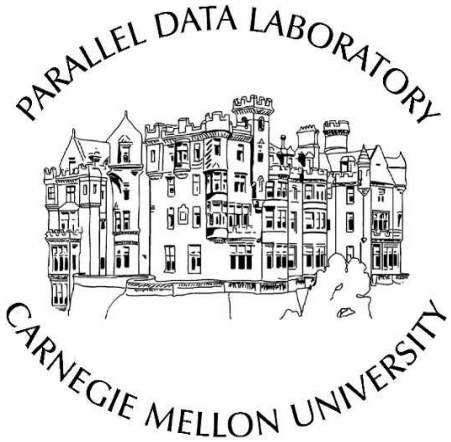


MultiMap: Preserving disk locality for multidimensional datasets

Minglong Shao, Steven W. Schlosser, Stratos Papadomanolakis, Jiri Schindler,
Anastassia Ailamaki, Christos Faloutsos, Gregory R. Ganger

CMU-PDL-05-102
March 2005

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

MultiMap is a new approach to mapping multidimensional datasets to the linear address space of storage systems. 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 overall I/O time by over 50% when compared to traditional naive layouts and by over 30% when compared to a Hilbert curve approach. For scans of the primary dimension, MultiMap and naive both provide almost two orders of magnitude higher throughput than the Hilbert curve approach.

We thank the members and companies of the PDL Consortium (including EMC, Hewlett-Packard, Hitachi, IBM, Intel, Microsoft, Network Appliance, Oracle, Panasas, Seagate, Sun, and Veritas) for their interest, insights, feedback, and support. This work is funded in part by NSF grants CCR-0113660, IIS-0133686, and CCR-0205544, as well as by an IBM faculty partnership award.

Keywords: multidimensional dataset, disk performance, database access, spacial locality

1 Introduction

The simple linear abstraction of storage devices offered by standard interfaces such as SCSI is 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 mapping requires making a choice between storing a 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 thanks to requests to sequential disk blocks, access in the other order is inefficient. The accesses at regular strides give random access-like performance and incur short seeks and rotational latencies. Similarly, range queries are inefficient single dimension. The problem is the same for datasets with higher dimensionality: sequentiality can only be preserved for a single dimension, and all other dimensions will be scattered across the disk.

The shortcomings of non-sequential disk drive accesses have sparked a healthy research effort to improve performance when accessing multidimensional datasets. These often start with a problem statement like this: find a mapping of points in an N-dimensional dataset to a one-dimensional abstraction of the storage device. Mapping schemes such as Z-ordering [15], Hilbert curves [11], and Gray-coded curve [7], can help preserve locality for multidimensional datasets, but they do not allow accesses along any one dimension to take advantage of streaming bandwidth, disk's peak performance. This is a high price to pay, since the difference between streaming bandwidth and non-sequential accesses is at least two orders of magnitude.

This paper introduces *MultiMap*, a new mapping model that properly matches N-dimensional datasets to physical characteristics of storage. Without breaking existing abstractions, MultiMap offers streaming disk bandwidth for access to one dimension and outperforms all existing schemes when accessing data blocks for other dimensions. MultiMap maps such blocks on nearby disk tracks to avoid rotational latency and to incur only the minimum seek time. This access is possible thanks to high disk track densities, which make the cost of seeking to nearby tracks (typically within 10-20 cylinders of the current position) nearly equal. MultiMap gives applications the flexibility to choose a specific block on any of these tracks whereby such blocks, called *adjacent blocks*, can be accessed without any rotational latency *and* for the same positioning cost.

We describe a general model for MultiMap and offer an API that exposes these adjacent disk blocks. This API hides from applications disk-specific details and maintains the abstractions established by existing storage interfaces. We evaluate MultiMap analytically and on a prototype implementation that uses a logical volume of real disk drives with 3-D and 4-D datasets The 4-D dataset represents an earthquake simulation workload. Our comparison of MultiMap to naive linearizing mapping scheme and a space-filling Hilbert curve shows that MultiMap speeds up a variety of spatial range queries on average by $1.3\times$–$2\times$.

This remainder of the paper is organized as follows. Section 2 describes related work. Section 3 defines adjacent blocks and describes the underpinnings for enabling efficient accesses to these blocks on nearby tracks. Section 4 describes an algorithm for mapping multidimensional datasets. Section 5 evaluates the performance of the system for 3-D, 4-D, and 8-D datasets.

2 Related work

There is a large amount of research on access methods for multidimensional datasets. Under the premise that disks are one-dimensional devices (as suggested by its abstraction of a linear space of fixed-size blocks), various multidimensional access methods have been proposed in literature. These methods attempt to order multidimensional data so that spatial locality can be preserved as much as possible on the one-dimensional disk abstraction. With the premise that objects that are close in the multidimensional space are also physically close in the disk space, they assume that fetching them will not incur inefficient random-like accesses. The property of preserving spatial locality is often called *clustering* in multidimensional applications.

Multidimensional access methods, which can be roughly classified into point access methods and spatial access methods, include hash structures (for example, grid files), hierarchical structure such as various trees, and space-filling curves. These spatial access methods, as surveyed by Gaede et al. [9], typically assume that all disk accesses to the tree nodes are random, and thus equally costly. By contrast, MultiMap deviates from that assumption, and achieves better performance.

Space-filling methods such as Gray-coded curves [7], Hilbert curve [11], and Z-ordering [15] use linear mapping: they traverse the multidimensional dataset linearly and impose a total order on the dataset when storing data on disks. Moon et al. [14] analyze the clustering properties of the Hilbert curve and indicate that it has better clustering than Z-ordering, and hence better I/O performance in terms of per-node average access time. Therefore, we compare the performance of MultiMap against Hilbert curve mapping.

The existing multidimensional data access methods proposed thus far focus are applicable at the database level and do not pay much attention to the specifics of the underlying storage devices. They assume a disk drive is a linear storage space with constant (average) access time per block. Recently, researchers focused on the lower level of the storage system in an attempt to improve performance of multidimensional queries [10, 19, 22, 23, 29]. Part of the work revisits the simple disk abstraction and proposes to expand the storage interfaces so that the applications can be more intelligent. Schindler et al. [19, 21] explored aligning of the access patterns to the disk drive track-boundary to get rid of the rotational latency. Gorbatenko et al. [10] and Schindler et al. [22] propose a secondary dimension on disks, which has been utilized to create a more flexible database page layout [26] for two-dimensional tables. Others have studies the opportunities of building two dimensional structures to support database applications with new alternative storage devices, such as MEMS-based storage devices [23, 29]. The work described in this paper further leverages the underlying characteristics of the storage device to efficiently map an arbitrary number of dimensions on logical volumes comprised of modern disk drives.

Another body of related research focuses on how to decluster multidimensional datasets [1, 3, 4, 8, 13, 16, 17] to optimize spatial access methods [12, 25] with multiple disks and improve the throughput parallel system. While all the existing declustering schemes assume that disk drives are purely one-dimensional, making it impossible to find a total ordering of blocks that preserves spatial locality for accesses to multidimensional datasets, our paper challenges this assumption. We exploit two important observations of modern disk drive technologies and show how to map such datasets for efficient access for *all* dimensions across disks in a multi-disk storage array. Naturally, much of the discussion focuses on how to organize multidimensional data so that spatial locality on a *single* disk is preserved. We discuss alternative ways for mapping data across multi-disk volumes and evaluate one such approach. However, exploring the tradeoffs of different declustering

techniques and their effects on the performance of specific spatial queries is beyond the scope of this paper.

