IEEE 23rd International Conference on Data Engineering (ICDE 2007) Istanbul, Turkey, April 2007.

MultiMap: Preserving disk locality for multidimensional datasets

Minglong Shao*, Steven W. Schlosser†, Stratos Papadomanolakis*, Jiri Schindler‡
Anastassia Ailamaki*, Gregory R. Ganger*

*Carnegie Mellon University †Intel Research Pittsburgh ‡ Network Appliance, Inc

Abstract

MultiMap is 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.

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 [15], Hilbert curves [11], and Gray-coded curves [7]. 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 [22] describes a new generalized model of disks, called the 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 paper 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 paper is organized as follows. Section 2 describes related work. Section 3 outlines the characteristics of modern disk technology that enable *MultiMap*. Section 4 introduces *MultiMap* in detail. Section 5 evaluates the performance of *MultiMap*, and section 6 concludes.

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 [10, 25, 29]. Data storage and management for massive multidimensional data have two primary tasks: data indexing, for quick *location* of the needed data, and data placement, which arranges data on storage devices for efficient *retrieval*. 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 [15], Hilbert [11], or Gray-coded curve [7]. Their goal is to order multidimensional data such that spatial locality can be preserved as much as possible within the 1-D disk abstraction. The property of preserving spatial locality, often called *clustering* [13], does not take advantage of the disk characteristics; access along each dimension has (nearly) identical cost. However, the cost, equivalent to short seek and a random rotational latency, is far greater than the efficient sequential access. In contrast, our *MultiMap* algorithm exploits the opportunity for efficient sequential access along one dimension and a minimal-overhead access for the remaining dimensions.

Optimizations on naive mappings [19] 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. [9] and Schindler et al. [20] proposed a secondary dimension on disks which has been utilized to store 2-D database tables. Multidimensional clustering in DB2 [16] 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 [21, 28].

Another body of related research focuses on how to *decluster* multidimensional datasets across multiple disks [1, 3, 4, 8, 17, 18] to optimize spatial access methods [12, 23] and improve throughput.

3 Multidimensional disk access

This section reviews the adjacency model on which we build *MultiMap*. Detailed explanations and evaluation of the disk technologies across a range of disks from a variety of vendors are provided by Schlosser et al. [22]. The adjacency model has two primary concepts: *adjacent blocks* and *semi-sequential access*. As described by Schlosser et al., the necessary disk parameters can be exposed to applications in an abstract, disk-generic way.

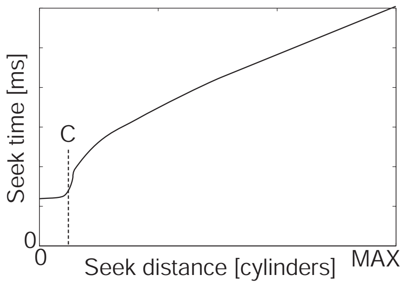
3.1 Adjacent disk blocks

The concept of adjacent blocks is based on two characteristics of modern disks, shown in Figure 1 [22]:

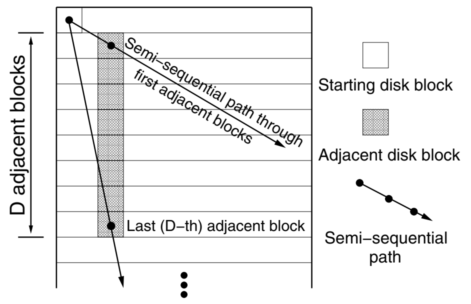
1. Short seeks of up to some cylinder distance, C, are dominated by the time to settle the head on a destination track;
2. Firmware features internal to the disk can identify and, thus, access blocks that require no rotational latency after a seek.

Figure 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 dominated by the time it takes for the disk head to settle on the destination track, referred to as *settle time*. Supposing 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. 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]. With such trends, more cylinders and, thus, more tracks, can be accessed within the settle time.

While each of these D tracks contain many disk blocks, there is one block on each track that can be accessed immediately after the head settles on the destination track, with



(a) Conceptual seek profile of modern disks.


D adjacent blocks
Semi–sequential path through first adjacent blocks
Last (D–th) adjacent block
Starting disk block
Adjacent disk block
Semi–sequential path

(b) Adjacent blocks and semi-sequential access.

Figure 1. Conceptual seek profile of modern disk drives and illustration of adjacent blocks.

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 another track.

