.3: μSS vs. TPC-H "scan bound" query. The graphs show the time breakdowns of μSS and TPC-H "scan bound" queries. For the μSS query, we vary its selectivity from 1% to 100% to show how selectivity affects the behavior.

bar shows the contributions of the three primary microarchitectural stall components (memory stalls, branch stalls, and resource stalls) as a percentage of the total query execution time.

These results corroborate prior findings [3, 9] that on average the processor is idle more than 80% of the time when executing the TPC-H queries. In both groups, the performance bottlenecks are memory-related and resource-related stalls, each accounting for approximately 25% to 50% of the execution time. While we can not measure the exact cause of the resource-related stalls, our conjecture is that they are related to the load/store unit due to the high overall fraction of memory accesses in these queries.

Not surprisingly, the queries in the "join bound" group have a higher computation time component because joins are more computationally intensive than sequential scans. Furthermore, control-flow in joins are data-dependent and irregular, and as such the "join bound" group exhibits a higher branch misprediction stall (over 15%) component as compared to the "scan bound" group whose execution is dominated by loops exhibiting negligible branch misprediction stall time.

Figure 3.2(b) depicts a breakdown of memory stall time. The figure indicates that the "scan bound" group's memory stalls are dominated (over 90%) by L2 data misses. These queries simply thrash the L2 cache by marching over the entire dataset and as such have no other relatively significant memory stall component.

Unlike the "scan bound" queries, the "join bound" queries suffer from frequent L1 i-cache and i-TLB misses. These queries exhibit large and dynamic i-cache footprints that can not fit in a 2-way associative 16KB cache. The dynamic footprint nature of these queries is also consistent with their irregular control flow nature and their high branch misprediction stalls. Moreover, frequent branch misprediction also inadvertently pollutes the i-cache with the wrong-path instructions, thereby increasing the miss rate.

3.5.2 Comparison to μTPC-H

In this section, we compare the microarchitecture-level performance behavior of the "scan bound" and "join bound" TPC-H queries against their μTPC-H counterparts. As before, TPC-H results assume a 10GB dataset while the μTPC-H results we present correspond to a significantly scaled down 100MB dataset.

Figure 3.3(a) compares the execution time breakdown of the μSS query and TPC-H queries in the "scan bound" group. The x-axis in the left graph reflects the selectivity of the predicate in the μSS query. These results indicate that the execution time breakdown of the TPC benchmark is closely mimicked by the DBmbench. Our measurements indicate that the absolute benchmark performances also match, averaging a CPI of approximately 4.1.

The μSS query with high selectivity sorts more records, thereby increasing the number of branches in the instruction stream. These branches do not exhibit any patterns and are difficult to predict, which unavoidably results in a higher branch misprediction rate. As shown in Figure 3.3(a), the μSS query successfully captures the representative characteristics of the TPC-H queries in the "scan bound" group: it exposes the same bottlenecks and has similar percentages of each component. Figure 3.3(b) compares the memory stall breakdowns of the μSS query and the "scan bound" queries. The μSS query exposes the same bottlenecks at the L2 (for data accesses) and L1 instruction caches.

To mimic the "join bound" queries, we focus on the nested loop join because it is the only join operator that appears to be dominant. To represent TPC-H's behavior accurately, we build an index on the join fields when evaluating μNJ. We do so because most join fields in the TPC-H workload have indices, and the index decreases the query execution time significantly.

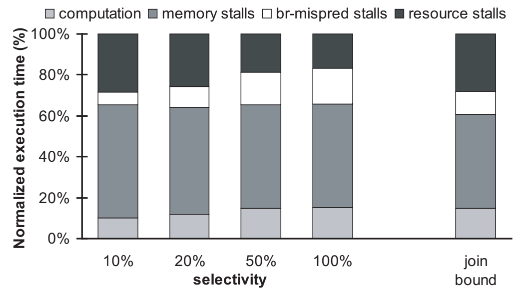
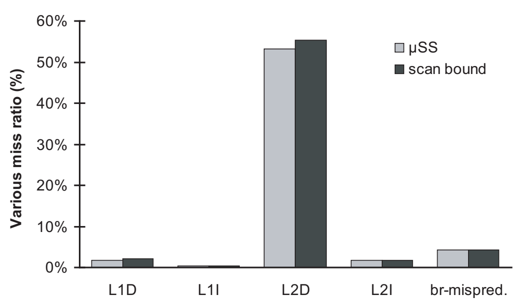
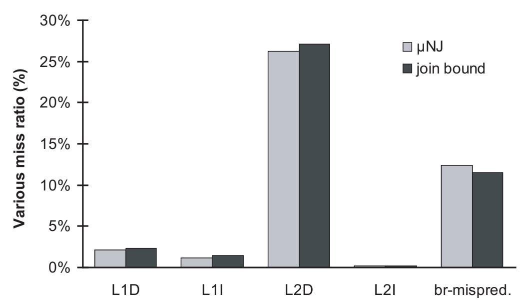


Figure 3.4: μNJ vs. TPC-H "join bound" query. The graph shows the time breakdowns of μNJ and TPC-H "join bound" queries. For the μNJ query, we vary its selectivity from 1% to 100% to show how selectivity affects the behavior.



(a) μ**SS and "scan bound" miss ratio comparison.**



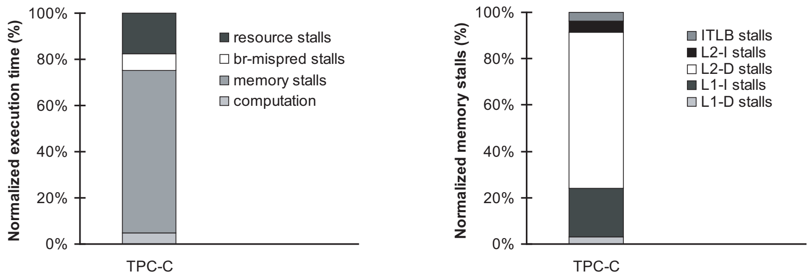
(b) μ**NJ and "join bound" miss ratio comparison.**

Figure 3.5: μTPC-H vs. TPC-H.The graphs compare the miss ratios of μTPC-H and TPC-H

Figure 3.4 examines the execution time breakdown of the μNJ query and the "join bound" queries. It shows that selectivity significantly affects the execution time breakdown of the μNJ query, and a 20% selectivity best represents the characteristics of a "join bound" query. We also verify that the absolute performance measured in CPI matches between the TPC queries and the scaled down DBmbench query with a 20% selectivity. The average CPI for these benchmarks are approximately 2.95.

Figure 3.5(a) and Figure 3.5(b) compare the stall event frequencies across the benchmarks suites. Much like the "scan bound" queries, the execution of μSS is dominated by L2 cache misses. Similarly, besides the high fraction of L2 cache stalls, the execution of μNJ much like the "join bound" queries also incurs a high rate of L1 i-cache misses and branch mispredictions. The L1 d-cache misses are often overlapped Moreover, the actual differences in event counts between the benchmark suites are negligible.

In summary, the simple μSS and μNJ queries in μTPC-H closely capture the microarchitecture-level performance behavior of the "scan bound" and "join bound" queries in the TPC-H workload



(a) **TPC-C execution time breakdown.** (b) **TPC-C memory stall breakdown.**

Figure 3.6: TPC-C time breakdowns.

respectively. μTPC-H reduces the number of queries in TPC-H from 22 to 2. Moreover, μTPC-H allows for scaling down the dataset with predictable behavior from 10GB to 100MB. We measure a reduction in the total number of instructions executed from 1.8 trillion in TPC-H to 1.6 billion in μTPC-H, making μTPC-H a suitable benchmark suite for microarchitecture simulation and research.

3.5.3 Analyzing the OLTP benchmarks

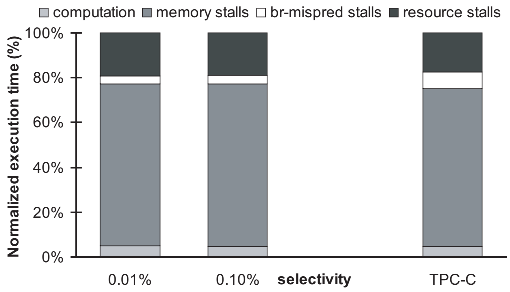
Figure 3.6 shows the execution time and memory stall breakdowns for a 150-warehouse, 100-client TPC-C workload corresponding to a 15GB dataset. Much like the TPC-H results, these results corroborate prior findings on microarchitecture-level performance behavior of TPC-C [3].

The effect of the high instruction cache miss rates result in an increased memory stall component, which is nevertheless dominated by L2 stall time due to data accesses. The reason is that, although the L2 data miss rate is not that high, in TPC-C each L2 data miss reflects I/O delays (TPC-C incurs I/O costs regardless of the dataset size, because it logs the transaction updates).

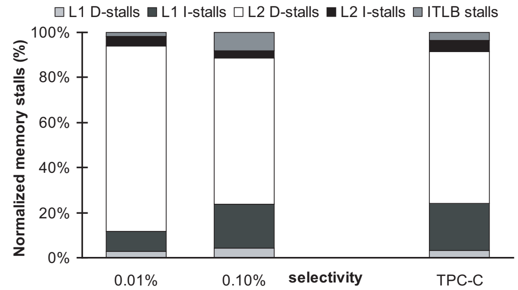
3.5.4 Comparison to μTPC-C

Figure 3.7(a) compares the execution time breakdown of the μIDX query and the TPC-C benchmark. It shows that the μIDX query with 0.01% and 0.1% selectivity mimics the execution time breakdown of TPC-C. Increasing the selectivity to 0.1%, however, also achieves the desired memory stall breakdown (shown in Figure 3.7(b)). The interesting result here is that selectivity is important to fine-tune memory stall breakdown.

The execution of a single μIDX query exhibits fewer stall cycles caused by L1 instruction cache misses. This is because the TPC-C workload has many concurrently running transactions which aggravate the localities in the instruction stream. We can improve the similarity by running



(a) **Execution time breakdown comparison.**



(b) **Memory stall breakdown comparison.**

Figure 3.7: μIDX vs. TPC-C. The graphs show the time breakdowns of μIDX and TPC-C.

multiple μIDX queries. We increase the L1 instruction cache miss rate from 0.017 to 0.032 with 10 currently running queries, which is similar to the L1 instruction cache miss rate of TPC-C ($\simeq 0.036$).

Figure 3.8 shows the miss ratios of μIDX with a 0.1% selectivity and TPC-C. We can see from the graph that the branch misprediction rate of the μIDX query is the performance metric that is far from the real TPC-C workload. The simpler execution path of the μIDX query might be the reason for this discrepancy. The branch misprediction rate cannot be improved with a higher degree of concurrency. Fortunately, this discrepancy does not affect the performance bottleneck, as shown in Figure 3.7(a). This branch prediction mismatch, however, results in a small overall CPI difference of 8.4 for μIDX as compared to 8.1 for TPC-C.

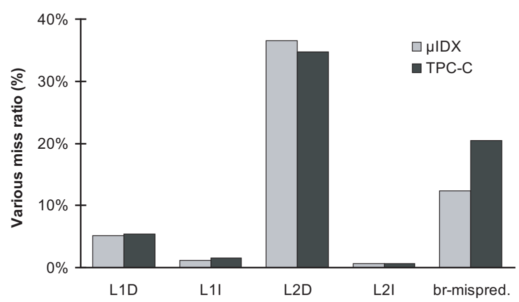


Figure 3.8: μIDX vs. TPC-C.The graph compare the miss ratios of μIDX and TPC-C

A single μIDX query is not enough to mimic the instruction-related performance behavior of TPC-C. We can achieve better approximation by running multiple μIDX queries. As for the data cache, the μIDX query represents TPC-C well on the L1 data cache. Less locality in our simple μIDX query's execution, however, causes a higher L2 data cache miss rate.

In summary, the μIDX query can expose the execution bottlenecks of TPC-C successfully. Running multiple μIDX queries (usually 10) can closely mimic the execution path of the TPC-C workload with a 7.06% relative error. The μIDX query fails to approximate the branch misprediction rate of the TPC-C workload. We should take this into account when predicting the branch behavior of the TPC-C benchmark. μTPC-C reduces the dataset size from 10GB to 100MB as compared to TPC-C. It also reduces five transactions containing approximately 50 queries to just a single query. The total number of instructions executed per transaction is reduced from 91.65 million in TPC-C to 2.75 million in μTPC-C.

3.6 Chapter summary

Database applications and systems are emerging as the popular (if not dominant) commercial workloads. Computer architects are increasingly relying on database benchmarks to evaluate future server designs. Unfortunately, conventional database benchmarks are prohibitively complex to set up, and too large to experiment with and analyze when evaluating microarchitecture-level performance bottlenecks.

This chapter proposes DBmbench, a benchmark suite representing DSS and OLTP workloads tailored to the requirements for microarchitecture research. The rationale of DBmbench is based on the observation that the execution of database benchmarks are decided by a small set of database operations. At the microarchitecture level, the execution of complicated database queries are broken down into executions of a handful database operations with each having a small code footprint and unique performance characteristics. By identifying these key database operations for DSS and OLTP workloads, DBmbench proposes a set of simple queries that use the key database operations as their building blocks. By doing so, DBmbench is able to preserve the performance characteristics of their large counterparts.

DBmbench identifies that sequential scan and nested loop join are the two operators that primarily dominate the performance of TPC-H, a widely used DSS workload. The two operators take more than 90% of the execution time of all TPC-H queries, and they show very different performance behavior. Therefore, we use two types of queries to mimic the TPC-H queries: the μSS query for sequential scan and the μNJ query for join. Both μSS and μNJ match the CPIs of sequential scan and join queries, which are 4.1 and 2.95, respectively. In addition, they also closely, usually with a less than 5% relative error, capture the detailed characteristics such as cache miss ratios at different levels of the memory hierarchy. We find that selectivity affects the μNJ query more than it does on μSS. Both queries reduce the number of instructions executed by $1000\times$.

OLTP workloads, such as TPC-C, present more challenges as there are more variations in the execution of these workloads due to the concurrency in transaction executions. The μIDX query, a read-only non-clustered index scan, captures the primary access pattern, random access, of TPC-C, thus its performance behavior resembles that of TPC-C, especially on exposing bottlenecks. But single μIDX query falls short of capturing the complex interaction among concurrently running transactions in the TPC-C execution. Running multiple μIDX queries (usually 10) improves the representativeness on detailed statistics such as cache miss rate with a 7.06% relative error. μTPC-C, consisting of 10 concurrently running μIDX queries, also reduces the number of instructions executed per transaction by $33\times$.

Although DBmbench is originally designed for microarchitectural research, it is also valuable for database system research, especially in sensitivity analysis because the simple queries in DBmbench do a good job at isolating operation executions. This isolation makes it easy to control experiments and to study the effect of a single operation on performance. The small set of tunable knobs in DBmbench also makes performance analysis and comparison more manageable. The other projects in this thesis use DBmbench to conduct sensitivity analysis on query selectivities and payloads.

Admittedly, DBmbench can not perfectly mimic the performance behavior of TPC-H and TPC-C, but it achieves its goal of providing a simple, more controllable, yet representative database workload for microarchitecture research to locate bottlenecks and gain insight into performance characteristics at the early design stage with short turn-around time.

Chapter 4

Fates database management system storage architecture

The life of data in Fates:
First, Clotho spins the thread of life with her distaff: generating request for data.
Then, Lachesis decides the length of the thread with her ruler: determining the I/O request size.
Finally, Atropos cuts the thread with her scissors: fetching and returning the data.

This chapter describes the design and implementation of the *Fates* database storage manager, with a focus on *Clotho*, and evaluates its performance under a variety of workloads using both disk arrays and simulated MEMS-based storage devices.

As database application performance depends on the utilization of the memory hierarchy, smart data placement plays a central role in increasing locality and in improving memory utilization. Existing techniques, however, do not optimize accesses to all levels of the memory hierarchy and for all the different workloads, because each storage level uses different technology (cache, memory, disks) and each application accesses data using different patterns. The *Fates* database storage manager addresses this problem by building pages of data that are tailored to match the characteristics of the medium in which they are stored, i.e., on disk, in main memory, and in CPU caches. On disk, data layout is tailored to the underlying storage devices such that tables can be accessed efficiently in either dimension. In memory, pages are tailored to individual queries, such that only the required fields of records are fetched from disk, saving disk bandwidth and reducing memory footprint. Memory pages are partitioned into minipages, which optimizes CPU cache performance, as in the *PAX* page layout [4]. In contrast to previous systems, in *Fates*, data page layouts are *decoupled*, meaning that their format is different at each level of the memory hierarchy. The *Fates* architecture consists of three main components, which are named for the three Fates of Greek mythology. *Clotho* is the buffer pool and storage management component which manages in-memory and on-disk data layout. *Atropos* is a logical volume manager which exposes device-specific details of underlying storage devices to allow efficient access to two-dimensional data structures. Lastly, the *Fates* system leverages device-specific performance characteristics provided by *Lachesis* [59] to tailor access to storage.

4.1 Introduction

Page structure and storage organization have been the subject of numerous studies [4, 11, 15, 31, 40], because they play a central role in database system performance. Research continues as no single data organization serves all needs within all systems. In particular, the access patterns resulting from queries posed by different workloads can vary significantly. One query, for instance, might access all the attributes in a table (*full-record access*), while another accesses only a subset of them (*partial-record access*). Full-record accesses are typical in transactional (OLTP) applications where insert and delete statements require the entire record to be read or written, whereas partial-record accesses are often found in decision-support system (DSS) queries. Moreover, when executing compound workloads, one query may access records sequentially while others access the same records "randomly" (e.g., via non-clustered index). Currently, database storage managers implement a single page layout and storage organization scheme, which is utilized by all applications running thereafter. As a result, in an environment with a variety of workloads, only a subset of query types can be serviced well.

Several data page layout techniques have been proposed in the literature, each targeting a different query type. Notably, the N-ary Storage Model (*NSM*) [52] stores records consecutively, optimizing for full-record accesses, while penalizing partial-record sequential scans. By contrast, the Decomposition Storage Model (*DSM*) [18] stores values of each attribute in a separate table, optimizing for partial-record accesses, while penalizing queries that need the entire record. More recently, *PAX* [4] optimizes cache performance, but not memory utilization. "Fractured mirrors" [53] reduce *DSM*'s record reconstruction cost by using an optimized structure and scan operators, but need to keep an *NSM*-organized copy of the database as well to support full-record access queries. None of the previously proposed schemes provides a universally efficient solution, however, because they all make a fundamental assumption that the pages used in main memory must have the same contents as those stored on disk.

This chapter describes how *Fates*, a dynamic and automated database storage manager, addresses the above problems by decoupling in-memory data layout from on-disk storage layout to exploit unique device-specific characteristics across the memory hierarchy.

Fates consists of three independent components, named after three goddesses in Greek mythology: *Atropos* is the volume manager [60] which enables efficient accesses to two dimensional data structures stored both in disk arrays and in MEMS-based storage devices (MEMStores) [63, 81] through new simple interfaces; *Lachesis* [59] utilizes the explicit device-specific information provided by *Atropos* to construct efficient I/O requests for varying and mixed workloads without manual tuning; *Clotho* [66] is the buffer pool manager which manages query-specific data pages dynamically based on queries' needs. The contents of in-memory query-specific pages could be different from the on-disk pages, hence the meaning of "decoupling".

This decoupling offers two significant advantages. First, it optimizes storage access and memory utilization by fetching from disk only the data accessed by a given query. Second, it allows new two-dimensional storage mechanisms to be exploited to mitigate the trade-off between the *NSM* and *DSM* storage models. The break of conventional view of single and static page layouts provides the flexibility of using different page layouts that perform best for the current workload at the current memory hierarchy level.

Among the three *Fates* components, this chapter focuses on the design and a prototype implementation of *Clotho* within the Shore database storage manager [13]. Experiments with disk arrays show that, with only a single storage organization, performance of DSS and OLTP workloads is comparable to the page layouts best suited for the respective workload (i.e., *DSM* and *PAX*, respectively). Experiments with a simulated MEMStore confirm that similar benefits will be realized with these future devices as well.

The remainder of this chapter is organized as follows. Section 4.2 introduces background knowledges and related work. Section 4.3 describes the decoupled data organization in *Fates*. Section 4.4 presents the *Fates* architecture and the relationship among its three components. Section 4.5 describes the design of *Atropos*. Section 4.6 details how *Clotho*, the new buffer pool manager, constructs query-specific pages using the new in-memory page layout. Section 4.7 describes our initial implementation, and Section 4.8 evaluates this implementation for several database workloads using both a disk array logical volume and a simulated MEMStore.

4.2 Background and related work

Conventional relational database systems store data in fixed-size pages (typically 4 to 64 KB). To access individual records of a relation (table) requested by a query, a scan operator of a database system accesses main memory. Before accessing data, a page must first be fetched from non-volatile storage (e.g., a logical volume of a disk array) into main memory. Hence, a page is the basic allocation and access unit for non-volatile storage. A database storage manager facilitates this access and sends requests to a storage device to fetch the necessary blocks.

A single page contains a header describing what records are contained within and how they are laid out. In order to retrieve data requested by a query, a scan operator must understand the page layout, (a.k.a. storage model). Since the page layout determines what records and which attributes of a relation are stored in a single page, the storage model employed by a database system has far reaching implications on the performance of a particular workload [3].

The page layout prevalent in commercial database systems, called N-ary storage model (*NSM*), is optimized for queries with full-record access common in an on-line transaction processing (OLTP) workload. *NSM* stores all attributes of a relation in a single page [52] and full records are stored within a page one after another. Accessing a full record is accomplished by accessing a particular record from consecutive memory locations. Using an unwritten rule that access to consecutive logical blocks (LBNs) in the storage device is more efficient than random access, a storage manager maps single page to consecutive LBNs. Thus, an entire page can be accessed by a single I/O request.

An alternative page layout, called the Decomposition Storage Model (*DSM*) [18], is optimized for decision support systems (DSS) workloads. Since DSS queries typically access a small number of attributes and most of the data in the page is not touched in memory by the scan operator, *DSM* stores only one attribute per page. To ensure efficient storage device access, a storage manager maps *DSM* pages with consecutive records containing the same attribute into extents of contiguous LBNs. In anticipation of a sequential scan through records stored in multiple pages, a storage manager can prefetch all pages in one extent with a single large I/O, which is more efficient than accessing each page individually by a separate I/O.

A page layout optimized for CPU cache performance, called *PAX* [4], offers good CPU-memory performance for both individual attribute scans of DSS queries and full-record accesses in OLTP workloads. The *PAX* layout partitions data across into separate minipages. A single minipage contains data of only one attribute and occupies consecutive memory locations. Collectively, a single page contains all attributes for a given set of records. Scanning individual attributes in *PAX* accesses consecutive memory locations and thus can take advantage of cache-line prefetch logic. With proper alignment to cache-line sizes, a single cache miss can effectively prefetch data for several records, amortizing the high latency of memory access compared to cache access. However, *PAX* does not address memory-storage performance.