3 Breaking one-dimensionality assumptions

Despite the established view that disk drives can efficiently access only one-dimensional data mapped to sequential logical blocks, modern disk drives allow for efficient access in more than one dimension. This fundamental change is based on two observations of technology in modern disks:

1. Short seeks are dominated by the time the head needs to settle on a new track.
2. It is possible to identify and thus access blocks with no rotational latency after a seek.

By combining these two observations, it is possible to construct a mapping and access patterns for efficient access to multidimensional data sets with the current abstractions of storage systems as a linear address space of fixed-size blocks. This section details the technological underpinnings that make this mapping possible. It presents an algorithm for selecting the proper blocks for mapping and efficient access to multidimensional datasets. Finally, it describes an API abstraction that allows database storage managers to take advantage of such access without revealing low-level disk details or breaking current storage interfaces with their established abstractions.

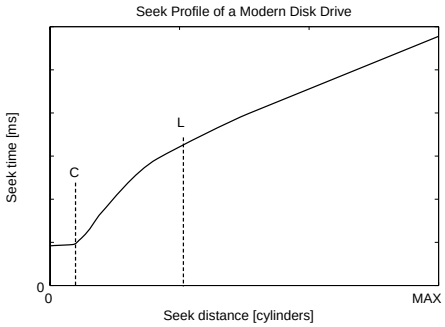
3.1 Adjacent blocks

It is well understood that disk drives are optimized for access to the next sequential disk block on the same track and that the time to access any other disk block is a function of that block's distance from the current head position. However, recent trends in track density and mechanical performance change this assumption. They define a set of blocks with respect to a starting location, here termed *adjacent*, whose cardinality is D. All of these D adjacent blocks can be accessed with constant amount of positioning time and without any rotational latency even though they are not equidistant from that starting location.

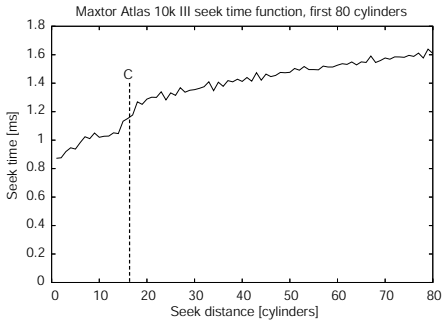
In addition to incurring constant positioning cost for accessing non-equidistant adjacent blocks, such access does not incur any rotational latency and hence results in efficient access. The cost is equal to the constant overhead of moving the disk head to any of the destination block in the set. When these adjacent are properly identified and exposed to applications, they can ensure efficient access to multidimensional data sets. To understand why disk drives can efficiently access adjacent blocks and what determines the value of D, it is necessary to describe in some detail certain disk drive characteristics and their trends, which is the topic of the next section.

3.2 Disk technology trends defining adjacent blocks

Figure 1(a) shows a conceptual view of seek time as a function of cylinder distance for modern disks. In the past the seek time for short distances between one and, say, L cylinders was approximately a square root of the seek distance [18]. In more recent disks, seek time is almost constant for distances of up to C cylinders, while it increases for distances between C and L cylinders as



(a) Conceptual seek profile.



(b) Actual seek profile of a modern disk drive.

Figure 1: **Typical seek profile of modern disk drives.** The seek profile consists of three distinct regions. For cylinder distances less that C, the seek time is constant, for distances between C and L, the seek time is approximately a square root of the seek distance, and for distances greater than L, the seek time is a linear function of seek distance in cylinders. To illustrate the trends more clearly, the X-axis in Figure 1(a) is not drawn to scale.

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

Table 1: **Adding extra conservatism to the base skew of 51° for the Atlas 10k III disk.**

before. Thus, seeks of up to C cylinders are dominated by the time it takes a disk head to settle on the new track. Figure 1(b) shows an example of a seek profile of a disk introduced in 2002 for which $C = 17$ and settle time is slightly less than 1 ms.

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. At lower track densities (e.g., in disks introduced before 2000), only a single cylinder can be reached within the settle period. However, with the large increase in track density since 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 [2], as shown in Figure 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 improves very little.

Given that each cylinder consists of R tracks, equal to the number of recording surfaces, we define the disk drive's *depth*, D, as the total number of tracks that are accessible within the settle time, where $D = R \times C$. We refer to these D tracks as being *adjacent*. Each of these tracks then contains one adjacent block that can be accessed from a starting block within the constant settle time, without any additional rotational latency. The Maxtor Atlas 10k III disk from our example in Figure 1(b) has $C = 17$, and up to 8 surfaces for a total capacity of 73 GB. Thus, it has 136 adjacent blocks according to our definition and $D = 136$.

Figure 3 shows a notational drawing of the layout of adjacent blocks on disk. For a given starting block, there are D adjacent disk blocks, one in each of the D adjacent tracks. During the settle time, the disk rotates by a fixed number of degrees, W, determined by the ratio of the settle

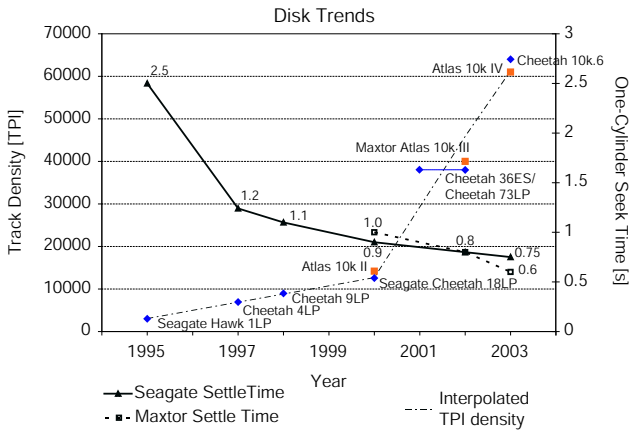


Figure 2: **Disk trends for 10,000 RPM disks.** Seagate introduced the Cheetah disk family in 1997 and Quantum/Maxtor introduced its line of Atlas 10k disks in 1999. Notice the dramatic increase in track density, measured in Tracks Per Inch (TPI), since 2000. The most recent disks introduced in 2004 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.

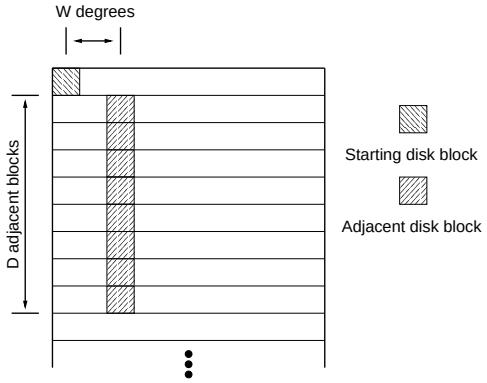


Figure 3: **An example of adjacent tracks.**

time to the rotational period of the disk. For example, with settle time of 1 ms and the rotational period of 6 ms (i.e., for a 10,000 RPM disk), $W = 60$ degrees $(1/6 \times 360^\circ)$. Therefore adjacent blocks are always physically offset from the starting block by the same amount.

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 suffer full revolution delay. Adding conservatism to the value of W increases the number of tracks, D, that can be accessed within the settle time at the cost of added rotational latency, as shown in Table 1 for the Atlas 10k III disk.