Figure 1(b) illustrates adjacent blocks on disk. For a given starting block, there are *D* adjacent disk blocks, one in each of the *D* adjacent tracks. All adjacent blocks are at the same physical offset from the starting block because the offset is determined by how many degrees disk platters rotate within the settle time.

3.2 Semi-sequential access

Accessing successive adjacent disk blocks enables *semi-sequential* disk access [9, 20], which is the second-most efficient disk access pattern after pure sequential access. Figure 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 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) *D*th adjacent block can be accessed for equal cost.

Semi-sequential access outperforms nearby access within *D* tracks by a factor of four, thanks to the elimination of all rotational latency. Modern disks can support many semi-sequential access paths since *D* is on the order of hundreds [22].

The adjacency model is exposed by our logical volume manager (LVM) to applications through two interface functions: `GETADJACENT` and `GETTRACKBOUNDARIES`.

4 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 *LBN*s without any impact on the applicability of our approach.

4.1 Examples

For simplicity, we 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

Table 1. Notation definitions.

The notations used in the examples and later discussions are listed in Table 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 2 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 the cell is mapped. Cells along the first dimension (i.e., Dim_0, or the row direction), are mapped sequentially to consecutive *LBN*s 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.

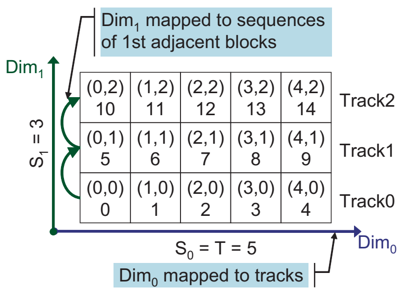


Figure 2. Mapping 2-D dataset.

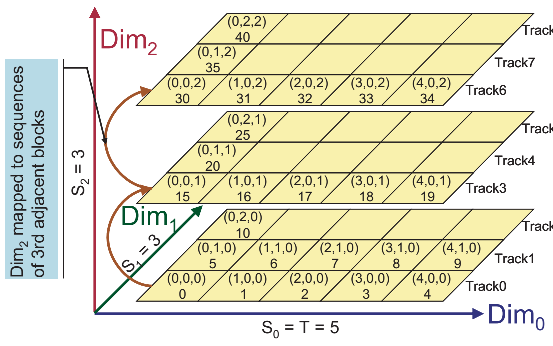


Figure 3. Mapping 3-D dataset.

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 2. 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 3, 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, 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.

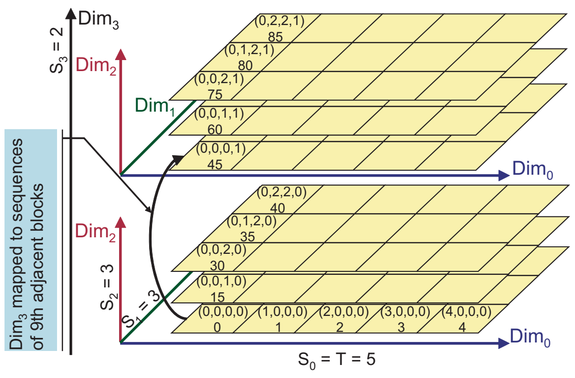


Figure 4. Mapping 4-D dataset.

Example of 4-D mapping. The 4-D example, shown in Figure 4, maps a dataset of the size $(5 \times 3 \times 3 \times 2)$ $(S_0 = T = 5,\ S_1 = 3,\ S_2 = 3,\ S_3 = 2)$. 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.

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

4.2 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:

$$K_0 \leq T \tag{1}$$

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

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

Equation 1 restricts the length of the first dimension of

```
$L :=$ MultiMap($x[0], x[1], \ldots, x[N-1]$) :
  lbn := (start_lbn + $x[0]$) mod $T$ + $\lfloor$start_lbn$/T\rfloor * T$
  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 $(0,\ldots,0)$)
GETADJACENT: get step-th adjacent block of lbn
```

Figure 5. Mapping a cell in space to an *LBN*.

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 the GETTRACKBOUNDARIES interface function exported by our LVM. Equation 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 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; $\ldots$; $Dim_{i+1}(1 \leq i \leq N-2)$ is mapped to a sequence of successive $(\prod_{i=1}^{i} K_i)$-th adjacent blocks.