All of the described storage models share the same characteristics. They (i) are highly optimized for one workload type, (ii) focus predominantly on one level of the memory hierarchy, (iii) use a static data layout that is determined *a priori* when the relation is created, and (iv) apply the same layout across all levels of the memory hierarchy, even though each level has unique (and very different) characteristics. As a consequence, there are inherent performance trade-offs for

Data Organization	Cache–Memory		Memory–Storage	
	OLTP	DSS	OLTP	DSS
NSM	$\surd$	$\times$	$\surd$	$\times$
DSM	$\times$	$\surd$	$\times$	$\surd$
PAX	$\surd$	$\surd$	$\surd$	$\times$

Table 4.1: Summary of performance with current page layouts.

each layout that arise when a workload changes. For example, *NSM* or *PAX* layouts waste memory capacity and storage device bandwidth for DSS workloads, since most data within a page is never touched. Similarly, a *DSM* layout is inefficient for OLTP queries accessing random full records. To reconstruct a full record with n attributes, n pages must be fetched and $n-1$ joins on record identifiers performed to assemble the full record. In addition to wasting memory capacity and storage bandwidth, this access is inefficient at the storage device level; accessing these pages results in random one-page I/Os. In summary, each page layout exhibits good performance for a specific type of access at a specific level of memory hierarchy. as shown in Table 4.1.

Several researchers have proposed solutions to address these performance trade-offs. Ramamurthy et al. proposed fractured mirrors that store data in both *NSM* and *DSM* layouts [53] to eliminate the need to reload and reorganize data when access patterns change. Based on the workload type, a database system can choose the appropriate data organization. Unfortunately, this approach doubles the required storage space and complicates data management; two physically different layouts must be maintained in synchrony to preserve data integrity. Hankins and Patel [31] proposed data morphing as a technique to reorganize data within individual pages based on the needs of workloads that change over time. Since morphing takes place within memory pages that are then stored in that format on the storage device, these fine-grained changes cannot address the trade-offs involved in accessing non-volatile storage. The multi-resolution block storage model (MBSM) [86] groups *DSM* table pages together into superpages, improving *DSM* performance when running decision-support systems.

MEMStores [14] are a promising new type of storage device that has the potential to provide efficient accesses to two-dimensional data. Schlosser et al. proposed data layout for MEMStores that exploits their inherent access parallelism [63]. Yu et al. devised an efficient mapping of database tables to this layout that takes advantage of the unique characteristics of MEMStores [81] to improve query performance.

In summary, these solutions either address only some of the performance trade-offs or are applicable to only one level of the memory hierarchy. *Fates* builds on the previous work and uses a decoupled data layout that can adapt to dynamic changes in workloads without the need to maintain multiple copies of data, to reorganize data layout, or to compromise between memory

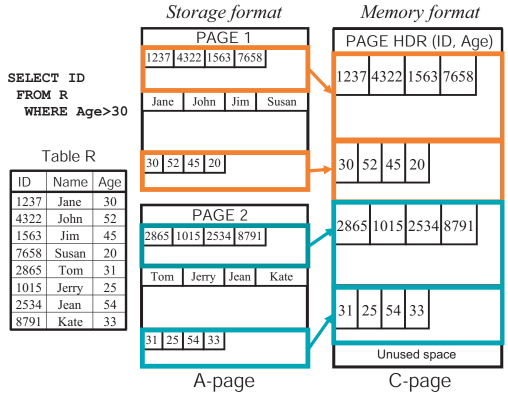


Figure 4.1: Decoupled on-disk and in-memory layouts.

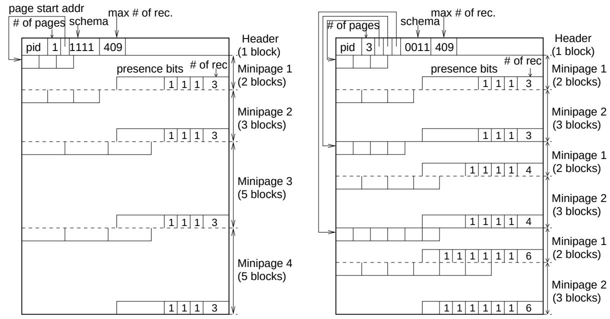
and I/O access efficiency.

4.3 Decoupling data organization

From the discussion in the previous section, it is clear that designing a static scheme for data placement in memory and on non-volatile storage that performs well across different workloads and different device types and technologies is difficult. Instead of accepting the trade-offs inherent to a particular page layout that affects all levels of the memory hierarchy, we propose decoupling the in-memory page layout from the storage organization. This section introduces the high-level picture of the data organization as well as detailed in-memory page layout used in *Fates* through a simple example. It also explains some terminologies used in later section.

4.3.1 An example of data organization in *Fates*

Fates allows for decoupled data layouts and different representations of the same table at the memory and storage levels. Figure 4.1 depicts an example table, R, with three attributes: `ID`, `Name`, and `Age`. At the storage level, the data is organized into A-pages. An A-page contains all attributes of the records; only one A-page needs to be fetched to retrieve a full record. Exploiting the idea used in *PAX* [4], an A-page organizes data into minipages that group values from the same attribute for efficient predicate evaluation, while the rest of the attributes are in the same A-page. The storage and organization of A-pages on storage devices are decided by *Atropos* and transparent to other components. This allows *Atropos* to use optimized methods for placing the contents of the A-page onto the storage medium. Therefore, not only does *Atropos* fully exploit sequential scan for evaluating predicates, but it also places A-pages carefully on the



(a) **Full records.** (b) **Partial records.**

Figure 4.2: C-page layout.

device to ensure near-sequential (or semi-sequential [60]) access when reconstructing a record. The placement of A-pages on the disk is further explained in Section 4.5.

The rightmost part of Figure 4.1 depicts a C-page, which is the in-memory representation of a page. The page frame is sized by the buffer pool manager and is on the order of 8 KB. A C-page is similar to an A-page in that it also contains attribute values grouped in minipages, to maximize processor cache performance. Unlike an A-page, however, a C-page *only* contains values for the attributes the query accesses. Since, the query in the example only uses the `ID` and `Age`, the C-page only includes these two attributes, maximizing memory utilization. Note that the C-page uses data from two A-pages to fill up the space "saved" from omitting `Name`. In the rest of this chapter, we refer to the C-page layout as the *Clotho* storage model (*CSM*). Details of *CSM* are discussed in Section 4.3.2. Note that how C-pages are stored in the main memory is independent of the organization of A-pages.

4.3.2 In-memory C-page layout

Figure 4.2 depicts two examples of C-pages for a table with four attributes of different sizes. In our design, C-pages only contain fixed-size attributes. Variable-size attributes are stored separately in other page layouts (see Section 4.7.2). A C-page contains a page header and a set of minipages, each containing data for one attribute and collectively holding all attributes needed by queries. In a minipage, a single attribute's values are stored in consecutive memory locations to maximize processor cache performance. The current number of records and presence bits are distributed across the minipages. Because the C-page only handles fixed-size attributes, the size of each minipage is determined at the time of table creation.

The page header stores the following information: page id of the first A-page, the number of partial A-pages contained, the starting address of each A-page, a bit vector indicating the schema of the C-page's contents, and the maximal number of records that can fit in an A-page.

Figure 4.2(a) and Figure 4.2(b) depict C-pages with complete and partial records, respectively. The leftmost C-page is created for queries that access full records, whereas the rightmost C-page is customized for queries touching only the first two attributes. The space for minipages 3 and 4 on the left are used to store more partial records from additional A-pages on the right. In this example, a single C-page can hold the requested attributes from three A-pages, increasing memory utilization by a factor of three.

On the right side of the C-page we list the number of storage device blocks each minipage occupies. In our example each block is 512 bytes. Depending on the relative attribute sizes, as we fill up the C-page using data from more A-pages there may be some unused space. Instead of performing costly operations to fill up that space, we choose to leave it unused. Our experiments show that, with the right page size and aggressive prefetching, this unused space does not cause a detectable performance deterioration (details about space utilization are in Section 4.8.7).

4.4 Overview of *Fates* architecture

Fates is a new storage architecture for database systems that enables the decoupled data organization. Thanks to the flexibility of query-specific pages, data processing in *Fates* is able to adapt to the vastly different characteristics at each level of the memory hierarchy. The challenge is to ensure that this decoupling works seamlessly within current database systems. This section describes the key components of the *Fates* architecture.

4.4.1 System architecture

The difficulty in building *Fates* lies in implementing the necessary changes without undue increase in system and code complexity. The three modules in *Fates* confine changes within each component that do not modify the query processing interface.

Figure 4.3 shows the three *Fates* components to highlight the interplay among them and their relationship to database systems as well. Each component can independently take advantage of enabling hardware/OS technologies at each level of the memory hierarchy, while hiding the details from the rest of the system. This section outlines the role of each component. The changes to the components are further explained in Sections 4.6 and 4.5 while details specific to our prototype implementation are provided in Section 4.7. We introduce each component from the top to the bottom.

The operators are essentially predicated scans that access data from in-memory pages stored

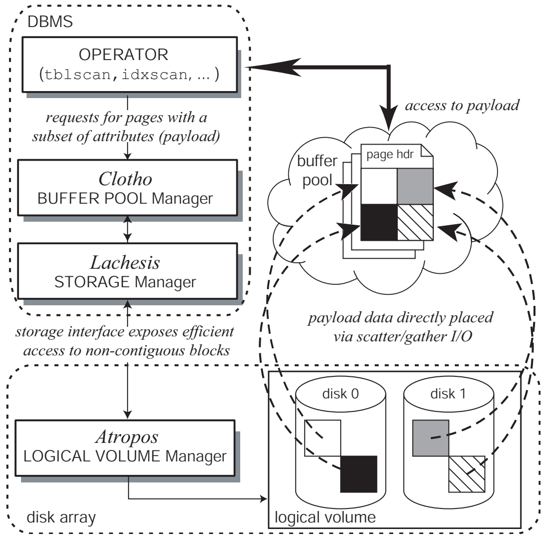


Figure 4.3: Interaction among three components in *Fates*.

in a common buffer pool. They take advantage of the query-specific page layout of C-pages that leverages the L1/L2 CPU cache characteristics and cache prefetch logic for efficient access to data. The scan operators are essentially a part of *Clotho*. They are discussed separately to give a clearer picture of the three components of *Fates*.

Clotho manages C-pages in the buffer pool and enables sharing across different queries that need the same data. In traditional buffer pool managers, a buffer page is assumed to have the same schema and contents as the corresponding relation. In *Clotho*, however, this page may contain a subset of the table schema attributes. To ensure sharing, correctness during updates, and high memory utilization, the *Clotho* buffer pool manager maintains a page-specific schema that denotes which attributes are stored within each buffered page (i.e., the page schema). The challenge of this approach is to ensure minimal I/O by determining sharing and partial overlapping across concurrent queries with minimal book-keeping overhead. Section 4.6 details the buffer pool manager operation in detail.

Lachesis maps A-pages to specific logical volume's logical blocks, called LBNs. Since the A-page format is different from the in-memory layout, the storage manager rearranges A-page data on-the-fly into C-pages using the query-specific *CSM* layout. Unlike traditional storage managers where pages are also the smallest access units, *Lachesis* storage manager selectively retrieves a portion of a single A-page. With scatter/gather I/O and direct memory access (DMA), the pieces of individual A-pages can be delivered directly into the proper memory frame(s) in the buffer pool as they arrive from the logical volume. *Lachesis* simply sets up the appropriate I/O vectors with the destination address ranges for the requested LBNs. The data is placed directly to its destinations without *Lachesis*'s involvement or the need for data shuffling and extraneous memory copies. To efficiently access data for a variety of access patterns, *Lachesis* relies on adjacent blocks provided by the logical volume manager, *Atropos*.

Atropos (Logical Volume Manager, LVM) maps volume LBNs to the physical blocks of the underlying storage device(s). It is independent of the database system and is typically implemented with the storage system (e.g., disk array). The *Atropos* LVM leverages the traditional sequential access path (e.g., all blocks mapped onto one disk or MEMStore track) to store minipages containing the same attributes for efficient scans of a subset of attributes (column direction access). It utilizes the semi-sequential access path consisting of consecutive first adjacent blocks to store a single A-page (row direction access). The *Atropos* LVM is briefly described in Section 4.5 and detailed elsewhere [60, 63]. *Clotho* need not rely on the *Atropos* LVM to create query-specific C-pages. With conventional LVMs, it can map a full A-page to a contiguous run of LBNs with each minipage mapped to one or more discrete LBNs. However, with these conventional LVMs, access only along one dimension will be efficient. Also, the actual storage devices managed by *Atropos* could be conventional magnetic disks or new MEMStore.

4.4.2 Advantages of *Fates* architecture

Fates is a dynamic, robust and autonomous storage architecture for database systems. It has the following advantages over existing approaches.

Leveraging unique device characteristics. At the volatile (main memory) level, *Fates* uses *CSM*, a data layout that maximizes processor cache utilization by minimizing unnecessary accesses to memory. *CSM* organizes data in C-pages and also groups attribute values to ensure that only useful information is brought into the processor caches [4, 31]. At the storage-device level, the granularity of accesses is naturally much coarser. The objective is to maximize memory utilization for all types of queries by only bringing into the buffer pool data that the query needs.

Query-specific memory layout. With memory organization decoupled from storage layout, *Fates* can decide what data is needed by a particular query, request only the needed data from a storage device, and arrange the data on-the-fly to an organization that is best suited for the particular query needs. This fine-grained control over what data is fetched and stored also puts less pressure on buffer pool and storage system resources. By not requesting data that will not be needed, a storage device can devote more time to servicing requests for other queries executing concurrently and hence speed up their execution.

Dynamic adaptation to changing workloads. A storage architecture with flexible data organization does not experience performance degradation when query access patterns change over time. Unlike systems with static page layouts, where the binding of data representation to workload occurs during table creation, this binding is done in *Fates* only during query execution. Thus, a system with decoupled data organizations can easily adapt to changing workloads and also fine-tune the use of available resources when they are under contention.

4.5 *Atropos* logical volume manager

This section briefly describes the storage device-specific data organization and the mechanisms exploited by *Atropos* in creating logical volumes that consist of either disk drives or a single MEMStore.

4.5.1 Atropos disk array LVM

To store a table, a 2-D data structure, on disks, *Atropos* utilizes the semi-sequential access path consisting of the consecutive first adjacent blocks, together with the traditional sequential access path to provide efficient I/O operations along both row direction and column direction. It also exploits automatically-extracted knowledge of disk track boundaries, using them as its stripe unit boundaries for achieving efficient sequential access. *Atropos* exposes these boundaries explicitly to *Clotho* so that it can use previously proposed "track-aligned extent" (*traxtents*), which provide substantial benefits for streaming patterns interleaved with other I/O activity [58, 59]. Finally, as with other logical volume managers, *Atropos* delivers aggregate bandwidth of all disks in the volume and offers the same reliability/performance trade-offs of traditional RAID schemes [49].

Please refer to Chapter 2 for the explanation of adjacent blocks and semi-sequential accesses. This section focuses on how a page is stored on disks.

4.5.2 Efficient database organization with *Atropos*

With *Atropos*, *Lachesis* can lay out A-pages such that access in one dimension of the table is sequential, and access to the other dimension is semi-sequential. Figure 4.4 shows the mapping of a simple table with 12 attributes and 1008 records to A-pages stored on an *Atropos* logical volume with four disks. A single A-page includes 63 records and maps to the diagonal semi-sequential LBNs, with each minipage mapped to a single LBN. When accessing one attribute from all records, *Atropos* can use four track-sized, track-aligned reads. For example, a sequential scan of attribute A1 results in a access of LBN 0 through LBN 15. Accessing a full A-page results in three semi-sequential accesses, one to each disk. For example, fetching attributes A1 through A12 for record 0 results in three semi-sequential accesses, each proceeding in parallel on different disks, starting at LBNs 0, 64, and 128.

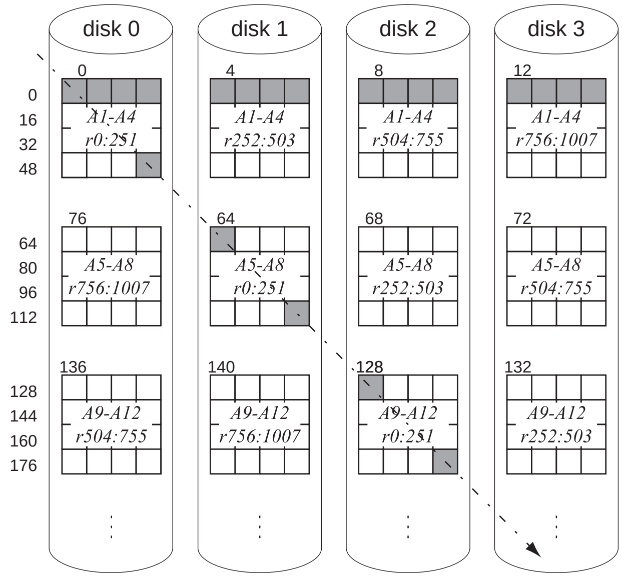


Figure 4.4: Mapping of a database table with 12 attributes onto *Atropos* with 4 disks.The numbers to the left of disk 0 are the LBNs mapped to the gray disk locations connected by the arrow and not the first block of each row. The arrow illustrates efficient semi-sequential access fetching single A-page with 63 records. A single sequential I/O for 16 LBNs can efficiently fetch one attribute from 1008 records striped across four disks.

Atropos hides the details of underlying storage devices through simple interfaces that support more than one access paths. It works harmoniously with other storage devices other than traditional disks, such as MEMStores. Details about *Atropos* and MEMStores can be found in the previous work [60, 63].

4.6 *Clotho* buffer pool manager

Due to the query-specific feature, a C-page can contain a single attribute (similar to *DSM*), a few attributes, or all attributes of a given set of records (similar to *NSM* and *PAX*) depending on query needs. The responsibility of *Clotho*, the new buffer pool manager in *Fates*, is to construct and manage C-pages to ensure both the query-specific feature and data sharing among multiple concurrently running queries. This section first discusses the general design choices for buffer pool managers, followed by the unique challenges facing *Clotho*. After that, we present the design details for Clotho.

4.6.1 Buffer pool manager design space

Before discussing the design space of a buffer pool manager, we first describe the performance metrics used in our analysis. The final performance of a buffer pool manager is evaluated by the average time to serve a request from a scan operator. For a better understanding of how each design aspect affects the performance, we break down the cause of the service time into two components: buffer pool miss rate and management cost.

- Buffer pool miss rate is the number of missed pages (page faults) divided by the total number of requested pages. It indicates how often a buffer pool manager needs to fetch pages from disks. Since fetching a page from disks is two orders of magnitude slower than visiting a page in main memory, buffer pool miss rate has a significant impact on the overall performance: waiting for pages from disks dramatically increases the average service time of a buffer pool manager.
- Management cost is the time spent on metadata maintenance, which implies the complexity of buffer pool management. It includes maintaining the lookup table structure, keep tracking of the status of all frames in memory, etc. Approaches trying to reduce buffer pool miss rate might have negative effects on management cost due to the complicated data structures or algorithms employed by these techniques.

We summarize the design choices for buffer pool managers in database systems as consisting of the following dimensions: in-memory data organization, page lookup algorithms, replacement policies, and metadata management. Among them, in-memory data organization is the most crucial dimension.

In-memory data organization refers to the way in which data are laid out in main memory. In almost all systems, buffer pool managers store data in frames of the same size as disk pages. *NSM*, *DSM*, *PAX*, and C-pages of *CSM* discussed in this chapter are examples of organizing data in pages with different layouts. The choice of in-memory data organization influences the designs of other three dimensions. Performance-wise, it affects both buffer pool miss rate and management cost. An in-depth analysis is presented in the next section.

A page lookup algorithm checks whether a requested page is present in the buffer pool (page hit or miss). It decides how fast a lookup operation can be performed. A hash table with page IDs as search keys is widely adopted by conventional buffer pool managers for its O(1) search complexity. Lookup operations are usually fast and not performance-crucial.

A replacement policy determines the victim pages to be evicted from the buffer pool when there is no enough space for newly requested pages. Based on the access history of pages, different algorithms have been proposed in the literature to exploit the temporal locality in data accesses, thus reducing buffer pool miss rate.

Metadata management includes data structures and algorithms used to keep track of the status of each buffer pool frame. It varies with the choices made for the other design dimensions.

Performance trade-offs exist among the above design dimensions. For instance, optimizing in-memory data organization, such as maximizing space utilization, could complicate management, resulting in more management overhead. For a traditional buffer pool manager that employs a static page layout, the design of in-memory data organization is straightforward. The in-memory page layout simply adopts the same structure as the on-disk layout which is selected a prior based on predicted workload characteristics. The flexibility introduced by *CSM* also brings in complexity and variations for this design dimension. The next section discusses the design choices for in-memory data organization and the new opportunities/challenges for *Clotho*.

4.6.2 Design spectrum of in-memory data organization

In-memory data organization affects buffer pool miss rate through effective memory capacity, which is the amount of memory storing relevant data. By holding more data to be accessed in the future, a buffer pool manager can reduce buffer pool miss rate. Therefore, we choose memory utilization rate as one optimization objective of in-memory data organization. In a multi-query execution environment, data sharing opportunities across different queries present another optimization target.

Memory utilization rate

Memory utilization rate is defined as the percentage of the entire allocated memory space used to buffer relevant data. Ideally, it would be 100%, meaning a buffer pool manager never wastes space to store undesired data. In reality, due to implementation artifact or preference for simple management, there are extraneous data and/or replicate data in the buffer pool. The first refers to the attributes that are not needed by any queries during their lifetime in the buffer pool; the second is the data that have multiple copies in the buffer pool. These two kinds of undesired data waste I/O bandwidth and memory space, increase the page miss rate, and thus impair the performance.

Memory utilization rate is workload-dependent. But generally speaking, buffer pool managers using the *NSM* and *PAX* page layouts suffer from extraneous data due to always fetching all attributes regardless of queries' needs. In contrast, buffer pool managers based on the *DSM* and *CSM* layouts can store only requested data (attributes) while trading some memory space to store replicate copies for faster data processing and less management cost.

The design spectrum of in-memory data organization on memory utilization rate spans from 0% to 100%, with *DSM* at 100% if no replicates and *NSM* and *PAX* at some point in between depending on workloads. Buffer pool managers using *CSM* have a wide range of memory utilization rate. Depending on the strategies of choosing C-page schemas in the presence of multiple queries, the order the queries enter the execution engine, and their relative execution speeds, the memory utilization rate could vary from 100% to less than 50% (in the most pessimistic cases).

We propose three buffer pool management strategies: the first one is simply retrieving full pages for all queries. This method is essentially the same as the traditional buffer pool manager using *NSM*-like page layouts. The second strategy is always fetching query-specific pages. The buffer pool is free of extraneous data but may contain replicate copies of some data, therefore extra structures are needed to maintain data consistency. In the second method, full pages are used for update queries. The last strategy is the one we apply to the *Clotho* prototype, which is discussed in details in the next section.

Data sharing opportunities

The data sharing discussed in this section happens at the granularity of buffer pool frames, i.e., if two queries access the same buffer pool frame, they are sharing this frame no matter whether they access the same set of attributes or the same set of records. Data sharing opportunities among concurrent queries are determined by the relationships of two sets, the attribute set and the record set, of queries and buffer pool frames. At any point of time, sharing a frame is possible only when both sets of the buffer pool frame contain those of the queries. The complexity of data sharing in *Clotho* comes from the fact that concurrent queries do not necessarily access the same set of attributes. Attribute sets of different queries may be disjoint, inclusive, or otherwise overlapping. The observation is also true for record sets.