Accessing successive adjacent disk blocks enables *semi-sequential* disk access [10, 22], which is the second most efficient disk access method after pure sequential access. The delay between each access is equal to a disk head settle time, which is the minimum mechanical positioning delay the disk can offer. However, the semi-sequential access in previous work utilizes only one of the adjacent blocks for efficiently accessing 2-D data structures. This work shows how to use up to D adjacent blocks to improve performance for multidimensional datasets.

```
/* Find an LBN adjacent to lbn and step tracks away */
L := GETADJACENT(lbn, step) :
   /* Find the required skew of target LBN */
   target_skew := (GETSKEW(lbn) + W) mod 360

   /* Find the first LBN in target track */
   base_lbn := lbn + (step * T)
   {low, high} := GETTRACKBOUNDARIES(base_lbn)

   /* Find the minimum skew of target track */
   low_skew := GETSKEW(low)

   /* Find the offset of target LBN from the start of target track */
   if (target_skew > low_skew) then
      offset_skew := target_skew - low_skew
   else
      offset_skew := target_skew - low_skew + 360
   end if

   /* Convert the offset skew into LBNs */
   offset_lbns := (offset_skew/360) * T
   RETURN(low + offset_lbns)

/* Find the physical skew of lbn, measured in degrees */
A := GETSKEW(lbn)

/* Find the boundaries of the track containing lbn */
{L, H} := GETTRACKBOUNDARIES(lbn)


T:  The track length
W:  Skew to add between adjacent blocks, measured in degrees
```

Figure 4: **Algorithm for the** `GETADJACENT` **function.**

3.3 An algorithm for identifying adjacent blocks

The current disk interface based on the abstraction of a storage device as a linear array of fixed-sized blocks does not expose adjacent blocks, even though such information is readily available inside modern disk drives. In particular, the firmware functions for mapping logical blocks to physical sectors use the skew W and the logic of functions for scheduling requests can identify blocks that can be accessed without incurring any rotational latency.

This section describes an algorithm for identifying the logical block numbers (*LBN*) of adjacent disk blocks for a given block in the absence of the proper storage interface functions. The algorithm uses a detailed model of the low-level disk layout taken from a storage system simulator called DiskSim [5]. The parameters can be extracted from SCSI disks by previously published methods [20, 28]. The algorithm uses two functions that abstract the disk-specific details of the disk model: `GETSKEW(lbn)`, which returns the physical angle between the physical location of an *LBN* on the disk and a "zero" position, and `GETTRACKBOUNDARIES(lbn)`, which returns the first and the last *LBN* at the ends of the track containing `lbn`.

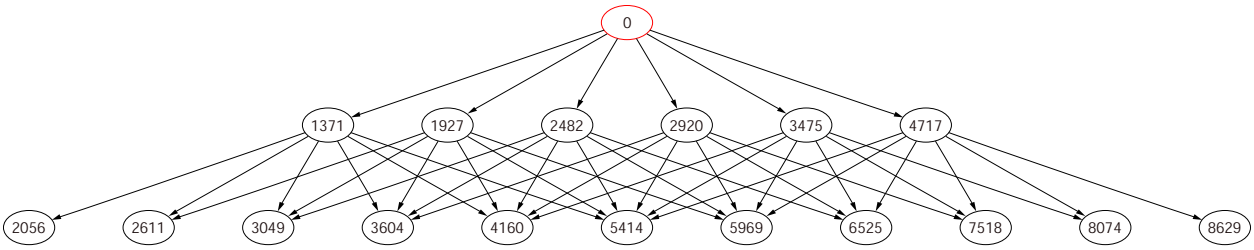


Figure 5: **Adjacency graph for*LBN* 0 of Atlas 10k III.** Only two levels adjacent blocks are shown. The *LBN*s are shown inside nodes.

For convenience, the algorithm also defines two parameters. First, the parameter T is the number of disk blocks per track and can be found by calling `GETTRACKBOUNDARIES`, and subtracting the low *LBN* from the high *LBN*. Second, the parameter W defines the angle between a starting block and its adjacent blocks, as described in Section 3.2. This angle can be found by calling the `GETSKEW` function twice for two consecutive *LBN*s mapped to two different tracks and computing the difference; disks skew the mapping of *LBN*s on consecutive tracks by W degrees to account for settle time and to optimize sequential access performance.

The `GETADJACENT` algorithm shown in Figure 4 takes as input a starting *LBN* (`lbn`) and finds the adjacent *LBN* that is W degrees ahead and `step` tracks away. Every disk block has an adjacent block within the D closest tracks, so the entire set of adjacent blocks is found by calling `GETADJACENT` for increasing values of `step` from 1 to D.

3.4 Hiding low-level details from software

While this paper advocates exposing to applications adjacent *LBN*s, it is unrealistic to burden application developers with such low-level details as settle time, physical skews, and data layout to implement the algorithm described above. The application need not know the reasons 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. Our prototype implementation of a logical volume manager described in Section 5 takes the latter approach when creating logical volumes from one or more disks.

A useful way to analyze the adjacency relationships between disk blocks is by constructing adjacency graphs, such as that shown in Figure 5. The graph nodes represent disk blocks and the edges connect blocks that are adjacent. The graph in the figure shows two levels of adjacency: the root node is the starting block, the nodes in the intermediate level are adjacent to that block, and the nodes in the bottom level are adjacent to the blocks in the intermediate level. Note that adjacent sets of adjacent blocks (i.e., those at the bottom level of the graph) overlap. This overlap limits the overall dimensionality of the disk space to two, and is discussed further in Section 4. For brevity, the graph shows only the first 6 adjacent blocks (i.e., $D = 6$), even though D is considerably larger. Thus, higher-dimensional data can be efficiently folded into the 1-D abstraction of the disk.

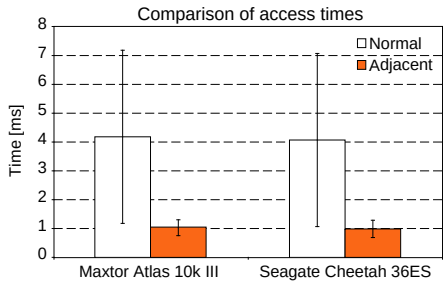


Figure 6: **Quantifying access times.** This graph compares the access times to blocks located within C cylinders. For both disks, the average rotational latency is 3 ms. For the Atlas 10k III disk, C = 17 and seek time within C ranges from 0.8 ms to 1.2 ms. For the Cheetah 36ES disk, C=12 and seek time ranges from 0.7 ms to 1.2 ms.

3.5 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, 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 taking advantage of accesses to adjacent blocks outperforms traditional systems. As shown in Figure 6, such 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, while the access time variability is much smaller for the Adjacent case; it entirely due to the difference in seek time within the C cylinders, as depicted by the error bars.

4 Mapping multidimensional data onto disks

Given the underpinnings for efficient access to adjacent blocks, we now discuss how to create a general model for mapping N-dimensional datasets to these adjacent blocks. We first discuss the intrinsic dimension of the adjacency model. Then, we show on examples of mapping 2-D, 3-D and 4-D datasets its basic ideas, and finally present a general algorithm for mapping N-dimensional datasets to adjacent blocks.

4.1 Intrinsic dimension of adjacency model

At a first glance, one might assume that a disk using the adjacency model with a depth of D can have D+1 dimensions. It turns out to be more complicated. To obtain an insight to the dimensionality of the adjacency model, we apply the power law method [24] to calculate its intrinsic dimension.

The *power law* formula $N(r) = r^a$ describes the internal structure of a dataset. Here, r denotes the steps from a point and $N(r)$ counts the average number of neighbors within r. The exponent a is referred to as intrinsic or fractal dimension and usually changes with different scales of r.