The *MultiMap* algorithm, shown in Figure 5, generalizes the above procedure. The inputs of MultiMap are the coordinate of a cell in the basic cube, and the output is the *LBN* to store that cell. MultiMap 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 $(K_1 \times K_2 \ldots \times K_{i-1})$ 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 3). So, requesting these (non-contiguous) blocks results in semi-sequential accesses.

4.3 Number of dimensions supported by a disk

The number of dimensions that can be supported by *MultiMap* is bounded by D and K_i. Realistic values of D and K_i allow for a substantial number of dimensions. The first dimension, Dim_0, is mapped along disk tracks, and the last dimension, Dim_{N-1}, is mapped along successive last (D-th) adjacent blocks. The remaining $N-2$ dimensions must fit in D tracks (refer to Equation 3). Consider basic cubes with equal length along all dimensions, $K_1 = \ldots = K_{N-2} = K$. Based on Equation 3, we get:

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

$$N_{max} = \lfloor 2 + \log_2 D \rfloor \tag{5}$$

For modern disks, D is typically on the order of hundreds [22], allowing mapping for more than 10 dimensions. For most physical simulations and OLAP applications, this number is sufficient.

4.4 Mapping large datasets

The basic cube defined in Section 4.2 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. 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 at a premium 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.

4.5 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) [14] and tomographic (e.g., the Visible Human Project for the National Library of Medicine) or other volumetric datasets [6]. 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 [26] that uses an octree as its index. Experimental results with this dataset are shown in Section 5.

4.6 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 [5]. Observation-based applications, such as telescope or satellite imaging systems [10], 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 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.

We also 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 *Naive* and *MultiMap* given disk parameters, the dimensions of the dataset, and the size of the query. Due to space limitations, we refer the interested reader to a technical report [24], which shows the details of this model.

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 [22]. The database storage manager maps multidimensional datasets by utilizing high-level functions exported 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.

5.2 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 *LBN*s 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 (along Dim_0) or semi-sequential (along other dimensions) accesses. The storage manager identifies those *LBN*s that contain the data and issues them directly to the disk. No sorting is required. For instance, in Figure 2, 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 2 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 bytes.

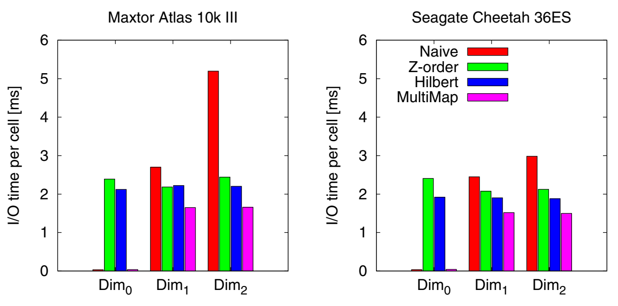
5.3 Synthetic 3-D dataset

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 $D = 128$.

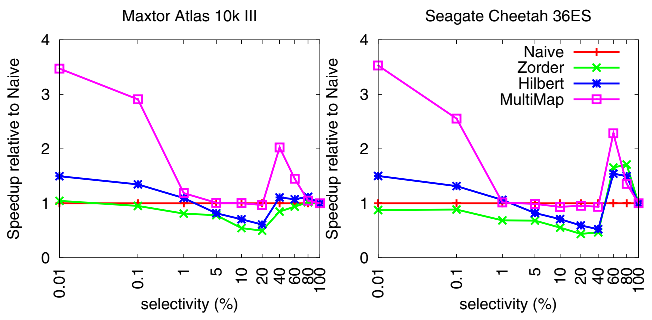
Beam queries. The results for beam queries along Dim_0, Dim_1, and Dim_2 are presented in Figure 6(a). The graphs show the average I/O time per cell (disk block). The values are averages over 15 runs, and the standard deviation is less than 1% of the reported times. Each run selects a random value between 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 of 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* ($57\times$ – $69\times$ 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



(a) **Beam queries.**



(b) **Range queries.**

Figure 6. Performance of queries on the synthetic 3-D dataset.

Z-order, which agrees with the theory that *Hilbert* curve has better clustering properties [13].

MultiMap delivers the best performance for beam queries along 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 6(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×, while *Z-order* and *Hilbert* mappings observe a maximum speedup of 1.54× and 1.11×, respectively.

