Appeared in Proceedings of the 19th ACM Symposium on Principles of Database Systems (PODS), Dallas, May 2000

(Almost) Optimal Parallel Block Access for Range Queries

Mikhail J. Atallah*
CERIAS and
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907
U.S.A.
mja@cerias.purdue.edu

Sunil Prabhakar
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907
U.S.A.
sunil@cs.purdue.edu

ABSTRACT

Range queries are an important class of queries for several applications including relational and spatial databases, visualization, and GIS applications. For large datasets, the performance of range queries is limited by disk I/O. Performance improvements are achieved by tiling the multidimensional data and distributing it among multiple disks or nodes. Consequently, in order to process a range query, it is necessary to access only those tiles or blocks that intersect with the query. Given k disks, a query that accesses m blocks needs a number of parallel block accesses that is at least $\lceil m/k \rceil$ (which is known to be unachievable except for a few special cases [1]). Though several schemes for the allocation of tiles to disks have been developed, no scheme with guaranteed worst-case performance is known. We establish that any range query on a $2^q \times 2^q$-block grid of blocks can be performed using $k = 2^t$ disks ($t \leq q$), in at most $\lceil m/k \rceil + O(\log k)$ parallel block accesses. We also give two natural generalizations of the scheme to higher dimensions. We achieve this result by judiciously distributing the blocks among the k nodes or disks. Experimental data show that the algorithm achieves very close to $\lceil m/k \rceil$ performance (on average less than 0.5 away from $\lceil m/k \rceil$, with a worst-case of 3). Although several declustering schemes for range queries have been developed, prior to our work no additive non-trivial performance bounds were known. Our scheme guarantees performance within a (small) *additive* deviation from $\lceil m/k \rceil$. This guarantee is true for any number of dimensions. Subsequent to this work, Bhatia et al. [4] have proved that such a performance bound is essentially optimal for this kind of scheme, and have also extended our results to the case where the number of disks is a product of the form $k_1 * k_2 * \cdots * k_t$ where the k_is need not all be 2.

*Portions of this work were supported by sponsors of the CERIAS Laboratory.

1. INTRODUCTION

Range queries are an important class of queries for several application domains including relational databases, spatial databases, visualization, and GIS applications. Given a multidimensional dataset, a range query specifies a range of values for each dimension. The result of the range query is the set of all items in the dataset that have values within the specified range in each dimension. As the size of the dataset grows, the amount of data that needs to be accessed to answer range queries also increases, resulting in poor performance. In order to improve performance, the dataset is typically tiled along each dimension. Therefore in order to process a range query, it is necessary to access only those tiles or blocks that intersect the query, resulting in reduced I/O and improved performance. Even with such tiling, the performance is limited by the disk I/O. To further improve performance, multiple disks or processing nodes can be used to access the blocks in parallel. The blocks of the dataset are distributed among the disks. The key to achieving gains from parallelism is in the allocation of the blocks to the disks. Note that the data could be placed on multiple disks connected to a single processor or stored on parallel nodes, each with local disk space. Similarly, at the primary storage level, access to rectangular ranges from main memory are needed to refresh rectangular windows in GUI displays, and for visualizing three dimensional objects. Main memory typically consists of several memory banks, each of which can retrieve a single data item in one memory cycle. These range accesses can be optimized by accessing the required data in parallel from the data banks. Once again, the key is to place the data on the multiple data banks so as to reduce the overall number of memory cycles needed to access the data. The allocations described in this paper are applicable to each of these problems. We will use the term disk to refer to such data banks, parallel disks, or nodes.

The goal of the allocation is to achieve optimal parallel access for each range query. The design of these allocation schemes has been an active research area, resulting in the development of several allocation schemes [5, 7, 6, 1, 9]. These schemes are developed under the following framework. Due to the relatively high cost of disk accesses, the

CPU processing time is ignored. Furthermore, since the disk accesses are random, the cost of a single disk access is assumed to be constant. Note that for the case of main memory banks, this model is completely accurate. Thus, given a query, the cost of executing the query is taken to be proportional to the number of disk accesses performed. When the data are accessed from multiple disks in parallel, the cost is proportional to the largest number of accesses performed on a single disk. For a query that intersects m blocks, the access cost using k disks is at least $OPT = \lceil m/k \rceil$. The worst-case performance of an allocation is, expressed as a function of m, the maximum number of parallel accesses over all possible queries that intersect m blocks. It has been established that for the 2-dimensional case, allocations that (in the worst case) achieve OPT exist in only a small number of cases [1]. In particular, allocations that achieve OPT exist if and only if: 1) the number of disks is 1,2,3 or 5; or 2) there are no more than two blocks in at least one of the dimensions; or 3) the number of disks is almost as large as the total number of blocks; and 4) a special case for a 4×4 tiling with 8 disks. The existence of allocations that achieve OPT for higher dimensions is expected to be at least as restrictive. The paper [1], also describes a scheme that produces an allocation that achieves OPT for two-dimensional data whenever one exists. Next, we describe the most prominent schemes that have previously been developed. In view of the provable impossibility of achieving OPT except in a small number of special cases, the notation "OPT" should be viewed as merely a shorthand for $\lceil m/k \rceil$ rather than as an achievable performance bound.