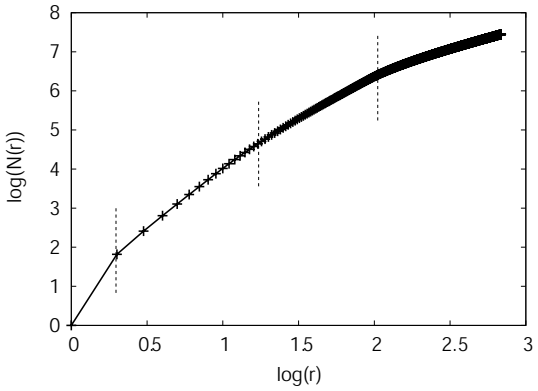


Figure 7: **Intrinsic dimensionality of the adjacency model.** Depth, D is 64, track length, T, is 686, and the skew, W, is 60°.

The intrinsic dimension is very helpful to understand complex datasets where traditional distributions are not applicable. It indicates how fast the dataset grows when increasing the range. For example, lines have intrinsic dimensions of 1 because the number of neighbors increases linearly with r. In practice, $N(r)$ and r are often plotted in log-log scales, so the slope of the curve is the "intrinsic dimension". We apply the fractal analytical method to the adjacency model to get an insight of its dimensionality.

In the following discussion, we count both the adjacent blocks (as shown in Figure 3) and the consecutive block on the same track as the neighbors of a given block. Varying the value of r from 1 to the track length, we plot the log-scale curve of $N(r)$ vs. r in Figure 7. The depth D of the adjacency model used in Figure 7 is 64.

Figure 7 shows that the slope decreases a the range r increases. We can split the curve into four regions based on the changing trends of the slope a. The initial region starts from $r = 1$, and ends at $r = 2$ with a slope of around 6. Next, the slope drops abruptly to 3 starting from $r = 2$, marking the beginning of the first plateau. The dimension of 3 lasts until $r = 9$. After that, it changes gradually into the second plateau with a slope of 2 until $r = 95$. The third region starts from $r = 95$, where the slope decreases gradually from 2 to 1 until it reaches the last region.

The changes of the slope, that is the intrinsic dimension of the adjacency model, are due to the complex and changing overlapping patterns among different adjacency sets at different r. The larger the r, the higher the degree of overlap, and the smaller the number of new neighbors brought into the neighbor set at each step. Therefore, the slope decreases with the increase of the range. Without discussing more details about the intrinsic dimension of the adjacency model, we observe that the adjacency model has an inherent limitation on the dimensions it can support. To get around the restriction in practice, we rely on the fact that to build an N-dimensional space, one block does not need more than N neighbors. For instance, to create a 3-D space, each block only needs three neighbors, even though the value of D is very large, as discussed in Section 3. If we choose the neighbors carefully, we can build a mapping for N-dimensional datasets with large N, which is the topic of the next section.

Notation	Definition
T	disk track length
D	depth of the adjacency model
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
σ-neighbor	sequential block on the same track
α-neighbor	adjacent block on another track
i-th α-neighbor	i-th adjacent block

Table 2: **Notation definitions.** For Dim, S, and K, $0 \leq i \leq N-1$.

4.2 Mapping examples

To map an N-dimensional dataset onto disks, we first impose an N-dimensional grid onto the dataset. Each cell in the grid will be assigned to an N-dimensional coordinate mapped to a single disk block with its own *LBN* address and whose size is typically 512 bytes. In the rest of the paper, a *point* refers to a cell, even though a single cell can actually contain multiple points of the original discrete N-dimensional space.

We next illustrate MultiMap through three concrete examples for 2-D, 3-D, and 4-D datasets, followed by a formal definition and a general algorithm. The notation used here is listed in Table 2. In the following examples, we assume that $T = 8$, $D = 4$, and the disk blocks start from *LBN* 0. As you will notice, the MultiMap algorithm need not know or use the value of skew W. This value is abstracted away by the `GETADJACENT` function, which in turn uses the `GETSKEW` function, which in turn builds upon the abstraction of the disk firmware mappings of logical block (*LBN*s) to physical sectors, which uses W.

4.2.1 Example of 2-D mapping

Figure 8 shows how a 2-D rectangle is mapped to the disk. Each row in the sketch represents a disk track and each cell corresponds to a disk block whose *LBN* is the number inside the box. In this example, we have three tracks, each having 8 blocks. For simplicity, we choose a 2-D space of the size $S_0 = T = 8$ for Dim_0 and $S_1 = 3$ for Dim_1.

The first dimension (Dim_0) is mapped sequentially to consecutive *LBN*s mapped to the same track, which ensures that accesses along Dim_0 get the performance of streaming bandwidth. The second dimension (Dim_1) is mapped to the first α-neighbors (that is, the first adjacent blocks) For example, *LBN* 8 is the first adjacent block of *LBN* 0, and *LBN* 16 is the first adjacent block of *LBN* 8. Although not as efficient as sequential access, he semi-sequential disk access along Dim_1 is still much more efficient than the accesses for the other schemes.

Now the rectangle is mapped to 3 contiguous tracks. The spatial localities on Dim_0 and Dim_1 are preserved on the disk. Note that in the 2-D mapping, we only leverage sequential blocks and the first α-neighbors.

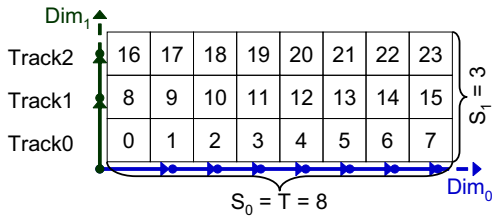


Figure 8: **Mapping 2-d dataset to disks.** The first dimension is mapped to the track; the second dimension is mapped to the sequences of the first α-neighbors.

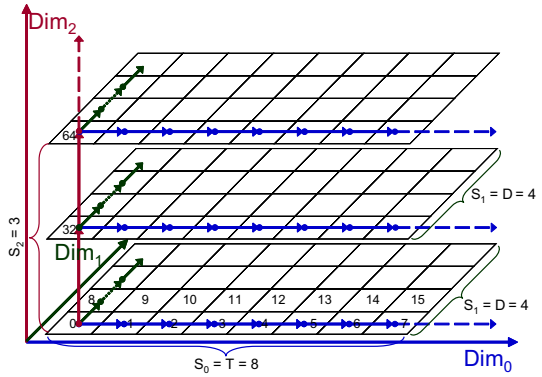


Figure 9: **Mapping 3-D dataset to disks.** Each surface is a 2-D structure as in Figure 8. The third dimension is mapped to the sequences of 4th-α neighbors.

4.2.2 Example of 3-D mapping

In this example, we use a 3-D dataset with the following parameters: $S_0 = T = 8$, $S_1 = 4$, and $S_2 = 3$. As shown in Figure 9, Dim_0 is still mapped to the track direction as in the 2-D case. To map the other two dimensions, we use $(S_2 \times S_1)$ adjacent tracks, partition them into S_2 pieces each occupying S_1 consecutive tracks (one piece corresponds to one layer in Figure 9). For each piece, we apply the same method from the 2-D example for mapping a $S_0 \times S_1$ surface. Then we "fold" these S_2 pieces along the direction of Dim_2 to build the third dimension.

A crucial step here is that the points along the third dimension are mapped to a sequence of the S_1-th α-neighbors. For instance, *LBN* 32 is the 4th α-neighbor of *LBN* 0, and *LBN* 64 is the 4th α-neighbor of *LBN* 32. Since $D = 4$, access along Dim_2 fetches adjacent blocks. So it has the same performance as the access along Dim_1. Therefore, the spacial locality of Dim_2 is also preserved (the locality of Dim_1 is guaranteed by the 2-D mapping). Note that that the size of S_1 is restricted by the value of D; to guarantee efficient along Dim_2 as well. We will discuss the case where $S_1 > D$ in the general mapping algorithm in the next section. The resulting 3-D mapping occupies $(S_1 \times S_2)$ contiguous tracks.