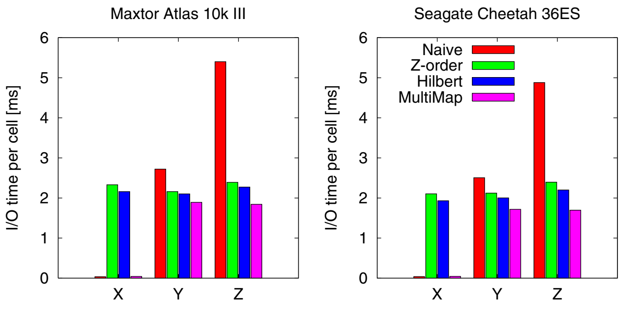
Given our dataset size and the range of selectivities from 0.01% to 100%, these queries fetch between 900 KB and 8.5 GB data from a single disk. The performance of range queries are determined by two factors: the closeness of the required blocks (the clustering property of the mapping algorithm) and 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 except in the selectivity range of 10%–40% on the Seagate Cheetah 36ES disk where it is 6% worse than *Naive*.

5.4 3-D earthquake simulation dataset

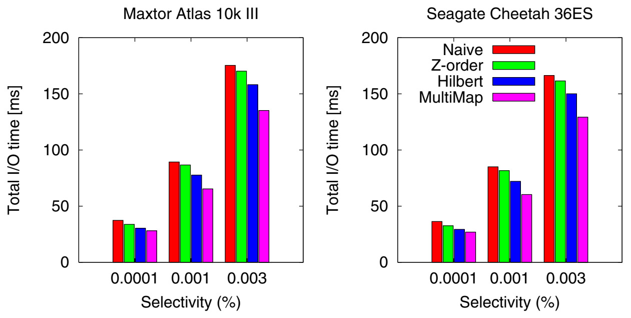
The earthquake dataset models earthquake activity in a 14 km deep slice of earth of a 38 × 38 km area in the vicinity of Los Angeles [27]. We use this dataset as an example of how to apply *MultiMap* to skewed datasets. The points in the 3-D dataset, called nodes, have variable densities and are packed into elements such that the 64 GB dataset is translated into a 3-D space with 113,988,717 elements indexed by an octree, with each element as a leaf node.

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 7, 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 compro-

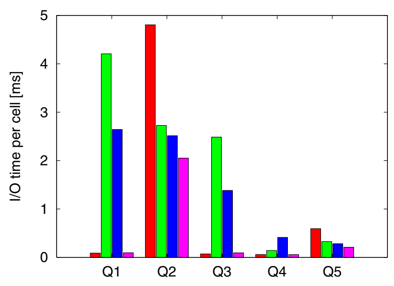


(a) **Beam queries.**

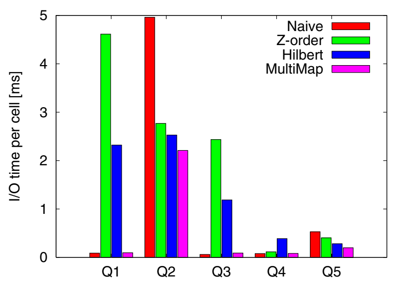


(b) **Range queries.**

Figure 7. Performance of queries on the 3-D earthquake dataset.



(a) **Maxtor Atlas 10k III.**



(b) **Seagate Cheetah 36ES.**

Figure 8. Performance of queries on the 4-D OLAP dataset.

mising the performance of spatial accesses in other dimensions. For range queries, we select representative selectivities for the applications.

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 [16]. 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 8. 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*.

6 Conclusions

MultiMap is a data placement technique for multidimensional datasets that leverages technological trends of modern disk drives to preserve spatial locality, delivering streaming bandwidth for accesses along one dimension and efficient semi-sequential accesses along the other dimensions. We measure substantial improvement over traditional mapping techniques for multidimensional datasets.

References

[1] K. A. S. Abdel-Ghaffar and A. E. Abbadi. Optimal Allocation of Two-Dimensional Data. *International Conference on Database Theory*, pp. 409-418, 1997.

[2] D. Anderson, J. Dykes, and E. Riedel. More than an interface: SCSI vs. ATA. *FAST*, pp. 245–257. USENIX, 2003.

[3] M. J. Atallah and S. Prabhakar. (Almost) Optimal Parallel Block Access for Range Queries. *ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pp. 205-215. ACM, 2000.

[4] R. Bhatia, R. K. Sinha, and C.-M. Chen. Declustering Using Golden Ratio Sequences. *ICDE*, pp. 271-280, 2000.