For ease of exposition, the following notation is used. For a d-dimensional dataset, each block is described by a set of coordinates $(x_0, x_1, \cdots, x_{d-1})$. Each coordinate, x_j, is in the range $[0, N_j - 1]$ and represents the order of the block in dimension j, where dimension j is divided into N_j blocks. The number of disks is k, and each disk is identified by a number ranging from 0 to $k-1$. The Disk Modulo (DM) scheme [5], developed by Du and Sobolewski and later extended for range queries and dynamic files in [8] allocates block $(x_0, \cdots, x_{d-1})$ to disk $(x_0 + x_1 + \cdots + x_{d-1}) \bmod k$. The Fieldwise eXclusive (FX) method proposed by Kim and Pramanik [7], allocates a block to the disk given by the lowest $\log_2 k$ bits of the bit-wise exclusive-OR of the binary representations of all the coordinates of the block. The Hilbert Curve Allocation Method (HCAM) [6], proposed by Faloutsos and Bhagwat, is based upon the Hilbert space-filling curve. Hilbert curves can be used to convert a discrete multidimensional space into a linear sequence such that spatial proximity is preserved as much as possible. After mapping the blocks into this linear sequence, the blocks are assigned to disks in a round-robin fashion. The allocation method that achieves OPT for 2-dimensional data, described in [1], allocates block (x_0, x_1) to disk $(x_0 + \lfloor \frac{k}{2} \rfloor x_1) \bmod k$. A class of declustering schemes called Cyclic allocation schemes was developed in [9] and [11]. These schemes allocate block $(x_0, \cdots, x_{d-1})$ to disk $(x_0 H_0 + x_1 H_1 + \cdots + x_{d-1} H_{d-1}) \bmod k$, where the values $H_0, H_1, \cdots, H_{d-1}$, are called *skip* values. Each scheme is the class is defined by a different choice of skip values. It is shown that the choice of skip values is critical in determining the quality of the allocation, and three different approaches for determining values that give good performance are also developed. More recently, a new scheme called GRS, based upon the golden ratio, has been developed [3]. Its performance is comparable to the best Cyclic scheme, but requires less computation. There has also been some recent work on declustering for similarity queries, also known as nearest-neighbor queries [2, 10]. Declustering for nearest-neighbor queries is a very different problem since the set of tiles needed to answer a given query depends upon the actual data and the search algorithm, unlike the case of range queries. The scheme described in [2] performs poorly for range queries. The scheme of [10] is a variant of the Cyclic schemes for range queries.

The relative performance of these schemes has been studied experimentally in earlier work [3, 9, 11]. It is seen that, on the average, the Cyclic schemes outperform the other schemes. However, prior to our work, there is no guarantee on the performance of a given range query for any of the existing schemes, except the GRS scheme. For two dimensions, the GRS scheme has been shown to guarantee performance within a *multiplicative* factor of 3 from OPT (i.e., $3OPT$). In this paper, we develop an allocation scheme which has guaranteed worst-case performance within an *additive* deviation from OPT: Within $OPT + O(\log k)$ for two dimensions. The allocation scheme can be extended naturally to higher dimensions. We are hopeful that we can extend the 2-dimensional performance bounds to the higher dimensional scheme, but with an additive factor of 2^d. However, we expect that this is a naive bound and an overestimate of the actual bound.

Our scheme requires that the number of disks available is $k = 2^t$, numbered from 1 to k. To generate the allocation for a dataset that has been divided into $N_1 \times N_2$ blocks along the two dimensions, we first extend the number of blocks in each dimension such that we have 2^q tiles in each dimension, where $q \geq t$. After generating the allocation for this larger grid of blocks, we simply ignore the extra blocks that were added, resulting in the allocation for the $N_1 \times N_2$ dataset. The disk allocation problem can be viewed as that of coloring the $N = 2^{2q}$ blocks by using colors numbered 1 to k, with the interpretation that a block of color i is to be stored in disk i. The number of parallel block accesses for processing a range query is the maximum occurrence of any color in the rectangular region of blocks defined by the range query.