4.2.3 Example of 4-D mapping

In the 4-D example, shown in Figure 10, we use a dataset of size ($S_0 = T = 8$, $S_1 = 2$, $S_2 = 2$, $S_3 = 3$). Note that the value of S_1 is different from the 3-D example. As before, Dim_0 is mapped along the track. To map the other three dimensions, we use $(S_3 \times S_2 \times S_1)$ consecutive tracks and partition them into S_3 pieces each mapped to $(S_2 \times S_1)$ consecutive tracks. The same 3-D mapping

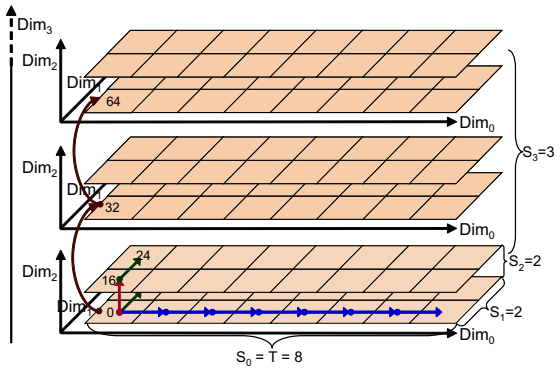


Figure 10: **Mapping 4-d dataset to disks.** Fold K_3 3-D structures as in Figure 9 to build the fourth dimension. The fourth dimension is mapped to the sequences of adjacent blocks with a step of $K_1 \times K_2$.

method is applied on each of the pieces.

We fold again the S_3 pieces to get the fourth dimension Dim_3. The points along Dim_3 are therefore mapped to a sequence of $(S_1 \times S_2)$-th α-neighbors. This time *LBN* 32, the 4th adjacent block of *LBN* 0, becomes the "next" block to *LBN* 0 along Dim_3. Access along Dim_3 involves fetching adjacent blocks, so it has same performance as fetching blocks along Dim_1 and Dim_2. Notice that in the 4-D example, S_1 and S_2 must satisfy the restriction: $(S_1 \times S_2) \leq D$.

4.3 Definition of the mapping problem

Following the discussion in the previous section, it becomes clear that mapping to the space of N dimensions is an iterative extension of the problem of mapping $N-1$ dimensions. In this section, we give the formal definition of the mapping problem and a general algorithm, MultiMap, to map N-dimensional datasets and discuss its limitations.

The mapping problem we are trying to solve is defined as follows: given D adjacent blocks and track length, T, map an N-dimensional dataset to the disk, so that the spatial locality of the dataset can be preserved. By preserving locality, we mean that neighbors of one point P in the original space will be stored in the blocks that are *adjacent* to the block storing P. In the following discussion, we assume that the size of the dataset is smaller than the disk capacity and relax it Section 4.5, when we describe how to map datasets that are use logical volumes built out of multiple disks.

Before describing the mapping of any N-dimensional grid to one or more disks, we first discuss how to map a single *basic cube* to a single disk. Basic cube is a cube in the N-dimensional space with a size of $\prod_{i=0}^{N-1} K_i$, satisfies the following requirements:

$$K_0 \leq T \quad (1)$$

$$K_{N-1} \leq S_{N-1} \quad (2)$$

$$\prod_{i=1}^{N-2} K_i \leq D \quad (3)$$

Equation 1 restricts the length of the first dimension of the basic cube to track length. Equation 2 indicates that the last dimension of the basic cube and the original dataset can have the same

```
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   = the first LBN of the basic cube
            = the LBN storing point (0, ..., 0)
```

Figure 11: **Mapping a point in space to an *LBN*.**

size, subject to the disk capacity. The last equation sets a limit on the lengths of K_1 to K_{N-2}. The volume of the $(N-2)$-dimensional space, that is $\prod_{i=1}^{N-2} K_i$, should be less than D, otherwise the locality of the last dimension cannot be preserved because accessing the consecutive points along the last dimension cannot be done within α.

4.4 Mapping the basic cube

We map the basic cube in the following way: Dim_0 is mapped to the track direction; Dim_1 is mapped to the sequence of the first α-neighbors; ...; $Dim_{i+1}(1 \leq i \leq N-2)$ is mapped to a sequence of $(K_1 \times K_2 \ldots \times K_i)$-th α-neighbors.

The above procedure is generalized in the MultiMap algorithm shown in Figure 11. The inputs of `MAP` are the coordinates of a point in the basic cube, the output is the *LBN* to store that point. `MAP` starts from the point $(0,0,\ldots,0)$. Each inner iteration proceeds one step along Dim_i, which on a disk corresponds to jump over $(K_1 \times K_2 \ldots \times K_{i-1})$ adjacent blocks. Therefore each iteration of the outer loop goes from point $(x[0],\ldots,x[i-1],0,\ldots,0)$ to point $(x[0],\ldots,x[i-1],x[i],0,\ldots,0)$.

The mapping algorithm has three important properties. First, Dim_0 is mapped to the disk track so that accesses on this dimension enjoy the best performance of the sequential scan. Second, all the other dimensions are mapped to a sequence of adjacent blocks with different steps. Any two neighboring blocks on all dimensions are mapped to adjacent blocks at most D tracks away. (see Equation 3). Requesting these (non-contiguous) blocks results in semi-sequential accesses that are much more efficient than accessing blocks mapped within D tracks at random. And third, the mapping algorithm preserves the spatial locality within the basic cube because neighbors in the cube are mapped to neighboring blocks (σ-neighbor and α-neighbor) within D tracks.

We require that $K_i \geq 2$ $(0 \leq i \leq N-1)$, otherwise the basic cube deteriorates to a cube in a space with less than N dimensions. Usually, we consider the basic cubes with equal length along dimensions $Dim_1,\ldots,Dim_{N-2}$, that is $K_1 = K_2 = \ldots = K_{N-2} = K$. With these assumptions, derived from Equation 3, we have

$$N \quad \leq \quad \lfloor 2 + \log_K D \rfloor \ (K \geq 2) \tag{4}$$

$$N_{max} = \lfloor 2 + \log_2 D \rfloor \quad (5)$$

For modern disks, D is typically on the order of hundreds, allowing mapping for space of more than 10 dimensions.

4.5 Mapping large dataset

The algorithm described above maps a basic cube to within D tracks of a single disk and determines the proper values for K_1 to K_{N-2}. As Equation 3 indicates, D is the maximum volume of the $(N-2)$-D cube. The track length, T, restricts the size of the basic cube along the first dimension. Finally, the total number of tracks, R_{max}, on a disk set an upper bound for K_{N-1}, according to the following inequality:

$$K_{N-1} \leq \left\lfloor \frac{R_{max}}{\prod_{i=1}^{N-2} K_i} \right\rfloor \quad (6)$$

If the length of the dataset's, and hence basic cube's, Dim_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 larger than T and set it as Dim_0. When $|Dim_0| \bmod T = A$ and $A = 0$, we have total coverage and no internal fragmentation of our *LBN* space. When $A > 0$, there will be at most A^N unallocated *LBN*s due to internal fragmentation when the entire dataset (i.e., all basic cubes) is allocated.

The basic cube defined in Section 4.3 serves as an allocation unit when we map larger datasets to the disks. If the original space is larger than the basic cube, we partition it into basic cubes to get a new N-dimensional cube with a reduced size of

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

To map each basic cube to the disk, we can apply any traditional algorithm used to linearize N-dimensional datasets. The only difference is the size of the "point" is many more (non-contiguous) *LBN*s instead of one disk block.