With the *NSM* and *PAX* page layouts, the attribute sets of buffer pool frames are the same as the relational table schemas stored in them because full pages are used regardless. In the *DSM* case, queries join multiple single-attribute pages to reconstruct records containing requested attributes. In both situations, pages brought into the buffer pool by one query can always be used by others. Therefore data sharing is easy and straightforward. The *CSM* layout with a query-specific attribute set brings up new issues. Should queries make use of the existing *CSM* pages with overlapping attributes? If yes, how? If no, how to enable data sharing among different queries? Generally speaking, data sharing is desirable. The challenge here is to keep a good balance between management cost and memory utilization rate.

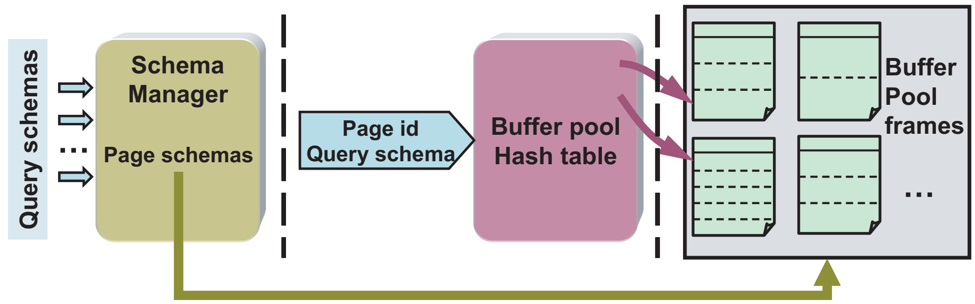


Figure 4.5: Components in *Clotho* buffer pool manager.

4.6.3 Design choices in *Clotho* buffer pool manager

This section describes the design choices made by *Clotho*, namely, the in-memory data organization, the data structure and algorithm for page lookup, the replacement policy, and the metadata management. Figure 4.5 depicts three important components of the *Clotho* buffer pool manager, separated by vertical dotted lines. The leftmost is called *Schema Manager* which keeps updating the latest information of active *page schemas* based on new and retired *query schemas* for each relation. A page schema is a bit vector describing the attributes held by a C-page. Active page schemas are the ones currently in Schema Manager. It is used to decide what attributes should be fetched from disks, indicated by the arrow from Schema Manager to the buffer pool. A query schema is also a bit vector, describing the attributes in a single relation that are referenced by a query. Both schemas are based on individual relations. A single relation may have multiple disjoint page schemas, and a single query may have multiple query schemas, each corresponding to a relation referenced by the query. Schema Manager belongs to the metadata management category and is discussed in details in Section 4.6.3. The middle component is a hash table to look up requested A-pages in the buffer pool. Its responsibility includes the page lookup operation as well as choosing victim pages based the adopted replacement policy. We explain how it works in Section 4.6.3. The rightmost depicts the shared buffer pool with 4 example C-pages.

During query execution, the three components collaborate in the following way: when a new query comes in, Schema Manager checks the query schemas and updates the page schemas for all relations involved. A page lookup searches for the requested page id *and* the corresponding query schema to decide whether it is a hit or miss. If it is a miss, an I/O request for the missing page is issued using the page id and a proper page schema to fetch data from disks.

The rest of this section first reviews the layout of C-pages briefly (the rightmost part), followed by the page lookup operation. The algorithm employed by Schema Manager to enable data sharing among different queries is discussed after that.

In-memory data organization

Clotho adopts C-page (Figure 4.2) as its in-memory data organization format to match various needs from different queries. Details of the C-page layout, such as the metadata in page headers and the arrangement of minipages, can be found in Section 4.3.2. We recapitulate some important characteristics about C-pages that are closely related to the topic of this section.

Each frame in the buffer pool stores a C-page which has a global header followed by one or more partial A-pages. The global header contains, among other metadata, the page schema, the page id of the first partial A-page, and the total number of partial A-pages.

C-pages introduce high flexibility to database systems, with the caveat that data sharing among multiple queries becomes a tricky problem: it is no longer a straightforward page id (record set) match, but a more complex decision involving both page ids and attribute sets. Before delving into the sharing algorithm details, I first describe the data structure and algorithm for page lookup. It is closely related to data sharing: being able to find C-pages containing needed attributes and records is a prerequisite of data sharing among concurrent queries.

Page lookup in *Clotho*

When looking for a page, a traditional database buffer pool manager searches for the corresponding page id in the hash table to determine whether the page is in memory. To support page lookup among query-specific C-pages in *Clotho*, the hash table of *Clotho* contains the page ids of all partial A-pages currently in the buffer pool, augmented by the page schemas of the corresponding C-pages that contain these partial A-pages. When looking for a page in the hash table, *Clotho* still uses a key generated based on the page id, but a hit now has to meet two conditions: first, the page id matches one A-page page id; second, the schema associated with the partial A-page subsumes the query schema. Note that a query schema, not a page schema is used to decide whether there is a match. This has the advantage that even when the page schema of a C-page is not active any more, the pages fetched earlier using this schema can still be utilized by later queries with a smaller attribute set.

Clotho adopts the traditional LRU algorithm to select a victim C-page when no free frames are available. Since the victim C-page may contain multiple partial A-pages, we have to make sure all the partial A-pages in that C-page are removed from the hash table. This is done by looking up the hash table using the A-page ids and the schema of that C-page.

4.6.4 Data sharing in *Clotho*

This section presents the ideas to enable data sharing in *Clotho*. Like all buffer pool managers, the *Clotho* buffer pool manager sets its performance goals for data sharing as follows: (a) maximize sharing, ensuring high memory utilization rate, (b) minimize book-keeping cost to keep buffer pool operations quick, and (c) maintain consistency in the presence of updates.

First, let us examine the options of data sharing in the presence of query-specific pages through an example of two queries with overlapped attribute sets. Assume that query Q1 asks for attributes a_1 and a_2 of a table T and query Q2 asks for attributes a_2 and a_3 of the same table. Using a simple approach, the buffer pool manager could create two separate C-pages tailored to each query. This approach ignores the sharing possibilities when these queries scan the table concurrently and could have shared the data of a_2. To achieve a better memory utilization rate, the buffer pool manager can instead keep track of minipages of a_2 and share them between the two queries. However, this approach incurs too much book-keeping overhead, and is inefficient in practice. It is basically equivalent to the management of *DSM* pages, but at a much smaller granularity.

The *Clotho* buffer pool manager balances memory utilization rate and management complexity. This is achieved by the collaboration between Schema Manager and the page lookup logic. The role of Schema Manager is to make sure that a C-page in the buffer pool can be utilized by as many queries as possible. In this sense, C-pages are more workload-specific than query-specific for a single query. The page lookup logic is to assist each query to find those C-pages, as described above.

For each active table, Schema Manager keeps a list of page schemas for C-pages that belong to the table and are *active*, meaning they are used to fetch data from disks.

Whenever a query starts executing, for each relation referenced by the query, Schema Manager notes the query schema and inspects the other, already active, page schemas. If the new query schema accesses a disjoint set of attributes from the other active queries, Schema Manager adds the new page schema to the list and uses it to fetch new C-pages. Otherwise, it merges the new schema with the most-efficient overlapping one already in memory. The algorithm in Figure 4.6 modifies the page schema list (p_sch), which is initially empty, based on the query schema (q_sch). Once the query is complete, the system removes the corresponding query schema from the list and adjusts the page schema list accordingly using the currently active query schemas.

During query execution the page schema list dynamically adapts to changing workloads depending on the concurrency degree and the overlaps among attribute sets accessed by queries. This list ensures that queries having common attributes can share data in the buffer pool while queries with disjoint attributes will not affect each other. In the above example, Q1 first comes

if read-only query **then**
  if $\exists$p_sch $\supseteq$ q_sch **then**
    Do nothing
  else if q_sch $\cap$ allp_sch $= \varnothing$ **then**
    Add q_sch to the schema list
  else
    New p_sch = $\cup$(q_sch, {p_sch | p_sch $\cap$ q_sch $\neq \varnothing$})
    Add the new p_sch to the list
  end if
else if it is a write query (update/delete/insert) **then**
  Use full schema as the q_sch
  Modify the list: only one full p_sch now
end if

Figure 4.6: Buffer pool manager algorithm.

along, the buffer pool manager creates C-pages with a_1 and a_2. When Q2 arrives, the buffer pool manager will create a C-page with a_1, a_2, and a_3 for these two queries. After Q1 finishes, C-pages with only a_2 and a_3 will be created for Q2.

One characteristic of the schema updating algorithm is that if a query with a large attribute set comes after a query with a smaller attribute set, yet they access the same set of records, the existing C-pages with the smaller schema are of no use to the second query because there will not be a buffer pool hit. In other words, *Clotho* does not fetch only the missing attributes and rearrange the existing partial A-pages to create a new C-page with a larger attribute set. This is opted out due to too much bookkeeping because the buffer pool manager has to keep track of the progress for all queries that are processing this C-page. In addition, fetching only the missed attributes and rearrange them in memory, in practice, does not bring substantially better performance. In our design, the existing queries can keep working on the old copy while the new query starts processing the new C-page with a enlarged page schema. Here, we are trading memory space for simple and quick management. However, the queries processing the old copy will start using the new C-page once they are done with the old C-page.

The memory utilization rate will decrease when the above scenario happens. In an unfavorable case, two or more queries with overlapping attribute sets ask for the same record set, but the queries with larger attribute sets always come after the queries with smaller attribute sets, forcing *Clotho* to fetch a new C-page with an enlarged schema every time a new query comes in. The theoretical worst case would be a query sequence in which the next query asks for one more attribute from the same A-page than the previous one. The length of the sequence can be as large as the number of total attributes in a full A-page. Take a specific example, for a page size of 8 KB and a block size of 512 Bytes, the maximum number of attributes in an A-page is 15 (one block for each attribute and one block for the page header.) We call the 15 attributes a_1 a_2, ... , a_{15}. Assume we have a sequence of 15 queries, Query 1 through Query 15, of which Query i comes after Query (i-1). All of them access the same A-page with a page id P. Query 1 asks for a_1; Query 2 asks for a_1 and a_2; so on and so forth. After Query 15 comes in, the buffer pool will have 15 C-pages, each containing a subset of A-page P. The attribute a_1 is in all C-pages constructed for the 15 queries, thus it has 15 copies; similarly, a_2 has 14 copies, etc. If these C-pages will not be used later, the memory utilization rate is as low as $1/15$. In a more general form, the theoretical lowest memory utilization rate is $1/(number\ of\ attributes\ in\ anA - page)$. Fortunately, the chance of the worst case is very small, if not zero, in real applications. Because a small variation in the assumptions, such as the arriving time of queries, the query schemas, the query execution time, and/or the attribute sizes, will completely change the picture and significantly boost the memory utilization rate. We have never observed the worst case in our experiments except in a carefully and deliberately-set experiment. The above is for the purpose of theoretical analysis.

We also employ the following idea to avoid unfavorable cases as much as possible. When the number of partial A-pages that can be held by a C-page is less than 2, *Clotho* will use the full schema to fetch the entire requested A-page. But Schema Manager does not update the schema list. The benefits are twofold: first, space in C-pages is not left blank, thus wasted; second, this C-page can be used by all queries. The extra cost to get the full A-page is negligible.

Most of the time, database systems benefit from the flexible C-pages, as the evaluation in Section 4.8 shows. The schema management algorithm will not miss any favorable cases, which are quite common in real applications. For instance, queries with disjoint attribute sets or disjoint record sets or disjoint attribute sets and record sets can run on the C-pages tailored to their needs without any duplicate data; queries with smaller attribute sets and record sets that are contained in exiting C-pages can readily make use of them. Even in the unfavorable cases, after the schema list is updated with a full schema, *Clotho* acts like a traditional buffer pool manager with the *NSM* or *PAX* layout.

4.6.5 Maintaining data consistency in *Clotho*

With the algorithm in Figure 4.6, the buffer pool may contain multiple copies of the same mini-page. To ensure data consistency when a transaction modifies a C-page, *Clotho* uses the mechanisms described below to fetch the latest copy of the data to other queries.

As described in Section 4.6.3, a buffer pool hit requires that the requested page id matches one of the A-page page ids, and the requested schema is a subset of the matched A-page's schema. In the case of write queries, i.e., insertions, deletions, and updates, *Clotho* uses full-schema C-pages. There are two reasons for the full-schema design: first, insertions and deletions need full-record access and modify all respective minipages regardless; second, full-schema pages for updates help keep buffer pool data consistent at little additional cost, as explained below.

When a write query is looking up an A-page, it invalidates all of the other buffered A-pages with the same page id and partial schemas. Thus, after the write operation, there is only one A-page containing the valid copy of the modified data and it has the full-schema. According to the page lookup algorithm, queries asking for updated records in the modified A-page automatically obtain the (only) correct dirty page from the buffer pool. Since the A-page with updated data has a full schema, the updated page will serve all other queries asking for records in this page until it is flushed to the disk. *Clotho* does not affect locking policies because page data organization is transparent to the lock manager. Since deletion of records always operates on full C-pages, *CSM* can work with any existing deletion algorithms, such as "pseudo deletion" [41]. Another reason for the full-schema design is that it is more efficient and easier to write out a whole A-page than writing out several C-pages with partial A-pages. The procedures that collect, coalesce, and write out dirty pages need no changes at all.

4.7 Implementation details

Clotho and *Lachesis* are implemented within the Shore database storage manager [13] while *Atropos* is implemented as a separate software library. This section describes the implementation of C-pages, scan operators, and *Atropos*. The implementation does not modify the layout of index pages.

4.7.1 Creating and scanning C-pages

We implemented *CSM* as a new page layout in Shore, according to the format described in Section 4.3.2. The only significant change in the internal Shore page structure is that the page header is aligned to occupy one block (512 B in our experiments). As described in Section 4.6, the original buffer pool manager is augmented with schema management information to control and reuse C-page contents. These modifications were minor and limited to the buffer pool module. To access a set of records, a scan operator issues a request to the buffer pool manager to return a pointer to the the C-page with the (first of the) records requested. This pointer consists of the first A-page id in the C-page plus the page schema id.

If there is no appropriate C-page in the buffer pool to serve the request, the buffer pool manager allocates a new frame for the requested page. It then fills the page header with schema information that allows the storage manager to determine which data (i.e., minipages) is needed. This decision depends on the number of attributes in the query payload and on their relative sizes. Once the storage manager determines from the header information what minipages to request, it constructs an I/O vector with memory locations for individual minipages and issues a batch of I/O requests to fetch them. Upon completion of the individual I/Os, the requested blocks with minipages are "scattered" to their appropriate locations.

We implemented two scan operators: S-scan is similar to a scan operator on *NSM* pages, with the only difference that it only scans the attributes accessed by the query. (in the predicate and in the payload). *Clotho* invokes S-scan to read tuples containing the attributes in the predicate and those in the payload, reads the predicate attributes, and if the condition is true returns the payload. The second scan operator, SI-scan, works similarly to an index scan. SI-scan first fetches and evaluates only the attributes in the predicates, then makes a list of the qualifying record ids, and finally retrieves the projected attribute values directly. Section 4.8.2 evaluates these two operators. To implement the above changes, we wrote about 2000 lines of C++ code.

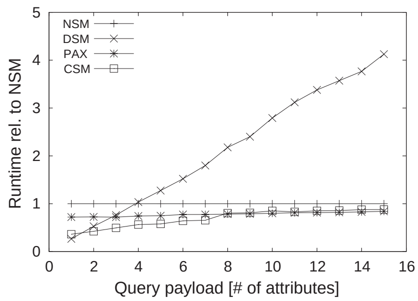
4.7.2 Storing variable-sized attributes

Our current implementation stores fixed-sized and variable-sized attributes in separate A-pages. Fixed-sized attributes are stored in A-pages as described in Section 4.3.1. Each variable-sized attribute is stored in a separate A-page whose format is similar to a *DSM* page. To fetch the full record of a table with variable-sized attributes, the storage manager issues one (batch) I/O to fetch the A-page containing all of the fixed-size attributes and an additional I/O for each variable-sized attribute in the table. As future work, we plan to design storage of variable-sized attributes in the same A-pages with fixed-sized attributes using attribute size estimations [4] and overflow pages whenever needed.

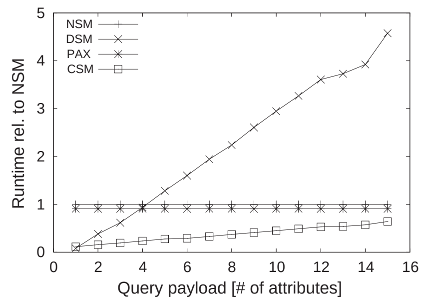
4.7.3 Logical volume manager

The *Atropos* logical volume manager prototype is implemented as a software library which is linked with the client application, in this case Shore. In practice, the functions of *Atropos* would be added to a traditional disk array logical volume manager. In our case, the software LVM determines how I/O requests are broken into individual disk I/Os and issues them directly to the attached SCSI disks using the `/dev/sg` Linux raw SCSI device.

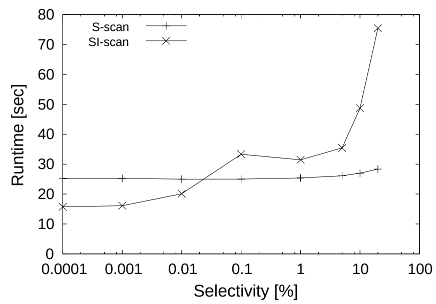
Since real MEMStores do not exist yet, the *Atropos* MEMStore LVM implementation relies on simulation. It uses an existing model of MEMS-based storage devices [61] integrated into the DiskSim storage subsystem simulator [21]. The LVM process runs the I/O timings through



(a) **Atropos disk LVM.**



(b) **MEMStore.**



(c) **Scan operators performance.**

Figure 4.7: Microbenchmark performance for different layouts. The graphs show the total microbenchmark query run time relative to *NSM*. The performance of S-scan and SI-scan is shown for *CSM* layout running on *Atropos* disk array.

DiskSim and uses main memory for storing data.

Disks don't always work the way you want them to. Both of these problems would be solved by adding proper batching semantics to SCSI requests. The disk would only schedule batches of requests once all of the requests in the batch have arrived.

The schedulers in the Cheetah (and maybe in the Atlas10kIII) can't handle not-ascending semi-sequential requests. The solution in our implementation is to only issue two requests to a disk at any one time.

Semi-sequential batches will always incur one half of a rotation of initial latency because the first request delivered always gets scheduled first. The solution here is to use a model of the disk to predict which request should be delivered first based on rotational position after the seek.

Zero-latency access doesn't give all of the benefits you would expect since the data must still be delivered to the host in order.

The real solution to these problems would be to define proper batching semantics in SCSI. There is provision to link requests together, which only says that unlinked requests will be deferred until all of the linked requests have been received.

4.8 Evaluation

This section evaluates the benefits of decoupling in-memory data layout from storage device organization using our *Fates* prototype. The evaluation is presented in two parts. The first part uses representative microbenchmarks [67] to perform a sensitivity analysis by varying several parameters such as the query payload (projectivity) and the selectivity in the predicate. The second part of the section presents experimental results from running DSS and OLTP workloads, demonstrating the efficiency of *Fates* when running these workloads with only one common storage organization. The microbenchmarks include queries with sequential and random access, point updates, and bulk insert operations and evaluate the performance of the worst- and best-case scenarios.

4.8.1 Experimental setup

The experiments are conducted on a two-way 1.7 GHz Pentium 4 Xeon workstation running Linux kernel v. 2.4.24 and RedHat 7.1 distribution. The machine for the disk array experiment has 1024 MB memory and is equipped with two Adaptec Ultra160 Wide SCSI adapters, each controlling two 36 GB Seagate Cheetah 36ES disks (ST336706LC). The *Atropos* LVM exports a single 35 GB logical volume created from the four disks in the experimental setup and maps it to the blocks on the disks' outermost zone.

An identical machine configuration is used for the MEMStore experiments; it has 2 GB of memory, with half used as data store. The emulated MEMStore parameters are based on the G2 MEMStore [61] that includes 6400 probe tips that can simultaneously access 16 LBNs, each of size 512 bytes; the total capacity is 3.46 GB.

All experiments compare *CSM* to the *NSM*, *DSM*, and *PAX* implementations in Shore. *NSM* and *PAX* are implemented as described in [4], whereas *DSM* is implemented in a tight, space-efficient form using the tuple-at-a-time reconstruction algorithm [53]. For *CSM*, the *Atropos* LVM uses its default configuration [60]. The *NSM*, *DSM*, or *PAX* page layouts don't take advantage of the semi-sequential access that *Atropos* provides. However, they still run over the logical volume which is effectively a conventional striped logical volume with the stripe unit size equal to individual disks' track size to ensure efficient sequential access. Unless otherwise stated, the buffer pool size in all experiments is set to 128 MB and page sizes for *NSM*, *PAX* and *DSM* are 8 KB. For *CSM*, both the A-page and C-page sizes are also set to 8 KB. The TPC-H queries used in our experiments (Q1, Q6, Q12, Q14) do not reference variable-sized attributes. TPC-C new-order transaction has one query asking for a variable-size attribute, `C_DATA`, which is stored separately as described in Section 4.7.2.

4.8.2 Microbenchmark performance

To establish *Fates* baseline performance, we first run a range query of the form `SELECT AVG(a1), AVG(a2), ... FROM R WHERE Lo < a2 < Hi`. R has 15 attributes of type `FLOAT`, and is populated with 8 million records (roughly 1 GB of data). All attribute values are uniformly distributed. We show the results of varying the query's payload by increasing the number of attributes in the select clause from one up to the entire record, and the selectivity by changing the values of Lo and Hi. We first run the query using sequential scan, and then using a non-clustered index to simulate random access. The order of the attributes accessed does not affect the performance results, because *Atropos* uses track-aligned extents [58] to fetch each attribute for sequential scans.

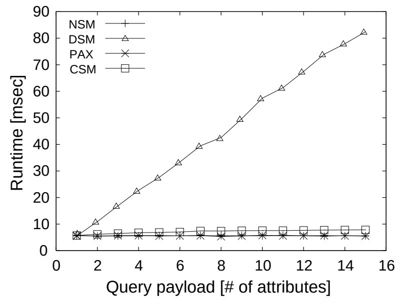
Queries using sequential scan

Varying query payload. Figure 4.7 compares the performance of the microbenchmark query with varying projectivity for the four data layouts. *CSM* uses the S-scan operator. The data are shown for a query with 10% selectivity; using 100% selectivity exhibits the same trends.

Fates shows the best performance at both low and high projectivities. At low projectivity, *CSM* achieves comparable performance to *DSM*, which is the best page layout when accessing a small fraction of the record. The slightly lower running time of *DSM* for the one attribute value in Figure 4.7(a) is caused by a limitation of the Linux operating system that prevents us from using DMA-supported scatter/gather I/O for large transfers[1]. As a result, it must read all data into a contiguous memory region and do an extra memory copy to "scatter" data to their final destinations. *DSM* does not experience this extra memory copy; its pages can be put verbatim to the proper memory frames. Like *DSM*, *CSM* effectively pushes the project to the I/O level. Attributes not involved in the query will not be fetched from the storage, saving I/O bandwidth, memory space, and accelerating query execution.