The rest of this paper is organized as follows. Section 2 describes the coloring (allocation) scheme used. Section 3 discusses some properties of that coloring scheme. Sections 4 proves that, for any 2-dimensional range query, if m is the number of blocks for that range query, then the coloring scheme we use can result in no more than $\lceil m/k \rceil + \gamma$ parallel block accesses where $\gamma = O(\log k)$. Section 5 presents the generalization of the coloring scheme to higher dimensions. In practice $O(\log k)$ is a considerable overestimate for γ: experimental data, some of which is presented in Section 6, reveals that γ is typically no larger than 3 for practical

problem sizes. Finally, Section 7 concludes the paper.

2. THE COLORING SCHEME

We partition the $2^q \times 2^q$ grid of blocks into a $2^{q-t} \times 2^{q-t}$ grid of *groups* each of which is itself a $k \times k$ grid of blocks (recall that $k = 2^t$). We next describe the coloring scheme for the blocks in a group (the same coloring scheme is used for all the groups). Row and column numbers in that description are *relative to that group* (not relative to the whole grid). We start with some definitions.

Let j be a column whose blocks have been colored, and let j' be another column whose coloring is to be derived from that of column j. We say that the coloring of column j' is a $k/2$-*swap* of the coloring of column j if we first copy the coloring of j into j' and then we "swap" the coloring of the upper half of column j' with the coloring of its lower half. For example, if the colors of the cells of column j are (in row order) $1, 2, \ldots, k$ then the colors for column j' would be $(k/2)+1, \ldots, k, 1, \ldots, (k/2)$. More generally, a $k/2^i$-swap of a column's coloring, for an integer $i \leq t$, is defined as follows:

- Partition the column into 2^{i-1} contiguous, non overlapping pieces of size $k/2^{i-1}$ each. Then, for each piece, swap the coloring of the piece's upper half with the coloring of the piece's lower half. (We call it a "$k/2^i$" swap because that is the size of each portion being swapped, as a mnemonic.)

For example, a 1-swap of a column's coloring consists of interchanging the colors of cells $2\ell - 1$ and 2ℓ, for all $1 \leq \ell \leq k/2$.

We are now ready to describe the coloring of a group of $k \times k$ blocks.

1. Assign the colors $1, \ldots, k$ to the cells of column 1. *Comment.* Although we assign the colors in sorted order, in fact any permutation would also work (as will soon become apparent).
2. For $\mu = 1, \ldots, t$ in turn, do the following: For $j = 1, \ldots, 2^{\mu-1}$ in turn, assign to column $2^{\mu-1} + j$ a coloring that is a $k/2^\mu$-swap of the coloring of column j.

Figure 1 gives an example of the above coloring for the case $t = 4$ (i.e., 16 colors). All columns are generated from column 1. For example, column 16 is a 1-swap of column 8, which is a 2-swap of column 4, which is a 4-swap of column 2, which is a 8-swap of column 1. Similarly, column 15 is a 1-swap of column 7, which is a 2-swap of column 3, which is a 4-swap of column 1.

3. PROPERTIES OF COLORING SCHEME

A coloring of a group is said to be *left-to-right legal* if it is obtained according to the process described in the previous section except that the process can be initiated with the first column holding *any* permutation of the k colors (not necessarily the sorted one we used in the previous section). The coloring is *right-to-left legal* if we do the same thing except that we start with the rightmost column of the group and proceed leftward from there. The notions of top-to-bottom legal and of bottom-to-top legal are defined using a similar coloring process that operates by rows rather than by columns.

We begin with the **group properties**, i.e., the properties that hold within each group:

1. Any of the following four properties of a group coloring implies the other three: { left-to-right legal, right-to-left legal, top-to-bottom legal, bottom-to-top legal }. Therefore the left-to-right legal coloring we produced for a block also has the other three properties. We henceforth use the word *legal coloring* as an abbreviation for these.
2. Each column of a group contains a permutation of the k colors.
3. Each row of a group contains a permutation of the k colors.

Group property 2 is an immediate consequence of the way a column is colored (because copying then permuting the coloring of a column results in another permutation of the colors).

Group property 1 will be proved (together with other properties) at the end of this section.

Group property 3 follows from group properties 1 and 2.

We now define a finer partition of the input grid than its partition into groups. This is not needed algorithmically and is done purely for the sake of the analysis. To avoid unnecessarily cluttering the analysis with "$\lfloor \cdot \rfloor$" notation, we assume t is even (it is easy to modify the analysis for odd t).