Given the mapping of basic cubes, spatial locality for points in different basic cubes is not preserved, and the cost of a jumps between basic cubes will be a seek that is slightly larger than settle time with some rotational latency. However, when requests for the destination basic cube are issued together, this rotational latency is still much smaller than 1/2 revolution; the firmware disk scheduler can reorder these requests to minimize overall rotational latency. Despite these discontinuities, MultiMap still outperforms existing solutions, as shown in Section 5.

When using multiple disks, we 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 the way how multidimensional data is organized on a single disk. The MultiMap scheme thus works nicely with the existing declustering methods and enjoys an increase in throughput brought by parallel I/O operations. In the reminder of our discussion, we focus on the performance of MultiMap on a single disk, with the understanding that multiple disks can deliver I/O throughput scalable with the number of disks. The access latency for each disk, however, remains the same regardless of the number of disks.

5 Evaluation

We evaluate *MultiMap*'s performance using a prototype implementation that runs queries against 3-D and 4-D datasets stored on real disks. In addition, we use an analytical model to estimate *MultiMap*'s performance for 8-D datasets.

We compare *MultiMap* to *Naive* and *Hilbert* schemes. *Naive* linearizes an N-dimensional space along a chosen dimension, called major order. In all experiments, we choose Dim_0 as the major order. *Hilbert* mapping orders the N-dimensional points according to their Hilbert curve values.

5.1 Experimental setup

The experiments are conducted on a two-way 1.7 GHz Pentium 4 Xeon workstation running Red-Hat 8.1 distribution with Linux kernel 2.4.24. The machine has 1024 MB of main memory and is equipped with one Adaptec Ultra160 Wide SCSI adapter connecting two 36.7 GB disks: Seagate Cheetah 36ES disk (ST336706LC) and Maxtor Atlas 10k III disk (KU036J4). The disks were introduced in late 2001 and early 2002, respectively and both have two platters (four surfaces, $R = 4$). The Cheetah disk maps logical blocks across 26,302 cylinders with track sizes between 736–574 sectors, while the Atlas disk uses 31,002 cylinders with track sizes between 686–396 sectors.

Our prototype system consists of two software components running on the same host machine: a logical volume manager (LVM) and a database storage manager. In a production system, the LVM would likely reside inside a storage array system separate from the host machine. The LVM exports a single logical volume mapped across multiple disks and identifies adjacent blocks using the algorithm described in Section 3.3. It uses a DiskSim disk model [5] whose parameters we obtained from our disks with an on-line extraction tool [20]. The database storage manager maps multidimensional datasets by utilizing the `GETADJACENT` and `GETTRACKBOUNDARIES` functions provided by the LVM. Based on the query type, it issues appropriate I/Os to the LVM, which then breaks these I/Os into proper requests to individual disks.

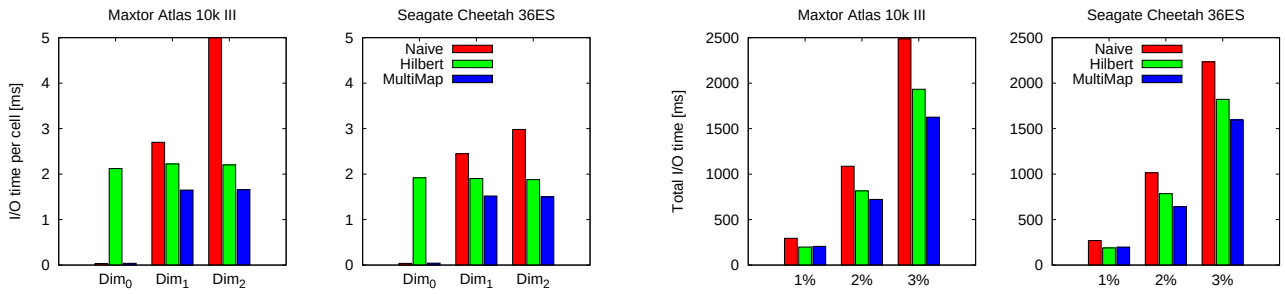
The datasets used for our experiments are stored on multiple disks in our LVM. Akin to commercial disk arrays, the LVM uses disks of the same type and utilizes only a part (slice) of the disk's total space [6]. The slices in our experiments are slightly less than half of the total disk capacity and span one or more zones (cylinders with equal number of sectors per track).

The LVM generates requests to all the disks during our experiments, but we report performance results from a single disk. The reason is that we examine 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.

5.2 Query workloads

Our experimental evaluation compares two classes of queries for 3-D and 4-D datasets. *Beam queries* are one-dimensional queries retrieving data points along the lines parallel to the dimensions. A beam query along Dim_k can be described in SQL-like syntax as follows:

`SELECT` all points X $(x_0, \ldots, x_{N-1})$ `FROM` *dataset* `WHERE` $x_i = q_i$, `AND` $0 \leq i \leq N-1$ `AND` $i \neq k$



(a) **Beam queries.** (b) **Range queries.**

Figure 12: **Performance of the 3-D dataset.**

The $(x_0,\ldots,x_{N-1})$ are the coordinates of point X in the N-dimensional space. And the query predicate says all $(x_0,\ldots,x_{N-1})$ coordinates are fixed except for x_k.

Range queries, called *p%-length cube queries*, fetch an N-dimensional cube with an edge length equal to the p% of the dataset's edge length. With $(q_0,\ldots,q_{N-1})$, $(Q_0,\ldots,Q_{N-1})$ as its lower-left and upper-right corners, a cube query has the following expression:

`SELECT` all points X $(x_0,\ldots,x_{N-1})$ `FROM` *dataset* `WHERE` x_i `BETWEEN` q_i `AND` Q_i `AND` $(0 \leq i \leq N-1)$

5.3 3-D dataset

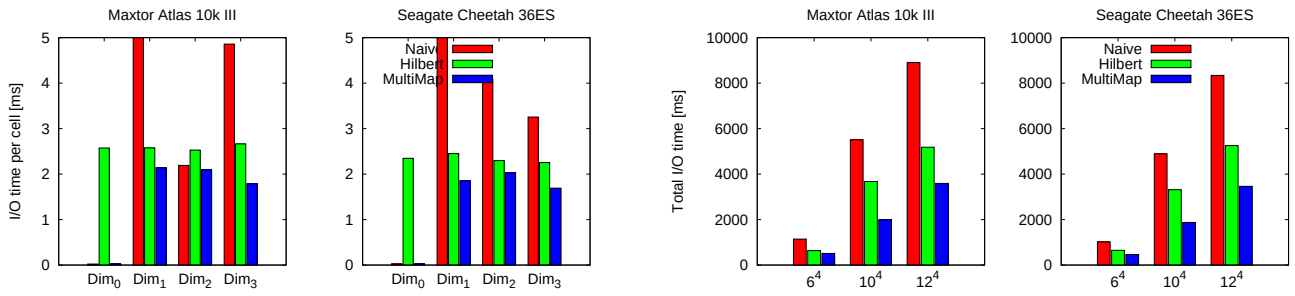
For these experiments, we use a dataset with $1024 \times 1024 \times 1024$ cells. We partition the space into chunks of at most $259 \times 259 \times 259$ cells that fit on a single disk. For both disks, *MultiMap* uses $D = 128$ and conservatism of 30°.