With increasing projectivity, *CSM* performance is better than or equal to the best case at the other end of the spectrum, i.e., *NSM* and *PAX*, when selecting the full record. *DSM*'s suboptimal performance at high projectivities is due to the additional joins needed between the table fragments spread out across the logical volume. *Clotho*, on the other hand, fetches the requested data in lock-step from the disk and places it in memory using *CSM*, maximizing spatial locality and eliminating the need for a join. *Clotho* performs a full-record scan over 3× faster when compared to *DSM*. As shown in Figure 4.7(b), the MEMStore performance shows the same results.

[1]The size of an I/O vector for scatter/gather I/O in Linux is limited to 16 elements, while commercial UNIX-es support up to 1024 elements.



(a) **Point queries.**

Runtime	
NSM	58 s
DSM	236 s
PAX	63 s
CSM	57 s

(b) **Full scan.**

Figure 4.8: Microbenchmark performance for *Atropos* LVM.

Comparison of S-scan and SI-scan. Figure 4.7(c) compares the performance of the above query for the S-scan and SI-scan operators. We vary selectivity from 0.0001% to 20% and use a payload of four attributes (the trend continues for higher selectivities). As expected, SI-scan exhibits better performance at low selectivities, whereas S-scan wins as the selectivity increases. The performance gain comes from the fact that only pages containing qualified records are processed. The performance deterioration of SI-scan with increasing selectivity is due to two factors. First, SI-scan must process a higher number of pages than S-scan. At selectivity equal to 1.6%, all pages will have qualifying records, because of uniform data distribution. Second, for each qualifying record, SI-scan must first locate the page, then calculate the record address, while S-scan uses a much simpler same-page record locater. The optimizer can use SI-scan or S-scan depending on which one will perform best given the estimated selectivity.

Point queries using random access

The worst-case scenario for *Clotho* data placement schemes is random point tuple access (access to a single record in the relation through a non-clustered index). As only a single record is accessed, sequential scan is never used; on the contrary, as the payload increases *CSM* is penalized more by the semi-sequential scan through the disk to obtain all the attributes in the record. Figure 4.8(a) shows that, when the payload is only a few attributes, *CSM* performs closely to *NSM* and *PAX*. As the payload increases the *CSM* performance becomes slightly worse, although it deteriorates much less that *DSM* performance.

Updates

Bulk updates (i.e., updates to multiple records using sequential scan) exhibit similar performance to queries using sequential scan, when varying either selectivity or payload. Similarly, point updates (i.e., updates to a single record) exhibit comparable performance across all data placement

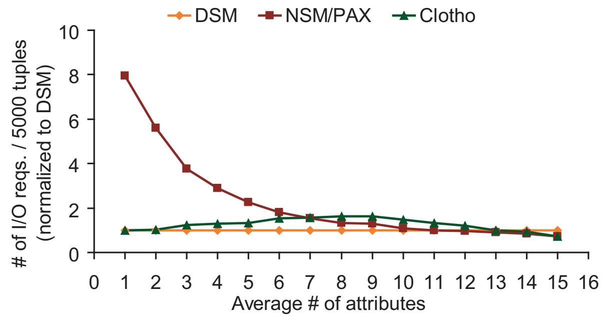


Figure 4.9: Performance of buffer pool managers with different page layouts.This figure compares the average I/O requests per 5000 tuples during the execution of a table-scan query with different page layouts.

methods as point queries. *Clotho* updates single records using full-schema C-pages, therefore its performance is always 22% worse than *NSM*, regardless of payload. To alleviate this behavior, we are currently investigating efficient ways to use partial–record C-pages for updates as we do for queries. As with point queries, the performance of *DSM* deteriorates much faster.

Full table scans and bulk inserts

When scanning the full table (full-record, 100% selectivity) or when populating tables through bulk insertions, *Clotho* exhibits comparable performance to *NSM* and *PAX*, whereas *DSM* performance is much worse, which corroborates previous results [4]. Figure 4.8(b) shows the total running time when scanning table R and accessing full records. The results are similar when doing bulk inserts. Our optimized algorithm issues track-aligned I/O requests and uses aggressive prefetching for all data placement methods. Because bulk loading is an I/O intensive operation, space efficiency is the only factor that will affect the relative bulk-loading performance across different layouts. The experiment is designed so that each layout is as space-efficient as possible (i.e., table occupies the minimum number of pages possible). *CSM* exhibits similar space efficiency and the same performance as *NSM* and *PAX*.

4.8.3 Buffer pool performance

This section evaluates the performance of the buffer pool manager. The first experiment runs the same microbenchmark described in Section 4.8.2 to compare the performance of *NSM*, *DSM*, and *Clotho*. In this experiment, we generate and run a sequence of table-scan queries accessing a random set of attributes with a pre-selected expected set size. The expected set size varies from 1 to 15 which corresponds to queries referencing a single attribute to queries referencing

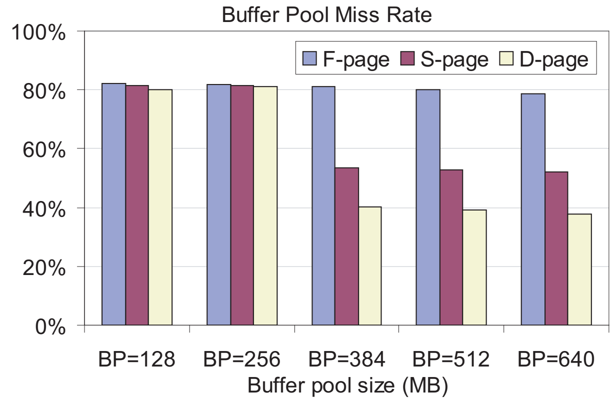


Figure 4.10: Miss rates of different buffer pool managers.This figure shows the miss rates of different buffer pool managers with various buffer pool sizes.

all attributes. We measure the average I/O requests issued per 5000 tuples, which is equivalent to the average buffer pool miss rate. Since *DSM* has the lowest miss rate due to only storing relevant data, we normalize the results of *NSM* and *Clotho* based on the result of *DSM*.

In Figure 4.9, the X axis is the average number of attributes referenced by the query sequence, increasing from 1 to 15. The Y axis is the normalized results of the I/O requests per 5000 tuples, therefore the lower the better. As we expected, *DSM* has the fewest I/O requests when accessing the same amount of tuples, especially when the attribute set size is small. This is because *DSM* only fetches and stores relevant data. By issuing one I/O request of the same size, the buffer pool with *DSM* obtain more relevant data than *NSM* and *PAX*. *Clotho* matches the performance of *DSM* at the leftmost end and both outperform *NSM* and *DSM*. As the attribute set size increases, it is more likely that two queries in *Clotho* have overlapped schemas, which may result in fetching duplicate data from disks. This is the reason for the increasing I/O request number. When the attribute set size continues increasing, the number of I/O requests of all page layouts converge since all fetch full pages from disks.

The second experiment compares the miss rates of the three methods mentioned in the Section 4.6.2. For a better presentation, we denote the first buffer pool as ”F-page” indicating that full pages will always be used. We call the second one as ”S-page” meaning that query- specific pages are used for each query except update queries where full pages will be fetched. The third one is referred to as ”D-page” because it dynamically changes page schemas in the buffer pool when queries come and go.

We run TPC-H 1 GB queries using different buffer pool managers with various buffer pool sizes. Figure 4.10 shows that when buffer pool size is smaller than the working set, all of the three buffer pool managers behave similarly. But with the increase of the buffer pool size, D-page

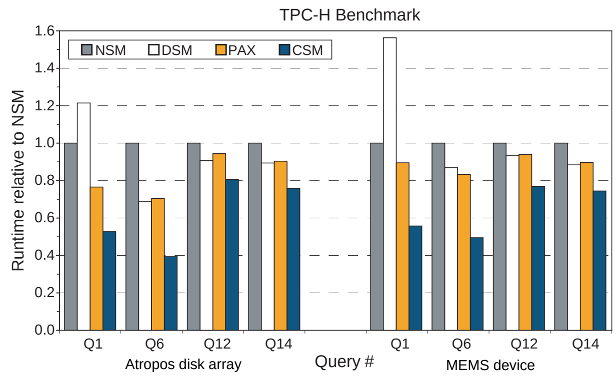


Figure 4.11: TPC-H performance for different layouts.The performance is shown relative to *NSM*.

outperforms the other two due to the fact that it balances the "commonality" and "uniqueness" of queries in a dynamic way. The buffer pool with the "D-page" algorithm uses the precious resource of main memory more efficiently by fetching only necessary information as much as possible.

4.8.4 DSS workload performance

To quantify the benefits of decoupled layout for database workloads, we run the TPC-H decision support benchmark on our Shore prototype. The TPC-H dataset is 1 GB and the buffer pool size is 128 MB.

Figure 4.11 shows execution times relative to *NSM* for four representative TPC-H queries (two sequential scans and two joins). The leftmost group of bars represents TPC-H execution on *Atropos*, whereas the rightmost group represents queries run on a simulated MEMStore. *NSM* and *PAX* perform the worst by a factor of 1.24× – 2.0× (except for *DSM* in Q1) because they must access all attributes. The performance of *DSM* is better for all queries except Q1 because of the benchmark's projectivity. *CSM* performs best because it benefits from projectivity and avoids the cost of the joins that *DSM* must do to reconstruct records. Again, results on MEMStore exhibit the same trends.

4.8.5 OLTP workload performance

The queries in a typical OLTP workload access a small number of records spread across the entire database. In addition, OLTP applications have several insert and delete statements as well as point updates. With *NSM* or *PAX* page layouts, the entire record can be retrieved by a single-page random I/O, because these layouts map a single page to consecutive LBNs. *Clotho* spreads

Layout	*NSM*	*DSM*	*PAX*	*CSM*
TpmC	1115	141	1113	1051

Table 4.2: TPC-C benchmark results with *Atropos* disk array LVM.

a single A-page across non-consecutive LBNs of the logical volume, enabling efficient sequential access when scanning a single attribute across multiple records and less efficient semi-sequential scan when accessing full records.

The TPC-C benchmark approximates an OLTP workload on our Shore prototype with all four data layouts using 8 KB page size. TPC-C is configured with 10 warehouses, 100 users, no think time, and 60 s warm-up time. The buffer pool size if 128 MB, so it only caches 10% of the database. The completed transactions per minute (TpmC) throughput is repeatedly measured over a period of 120 s.

Table 4.2 shows the results of running the TPC-C benchmark. As expected, *NSM* and *PAX* have comparable performance, while *DSM* yields much lower throughput. Despite the less efficient semi-sequential access, *CSM* achieves only 6% lower throughput than *NSM* and *PAX* by taking advantage of the decoupled layouts to construct C-pages that are shared by the queries accessing only partial records. On the other hand, the frequent point updates penalize *CSM*'s performance: the semi-sequential access to retrieve full records. This penalty is in part compensated by the buffer pool manager's ability to create and share pages containing only the needed data.

4.8.6 Compound OLTP/DSS workload

Benchmarks involving compound workloads are important in order to measure the impact on performance when different queries access the same logical volume concurrently. With *Fates*, the performance degradation may be potentially worse than with other page layouts. The originally efficient semi-sequential access to disjoint LBNs (i.e., for OLTP queries) could be disrupted by competing I/Os from the other workload creating inefficient access. This does not occur for other layouts that map the entire page to consecutive LBNs that can be fetched in one media access.

We simulate a compound workload with a single-user DSS (TPC-H) workload running concurrently with a multi-user OLTP workload (TPC-C) against our *Atropos* disk LVM and measure the differences in performance relative to the isolated workloads. The respective TPC workloads are configured as described earlier. In previous work [59], we demonstrated the effectiveness of track-aligned disk accesses on compound workloads; here, we compare all of the page layouts using these efficient I/Os to achieve comparable results for TPC-H.

As shown in Figure 4.12, undue performance degradation does not occur: *CSM* exhibits

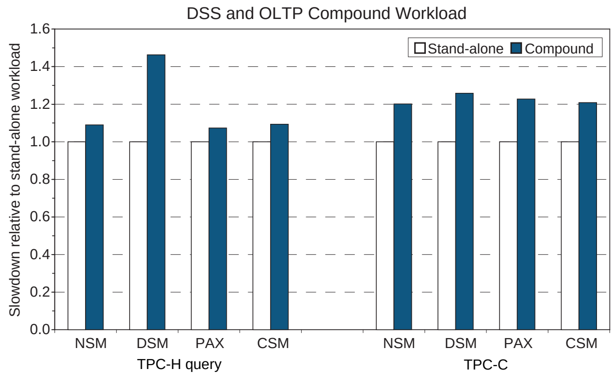


Figure 4.12: Compound workload performance for different layouts.This figure shows the slowdown off TPC-H query 1 running time when it runs with TPC-C benchmark relative to the case when it runs in isolation and the impact on TPC-C performance.

the same or lesser relative performance degradation than the other three layouts. The figure shows indicative performance results for TPC-H query 1 (others exhibit similar behavior) and for TPC-C, relative to the base case when OLTP and DSS queries run separately. The larger performance impact of compound workloads on DSS with *DSM* shows that small random I/O traffic aggravates the impact of seeks necessary to reconstruct a *DSM* page. Comparing *CSM* and *PAX*, the 1% lesser impact of *PAX* on TPC-H query is offset by 2% bigger impact on the TPC-C benchmark performance.

4.8.7 Space utilization

Since the *CSM* A-page partitions attributes into minipages whose minimal size is equal to the size of a single LBN, *CSM* is more susceptible to the negative effects of internal fragmentation than *NSM* or *PAX*. Consequently, a significant amount of space may potentially be wasted, resulting in diminished access efficiency. With *PAX*, minipage boundaries can be aligned on word boundaries (i.e., 32 or 64 bits) to easily accommodate schemas with high variance in attribute sizes. In that case, *Clotho* may use large A-page sizes to accommodate all the attributes without undue loss in access efficiency due to internal fragmentation.

To measure the space efficiency of the *CSM* A-page, we compare the space efficiency of *NSM* and *CSM* layouts for the TPC-C and TPC-H schemas. *NSM* exhibits the best possible efficiency among all four page layouts. Figure 4.13 shows the space efficiency of *CSM* relative to *NSM* for all tables of TPC-C and TPC-H as a function of total page size. Space efficiency is defined as the ratio between the maximum number of records that can be packed into a *CSM* page and the number of records that fit into an *NSM* page.

A 16 KB A-page suffices to achieve over 90% space utilization for all but the customer and

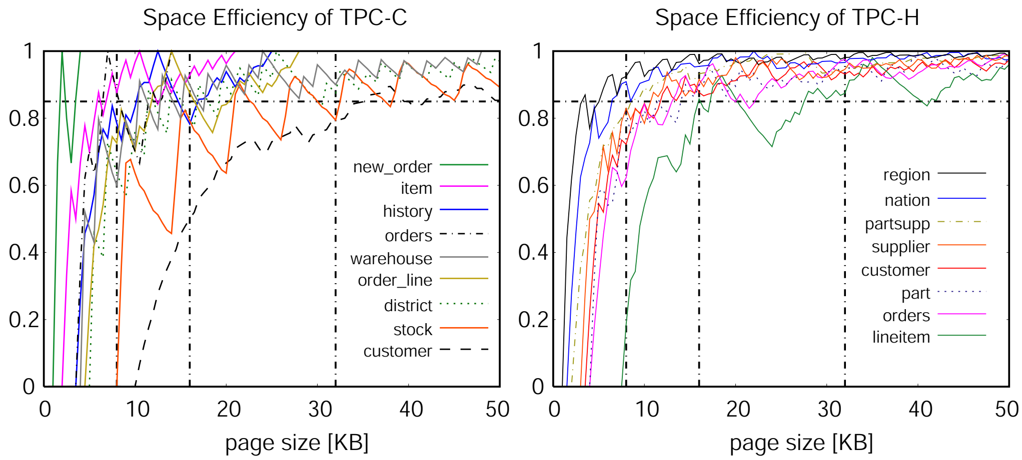


Figure 4.13: Space efficiencies with *CSM* page layout.

stock tables of the TPC-C benchmark. A 32 KB A-page size achieves over 90% space efficiency for the remaining two tables. Both customer and stock tables include an attribute that is much larger than all other attributes. The customer table includes a 500 byte long `C_DATA` attribute containing “miscellaneous information”, while the next largest attribute has a size of 20 bytes. The stock table includes a 50 byte `S_DATA` attribute, while the next largest attribute is 24 bytes. Both of these attributes are rarely used in the TPC-C benchmark.

4.9 Chapter summary

Page layouts for relational tables are among the most fundamental data structures in database systems. How relational tables are organized in memory and on disks affects all aspects of data processing. Due to preferences for simplicity, in-memory page layouts have been chosen to be the same as on-disk page layouts. While this design choice greatly reduces the design and implementation complexity, it imposes inflexibility on page layout designs in a dynamic environment, which results in performance trade-offs for different workloads. Applications have to select one layout a priori that is optimized for predicted prevalent workload characteristics. The reality in database workloads is that the two most common workloads, OLTP and DSS, have exactly conflict preferences for page layouts, namely *NSM* and *DSM*. Therefore, one page layout optimized for one type of workloads penalizes severely the performance of the other type of workloads. For example, OLTP workloads can run 20 times faster on *NSM* than on *DSM*, while DSS workloads can run 5–6 times faster on *DSM* than on *NSM* in our prototype evaluation. This long-standing performance trade-off has been bothering researchers for years.

Fates’s contribution is eliminating this trade-off by decoupling in-memory page layout from on-disk data organization. The rationale of *Fates* is that page layouts at different levels of the storage hierarchy should be designed such that they can exploit distinct features of different storage hardware. *Fates* fetches only relevant data and dynamically assembles query-specific in-memory pages for running queries. The design of *Fates* has its own trade-off between flexibility and implementation simplicity. On the spectrum of flexibility and simplicity, existing approaches using static and uniform page layouts can be viewed at one end with the maximum simplicity whereas approaches using fully dynamic pages in memory are at the other end with the maximum flexibility. *Fates* makes the following design choices to stay balanced on the spectrum. First, *Fates* uses the same page size for both in-memory and on-disk pages; second, only read-only queries are considered using query-specific pages; third, *Fates* allows duplicate data in memory and guarantees data integrity by the algorithm in Figure 4.6.

Another contribution of *Fates* is its modularized architecture. *Fates* consists of three independent but closely interacting components, *Clotho*, *Lachesis*, and *Atropos*. Each component has well-designed interfaces that hide details from other modules while providing necessary information to enhance inter-module interaction. *Fates* greatly improves the interaction between workloads and hardware by passing down the query payloads information to the buffer pool manager and exposing the semi-sequential access paths on modern disks to the storage manager.

Experiments with our *Fates* implementation show that it achieves the best case performance of both *NSM* and *DSM* on OLTP and DSS workloads respectively: *Fates* outperforms *DSM* on TPC-C by a factor of $10\times$ and outperforms *NSM* on TPC-H by a factor of $1.24\times$–$2.0\times$. In addition, for non-best case workloads, *Fates* prevails by as much as 2 times and 8 times for a real disk array and future MEMS-based storage devices.

Last but not least, *Fates*'s achievement can be summarized from another interesting angle which inspires the following work of *MultiMap*. The disk layout of A-pages, the on-disk *CSM* pages, proposes an elegant way to store two-dimensional data structures on disks so that accesses along both dimensions are efficient, namely sequential access and semi-sequential access. A natural following-up question is that can we expand the layout to multidimensional structures. We address this question in the next Chapter.

Chapter 5

MultiMap: Preserving disk locality for multidimensional datasets

Build a multidimensional disk.

This chapter introduces, *MultiMap*, an algorithm for mapping multidimensional datasets so as to preserve the data's spatial locality on disks. Without revealing disk-specific details to applications, MultiMap exploits modern disk characteristics to provide full streaming bandwidth for one (primary) dimension and maximally efficient non-sequential access (i.e., minimal seek and no rotational latency) for the other dimensions. This is in contrast to existing approaches, which either severely penalize non-primary dimensions or fail to provide full streaming bandwidth for any dimension. Experimental evaluation of a prototype implementation demonstrates MultiMap's superior performance for range and beam queries. On average, MultiMap reduces total I/O time by over 50% when compared to traditional linearized layouts and by over 30% when compared to space-filling curve approaches such as Z-ordering and Hilbert curves. For scans of the primary dimension, MultiMap and traditional linearized layouts provide almost two orders of magnitude higher throughput than space-filling curve approaches.

5.1 Introduction

Applications accessing multidimensional datasets are increasingly common in modern database systems. The basic relational model used by conventional database systems organizes information with tables or relations, which are 2-D structures. Spatial databases directly manage multidimensional data for applications such as geographic information systems, medical image databases, multimedia databases, etc. An increasing number of applications that process multidimensional data run on spatial databases, such as scientific computing applications (e.g., earthquake simulation and oil/gas exploration) and business support systems using online analytical processing (*OLAP*) techniques.

Existing mapping algorithms based on the simple linear abstraction of storage devices offered by standard interfaces such as SCSI are insufficient for workloads that access out-of-core multidimensional datasets. To illustrate the problem, consider mapping a relational database table onto the linear address space of a single disk drive or a logical volume consisting of multiple disks. A naive approach requires making a choice between storing the table in row-major or column-major order, trading off access performance along the two dimensions. While accessing the table in the primary order is efficient, with requests to sequential disk blocks, access in the other order is inefficient: accesses at regular strides incur short seeks and variable rotational latencies, resulting in near-random-access performance. Similarly, range queries are inefficient if they extend beyond a single dimension. The problem is more serious for higher dimensional datasets: sequentiality can only be preserved for a single dimension and all other dimensions will be, essentially, scattered across the disk.

The shortcomings of non-sequential disk drive accesses have motivated a healthy body of research on mapping algorithms using space-filling curves, such as Z-ordering [47], Hilbert curves [32], and Gray-coded curves [23]. These approaches traverse the multidimensional dataset and impose a total order on the dataset when storing data on disks. They can help preserve locality for multidimensional datasets, but they do not allow accesses along any one dimension to take advantage of streaming bandwidth, the best performance a disk drive can deliver. This is a high price to pay, since the performance difference between streaming bandwidth and non-sequential accesses is at least two orders of magnitude.

Recent work [62] describes a new generalized model of disks, called adjacency model, for exposing multiple efficient access paths to fetch non-contiguous disk blocks. With this new model, it becomes feasible to create data mapping algorithms that map multiple data dimensions to physical disk access paths so as to optimize access to more than one dimension.

This chapter describes *MultiMap*, a data mapping algorithm that preserves spatial locality of multidimensional datasets by taking advantage of the adjacency model. *MultiMap* maps neighboring blocks in the dataset into specific disk blocks on nearby tracks, called *adjacent blocks*, such that they can be accessed for equal positioning cost *and* without any rotational latency. We describe a general algorithm for *MultiMap* and evaluate *MultiMap* on a prototype implementation that uses a logical volume of real disk drives with 3-D and 4-D datasets. The results show that, on average, *MultiMap* reduces total I/O time by over 50% when compared to the naive mapping and by over 30% when compared to space-filling curve approaches.