[5] The Office of Science Data-Management Challenge, 2005.

[6] T. T. Elvins. A survey of algorithms for volume visualization. *Computer Graphics*, **26**(3):194–201, 1992.

[7] C. Faloutsos. Gray codes for partial match and range queries. *Workshop on Software Testing Papers*. Published as *IEEE Transactions on Software Engineering*, **14**(10):1381–1393. IEEE, 1986.

[8] C. Faloutsos and P. Bhagwat. Declustering Using Fractals. *International Conference on Parallel and Distributed Information Systems*, 1993.

[9] G. G. Gorbatenko and D. J. Lilja. *Performance of two-dimensional data models for I/O limited non-numeric applications*. Technical report ARCTiC–02–04. University of Minnesota, 2002.

[10] J. Gray, D. Slutz, A. Szalay, A. Thakar, J. vandenBerg, P. Kunszt, and C. Stoughton. *Data Mining the SDSS Sky-server Database*. Technical report. MSR, 2002.

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

[12] I. Kamel and C. Faloutsos. Parallel R-trees. *SIGMOD*, pp. 195-204, 1992.

[13] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz. *Analysis of the clustering properties of Hilbert space-filling curve*. Technical report. UMCP, 1996.

[14] B. Nam and A. Sussman. Improving Access to Multi-dimensional Self-describing Scientific Datasets. *International Symposium on Cluster Computing and the Grid*, 2003.

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

[16] S. Padmanabhan, B. Bhattacharjee, T. Malkemus, L. Cranston, and M. Huras. Multi-Dimensional Clustering: A New Data Layout Scheme in DB2. *SIGMOD*, 2003.

[17] S. Prabhakar, K. Abdel-Ghaffar, D. Agrawal, and A. E. Abbadi. Efficient Retrieval of Multidimensional Datasets through Parallel I/O. *ICHPC*, pp. 375–386. IEEE, 1998.

[18] S. Prabhakar, K. A. S. Abdel-Ghaffar, D. Agrawal, and A. E. Abbadi. Cyclic Allocation of Two-Dimensional Data. *ICDE*. IEEE, 1998.

[19] S. Sarawagi and M. Stonebraker. Efficient Organization of Large Multidimensional Arrays. *ICDE*, pp. 328-336, 1994.

[20] J. Schindler, S. W. Schlosser, M. Shao, A. Ailamaki, and G. R. Ganger. Atropos: a disk array volume manager for orchestrated use of disks. *FAST*. USENIX, 2004.

[21] S. W. Schlosser, J. Schindler, A. Ailamaki, and G. R. Ganger. *Exposing and exploiting internal parallelism in MEMS-based storage*. Technical Report CMU–CS–03–125. Carnegie-Mellon University, March 2003.

[22] S. W. Schlosser, J. Schindler, S. Papadomanolakis, M. Shao, A. Ailamaki, C. Faloutsos, and G. R. Ganger. On multidimensional data and modern disks. *FAST*. USENIX, 2005.

[23] B. Seeger and P. A. Larson. Multi-disk B-trees. *SIGMOD*, pp. 436–445. ACM, 1991.

[24] M. Shao, S. Papadomanolakis, S. W. Schlosser, J. Schindler, A. Ailamaki, C. Faloutsos, and G. R. Ganger. *MultiMap: Preserving disk locality for multidimensional datasets*. CMU–PDL–05–102. CMU, April 2005.

[25] K. Stockinger, D. Dullmann, W. Hoschek, and E. Schikuta. Improving the Performance of High-Energy Physics Analysis through Bitmap Indices. *Database and Expert Systems Applications*, pp. 835–845, 2000.

[26] T. Tu, D. O'Hallaron, and J. Lopez. Etree: A Database-oriented Method for Generating Large Octree Meshes. *Eleventh International Meshing Roundtable*, pp. 127–138, Sep 2002.

[27] T. Tu and D. R. O'Hallaron. A Computational Database System for Generating Unstructured Hexahedral Meshes with Billions of Elements. *SC*, 2004.

[28] H. Yu, D. Agrawal, and A. E. Abbadi. Tabular placement of relational data on MEMS-based storage devices. *VLDB*, pp. 680–693, 2003.

[29] H. Yu, K.-L. Ma, and J. Welling. A Parallel Visualization Pipeline for Terascale Earthquake Simulations. *ACM/IEEE Conference on Supercomputing*, pp. 49, 2004.