Beam queries. The results for beam queries are presented in Figure 12(a). We run beam queries along all three dimensions, Dim_1, Dim_2, and Dim_3. 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 that 1% of the reported times. Each run selects a random value between 0 and 258 for the two fixed dimensions and fetches all points (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 as neighboring cells along Dim_1 and Dim_2 are stored respectively 259 and 67081 (259×259) blocks apart. 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 both disks) followed by rotational latency. Even though the jumps for Dim_2 are evenly spaced on both disks, the overall performance of the Dim_2 on the Cheetah disk is better. The rotational offset of two consecutive *LBN*s on the Atlas disk results in rotational latency of 3.8 ms on top of the 1.3 ms settle time, while it is only 2.1 ms for the Cheetah disk.

True to its goals, the *Hilbert* scheme achieves balanced performance across all dimensions. It sacrifices the performance of sequential accesses that *Naive* can achieve for Dim_0, resulting in 2.0 ms per cell vs. 0.035 ms for *Naive* (almost 57× worse). *Hilbert* outperforms *Naive* for the other two dimensions, achieving 22%–136% better performance for the two different disks.

The *MultiMap* scheme 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



(a) **Beam queries.** (b) **Range queries.**

Figure 13: **Performance of the 4-D earthquake dataset.**

cube to the next one. More importantly, *MultiMap* outperforms *Hilbert* for Dim_1 and Dim_2 by 25%—35% and *Naive* by 62%–214% for the two disks. Finally, notice that *MultiMap* achieves almost identical performance on both disks unlike *Hilbert* and *Naive*. That is because these disks have comparable settle times, which affect the performance of accessing adjacent blocks for Dim_1 and Dim_2. We observed that by using a smaller conservatism value the average I/O times for the Atlas disk is lowered by another 2–10% (not shown in the graphs).

Range queries. The first set of three bars, labeled 1_per, in Figure 12(b) shows the performance of 1%-length cube queries expressed as their total runtime. As before, the performance of each scheme follows the trends observed for the beam queries. *MultiMap* improves the query performance by 37% and 11% respectively compared to *Naive* and *Hilbert*.

Both *MultiMap* and *Hilbert* outperform *Naive* as it can not employ sequential access for range queries. *MultiMap* outperforms *Hilbert* as it must fetch some cells from physically distant disk blocks, although they are geographically close. These jumps make *Hilbert* less efficient compared to *MultiMap*'s semi-sequential accesses.

To examine the sensitivity of the cube query size, we also run 2%-length and 3%-length cube queries, whose results are presented in the second and third bar sets in Figure 12(b). The trends are similar, with *MultiMap* outperforming *Hilbert* and *Naive*. The total run time increases because each query fetches more data.

5.4 4-D earthquake simulation dataset

To evaluate performance for 4-D data sets, we use 4-D earthquake simulation data. In earthquake simulations [27], the logical dataset is a 3-D region of the earth, modeled as a set of discrete points, organized in a 3-D grid. The simulation computes the motion of the ground at each point, for a number of discrete time steps. The 4-D simulation output contains a set of 3-D grids, one for each step.

The dataset we evaluate here models earthquake activity in a 14 km deep slice of earth of a 38×38 km area in the vicinity of Los Angeles. The grid contains approximately one node per 200 m (unevenly distributed) and 2000 time steps. Given this dataset size and packing of up to 10 nodes per cell, this grid translates into a 4-D space of $2000 \times 64 \times 64 \times 64$ cells for a total size of 250 GB of data. We choose time as the major order to the Naive and the MultiMap schemes. Similar to the 3-D dataset, we partition the space into basic cubes of $530 \times 32 \times 32 \times 32$ cells to fit in a single disk. We query the 4-D dataset in two ways: *Time-varying queries*, or beam queries,

3-D experiments						
	Naive			MultiMap		
Query	*Measured*	*Calculated*		*Measured*	*Calculated*	
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 3: **Comparison of predicted and measured I/O costs.**

retrieve the velocity of a particular point for multiple time-steps. *Space-varying queries*, or range queries, retrieve the velocity for one time step, of all the nodes in a specific region.

The results, presented in Figure 13, exhibit the same trends as the 3-D experiments. The MultiMap model again achieves the best performance for all beam and range queries. The range queries, In Figure 13(a), the unusually good performance of *Naive* on Dim_2 is due to a particularly fortunate mapping that results in strided accesses that do not incur any rotational latency. The ratio of strides to track sizes also explains the counterintuitive trend of the *Naive* scheme performance on the Cheetah disk where Dim_3 outperforms Dim_2, and Dim_2 outperforms Dim_1. The range queries, shown in Figure 13(b), perform on both disks as expected from the 3-D case. In summary, *MultiMap* is efficient for processing queries against spatio-temporal datasets, such as this earthquake simulation output, and is the only scheme that can combine streaming performance for time-varying accesses with efficient spatial access.

5.5 Performance for higher dimensional space

We developed an analytical model that predicts the performance for the *Naive* and *MultiMap* schemes for higher dimensional space. The model predicts the per-cell access time for any spatial query in an N-dimensional cube based on disk characteristics and dataset dimensions.

Table 3 presents a comparison of the model's predictions with the measurements from the experiments described previously to establish the model's validity. As shown, the prediction error is for most cases less than 8% and 5 out of 24 predicted values are within 20%. Details of the model are discussed in Appendix A.

Having demonstrated the model's high accuracy, we can use the model to predict the performance trends for N-dimensional space with $N > 4$. Figure 14 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.

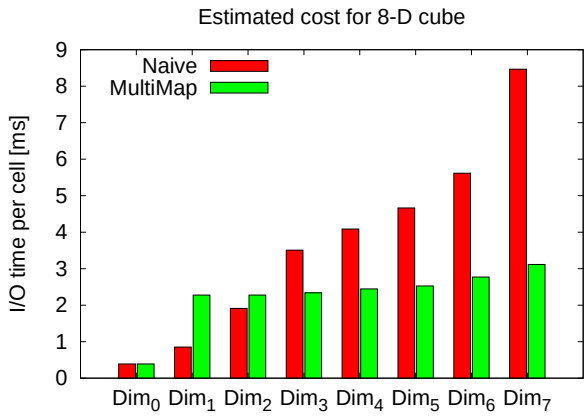


Figure 14: **Predicted performance of beam queries in 8-D space.**

6 Conclusions

The MultiMap mapping scheme for N-dimensional datasets leverages technological trends of modern disk drives to preserve spatial locality and deliver streaming bandwidth for accesses along one dimension and efficient semi-sequential accesses in the other dimensions. It does so without breaking established storage interfaces or exposing disk specific details, thanks to the logical volume abstraction offered by the `GETADJACENT` function.

While the MultiMap scheme works for arrays of disks by simply mapping successive basic cubes to different disks, our future work will investigate the effects of fine-grain striping within the basic cube to potentially improve performance of (very small) point or range queries. However, even without fine-grain striping, all spatial queries across 3-D, 4-D, and 8-D datasets evaluated in this paper enjoy the scalable bandwidth of parallel disk accesses.

References

[1] Khaled A. S. Abdel-Ghaffar and Amr El Abbadi. Optimal Allocation of Two-Dimensional Data. *International Conference on Database Theory* (Delphi, Greece), pages 409-418, January 8-10, 1997.

[2] Dave Anderson, Jim Dykes, and Erik Riedel. More than an interface: SCSI vs. ATA. *Conference on File and Storage Technologies* (San Francisco, CA, 31 March–02 April 2003), pages 245–257. USENIX Association, 2003.

[3] Mikhail J. Atallah and Sunil Prabhakar. (Almost) Optimal Parallel Block Access for Range Queries. *ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (Dallas, Texas, USA), pages 205-215. ACM, 2000.

[4] Randeep Bhatia, Rakesh K. Sinha, and Chung-Min Chen. Declustering Using Golden Ratio Sequences. *ICDE*, pages 271-280, 2000.