The remainder of the chapter is organized as follows. Section 5.2 describes related work.

Section 5.3 introduces *MultiMap* in detail. Section 5.4 provides a analytical cost model for the *Naive* and *MultiMap* mapping algorithms. Section 5.5 evaluates the performance of *MultiMap*. Section 5.6 summarizes the work of *MultiMap*.

5.2 Related work

Organizing multidimensional data for efficient access has become increasingly important in both scientific computing and business support systems, where dataset sizes are terabytes or more. Queries on these datasets involve data accesses on different dimensions with various access patterns [29, 72, 82]. Data storage and management for massive multidimensional data have two primary tasks: data indexing, whose goal is to quickly *locate* the needed data, and data placement, which arranges data on storage devices so that *retrieving* them is fast. There is a large body of previous work on the two closely related but separate topics as they apply to multidimensional datasets. Our work focuses on data placement which happens after indexing.

Under the assumption that disks are one-dimensional devices, various data placement methods have been proposed in the literature, such as the *Naive* mapping, described in the previous section, and spacing-filling curve mappings utilizing Z-ordering [47], Hilbert [32], or Gray-coded curve [23]. The goal is to order multidimensional data such that spatial locality can be preserved as much as possible within the 1-D disk abstraction. With the premise that nearby objects in the multidimensional space are also physically close on disk, they assume that fetching them will not incur inefficient random-like accesses. The property of preserving spatial locality is often called *clustering* [42].

Optimizations on naive mappings [57] such as dividing the original space into multidimensional tiles based on predicted access patterns and storing multiple copies along different dimensions improve performance for pre-determined workloads. However, the performance deteriorates dramatically for workloads with variable access patterns, the same problem as *Naive*.

Recently, researchers have focused on the lower level of the storage system in an attempt to improve performance of multidimensional queries. Part of that work proposes to expand the storage interfaces so that the applications can optimize data placement. Gorbatenko et al. [25] and Schindler et al. [60] proposed a secondary dimension on disks which has been utilized to store 2-D database tables. Multidimensional clustering in DB2 [48] consciously matches the application needs to the disk characteristics to improve the performance of OLAP applications. Others have studied the opportunities of building two dimensional structures to support database applications with new alternative storage devices, such as MEMS-based storage devices [63, 81].

Another body of related research focuses on how to *decluster* multidimensional datasets across multiple disks [1, 7, 10, 24, 50, 51] to optimize spatial access methods [33, 65] and improve throughput.

5.3 Mapping multidimensional data

To map an N-dimensional (N-D) dataset onto disks, we first impose an N-D grid onto the dataset. Each discrete cell in the grid is assigned to an N-D coordinate and mapped to one or more disk blocks. A cell can be thought of as a page or a unit of memory allocation and data transfer, containing one or more points in the original geometric space. For clarity of explanation, we assume that a single cell occupies a single LBN (logical block number) on the disk, whose size is typically 512 bytes. In practice, a single cell can occupy multiple LBNs without any impact on the applicability of our approach.

MultiMap exploits the new adjacency model to map multidimensional datasets to disks. The terminologies of the adjacency model and a brief explanation of them can be found in Chapter 2.

5.3.1 Examples

For simplicity, we first illustrate *MultiMap* through three concrete examples for 2-D, 3-D, and 4-D uniform datasets. The general algorithm for non-uniform datasets are discussed in later sections.

Notation	Definition
T	disk track length (varies by disk zone)
D	number of blocks adjacent to each LBN
N	dimensions of the dataset
Dim_i	notations of the N dimensions
S_i	length of Dim_i
K_i	length of Dim_i in the basic cube
σ	time to access next sequential block
α	time to access any adjacent block

Table 5.1: Notation definitions. For Dim, S, and K, $0 \leq i \leq N-1$.

The notations used in the examples and later discussions are listed in Table 5.1. In the following examples, we assume that the track length is 5 ($T = 5$), each block has 9 adjacent blocks ($D = 9$), and the disk blocks start from LBN 0.

Example of 2-D mapping. Figure 5.1 shows how *MultiMap* maps a (5×3) 2-D rectangle to a disk. The numbers in each cell are its coordinate in the form of (x_0, x_1) and the LBN to which

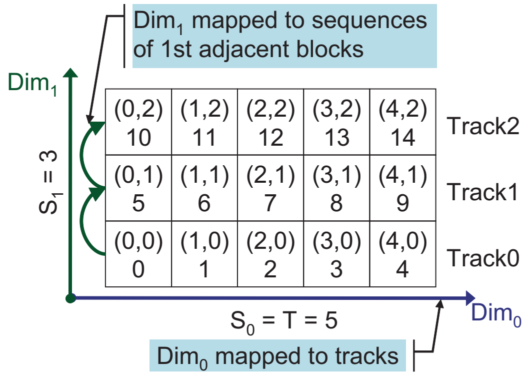


Figure 5.1: Mapping 2-D dataset

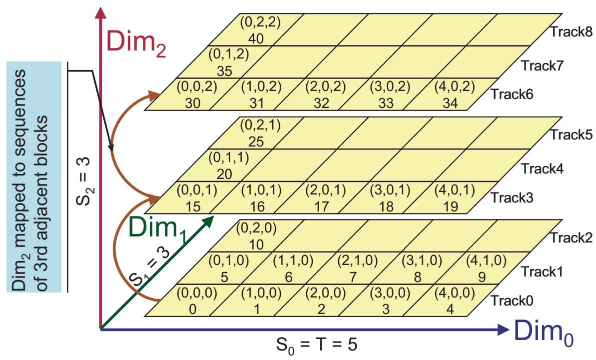


Figure 5.2: Mapping 3-D dataset.

the cell is mapped. Cells along the first dimension (i.e., Dim_0, or the row direction), are mapped sequentially to consecutive LBNs on the same track. For example, the five cells on the bottom row are mapped to LBN 0 through LBN 4 on the same track.

Cells along the second dimension (Dim_1, or the column direction) are mapped to successive first adjacent blocks. Suppose LBN 5 is the first adjacent block of LBN 0 and LBN 10 is the first adjacent block of LBN 5, then the cells of $(0, 1)$ and $(0, 2)$ are mapped to LBN 5 and 10, as shown in Figure 5.1. In this way, spatial locality is preserved for both dimensions: fetching cells on Dim_0 achieves sequential access and retrieving cells on Dim_1 achieves semi-sequential access, which is far more efficient than random access. Notice that once the mapping of the left-most cell $(0, 0)$ is determined, mappings of all other cells can be calculated. The mapping occupies $S_1 = 3$ contiguous tracks.

Example of 3-D mapping. In this example, we use a 3-D dataset of the size $(5 \times 3 \times 3)$. The mapping is iterative, starting with mapping 2-D layers. As shown in Figure 5.2, the lowest 2-D layer is mapped in the same way described above with the cell $(0, 0, 0)$ stored in LBN 0. Then, we use the third adjacent block of LBN 0, which is LBN 15, to store the cell $(0, 0, 1)$. After that,

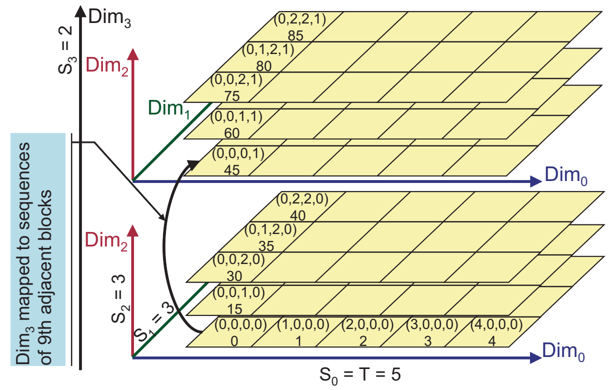


Figure 5.3: Mapping 4-D dataset.

the second 2-D layer can be mapped in the similar way as the 2-D example. Continuing this procedure, we map the cell $(0, 0, 2)$ to the third adjacent block of LBN 15 (LBN 30) and finish the mapping of all cells on the last layer after that.

Since $D = 9$, access along Dim_2 also achieves semi-sequential bandwidth by fetching successive adjacent blocks. Therefore, the spatial locality of Dim_2 is also preserved (the locality of Dim_0 and Dim_1 are guaranteed by the 2-D mapping). Note that the width of each layer (S_1) is restricted by the value of D to guarantee efficient access along Dim_2 as well. We will discuss the case where $S_1 > D$ in the general mapping algorithm. The resulting 3-D mapping occupies $(S_1 \times S_2 = 3 \times 3 = 9)$ contiguous tracks.

Example of 4-D mapping. The 4-D example, shown in Figure 5.3, maps a dataset of the size $(5 \times 3 \times 3 \times 2)$ ($S_0 = T = 5$, $S_1 = 3$, $S_2 = 3$, $S_3 = 2$). Iteratively, we start by mapping the first 3-D cube in the 4-D space using the same approach described in the 3-D example. Then, we use the ninth adjacent block of LBN 0 (LBN 45) to store the cell $(0, 0, 0, 1)$. Once the mapping of $(0, 0, 0, 1)$ is determined, the second 3-D cube can be mapped using the same 3-D mapping approach and so on.

As described, access along Dim_3 also achieves semi-sequential bandwidth, as long as S_1 and S_2 satisfy the restriction: $(S_1 \times S_2) \leq D$.

5.3.2 The *MultiMap* algorithm

As illustrated in the previous section, mapping an N-D space is an iterative extension of the problem of mapping $(N-1)$-D spaces. In addition, the size of the dataset one can map to disks while preserving its locality is restricted by disk parameters. We define a *basic cube* as the largest data cube that can be mapped without losing spatial locality. K_i, the length of Dim_i in the basic cube, must satisfy the following requirements:

```
L := MAP(x[0], x[1], ..., x[N − 1]) :
  lbn := (start_lbn + x[0]) mod T + ⌊start_lbn/T⌋ ∗ T
  step := 1
  i := 1
  repeat
    for j = 0 to x[i] − 1 do
      lbn := GETADJACENT(lbn, step)
    end for
    step := step ∗ K[i]
    i := i + 1
  until (i >= N)
  RETURN(lbn)
K[i]        = K_i
start_lbn   = 1st LBN of basic cube (storing cell (0, ..., 0))
GETADJACENT: get step-th adjacent block of lbn
```

Figure 5.4: Mapping a cell in space to an LBN.

$$K_0 \leq T \tag{5.1}$$

$$K_{N-1} \leq \left\lfloor \frac{\textit{Number of tracks in a zone}}{\prod_{i=1}^{N-2} K_i} \right\rfloor \tag{5.2}$$

$$\prod_{i=1}^{N-2} K_i \leq D \tag{5.3}$$

Equation 5.1 restricts the length of the first dimension of the basic cube to the track length. Note that track length is not a constant value due to zoning on disks, but is found through GETTRACKBOUNDARIES. Equation 5.2 indicates that the last dimension of the basic cube is subject to the total number of tracks in each zone, and zones with the same track length are considered a single zone. Equation 5.3 sets a limit on the lengths of K_1 to K_{N-2}. The volume of the $(N-2)$-D space, $\prod_{i=1}^{N-2} K_i$, must be less than D. Otherwise, the locality of the last dimension cannot be preserved because accessing the consecutive cells along the last dimension cannot be done within the settle time.

The basic cube is mapped as follows: Dim_0 is mapped along each track; Dim_1 is mapped to the sequence of successive first adjacent blocks; ...; $Dim_{i+1} (1 \leq i \leq N-2)$ is mapped to a sequence of successive $(\prod_{i=1}^{i} K_i)$-th adjacent blocks.

The *MultiMap* algorithm, shown in Figure 5.4, generalizes the above procedure. The inputs of MAP are the coordinate of a cell in the basic cube, and the output is the LBN to store that cell. MAP starts from the cell $(0, 0, \ldots, 0)$. Each inner iteration proceeds one step along Dim_i, which on a disk corresponds to a jump over $(K_1 \times K_2 \cdots \times K_{i-1})$ adjacent blocks. Therefore, each iteration of the outer loop goes from cell $(x[0], \ldots, x[i-1], 0, \ldots, 0)$ to cell $(x[0], \ldots, x[i-1], x[i], 0, \ldots, 0)$.

Because of the zoning on disks, the track length decreases from the outer zones to the inner zones. The parameter of T in the algorithm refers to the track length within a single zone. User applications can obtain the track length information from the proposed `GETTRACKBOUNDARIES` call implemented either in the storage controller or in a device driver. A large dataset can be mapped to basic cubes of different sizes in different zones. *MultiMap* does not map basic cubes across zone boundaries.

MultiMap preserves spatial locality in data placement. Dim_0 is mapped to the disk track so that accesses along this dimension achieve the disk's full sequential bandwidth. All the other dimensions are mapped to a sequence of adjacent blocks with different steps. Any two neighboring cells on each dimension are mapped to adjacent blocks at most D tracks away (see Equation 5.3). So, requesting these (non-contiguous) blocks results in semi-sequential accesses.

5.3.3 Maximum number of dimensions supported by a disk

The number of dimensions that can be supported by *MultiMap* is bounded by D and the values of K_i. But, a substantial number of dimensions can usually be supported. One can always map the first dimension, Dim_0, along disk tracks, and map the last dimension, Dim_{N-1}, along successive last (D-th) adjacent blocks. The rest $N-2$ dimensions must fit in D tracks (refer to Equation 5.3). Consider basic cubes with equal length along all dimensions, $K_1 = \cdots = K_{N-2} = K$. Based on Equation 5.3, we get:

$$N \leq \lfloor 2 + \log_K D \rfloor \ (K \geq 2) \tag{5.4}$$

$$N_{max} = \lfloor 2 + \log_2 D \rfloor \tag{5.5}$$

For modern disks, D is typically on the order of hundreds, allowing mapping for more than 10 dimensions. For most physical simulations and OLAP applications, this is more than sufficient.

5.3.4 Mapping large datasets

The basic cube in Section 5.3.2 serves as an allocation unit when we map larger datasets to disks. If the original space is larger than the basic cube, we partition it into basic cubes to get a new

N-D cube with a reduced size of

$$\left(\left\lceil \frac{S_0}{K_0} \right\rceil, \ldots, \left\lceil \frac{S_{N-1}}{K_{N-1}} \right\rceil\right)$$

Under the restrictions of the rules about the basic cube size, a system can choose the best basic cube size based on the dimensions of its datasets. Basically, the larger the basic cube size, the better the performance because the spatial locality of more cells can be preserved. The least flexible size is K_0, because the track length is not a tunable parameter. If the length of the dataset's, and hence basic cube's, S_0 (also K_0) is less than T, we simply pack as many basic cubes next to each other along the track as possible. Naturally, if at all possible, it is desirable to select a dimension whose length is at least T and set it as Dim_0.

In the case where $S_0 = K_0 < T$, *MultiMap* will waste $(T \bmod K_0) * \prod_{i=1}^{N-1} K_i$ blocks per $\lceil T/K_0 \rceil$ basic cubes due to unmapped space at the end of each track. So, the percentage of the wasted space is $(T \bmod K_0)/T$. In the worst case, it can be 50%. Note this only happens to datasets where all dimensions are much shorter than T. If space is a big concern and datasets do not favor *MultiMap*, a system can simply revert to linear mappings. In the case where $S_0 > K_0 = T$, *MultiMap* will only have unfilled basic cubes at the very end. Within a cell, *MultiMap* uses the same format as other mapping algorithms, and therefore it has the same in-cell space efficiency.

When using multiple disks, *MultiMap* can apply existing declustering strategies to distribute the basic cubes of the original dataset across the disks comprising a logical volume just as traditional linear disk models decluster stripe units across multiple disks. The key difference lies in how multidimensional data is organized on a single disk. *MultiMap* thus works nicely with existing declustering methods and can enjoy the increase in throughput brought by parallel I/O operations. In the rest of our discussion, we focus on the performance of *MultiMap* on a single disk, with the understanding that multiple disks will scale I/O throughput by adding disks. The access latency for each disk, however, remains the same regardless of the number of disks.

5.3.5 Mapping non-grid structure datasets

MultiMap can be directly applied to datasets that are partitioned into regular grids, such as the satellite observation data from NASA's Earth Observation System and Data Information System (EOSDIS) [44] and tomographic (e.g., the Visible Human Project for the National Library of Medicine) or other volumetric datasets [22]. When the distribution of a dataset is skewed, a grid-like structure applied on the entire dataset would result in poor space utilization. For such datasets, one should detect uniform subareas in the dataset and apply *MultiMap* locally.

Since the performance improvements of *MultiMap* stem from the spatial locality-preserving mapping *within* a basic cube, non-grid datasets will still benefit from *MultiMap* as long as there exist subareas that can be modeled with grid-like structures and are large enough to fill a basic cube. The problem of mapping skewed datasets thus reduces to identifying such subareas and mapping each of them into one or more basic cubes.

There are several existing algorithms that one can adopt to find those areas, such as density-based clustering methods. In this paper, we use an approach that utilizes index structures to locate the sub-ranges. We start at an area with a uniform distribution, such as a leaf node or an interior node on an index tree. We grow the area by incorporating its neighbors of similar density. The decision of expanding is based on the trade-offs between the space utilization and any performance gains. We can opt for a less uniform area as long as the suboptimal space utilization will not cancel the performance benefit brought by *MultiMap*. As a last resort, if such areas can not be found (e.g, the subarea dimensions do not fit the dimensions of the basic cubes), one can revert to traditional linear mapping techniques.

We demonstrate the effectiveness of this method by mapping a real non-uniform dataset used in earthquake simulations [78] that uses an octree as its index. Experimental results with this dataset are shown in Section 5.5.

5.3.6 Supporting variable-size datasets

MultiMap is an ideal match for the static, large-scale datasets that are commonplace in science. For example, physics or mechanical engineering applications produce their datasets through simulation. After a simulation ends, the output dataset is heavily queried for visualization or analysis purposes, but never updated [46]. As another example, observation-based applications, such as telescope or satellite imaging systems [29], generate large amounts of new data at regular intervals and append the new data to the existing database in a bulk-load fashion. In such applications, *MultiMap* can be used to allocate basic cubes to hold new points while preserving spatial locality.

For applications that need to perform online updates to multidimensional datasets, *MultiMap* can handle updates just like existing linear mapping techniques. To accommodate future insertions, it uses a tunable fill factor of each cell when the initial dataset is loaded. If there is free space in the destination cell, new points will be stored there. Otherwise, an overflow page will be created. Space reclaiming of underflow pages are triggered also by a tunable parameter and done by dataset reorganization, which is an expensive operation for any mapping technique.

5.4 Analytical cost model

To further evaluate the effectiveness of *MultiMap*, we developed an analytical model to estimate the I/O cost for any query against a multidimensional dataset. The model calculates the expected cost in terms of total I/O time for the *Naive* and the *MultiMap* mappings given disk parameters, the dimensions of the dataset and the size of the query. Our model does not predict the total cost for the the space curve mapping. Although it is possible to estimate the number of *clusters*, which are groups of consecutive LBNs for a given query [42], no work has been done on investigating the distribution of distances between clusters, which would be required for an accurate analytical model. Such investigation, as well as the derivation of such a model, is beyond the scope of this work.

In the following discussion, we use the notations defined in Table 5.1 in Section 5.3.1. As with traditional disk I/O cost models, we express the total I/O cost of a query as a sum of two parts: the total positioning time overhead, C_{pos}, and the total data transfer time, C_{xfer}. The positioning time overhead is the time needed to position the disk heads to the correct locations before data can be accessed. The transfer time is the time to read data from the media. The model does not include the cost of transporting the data to the host. This constant overhead is the same for both Naive and *MultiMap* mappings and depends on the interconnect speed and the number of blocks returned, here referred to as the volume of the query.

Suppose we have a range query of the size $(q_0, q_1, \ldots, q_{n-1})$, where q_i is the size of Dim_i, in an N-dimensional cube of size $(S_0, S_1, \ldots, S_{n-1})$. C_{xfer} is calculated by the volume of the query multiplied by the transfer time per disk block, σ. C_{pos} is a function of the number of movements of the disk heads, referred to as *jumps* in the LBN sequence, and their corresponding distances. The total I/O cost of a query is thus expressed as:

$$
\begin{aligned}
C_{cost} &= C_{pos} + C_{xfer} \\
C_{cost} &= \sum_{j=1}^{N_{jmp}} (Seek(d_j) + l_j) + \sigma\ Volume_{query}
\end{aligned}
$$

where N_{jmp} is the number of jumps incurred during the query execution, d_j is the *distance* the disk heads move in the j-th jump, expressed as the number of tracks, and l_j is the rotational latency incurred during each jump after the disk heads finish seeking. The function $Seek(d)$ determines the seek time for a given distance d from a pre-measured seek time profile.

Given the mapping of an N-dimensional cube for both the Naive and the *MultiMap* mappings, the most efficient path to fetch all points is first through Dim_0, then Dim_1, etc. We define *non-consecutive points* for Dim_i as two adjacent points whose coordinates for all but Dim_i are the same and that are mapped to non-contiguous LBNs. Therefore, a jump occurs when we fetch these non-consecutive points and the distance of the jump can be calculated as the difference of the LBNs storing these two non-consecutive points. We build the cost model based on the above analysis. We assume that in addition to a seek, each jump incurs a constant rotational latency equal to half a revolution, $RotLat$, which is the average rotational latency when accessing a randomly chosen pair of LBNs. Finally, we assume some initial cost for positioning disk heads prior to fetching the first block of Dim_0 and denote this overhead C_{init}.

5.4.1 Analytical cost model for Naive mapping

The following equations calculate the query I/O cost for the Naive model.

$$C_{xfer} = \sigma \prod_{i=0}^{n-1} q_i \tag{5.6}$$

$$C_{pos} = C_{init} + \sum_{i=1}^{n-1} \left[(Seek(d_i) + RotLat)\ (q_i - 1) \prod_{j=i+1}^{n-1} q_j \right] \tag{5.7}$$

$$d_i = \left\lceil \frac{\prod_{j=0}^{i-1} S_j - q_0 + 1 - \sum_{j=1}^{i-1} \left((q_j - 1) \prod_{k=0}^{j-1} S_k \right)}{T} \right\rceil \tag{5.8}$$

Equation 5.6 calculates the total transfer time; the per-block transfer time, σ, is multiplied by the query volume, which is the product of all cells returned by the query. Equation 5.7 calculates the positioning overhead. The term $(q_i - 1) \prod_{j=i+1}^{n-1} q_j$ calculates the number of jumps along Dim_i, where $i > 0$. Equation 5.8 computes the distance of such jumps along Dim_i. This distance is the LBN difference of the two points $(x_0, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_{n-1})$ and $(x_0 + q_0, \ldots, x_{i-1} + q_{i-1}, x_i + 1, x_{i+1}, \ldots, x_{n-1})$ divided by the track length, T.

5.4.2 Analytical cost model for *MultiMap* mapping

The following equations calculate the query I/O cost for the *MultiMap* model.

$$
\begin{aligned}
C_{xfer} &= \sigma \prod_{i=0}^{n-1} q_i & (5.9)\\
C_{pos} &= C_{init} + (Seek(d_0) + RotLat)\ N_{jmp}(q_0, K_0, S_0) + \\
\sum_{i=1}^{n-1} & \left[(\ \alpha\,(q_i - N_{jmp}(q_i, K_i, S_i)) + (Seek(d_i) + RotLat)\,N_{jmp}(q_i, K_i, S_i)) \prod_{j=i+1}^{n-1} q_j \right] & (5.10)\\
d_i &= \left\lceil \frac{\prod_{i=0}^{n-1} K_i}{T} \right\rceil \prod_{j=0}^{i-1} \left\lceil \frac{S_j}{K_j} \right\rceil & (5.11)\\
N_{jmp}(q_i, K_i, S_i) &= \left(\left\lceil \frac{q_i}{K_i} \right\rceil - 1 \right) \frac{Loc_S(q_i, K_i, S_i)}{S_i - q_i + 1} + \left\lceil \frac{q_i}{K_i} \right\rceil \left(1 - \frac{Loc_S(q_i, K_i, S_i)}{S_i - q_i + 1} \right) & (5.12)\\
Loc_S(q_i, K_i, S_i) &= Loc_K(q_i, K_i) \left(\left\lfloor \frac{S_i}{K_i} \right\rfloor - \left\lfloor \frac{q_i}{K_i} \right\rfloor \right) + \left\lfloor \frac{S_i \bmod K_i}{Loc_K(q_i, K_i)} \right\rfloor & (5.13)\\
Loc_K(q_i, K_i) &= K_i - (q_i \bmod (K_i + 1)) + 1 & (5.14)
\end{aligned}
$$

The transfer cost (Equation 5.9) is the same as the transfer cost of the *Naive* model. Equation 5.10 calculates the positioning cost and consists of three terms. The first term, C_{init}, is the initial positioning cost, the second term is the positioning cost when reading along Dim_0, and the third term is the positioning cost when reading along Dim_i, where $i > 0$. Since ***MultiMap*** partitions the N-dimensional cube into smaller basic cubes, the sequential accesses along Dim_0 are broken when moving from one basic cube to the next. Thus, the term $(Seek(d_0) + RotLat)\ N_{jmp}(q_0, K_0, S_0)$ accounts for the overhead of seek and rotational latency multiplied by the expected number of such jumps (determined by Equation 5.12).

The third term of Equation 5.10 is similar to the second term in Equation 5.7 for the Naive model, but includes two additional expressions for calculating the cost along Dim_i. The expression $\alpha\ (q_i - N_{jmp}(q_i, K_i, S_i))$ accounts for the cost of semi-sequential accesses to adjacent blocks when retrieving the points along the i-th dimension. The expression $(Seek(d_i) + RotLat)\ N_{jmp}(q_i, K_i, S_i)$ accounts for the cost of jumps from one basic cube to the next.

Equation 5.11 calculates the distance of each jump on Dim_i. This distance depends on the volume of each basic cube, $\prod_{i=0}^{n-1} K_i$, which determines the number of LBNs needed when mapping one basic cube. The term $\prod_{j=0}^{i-1} \left\lceil \frac{S_j}{K_j} \right\rceil$ determines how many basic cubes are mapped between two basic cubes containing two successive points $(x_0, \ldots, x_i, \ldots, x_{n-1})$ and $(x_0, \ldots, x_i + K_i, \ldots, x_{n-1})$.

Given query size q_i, basic cube size K_i, and original space size S_i, $N_{jump}(q_i, K_i, S_i)$ (Equation 5.12) calculates the expected number of jumps across basic cubes along that dimension.

With query size of q_i, there are $S_i - q_i + 1$ possible starting locations in the original space. Loc_S of the possible starting locations will cause $\left\lceil \frac{q_i}{K_i} \right\rceil - 1$ jumps along Dim_i while the remaining locations will cause one more jump. To calculate the expected number of jumps, we simply add the probabilities for each possible type of starting locations multiplied by the number of such jumps. The probability of the minimal number of jumps is the ratio of $Loc_S(q_i, K_i, S_i)$ and $S_i - q_i + 1$. Therefore, the probability of $\left\lceil \frac{q_i}{K_i} \right\rceil$ jumps is $1 - \frac{Loc_S(q_i,K_i,S_i)}{S_i - q_i + 1}$.

To determine Loc_S (Equation 5.13), we first count the locations within a basic cube that will cause the minimal number of jumps, denoted as Loc_K (Equation 5.14), and multiply it by the total number of complete basic cubes that the possible starting locations can span, given by the expression $\left\lfloor \frac{S_i}{K_i} \right\rfloor - \left\lfloor \frac{q_i}{K_i} \right\rfloor$. Finally, we add the number of locations causing the minimal number of jumps in the last (possibly incompletely mapped) basic cube, which is $\left\lfloor \frac{S_i \bmod K_i}{Loc_K(q_i, K_i)} \right\rfloor$.

5.5 Evaluation

We evaluate *MultiMap*'s performance using a prototype implementation that runs queries against multidimensional datasets stored on a logical volume comprised of real disks. The three datasets used in our experiments are a synthetic uniform 3-D grid dataset, a real non-uniform 3-D earthquake simulation dataset with an octree index, and a 4-D OLAP data cube derived from TPC-H. For all experiments, we compare *MultiMap* to three linear mapping algorithms: *Naive*, *Z-order*, and *Hilbert*. *Naive* linearizes an N-D space along Dim_0. *Z-order* and *Hilbert* order the N-D cells according to their curve values.

5.5.1 Experimental setup

We use a two-way 1.7 GHz Pentium 4 Xeon workstation running Linux kernel 2.4.24 with 1024 MB of main memory and one Adaptec Ultra160 SCSI adapter connecting two 36.7 GB disks: a Seagate Cheetah 36ES and a Maxtor Atlas 10k III. Our prototype system consists of a logical volume manager (LVM) and a database storage manager. The LVM exports a single logical volume mapped across multiple disks and identifies adjacent blocks [62]. The database storage manager maps multidimensional datasets by utilizing high-level functions expected by the LVM.

The experiment datasets are stored on multiple disks.The LVM generates requests to all the disks during our experiments, but we report performance results from a single disk. This approach keeps the focus on average I/O response times, which depend only on the characteristics of a single disk drive. Using multiple drives improves the overall throughput of our experiments, but does not affect the relative performance of the mappings we are comparing.

We run two classes of queries in the experiments. *Beam queries* are 1-D queries retrieving data cells along lines parallel to the dimensions. Queries on the earthquake dataset examining velocity changes for a specific point over a period of time are examples of beam query in real applications. *Range queries* fetch an N-D equal-length cube with a selectivity of $p\%$. The borders of range queries are generated randomly across the entire domain.

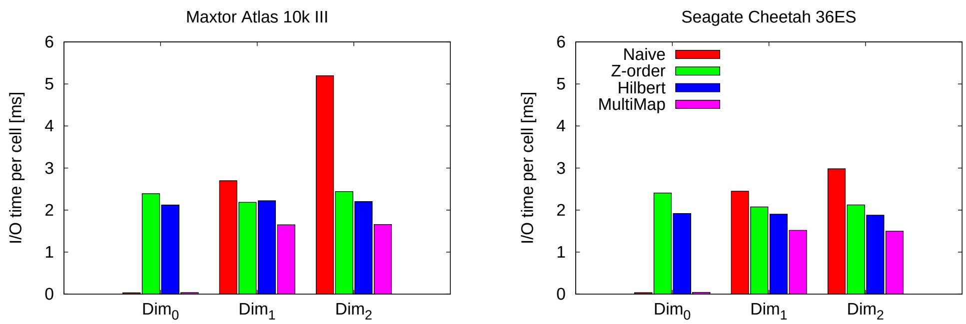
5.5.2 Implementation

Our implementation of the *Hilbert* and *Z-order* mappings first orders points in the N-D space, according to the corresponding space-filling curves. These points are then packed into cells with a fill factor of 1 (100%). Cells are stored sequentially on disks with each occupying one or more disk blocks, depending on the cell size. As we are only concerned with the cost of retrieving data from the disks, we assume that some other method (e.g. an index) has already identified all data cells to be fetched. We only measure the I/O time needed to transfer the desired data.

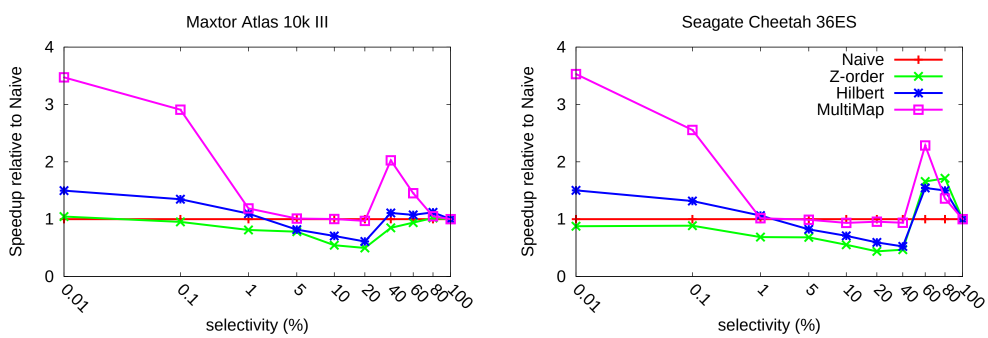
For *Hilbert* and *Z-order* mappings, the storage manager issues I/O requests for disk blocks in the order that is optimal for each technique. After identifying the LBNs containing the desired data, the storage manager sorts those requests in ascending LBN order to maximize disk performance. While the disk's internal scheduler should be able to perform this sorting itself (if all of the requests are issued together), it is an easy optimization for the storage manager that significantly improves performance in practice.

When executing beam queries, *MultiMap* utilizes sequential (Dim_0) or semi-sequential (other dimensions) accesses. The storage manager identifies those LBNs that contain the data and issues them directly to the disk. No sorting is required. For instance, in Figure 5.1, if a beam query asks for the first column (LBN 0, 5, and 10), the storage manager generates an I/O request for each block and issues them all at once. The disk's internal scheduler will ensure that they are fetched in the most efficient way, i.e., along the semi-sequential path.

When executing a range query using *MultiMap*, the storage manager will favor sequential access over semi-sequential access. Therefore, it will fetch blocks first along Dim_0, then Dim_1, and so on. Looking at Figure 5.1 again, if the range query is for the first two columns of the dataset (0, 1, 5, 6, 10, and 11), the storage manager will issue three sequential accesses along Dim_0 to fetch them. That is, three I/O requests for (0, 1), (5, 6), and (10, 11). Favoring sequential over semi-sequential access for range queries provides better performance as sequential access is still significantly faster than semi-sequential access. In our implementation, each cell is mapped to a single disk block of 512 Byte.



(a) **Beam queries.**



(b) **Range queries.**

Figure 5.5: Performance of queries on the synthetic 3-D dataset.

5.5.3 Synthetic 3-D dataset

For these experiments, we use a uniform synthetic dataset with $1024 \times 1024 \times 1024$ cells. We partition the space into chunks of at most $259 \times 259 \times 259$ cells that fit on a single disk and map each chunk to a different disk of the logical volume. For both disks in our experiments, *MultiMap* uses $D = 128$.

Beam queries. The results for beam queries along Dim_0, Dim_1, and Dim_2 are presented in Figure 5.5(a). The graphs show the average I/O time per cell (disk block). The values are averages over 15 runs, and the standard deviation is less than 1% of the reported times. Each run selects a random value between 0 and 258 for the two fixed dimensions and fetches all cells (0 to 258) along the remaining dimension.

As expected, *Naive* performs best along Dim_0, the major order, as it utilizes efficient sequential disk accesses with average time of 0.035 ms per cell. However, accesses along the non-major orders take much longer, since neighboring cells along Dim_1 and Dim_2 are stored 259 and 67081 (259×259) blocks apart, respectively. Fetching each cell along Dim_1 experiences mostly just rotational latency; two consecutive blocks are often on the same track. Fetching cells along Dim_2 results in a short seek (1.3 ms for each disk) followed by rotational latency.

True to their goals, *Z-order* and *Hilbert* achieve balanced performance across all dimensions. They sacrifice the performance of sequential accesses that *Naive* can achieve for Dim_0, resulting in 2.4 ms per cell in *Z-order* mapping and 2.0 ms per cell in *Hilbert*, versus 0.035 ms for *Naive* (almost 57× worse). *Z-order* and *Hilbert* outperform *Naive* for the other two dimensions, achieving 22%–136% better performance for each disk. *Hilbert* shows better performance than *Z-order*, which agrees with the theory that *Hilbert* curve has better clustering properties [42].

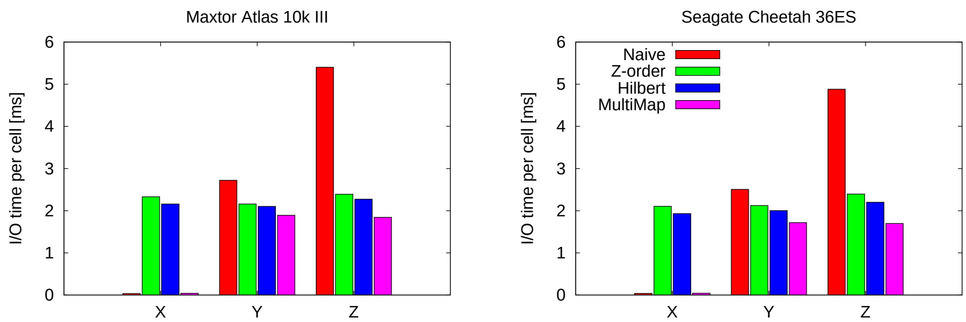
MultiMap delivers the best performance for all dimensions. It matches the streaming performance of *Naive* along Dim_0 despite paying a small penalty when jumping from one basic cube to the next one. As expected, *MultiMap* outperforms *Z-order* and *Hilbert* for Dim_1 and Dim_2 by 25%—35% and *Naive* by 62%–214% for each disk. Finally, *MultiMap* achieves almost identical performance on both disks, unlike the other techniques, because these disks have comparable settle times, and thus the performance of accessing adjacent blocks along Dim_1 and Dim_2.

Range queries. Figure 5.5(b) shows the speedups of each mapping technique relative to *Naive* as a function of selectivity (from 0.01% to 100%). The X axis uses a logarithmic scale. As before, the performance of each mapping follows the trends observed for the beam queries. *MultiMap* outperforms other mappings, achieving a maximum speedup of $3.46\times$, while *Z-order* and *Hilbert* mappings observe a maximum speedup of $1.54\times$ and $1.11\times$, respectively.

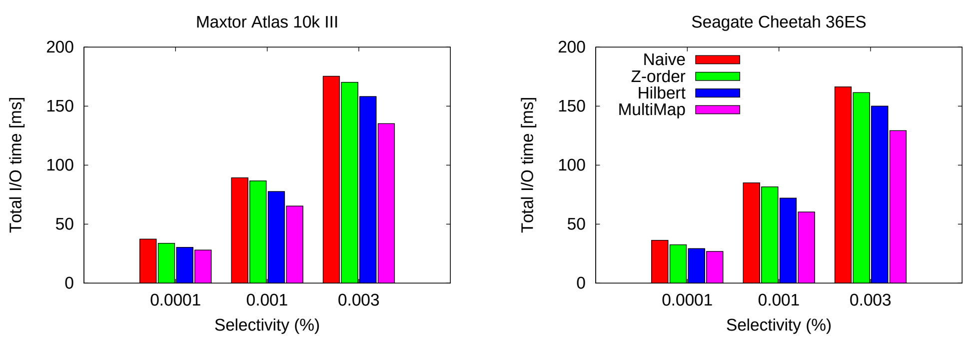
Given our dataset size and the range of selectivities from 0.01% to 100%, these queries fetch between 900 KB and 8.5 GB data from a single disk. The performance of range queries are determined by two factors: the closeness of the required blocks (the clustering property of the mapping algorithm) and the degree of sequentiality in these blocks. In the low selectivity range, the amount of data fetched is small and there are few sequential accesses. Therefore, *Hilbert* (up to 1%) and *Z-order* (up to 0.1%) outform *Naive* due to their better clustering property. As the value of selectivity increases, *Naive* has relatively more sequential accesses. Thus, its overall performance improves, resulting in lower speedups of other mappings. This trend continues until the selectivity hits a point (around 40% in our experiment) where all mappings have comparable sequential accesses but different degrees of clustering. In this case, *Hilbert* and *Z-order* again outperform *Naive*. As we keep increasing the value of selectivity to fetch nearly the entire dataset, the performance of all mapping techniques converge, because they all retrieve the cells sequentially. The exact turning points depend on the track length and the dataset size. Most importantly, *MultiMap* always performs the best.

5.5.4 3-D earthquake simulation dataset

The earthquake dataset models earthquake activity in a 14 km deep slice of earth of a 38×38 km area in the vicinity of Los Angeles [77]. We use this dataset as an example of how to apply



(a) **Beam queries.**



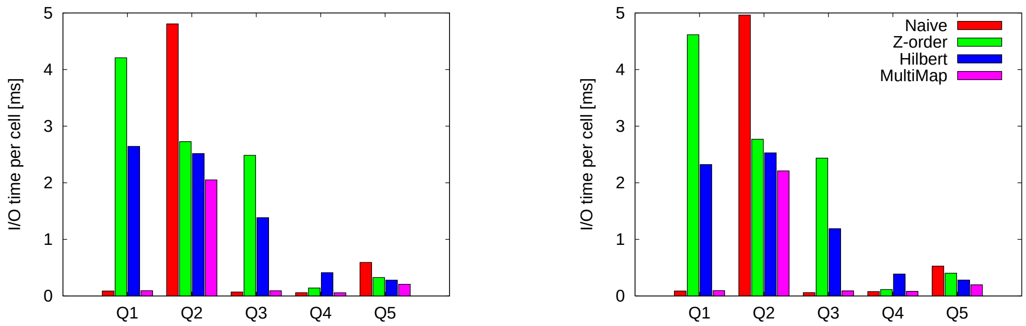
(b) **Range queries.**

Figure 5.6: Performance of queries on the 3-D earthquake dataset.

MultiMap to skewed datasets. The points in the 3-D dataset, called nodes, have variable densities and are packed into elements such that the 64 GB dataset is translated into a 3-D space with 113,988,717 elements indexed by an octree. Each element is a leaf node in the octree.

In our experiments, we use an octree to locate the leaf nodes that contain the requested points. *Naive* uses X as the major order to store the leaf nodes on disks whereas *Z-order* and *Hilbert* order the leaf nodes according to the space-filling curve values. For *MultiMap*, we first utilize the octree to find the largest sub-trees on which all the leaf nodes are at the same level, i.e., the distribution is uniform on these sub-trees. After identifying these uniform areas, we start expanding them by integrating the neighboring elements that are of the similar density. With the octree structure, we just need to compare the levels of the elements. The earthquake dataset has roughly four uniform subareas. Two of them account for more than 60% elements of the total datasets. We then apply *MultiMap* on these subareas separately.

The results, presented in Figure 5.6, exhibit the same trends as the previous experiments. *MultiMap* again achieves the best performance for all beam and range queries. It is the only mapping technique that achieves streaming performance for one dimension without compromising the performance of spatial accesses in other dimensions. For range queries, we select



(a) **Maxtor Atlas 10k III.** (b) **Seagate Cheetah 36ES.**

Figure 5.7: Performance of queries on the 4-D OLAP dataset.

representative selectivities for the applications.

5.5.5 4-D OLAP dataset

In this section, we run experiments on an OLAP cube derived from the TPC-H tables as follows:

```
CREATE TABLE Sales(
  int OrderDay, int ProductType,
  int NationID, int Quantity,
  <other information>)
```

This table schema is similar to the one used in the IBM's Multi-Dimensional Clustering paper [48]. We choose the first four attributes as the four dimensions of the space and form an OLAP cube of size (2361, 150, 25, 50) according to the unique values of these attributes. Since each unique combination of the four dimensions does not have enough points to fill a cell or disk block, we roll up along `OrderDay` to increase the number of points per combination, i.e., combine two cells into one cell along `OrderDay`. This leads to a cube of size (1182, 150, 25, 50) for a 100 GB TPC-H dataset. Each cell in the cube corresponds to the sales of a specific order size for a specific product sold to a specific country within 2 days.

The original cube is partitioned into chunks to fit on each disk, whose dimensions are (591, 75, 25, 25). The value of D is the same as the 3-D experiments, and the results are presented in Figure 5.7. For easy comparison across queries, we report the average I/O time per cell. The details of OLAP queries are as follows:

Q1: "*How much profit is made on product P with a quantity of Q to country C over all dates?*"

Q2: "*How much profit is made on product P with a quantity of Q ordered on a specific date over all countries?*"

Q1 and Q2 are beam queries on the major order (OrderDay) and a non-major dimension (NationID), respectively. As expected, *Naive* outperforms *Hilbert* and *Z-order* by two orders of magnitude for Q1, while *Z-order* and *Hilbert* are almost twice as fast as *Naive* for Q2. *MultiMap* achieves the best performance for both.

Q3: "*How much profit is made on product P of all quantities to country C in one year?*" The 2-D range query Q3 accesses the major order (OrderDay) and one non-major order (Quantity), so *Naive* can take advantage of sequential access to fetch all requested blocks along the major dimension then move to the next line on the surface. Hence, *Naive* outperforms *Z-order* and *Hilbert*. *MultiMap* matches *Naive*'s best performance, achieving the same sequential access on the major order.

Q4: "*How much profit is made on product P over all countries, quantities in one year?*" Q4 is a 3-D range query. Because it also involves the major order dimension, *Naive* shows better performance than the space-filling curve mappings by at least one order of magnitude. *MultiMap* slightly outperforms *Naive*because it also preserves locality along other dimensions.

Q5: "*How much profit is made on 10 products with 10 quantities over 10 countries within 20 days?*" Q5 is a 4-D range query. As expected, both *Z-order* and *Hilbert* demonstrate better performance than *Naive*. *MultiMap* performs the best. For the two different disks, it achieves 166%–187% better performance than *Naive*, 58%–103% better performance than *Z-order* and 36%–42% better performance than *Hilbert*.

5.5.6 Analytical cost model and higher dimensional datasets

For the results presented in this section, we used parameters that correspond to the Atlas 10 k III disk. We determined the values empirically from measurements of our disks. The average rotational latency $RotLat = 3$ ms, $C_{init} = 8.3$ ms, which is the average rotational latency plus the average seek time, defined as the third of the total cylinder distance. Based on our measurements, we set $\sigma = 0.015$ ms and $\alpha = 1.5$ ms with the conservatism of 30° (Table 2.1 in Section 2.2). We set $T = 686$, which is equal to the number of sectors per track in the outer-most zone. To evaluate the model's accuracy, we first compare its predictions with our measurements from the experiments on 3-D and 4-D data sets and summarize these findings in Table 5.2. As shown, the prediction error our model is for most cases less than 8% and 5 out of 24 predicted values are within 20%. The model estimates the cost based on the cells-to-LBNs mappings, which are accurate no matter what the dimensions are. Therefore, the prediction error is independent of the dimensions.

We can use the model to predict the performance trends for N-D space with $N > 4$. Figure 5.8 shows the predicted performance for beam queries with an 8-D dataset with each dimension being 12 cells. These results exhibit the same trends as those for 3-D and 4-D, demonstrating that *MultiMap* is effective for high-dimensional datasets.