[5] The DiskSim Simulation Environment (Version 3.0). http://www.pdl.cmu.edu/DiskSim/index.html.

[6] EMC Corporation. *EMC Symmetrix DX3000 Product Guide*, http://www.emc.com/products/systems/DMX_series.jsp, 2003.

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

[8] Christos Faloutsos and Pravin Bhagwat. Declustering Using Fractals. *International Conference on Parallel and Distributed Information Systems* (San Diego, CA, USA), 1993.

[9] Volker Gaede and Oliver Günther. Multidimensional Access Methods. *ACM Comput. Surv.*, **30**(2):170-231, 1998.

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

[11] D Hilbert. Über die stetige Abbildung einer Linie auf Flächenstück. *Math. Ann*, **38**:459–460, 1891.

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

[13] Nick Koudas, Christos Faloutsos, and Ibrahim Kamel. Declustering Spatial Databases on a Multi-Computer Architecture. *International Conference on Extending Database Technology* (Avignon, France), 1996.

[14] Bongki Moon, H. V. Jagadish, Christos Faloutsos, and Joel H. Saltz. *Analysis of the clustering properties of Hilbert space-filling curve*. Technical report. University of Maryland at College Park, 1996.

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

[16] Sunil Prabhakar, Khaled Abdel-Ghaffar, Divyakant Agrawal, and Amr El Abbadi. Efficient Retrieval of Multi-dimensional Datasets through Parallel I/O. *the Fifth International Conference on High Performance Computing*, page 375. IEEE Computer Society, 1998.

[17] Sunil Prabhakar, Khaled A. S. Abdel-Ghaffar, Divyakant Agrawal, and Amr El Abbadi. Cyclic Allocation of Two-Dimensional Data. *International Conference on Data Engineering* (Orlando, Florida, USA). IEEE Computer Society, February 23-27, 1998.

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

[19] Jiri Schindler, Anastassia Ailamaki, and Gregory R. Ganger. Lachesis: robust database storage management based on device-specific performance characteristics. *International Conference on Very Large Databases* (Berlin, Germany, 9–12 September 2003). Morgan Kaufmann Publishing, Inc., 2003.

[20] Jiri Schindler and Gregory R. Ganger. *Automated disk drive characterization*. Technical report CMU–CS–99–176. Carnegie-Mellon University, Pittsburgh, PA, December 1999.

[21] Jiri Schindler, John Linwood Griffin, Christopher R. Lumb, and Gregory R. Ganger. Track-aligned extents: matching access patterns to disk drive characteristics. *Conference on File and Storage Technologies* (Monterey, CA, 28–30 January 2002), pages 259–274. USENIX Association, 2002.

[22] Jiri Schindler, Steven W. Schlosser, Minglong Shao, Anastassia Ailamaki, and Gregory R. Ganger. Atropos: a disk array volume manager for orchestrated use of disks. *Conference on File and Storage Technologies* (San Francisco, CA, 1–2 April 2004), pages 27–41. USENIX Association, 2004.

[23] Steven W. Schlosser, Jiri Schindler, Anastassia Ailamaki, and Gregory R. Ganger. *Exposing and exploiting internal parallelism in MEMS-based storage*. Technical Report CMU–CS–03–125. Carnegie-Mellon University, Pittsburgh, PA, March 2003.

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

[25] Bernhard Seeger and Per-Åke Larson. Multi-Disk B-trees. *SIGMOD*, pages 436-445, 1991.

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

[27] Tiankai Tu and David R. O'Hallaron. A Computational Database System for Generating Unstructured Hexahedral Meshes with Billions of Elements. *SC*, 2004.

[28] Bruce L. Worthington, Gregory R. Ganger, Yale N. Patt, and John Wilkes. On-line extraction of SCSI disk drive parameters. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Ottawa, Canada), pages 146–156, May 1995.

[29] Hailing Yu, Divyakant Agrawal, and Amr El Abbadi. Tabular placement of relational data on MEMS-based storage devices. *International Conference on Very Large Databases* (Berlin, Germany, 09–12 September 2003), pages 680–693, 2003.

A Analytical cost model

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 Hilbert curve mapping. Although it is possible to estimate the number of *clusters*, which are groups of consecutive *LBN*s for a given query [14], no work has been done on investigating the distribution of distances between clusters, which would be required for an accurate analytical model. Given the complexity of the disk access analysis for the Hilbert curve model, deriving such a model is beyond the scope of this work.

In the following discussion, we use the notations defined in Table 2 in Section 4.2. 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 seek time profile such as the one shown in Figure 1(b).

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 *LBN*s. 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 *LBN*s 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 *LBN*s. 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}.

A.1 Analytical cost model for Naive mapping

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

$$C_{xfer} = \sigma \prod_{i=0}^{n-1} q_i \tag{7}$$

$$C_{pos} = C_{init} + \sum_{i=1}^{n-1} \left[(Seek(d_i) + RotLat)\ (q_i - 1) \prod_{j=i+1}^{n-1} q_j \right] \tag{8}$$

$$d_i = \left\lceil \frac{\prod_{j=0}^{i-1} S_j - q_0 + 1 - \sum_{j=1}^{i-1} \left((q_j - 1) \prod_{k=0}^{j-1} S_k \right)}{T} \right\rceil \tag{9}$$

Equation 7 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 8 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 9 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.

A.2 Analytical cost model for MultiMap mapping

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

$$C_{xfer} = \sigma \prod_{i=0}^{n-1} q_i \tag{10}$$

$$C_{pos} = C_{init} + (Seek(d_0) + RotLat)\ N_{jmp}(q_0, K_0, S_0) + \sum_{i=1}^{n-1} \left[(\ \alpha(q_i - N_{jmp}(q_i, K_i, S_i)) + (Seek(d_i) + RotLat) N_{jmp}(q_i, K_i, S_i)) \prod_{j=i+1}^{n-1} q_j \right] \tag{11}$$

$$d_i = \left\lceil \frac{\prod_{i=0}^{n-1} K_i}{T} \right\rceil \prod_{j=0}^{i-1} \left\lceil \frac{S_j}{K_j} \right\rceil \tag{12}$$

$$N_{jmp}(q_i, K_i, S_i) = \left(\left\lceil \frac{q_i}{K_i} \right\rceil - 1 \right) \frac{Loc_S(q_i, K_i, S_i)}{S_i - q_i + 1} + \left\lceil \frac{q_i}{K_i} \right\rceil \left(1 - \frac{Loc_S(q_i, K_i, S_i)}{S_i - q_i + 1} \right) \tag{13}$$

$$Loc_S(q_i, K_i, S_i) = Loc_K(q_i, K_i) \left(\left\lfloor \frac{S_i}{K_i} \right\rfloor - \left\lfloor \frac{q_i}{K_i} \right\rfloor \right) + \left\lfloor \frac{S_i \bmod K_i}{Loc_K(q_i, K_i)} \right\rfloor \tag{14}$$

$$Loc_K(q_i, K_i) = K_i - (q_i \bmod (K_i + 1)) + 1 \tag{15}$$

The transfer cost (Equation 10) is the same as the transfer cost of the *Naive* model. Equation 11 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 13).

The third term of Equation 11 is similar to the second term in Equation 8 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 12 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 13) 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 14), we first count the locations within a basic cube that will cause the minimal number of jumps, denoted as Loc_K (Equation 15), 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 $\left\lfloor \frac{S_i \bmod K_i}{Loc_K(q_i, K_i)} \right\rfloor$.

A.3 Model parameters

For the results presented in Section A, 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 1 in Section 3.2). We set $T = 686$, which is equal to the number of sectors per track in the outer-most zone.