3-D experiments						
	Naive			*MultiMap*		
Query	*Measured (ms)*	*Model (ms)*	Δ	*Measured (ms)*	*Measured (ms)*	Δ
Dim_0	0.03	0.03	0%	0.04	0.03	13%
Dim_1	2.7	2.7	0%	1.7	1.5	8%
Dim_2	5.2	4.3	17%	1.5	1.5	0%
1%	294	291	1%	205	168	18%
2%	1086	1156	6%	722	723	0%
3%	2485	2632	5%	1626	1758	8%
4-D experiments						
Dim_0	0.02	0.02	0%	0.02	0.02	0%
Dim_1	5.5	5.0	9%	2.1	1.9	9%
Dim_2	2.2	4.0	82%	2.1	1.9	9%
Dim_3	4.86	5.1	5%	1.8	1.6	11%
6^4	1142	1065	7%	513	436	15%
10^4	5509	5016	9%	1996	2150	8%
12^4	8905	8714	2%	3590	3960	9%

Table 5.2: Comparison of measured and predicted I/O times.

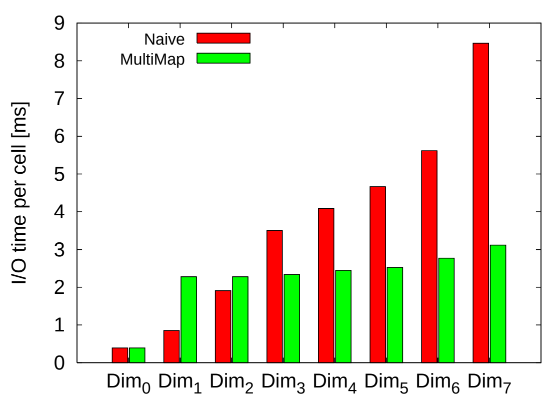


Figure 5.8: Estimated cost of beam queries in 8-D space.

5.6 Chapter Summary

MultiMap is a breakthrough on disk layouts for multidimensional data in that it extends the traditional linear abstraction of disks. In the world of *MultiMap*, disks are not in a linear space anymore: they are in a multidimensional space where the well-known sequential path and the new semi-sequential paths construct different dimensions in the space.

The contributions of *MultiMap* are twofolds. First, it presents a simple interface that takes the coordinates of a point in a multidimensional space as inputs and returns the `lbn` to store the point. This interface depicts disks as a multidimensional space. Second and more importantly, by using this interface, *MultiMap* solves the fundamental problem of preserving spatial locality for multidimensional data on disks because it does not need to do linearization as existing solutions have to. As a result, *MultiMap* does not face the performance trade-offs among different dimensions, thus demonstrating substantial improvement over traditional mapping techniques. On average, *MultiMap* reduces total I/O time by over 50% when compared to traditional linearized layouts and by over 30% when compared to space-filling curve approaches such as Z-ordering and Hilbert curves. For scans of the primary dimension, *MultiMap* and traditional linearized layouts provide almost two orders of magnitude higher throughput than space-filling curve approaches.

MultiMap achieves its goals by successfully exploiting the detailed device-specific information about semi-sequential access. The design of *MultiMap* is about how to leverage this new technique without increasing the implementation complexity. The interface mapping a multi-dimensional point to a disk block carefully hides the semi-sequential access details from user applications without compromising the functionality.

Last but not least, the disk technology development trends of fast increasing track density and slow improvement on seek time and rotational latency imply more dimensions supported on disks, which makes *MultiMap* even more useful in the future. Currently, the typical maximum number of dimensions supported by a modern disk is 10.

This chapter is a further exploration of how to utilize semi-sequential access paths to eliminate the performance trade-offs among multiple dimensions. The next chapter continues the journey from another angle: how to eliminate the performance trade-offs among different execution phases.

Chapter 6

Data organization for hash join and external sorting

This chapter discusses how a database system can utilize the semi-sequential access to organize intermediate results for hash join and external sorting. We discover that the performance trade-offs shown in accessing multidimensional datasets also exist in the two phases of the *GRACE* hash join algorithm and external sorting algorithms. The conflicting I/O access patterns are not caused by the high dimensionality of datasets, but rather by the different orders in which the two operators process data. In existing systems, one phase of these two operators achieves the sequential bandwidth while the other phase suffers the inefficiency of random accesses.

The solution proposed in this dissertation exploits the semi-sequential access path to store the intermediate results between the two phases. While keeping the good performance of the sequential access in one phase, the solution replaces expensive random accesses with semi-sequential accesses in the other phase, resulting in up to a 50% shorter I/O time. This performance is achieved without modifying the kernel algorithms. The rest of this chapter explains the conflicting I/O accesses, followed by the solutions there are uncovered in this project. The last part of the chapter presents the experimental results to evaluate the performance of the new solution on a prototype system.

6.1 Hash join

Hash join [27, 36, 43, 68, 83], as an efficient algorithm to implement equijoins, is widely used in commercial database systems. In its simplest form, when the available memory is large enough, the algorithm first reads in the smaller (*build*) relation and builds a hash table based on the value of the join attributes. Then it scans the larger (*probe*) relation, and for each tuple from the probe relation, it probes the hash table to find matches. In the simplest scenario, the only I/O operations are one scan on each join relation and one sequential write for the output.

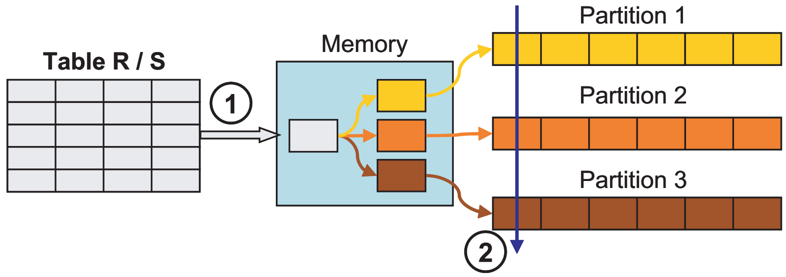
In the real world, database systems adopt more sophisticated algorithms, such as the *GRACE* hash join algorithm [36], to deal with the problem of excessive disk accesses when the memory is too small to hold the build relation and/or the hash table. Practical hash join algorithms usually take a two-phase approach. The first phase is called "partition phase" in which the algorithm partitions the two joining relations into smaller sub-relations (also called "partitions") using a hash function on the join attributes. In the second phase, the "join phase", the algorithm joins each pair of the build and probe partitions that have the same hash value as in the simple scenario. The number of partitions is based on the sizes of the available memory and the size of the build relation. A key point is that the build partition and the hash table built upon it must fit in memory. If a build partition turns out to be too big for the memory because of a skewed hash distribution, the above two-phase process is applied recursively.

Due to multiple read and write operations, hash join algorithms are I/O-intensive in nature. Since the introduction of the *GRACE* hash join algorithm, a lot of work has been done to minimize the I/O operations by keeping as many intermediate partitions in memory as possible [27, 36, 43, 68, 83]. The approach presented in this chapter takes a different angle by exploring more efficient ways to organize intermediate partitions on disks. If partitions must be read/written, what are the best ways to do it?

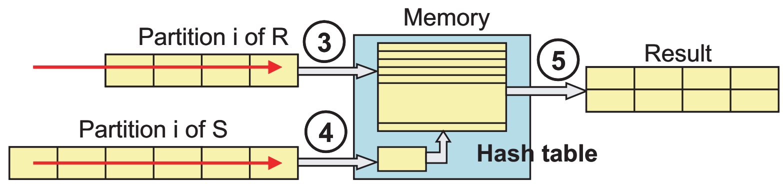
The next section explains the challenges of optimizing I/O performance for hash join algorithms, followed by our solution.

6.1.1 Opposite I/O accesses in partition phase and join phase

This section delineates the I/O access patterns in the two phases of the *GRACE* hash join algorithm. Figure 6.1 shows the procedure for joining two tables, R and S. The numbers in the circles denote the execution order. Following the order, the algorithm first partitions the join relations (Figure 6.1(a)) as follows. Step 1 reads in the join relation (e.g. relation R) in chunks with a typical size of several KB. A hash function is used to calculate the hash value of each tuple and put it into corresponding buckets. In this example, the hash function has three buckets (thus three partitions), coded with different colors: yellow, orange, and brown. When a bucket is full, the algorithm writes it to disk in chunks, as denoted by step 2. Assuming that the hash function is adequate, each final partition contains roughly the same number of records. In this example, each partition occupies 6 disk chunks. More importantly, in practice, the three buckets in memory are usually filled and written out alternately, resulting in interleaved writes to different partitions, as illustrated by the vertical arrow in Figure 6.1(a). The interleaved writes to disks



(a) **Partition phase.**



(b) **Join phase.**

Figure 6.1: *GRACE* hash join algorithm. The graph illustrates the two phases in the *GRACE* hash join algorithm. The vertical arrow at step 2 indicates interleaved writes to three partitions, but not a writing order from partition 1 to partition 3.

form a random-like access pattern.

After both join relations are partitioned, the algorithm enters the join phase (Figure 6.1(b)). In this phase, the algorithm joins pairs of partitions that have the same hash value in the previous phase. Without losing generality, the graph uses the pair of partition i as an example to illustrate the procedure. Step 3 reads in the entire partition i of R and builds a hash table based on the join attributes. Step 4 reads in the partition i of S in chunks, probes the hash table with each tuple in the chunk, and outputs the matched ones, as shown in step 5.

In today's systems, the I/O access patterns in steps 3 and step 4, which scan the join partitions (denoted by the horizontal arrow), conflict with the I/O access patterns in step 2. Sequential accesses at step 3 and 4 result in random accesses at step 2 because the interleaved I/O operations at step 2 are now writing to non-contiguous blocks spread across disks.

6.1.2 Organizing partitions along the semi-sequential path

The performance trade-offs caused by the conflicting I/O access patterns in the hash join algorithm can be resolved by utilizing the semi-sequential access path. This idea is similar to *MultiMap* in which the random access at step 2 is replaced with the much more efficient semi-sequential access. The high-level idea is to align all partitions along a semi-sequential access path. In other words, the chunks of different partitions that are of the same offset in the partition

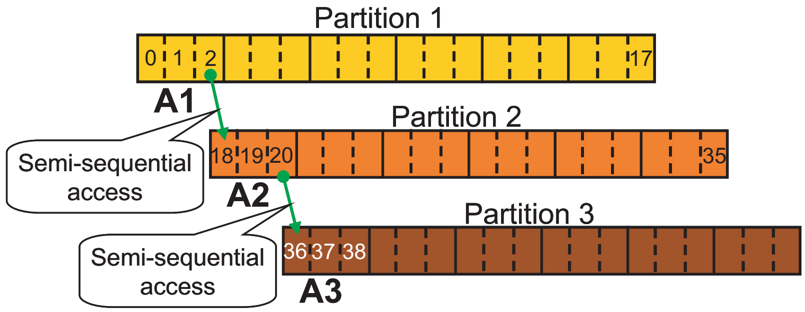


Figure 6.2: Proposed disk organization for partitions.

will be stored in the consecutive adjacent blocks. Figure 6.2 depicts the data placement strategy with more details.

Figure 6.2 is a detailed picture of step 2 taken from Figure 6.1(a) where the algorithm writes full buckets to the disk. Boxes with solid sides in the example denote chunks, the basic unit of a read/write operation. Boxes with dotted sides denote disk blocks, usually of 512 bytes. The chunk size in this example is thus 3 disk blocks. For simplicity, the example assumes the size of each partition equals the track length (i.e., 18 blocks in the graph) and disk block 0 is the starting disk block. A1, A2, and A3 are the first chunks in the three partitions, and are defined to be of the same offset from the beginning of the partitions to which they belong. Numbers in the dotted boxes (disk blocks) are LBNs. Suppose that chunk A1 is stored in the first 3 blocks, from LBN 0 to LBN 2. With the knowledge of the adjacency model, the new data placement algorithm picks the adjacent block of LBN 2 to be the first block of Chunk A2. For example, if LBN 18 is the adjacent block of LBN 2, then Chunk A2 is stored in the blocks starting from LBN 18. Similarly, the starting block of Chunk A3 is stored in the adjacent block of LBN 20, which is LBN 36 in the example. In this way, writing A1, A2, and A3 incurs semi-sequential access. By only paying the cost of settle time, the disk head can read LBN 18 after accessing LBN 2, without any rotational latency. This also holds true for the access from LBN 20 to LBN 36.

Implementation concerns, such as how to issue I/O requests and how to handle larger partitions, are shared by hash join and external sorting. Therefore, they are discussed together in Section 6.3.1.

6.2 External sorting

External sorting algorithms are used to order large datasets that cannot fit into memory. Similar to hash join algorithms, they typically have two phases. The first phase generates a set of files and the second phase processes these files to output the sorted result. Based on how the first phase generates the set of files, external sorting can be roughly classified into two groups [79]:

1. Distribution-based sorting: The input data file is partitioned into N partitions, with each partition covering a range of values [37]. All elements in partition i are larger than the elements in the partitions from 0 to (i-1) and smaller than the elements in the partitions from (i+1) to (N-1). The second phase then sorts each partition separately and concatenates them as the final output.
2. Merge-based sorting: The first phase partitions the input file into equal-sized chunks and sorts them separately in main memory. The sorted chunks, also called *runs*, are written to disks. The second phase merges the sorted runs and outputs the final sorted file.

External memory sorting performance is often limited by I/O performance. Extensive research has been done on formalizing the I/O cost in theory and minimizing the amount of data written to and read from disks in implementation [2, 38, 45, 56, 79]. Zhang et. al. [84] improve the performance of external mergesort by dynamically adjusting the memory allocated to the operation. Equal buffering and extended forecasting [85] are enhanced versions of the standard double buffering and forecasting algorithms that improve the external mergesort performance by exploiting the fact that virtually all modern disks perform caching and sequential prefetch. Some efforts aim at using a co-processor, such as GPU, to offload compute-intensive and memory-intensive tasks to the GPU to achieve higher I/O performance and better main memory performance [26]. Other approaches seek to minimize I/O time by reading in run chunks in a particular order that involves minimal disk seeks, such as the technique of *clustering* [85]. This dissertation chooses a similar angle as *clustering* towards optimizing the I/O performance by reading in run chunks in a disk-concious way. Different from the previous research, our idea is to organize the intermediate partitions along the new semi-sequential path.

6.2.1 Opposite I/O accesses in two phases

The conflict I/O access patterns in the distribution-based sorting algorithm are the same as in hash join. The writes to different partitions in the first phase are random if the algorithm chooses to optimize the later read operations by storing each partition sequentially.

The merge-based sorting algorithm is different, depicted in Figure 6.3. This diagram shows the procedure of sorting a file named D. In the partition phase (Figure 6.3(a)), at step 1, the algorithm reads in sequentially a certain amount of data that can be held in main memory and sorts it using any existing sorting algorithms. Then the sorted chunk, called a *run*, is written out sequentially at step 2. The graph shows the generation of the i-th run. After the entire file is consumed, the merge phase applies merge sort on all runs (step 3), and outputs the final sorted

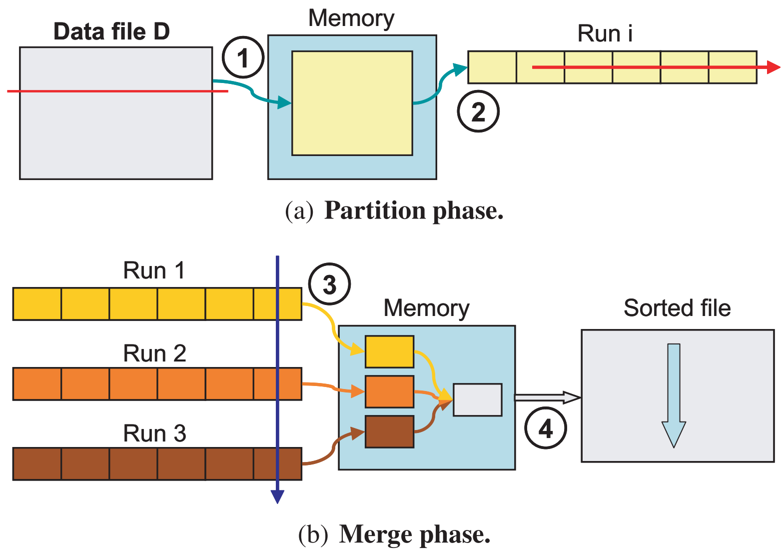


Figure 6.3: Merge-based external sorting algorithm. The graph illustrates the two phases in the merge-based external sorting algorithm.

file (step 4). At step 3, the three runs are fetched into memory in chunks. If the data being sorted is not highly skewed, reading in chunks from different runs are usually interleaved, which causes random access patterns on disks.

6.2.2 Organizing runs along the semi-sequential path

Adopting an idea similar to the one used in the solution of hash join, the runs are aligned along the semi-sequential access path. That is, the chunks of different runs that are of the same offset in the run will be stored in the consecutive adjacent blocks. The layout details of each run, such as the adjacent block selection, are the same as in hash join and have been discussed in Section 6.1.2 (Refer to Figure 6.2).

6.3 Evaluation

This section describes the implementation of a prototype developed on a logical volume comprised of real disks that is used to evaluate the performance of the new data organization for hash join and external sorting.

6.3.1 Implementation

The prototype system is implemented on a two-way 1.7 GHz Pentium 4 Xeon workstation running Linux kernel 2.4.24 with 1024 MB of main memory and one Adaptec Ultra160 SCSI adapter connecting two 36.7 GB disks: a Seagate Cheetah 36ES and a Maxtor Atlas 10k III. This prototype system consists of a logical volume manager (LVM) and the operators of the *GRACE* hash join algorithm [68] and the merge-based external sorting [37].

Hash join and external sorting algorithm implementation. We adopt the optimized hash join implementation from Chen et al [16]. It uses a slotted page structure with support for fixed-length and variable-length attributes in tuples. The hash function is a simple XOR and shift-based function. It converts join attributes of any length to 4-byte hash codes. The same hash codes are used in both the partition and the join phase. Partition numbers are hash codes modulo the total number of partitions which are calculated based on the statistics of the relations. Hash bucket numbers in the join phase are the hash codes modulo the hash table size. The merge-based external sorting algorithm is implemented using Quick sort in the partition phase to sort each run.

Double buffering. We also implement a simple buffer pool manager with the function of double buffering. Double buffering is a commonly used technique to exploit CPU resources during I/O operation by overlapping CPU and I/O processing. It also helps to accommodate the different filling (or consuming) speeds of different partitions (or runs). In the hash join operator, when a bucket is full, it is put in a full bucket list, and a free bucket will be linked in to fill in the vacancy. When there are full buckets for all partitions or there are no more free buckets, a worker thread will write them out and put the buckets back on the free list. In external mergesort, reading run chunks is triggered when the memory space for a run is half empty.

Skewed datasets. The approaches proposed in this chapter work nicely with uniformly distributed datasets, meaning the hash function in hash join does a good job at distributing tuples equally or the values of sort keys in external mergesort are randomly distributed. Datasets with highly skewed distributions pose some obstacles. For example, in an extreme case, the writing of hash join partitions or the reading of external sort runs are sequential, no interleaving reads or writes at all. In this case, our approach would not help to improve performance because first, no semi-sequential access batches will be issued; second, the sequential access is the fastest. Generally speaking, the skewness of datasets determines whether writing join partitions (hash join) and reading run chunks (external sorting) are interleaved. In other words, skewed datasets have better I/O performance than uniformly distributed datasets. From our approaches' point of view, as far as the interleaving reads/writes exist, it can help improve the I/O performance. If no such an interleaving access pattern, it will not hurt performance either.

Track-aligned extents. In the current prototype, disk space for tables, partitions, or runs is allocated in extents. An extent is a set of contiguous disk blocks. This prototype utilizes previously proposed "track-aligned extents" (*traxtents*) [58, 59] to achieve efficient sequential access. The idea of track-aligned extents exploits automatically-extracted knowledge of disk track boundaries, using them as the extent size. If a partition or a run is larger than the track capacity, the system just allocates another extent on a new track for it.

Large memory. The performance improvement gained from our approaches comes from the idea of replacing random read/write of chunks with semi-sequential access. If the chunk size is very big, the time it takes to read/write a chunk will be long, which makes the overhead of disk seeks less important or even negligible. In a system with ample memory, it tends to use large chunk sizes. In this case, our approaches cease to be helpful. The performance diminishing is shown in the experiments in the next section.

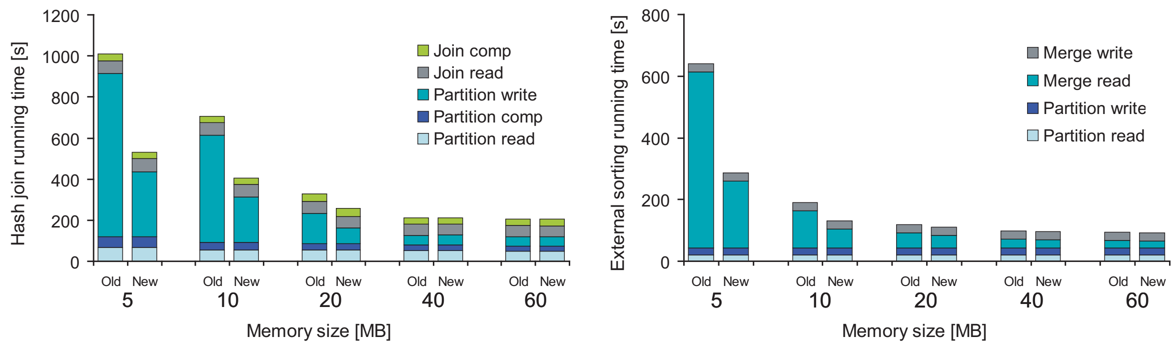
The next section compares our solutions to existing approaches where no care is taken to avoid random accesses.

6.3.2 Experiment results

The first half of this section evaluates the hash join algorithm. In the experiments, build relations and probe relations have the same schema which consists of a 4-byte join key and a fixed-length payload. No selection or projection operations are performed because they are orthogonal to the study. Output records consist of all fields from the two join relations. The tuple size in the following experiments is 100 bytes, and the sizes of the relations are 1 GB and 2 GB. Values of the join key are randomly distributed. All tuples in the build relation have matches in the probe relation. All partitions and the hash table built on them fit tightly in main memory; therefore, no recursion is involved.

We vary memory size from 5 MB to 60 MB and measure the total running time of the hash join algorithm, shown in Figure 6.4(a). By "Old", we denote the traditional semi-sequential-oblivious approach, and by "New", we refer to the new solution discussed in the previous sections. The total running time is broken down into five components, from the bottom to the top: (a) "partition read" is the total time spent reading in the join relations; (b) "partition comp" refers to the computational cost in the partition phase to calculate hash keys and to distribute tuples; (c) "partition write" is the time spent writing filled buckets which is the optimization target; (d) "join read" and (e) "join comp" are the costs in the join phase which consists of the time to read in the join partition and to build/probe the hash table.

When the memory size is small, the bucket size is also small. Therefore, writing out the partitions incurs more I/O requests for small data chunks, resulting in worse performance. The new



(a) **Hash join.** (b) **External sorting.**

Figure 6.4: Running time breakdown for hash join and external sorting algorithms. The graph plots the running time of the "Old" and "New" approaches with increasing memory sizes. The "old" solution refers to the existing implementation that does not take the advantage of semi-sequential access while the "new" solution exploits this feature to store the intermediate partitions.

organization of aligning partitions along the semi-sequential path helps to mitigate the problem. In the case where the memory size is 5 MB, the "New" approach reduces the "partition write" time by a factor of 2. The performance improvement diminishes as memory size increases because of the increased bucket size. Large bucket size implies few I/O requests with large data chunks. In this case, the benefit of saving rotational latency becomes less manifest.

The same trend is observed in the performance evaluation on the external sorting algorithm, as Figure 6.4(b) shows. In this experiment, a table of 1 GB is sorted using the same set of memory sizes (i.e., from 5 MB to 60 MB). Values of the sort attribute are uniformly distributed. The legend used in Figure 6.4(b) refers to different operations performed in the external sorting algorithm, from the bottom to the top: (a) "partition read" is the time spent reading in the data file at the beginning; (b) "partition write" is the time spent writing the sorted runs; (c) "merge read" refers to the cost of reading data from different partitions in the merge phase; and (d) "merge write" is the time spent writing out the final sorted file. True to its goal, our solution successfully reduces the cost of "merge read" by a factor of 2 when the memory size is small. The advantage of organizing intermediate partitions along the semi-sequential path decreases as more memory is available.

In real applications, the memory available for the operations of hash join and external sorting is quite limited because this part of memory is allocated from the private space of each execution process. In a system with a high concurrency degree, this new approach helps to sustain good performance with less memory. This is also beneficial to the other modules in the entire system.

6.4 Chapter summary

In this chapter, we propose a new way to optimize the I/O performance for two important database operators, hash join and external sorting, by exploiting the semi-sequential access path for organizing intermediate partitions. Our purpose is to improve the performance of the worst scenarios in hash join and external sorting, where the reads/writes of partitions/runs result in a random disk access pattern.

The performance improvement is achieved by eliminating expensive random accesses during the execution of the two operators in current systems. The experiments demonstrate that our prototype exhibits a speedup of a factor of two when compared to traditional techniques when the available memory is limited (less than 1% of the dataset size). Moreover, we achieve this speedup without the need of modifying existing algorithms. This project is another example that a deeper understanding of hardware features can facilitate data organization in database systems. However, we also find out that the performance improvement diminishes as more memory is available. The approach achieves the same performance as existing solutions when the memory size is larger than 6% of the dataset. Large memory size implies large I/O request size, which makes the overhead of a random disk access, seek time and rotational latency, less important in performance.

Although the memory size in a system will keep increasing rapidly in the future, the available memory to hash join and external sorting will not increase at a similar speed. In fact, since this part of memory is allocated on a per thread basis, the increasing concurrency degree will put a restriction on the memory space allocated to hash join and external sorting, which makes the approaches in this chapter stay valuable in the future.

Chapter 7

Conclusions

Thesis statement

"*Database Management Systems can become more robust by eliminating performance trade-offs related to inflexible data layout in the memory and disk hierarchy. The key to improving performance is to adapt the data organization to workload characteristics, which is achieved (a) by intelligently using query payload information when managing the buffer pool, (b) by decoupling the in-memory layout from the storage organization, and (c) by exposing the semi-sequential access path available on modern disks to the storage manager.*".

Existing database management systems use static data organization across all layers of the memory hierarchy: the same data layout is used in CPU caches, main memory, and disks. The single format design simplifies the implementation of database systems and works well in the early days when most hardware were dumb devices and major workloads shared similar characteristics. Unfortunately, the "one layout fits all" solution does not meet the ever-increasing performance requirements of today's applications where the high diversity of workloads asks for different, sometimes even conflict, designs of data layouts. In the meantime, this simple solution fails to leverage many advanced features being added to storage devices, missing opportunities which could have been exploited to improve performance.

In this thesis, we propose flexible and efficient data organization and management for database systems to address these problems. Our solutions include (a) a scaled-down database benchmark suite for quick evaluation; (b) an adaptive buffer pool manager which applies *CSM*, a dynamic page layout, to customize in-memory content to query payloads; (c) a new mapping algorithm, *MultiMap*, that exploits the new adjacency model of disks to provide a multidimensional structure on top of the linear disk space for efficiently storing multidimensional data; and (d) a new way to organize intermediate results for external sorting and hash join.

First, DBmbench provides a significantly scaled-down database benchmark suite that accurately mimics the characteristics of the most commonly used DSS and OLTP workloads, TPC-H and TPC-C, at the microarchitecture-level. This is done by identifying and isolating a small set of operations that primarily dominate the executions of the workloads. DBmbench simplifies experiment setup, reduces experiment running time, and ease performance analysis. More importantly, DBmbench faithfully preserves the major characteristics of their large-scale counterparts. DBmbench is also valuable database system research, especially in sensitivity analysis. The other projects in this thesis employ it to study the performance effects of individual parameters on the entire system.

Clotho is a buffer pool manager which adapts to changing workloads by decoupling the in-memory page layout from the storage organization. The decoupling solves the problem inherent to existing static page layouts, i.e., the performance trade-offs between two major database workloads TPC-H and TPC-C. Like *DSM*, *Clotho* asks for only requested attributes, which makes it the best choice for TPC-H-like workloads. Like *NSM*, *Clotho* also performs well for full record accesses as in TPC-C-like workloads, thanks to the new semi-sequential access path on disks. However, *Clotho* is not burdened with the details of the semi-sequential access path. The decoupling allows the volume manager to organize pages on disks that best exploits the characteristics of the storage devices.

MultiMap is a mapping algorithm to store multidimensional datasets on disks. Instead of using the over-simplified linear abstraction of disks, *MultiMap* utilizes the semi-sequential access path to build a multidimensional view on disks, thus eliminating the restrictions posed by the linear abstraction. Thanks to the multidimensional abstraction, the spatial locality of multidimensional datasets can be preserved on disks which translates to superior performance for range and beam queries. On average, *MultiMap* reduces total I/O time by over 50% when compared to traditional linearized layouts and by over 30% when compared to space-filling curve approaches.

Continuing the exploration of building hardware-aware algorithms, We investigate the idea of utilizing the semi-sequential access path to organize intermediate partitions in hash join and external sorting. It eliminates the costly random accesses existing in the current implementation, thus optimizing the I/O performance without modifying the kernel algorithms. This solution is especially useful in systems with scarce resources. The experiments on our prototype demonstrate a speedup of a factor of two over the traditional implementation.

From *Clotho* to the data placement for hash join and external sorting, the projects of this dissertation revolve around the theme of enhancing the interaction between software and hardware and demonstrate the significant performance benefits brought by it from various aspects.

The evolution of computer system has shown two evident developing trends. One is that computer hardware, such as processors, main memory, and disks, keeps evolving with more sophisticated built-in intelligence; the other is that computer software, from low-level firmware up to user applications, is becoming more and more complicated and diversified due to the ever increasing demands for performance and features. Under these two trends, it is increasingly crucial to design software that collaborates with hardware devices, thus exploiting their characteristics to the fullest.

Bibliography

[1] Khaled A. S. Abdel-Ghaffar and Amr El Abbadi. Optimal allocation of two-dimensional data. In *International Conference on Database Theory*, pages 409–418, 1997.

[2] Alok Aggarwal and S. Vitter Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.

[3] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. DBMSs on a modern processor: where does time go? In *Proceedings of International Conference on Very Large Databases*, pages 266–277. Morgan Kaufmann Publishing, Inc., 1999.

[4] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. Weaving relations for cache performance. In *International Conference on Very Large Databases*, pages 169–180. Morgan Kaufmann Publishing, Inc., 2001.

[5] Dave Anderson, Jim Dykes, and Erik Riedel. More than an interface: SCSI vs. ATA. In *FAST*, pages 245–257. USENIX, 2003.

[6] Murali Annavaram, Trung Diep, and John Shen. Branch behavior of a commercial OLTP workload on Intel IA32 processors. In *Proceedings of International Conference on Computer Design*, September 2002.

[7] Mikhail J. Atallah and Sunil Prabhakar. (almost) optimal parallel block access for range queries. In *ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 205–215. ACM, 2000.

[8] Luiz A. Barroso, Kourosh Gharachorloo, Andreas Nowatzyk, and Ben Verghese. Impact of chip-level integration on performance of OLTP workloads. In *Proceedings of International Symposium on High-Performance Computer Architecture*, January 2000.

[9] Luiz A. Barroso, Kourosh Gharachorloo, and Edouard Bugnion. Memory system characterization of commercial workloads. In *Proceedings of International Symposium on Computer Architecture*, pages 3–14, June 1998.

[10] Randeep Bhatia, Rakesh K. Sinha, and Chung-Min Chen. Declustering using golden ratio

sequences. In *ICDE*, pages 271–280, 2000.

[11] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. Database architecture optimized for the new bottleneck: memory access. In *International Conference on Very Large Databases*, pages 54–65. Morgan Kaufmann Publishers, Inc., 1999.

[12] Qiang Cao, Pedro Trancoso, Josep-Lluis Larriba-Pey, Josep Torrellas, Robert Knighten, and Youjip Won. Detailed characterization of a quad Pentium Pro server running TPC-D. In *Proceedings of International Conference on Computer Design*, October 1999.

[13] Michael J. Carey, David J. DeWitt, Michael J. Franklin, Nancy E. Hall, Mark L. McAuliffe, Jeffrey F. Naughton, Daniel T. Schuh, Marvin H. Solomon, C. K. Tan, Odysseas G. Tsatalos, Seth J. White, and Michael J. Zwilling. Shoring up persistent applications. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, May 24-27, 1994*, pages 383–394. ACM Press, 1994.

[14] L. R. Carley, J. A. Bain, G. K. Fedder, D. W. Greve, D. F. Guillou, M. S. C. Lu, T. Mukherjee, S. Santhanam, L. Abelmann, and S. Min. Single-chip computers with microelectromechanical systems-based magnetic memory. *Journal of applied physics*, 87(9):6680–6685, May 2000. ISSN 0021-8979.

[15] Shimin Chen, Phillip B. Gibbons, and Todd C. Mowry. Improving index performance through prefetching. In *SIGMOD Conference*, 2001.

[16] Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. Improving hash join performance through prefetching. In *ICDE*, pages 116–127, 2004.

[17] Edgar Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13 (6):377–387, 1970.

[18] George P. Copeland and Setrag Khoshafian. A decomposition storage model. In *ACM SIGMOD International Conference on Management of Data*, pages 268–279. ACM Press, 1985.

[19] DB2Manual. *IBM DB2 Universal Database Administration Guide: Implementation*, 2000.

[20] Trung Diep, Murali Annavaram, Brian Hirano, and John P. Shen. Analyzing performance characteristics of OLTP cached workloads by linear interpolation. In *Workshop on Computer Architecture Evaluation using Commercial Workloads*, September 2002.

[21] disksim. The disksim simulation environment (version 3.0), 2006. http://-www.pdl.cmu.edu/DiskSim/index.html.

[22] T. Todd Elvins. A survey of algorithms for volume visualization. *Computer Graphics*, 26 (3):194–201, 1992. URL `citeseer.ist.psu.edu/elvins92survey.html`.

[23] Christos Faloutsos. Multiattribute hashing using Gray codes. In *ACM SIGMOD*, pages 227–238, 1986.

[24] Christos Faloutsos and Pravin Bhagwat. Declustering using fractals. In *International Conference on Parallel and Distributed Information Systems*, 1993.

[25] George G. Gorbatenko and David J. Lilja. Performance of two-dimensional data models for I/O limited non-numeric applications. Technical Report ARCTiC 02-04, Laboratory for Advanced Research in Computing Technology and Compilers, University of Minnesota, 2002.

[26] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. Gputerasort: high performance graphics co-processor sorting for large database management. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 325–336. ACM Press, 2006.

[27] Goetz Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25 (2):73–170, 1993.

[28] Jim Gray. *The Benchmark Handbook for Database and Transaction Processing Systems*. Morgan Kaufmann Publishers, Inc., second edition, 1993.

[29] Jim Gray, Donald Slutz, Alexander Szalay, Ani Thakar, Jan vandenBerg, Peter Kunszt, and Chris Stoughton. Data mining the SDSS skyserver database. Technical Report MSR-TR-2002-01, MSR, 2002.

[30] Richard Hankins, Trung Diep, Murali Annavaram, Brian Hirano, Harald Eri, Hubert Nueckel, and John P. Shen. Scaling and characterizing database workloads: bridging the gap between research and practice. In *Proceedings of International Symposium on Microarchitecture*, December 2003.

[31] Richard A. Hankins and Jignesh M. Patel. Data morphing: an adaptive, cache–conscious storage technique. In *International Conference on Very Large Databases*, pages 1–12. VLDB, 2003.

[32] David Hilbert. Über die stetige abbildung einer linie auf flächenstück. *Math. Ann*, 38: 459–460, 1891.

[33] Ibrahim Kamel and Christos Faloutsos. Parallel R-trees. In *SIGMOD*, pages 195–204, 1992.

[34] Kimberly Keeton and David A. Patterson. *Towards a simplified database workload for computer architecture evaluations*, chapter 4. Kluwer Academic Publishers, 2000.

[35] Kimberly Keeton, David A. Patterson, Yong Qiang He, Roger C. Raphael, and Walter E.

Baker. Performance characterization of a quad Pentium Pro SMP using OLTP workloads. In *Proceedings of International Symposium on Computer Architecture*, pages 15–26, June 1998.

[36] Masaru Kitsuregawa, Hidehiko Tanaka, and Tohru Moto-Oka. Application of hash to data base machine and its architecture. *New Generation Comput.*, 1(1):63–74, 1983.

[37] Donald E. Knuth. *The art of computer programming, volume 3: sorting and searching*. Addison-Wesley, Reading, 2nd edition, 1998. ISBN 0–201–89685–0.

[38] Per-Åke Larson. External sorting: Run formation revisited. *IEEE Trans. Knowl. Data Eng.*, 15(4):961–972, 2003.

[39] Jack L. Lo, Luiz Andre Barroso, Susan J. Eggers, Kourosh Gharachorloo, Henry M. Levy, and Sujay S. Parekh. An analysis of database workload performance on simultaneous multi-threaded processors. In *Proceedings of International Symposium on Computer Architecture*, June 1998.

[40] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. Generic database cost models for hierarchical memory systems. In *International Conference on Very Large Databases*, pages 191–202. Morgan Kaufmann Publishers, Inc., 2002.

[41] C. Mohan, Donald J. Haderle, Bruce G. Lindsay, Hamid Pirahesh, and Peter M. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial roll-backs using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992.

[42] Bongki Moon, H. V. Jagadish, Christos Faloutsos, and Joel H. Saltz. Analysis of the clustering properties of Hilbert space-filling curve. Technical Report UMIACS-TR-96-20, University of Maryland at College Park, 1996.

[43] Masaya Nakayama, Masaru Kitsuregawa, and Mikio Takagi. Hash-partitioned join method using dynamic destaging strategy. In *VLDB*, pages 468–478, 1988.

[44] Beomseok Nam and Alan Sussman. Improving access to multi-dimensional self-describing scientific datasets. *International Symposium on Cluster Computing and the Grid*, 2003.

[45] Mark H. Nodine and Jeffrey Scott Vitter. Greed sort: optimal deterministic sorting on parallel disks. *J. ACM*, 42(4):919–933, 1995.

[46] Office of Science Data-Management Workshops. The office of science data-management challenge. Technical report, Department of Energy, 2005.

[47] Jack A. Orenstein. Spatial query processing in an object-oriented database system. In *ACM SIGMOD*, pages 326–336. ACM Press, 1986.

[48] Sriram Padmanabhan, Bishwaranjan Bhattacharjee, Tim Malkemus, Leslie Cranston, and Matthew Huras. Multi-dimensional clustering: A new data layout scheme in DB2. In *ACM SIGMOD*, 2003.

[49] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks RAID. In *ACM SIGMOD International Conference on Management of Data*, pages 109–116, 1–3 June 1988.

[50] Sunil Prabhakar, Khaled Abdel-Ghaffar, Divyakant Agrawal, and Amr El Abbadi. Efficient retrieval of multidimensional datasets through parallel I/O. In *ICHPC*, pages 375–386. IEEE, 1998.

[51] Sunil Prabhakar, Khaled A. S. Abdel-Ghaffar, Divyakant Agrawal, and Amr El Abbadi. Cyclic allocation of two-dimensional data. In *ICDE*. IEEE, 1998.

[52] Raghu Ramakrishnan and Johannes Gehrke. *Database management systems*. McGraw-Hill, 3rd edition, 2003.

[53] Ravishankar Ramamurthy, David J. DeWitt, and Qi Su. A case for fractured mirrors. In *International Conference on Very Large Databases*, pages 430–441. Morgan Kaufmann Publishers, Inc., 2002.

[54] Parthasarathy Ranganathan, Sarita V. Adve, Kourosh Gharachorloo, and Luiz A. Barroso. Performance of database workloads on shared-memory systems with out-of-order processors. In *Proceedings of Architectural Support for Programming Languages and Operating Systems*, volume 33, pages 307–318, November 1998. ISBN ISSN 0362–1340.

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

[56] Betty Salzberg, Alex Tsukerman, Jim Gray, Michael Stuewart, Susan Uren, and Bonnie Vaughan. Fastsort: a distributed single-input single-output external sort. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, pages 94–101. ACM Press, 1990. ISBN 0-89791-365-5.

[57] Sunita Sarawagi and Michael Stonebraker. Efficient organization of large multidimensional arrays. In *ICDE*, pages 328–336, 1994.

[58] Jiri Schindler, John Linwood Griffin, Christopher R. Lumb, and Gregory R. Ganger. Track-aligned extents: matching access patterns to disk drive characteristics. In *Conference on File and Storage Technologies*, pages 259–274. USENIX Association, 2002. ISBN ISBN 1–880446–03–0.

[59] Jiri Schindler, Anastassia Ailamaki, and Gregory R. Ganger. Lachesis: robust database

storage management based on device-specific performance characteristics. In *International Conference on Very Large Databases*, pages 706–717. Morgan Kaufmann Publishing, Inc., 2003.

[60] Jiri Schindler, Steven W. Schlosser, Minglong Shao, Anastassia Ailamaki, and Gregory R. Ganger. Atropos: a disk array volume manager for orchestrated use of disks. In *Conference on File and Storage Technologies*, pages 27–41. USENIX Association, 2004.

[61] Steven W. Schlosser, John Linwood Griffin, David F. Nagle, and Gregory R. Ganger. Designing computer systems with MEMS-based storage. In *Architectural Support for Programming Languages and Operating Systems*, pages 1–12, 2000.

[62] Steven W. Schlosser, Jiri Schindler, Minglong Shao, Stratos Papadomanolakis, Anastassia Ailamaki, Christos Faloutsos, and Gregory R. Ganger. On multidimensional data and modern disks. In *Conference on File and Storage Technologies*. USENIX Association, 2005.

[63] Steven W. Schlosser, Jiri Schindler, Anastassia Ailamaki, and Gregory R. Ganger. Exposing and exploiting internal parallelism in MEMS-based storage. Technical report, Technical Report CMU–CS–03–125, March 2003.

[64] Manfred Schroeder. *Fractals, Chaos, Power Laws: Minutes From an Infinite Paradise*. W. H. Freeman, 1991.

[65] Bernhard Seeger and Per-Åke Larson. Multi-disk B-trees. In *SIGMOD*, pages 436–445, 1991.

[66] Minglong Shao, Jiri Schindler, Steven W. Schlosser, Anastassia Ailamaki, and Gregory R. Ganger. Clotho: Decoupling memory page layout from storage organization. In *International Conference on Very Large Databases*, pages 696–707, 2004.

[67] Minglong Shao, Anastassia Ailamaki, and Babak Falsafi. DBmbench: Fast and accurate database workload representation on modern microarchitecture. In *Conference of the Centre for Advanced Studies on Collaborative Research*, pages 254–267, 2005.

[68] Leonard D. Shapiro. Join processing in database systems with large main memories. *ACM Trans. Database Syst.*, 11(3):239–264, 1986.

[69] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[70] simplescalar. SimpleScalar tool set. SimpleScalar LLC. http://www.simplescalar.com.

[71] Spec. *SPEC CPU Benchmark*. The Standard Performance Evaluation Corporation. http://www.specbench.org.

[72] Kurt Stockinger, Dirk Dullmann, Wolfgang Hoschek, and Erich Schikuta. Improving the performance of high-energy physics analysis through bitmap indices. In *Database and Expert Systems Applications*, pages 835–845, 2000. URL `citeseer.ist.psu.edu/stockinger00improving.html`.

[73] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. C-store: A column-oriented DBMS. In *VLDB*, pages 553–564, 2005.

[74] Shreekant S. Thakkar and Mark Sweiger. Performance of an OLTP application on symmetry multiprocessor system. In *Proceedings of International Symposium on Computer Architecture*, 1990.

[75] Pedro Trancoso, Josep-L. Larriba-Pey, Zheng Zhang, and Josep Torrellas. The memory performance of DSS commercial workloads in shared-memory multiprocessors. In *Proceedings of International Symposium on High-Performance Computer Architecture*, Feburary 1997.

[76] tpcmanual. *TPC benchmarks*. Transaction Processing Performance Countil. http://-www.tpc.org.

[77] Tiankai Tu and David R. O'Hallaron. A computational database system for generating unstructured hexahedral meshes with billions of elements. In *SC*, 2004.

[78] Tiankai Tu, David O'Hallaron, and Julio Lopez. Etree: A database-oriented method for generating large octree meshes. In *Eleventh International Meshing Roundtable*, pages 127–138, 2002.

[79] Jeffrey Scott Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Comput. Surv.*, 33(2):209–271, 2001.

[80] Roland E. Wunderlich, Thomas F. Wenisch, Babak Falsafi, and James C. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of International Symposium on Computer Architecture*, June 2003.

[81] Hailing Yu, Divyakant Agrawal, and Amr El Abbadi. Tabular placement of relational data on MEMS-based storage devices. In *International Conference on Very Large Databases*, pages 680–693, 2003.

[82] Hongfeng Yu, Kwan-Liu Ma, and Joel Welling. A parallel visualization pipeline for terascale earthquake simulations. In *Proceedings of Supercomputing '04*, page 49, 2004. ISBN 0-7695-2153-3. doi: http://dx.doi.org/10.1109/SC.2004.6.

[83] Hansjörg Zeller and Jim Gray. An adaptive hash join algorithm for multiuser environments. In *VLDB*, pages 186–197, 1990.

[84] Weiye Zhang and Per-Åke Larson. Dynamic memory adjustment for external mergesort. In *VLDB*, pages 376–385, Athens, Greece, 1997. Morgan Kaufmann Publishers Inc.

[85] Weiye Zhang and Per-Åke Larson. Buffering and read-ahead strategies for external mergesort. In *VLDB*, pages 523–533. Morgan Kaufmann Publishers Inc., 1998.

[86] Jingren Zhou and Kenneth A. Ross. A multi-resolution block storage model for database design. In *International Database Engineering & Applications Symposium*, 2003.