Partition each $2^t \times 2^t$ group into a $2^{t/2} \times 2^{t/2}$ grid of *superblocks* each of which is itself a $2^{t/2} \times 2^{t/2}$ grid of blocks (observe that $2^{t/2} = \sqrt{k}$). A range query is said to *vertically span* a superblock if it does not completely contain that superblock, and its intersection with that superblock is a contiguous set of columns of that superblock (horizontal span is defined similarly with respect to rows). Let S be a set of superblocks that are vertically contiguous to each other (i.e., each of them is "on top" of another one of them). A range query is said to vertically span S if it is contained in S and it vertically spans all the superblocks of S *at corresponding sets of columns*, i.e., if its intersection with a superblock x of S is the same interval of columns $[j, j']$ for all such x (horizontal span is analogously defined).

The following **superblock properties** hold:

1. Each superblock of $\sqrt{k} \times \sqrt{k}$ blocks contains the k

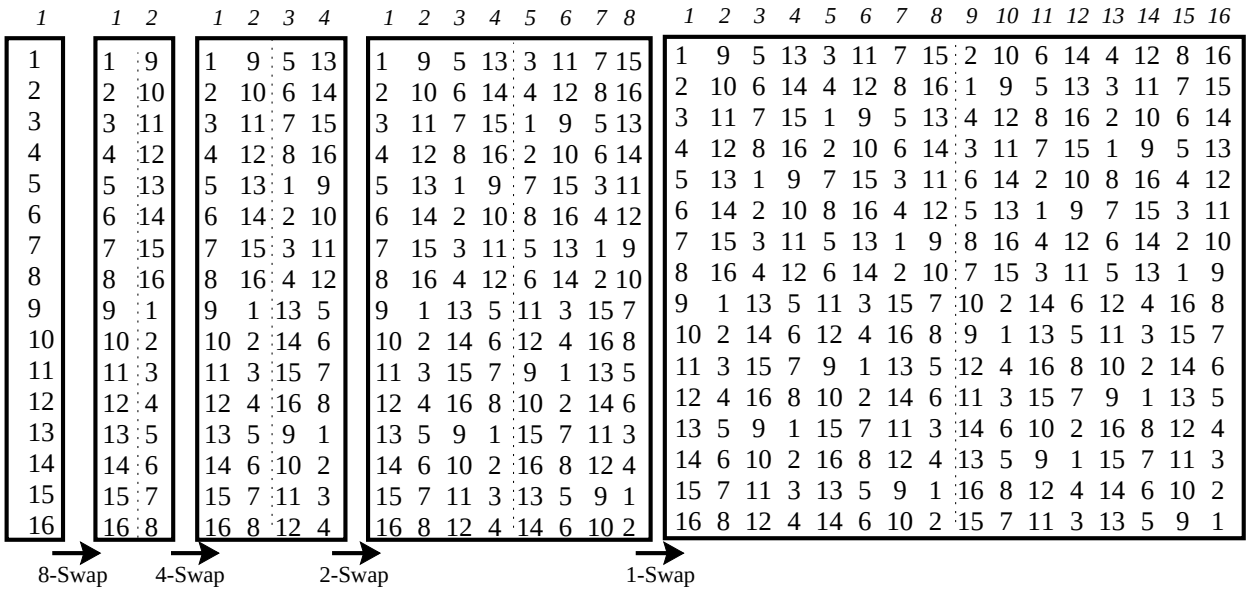

Figure 1: An example of the allocation scheme for 16 disks

colors (i.e., one occurrence of each color).

2. Let S be a set of superblocks that are vertically contiguous to each other. For any range query that vertically spans S, the legal coloring described in the previous section is within an additive $2^{-1}\log k$ of OPT for that query (i.e., results in at most $\lceil m/k \rceil + 2^{-1}\log k$ parallel block accesses where m is the number of blocks touched by the query).

3. Let S be a set of superblocks that are horizontally contiguous to each other. For any range query that horizontally spans S, the legal coloring described in the previous section is within an additive $2^{-1}\log k$ of OPT for that query (i.e., results in $\lceil m/k \rceil + 2^{-1}\log k$ parallel block accesses).

The above superblock properties are proved below (together with group property 1).

Proof sketch of group property 1 and superblock properties 2 and 3

We give the proof for the general case where the coloring process is initiated with an arbitrary permutation of k distinct symbols (rather then the particular sorted permutation of the integers 1 to k we used in Section 2).

The proof is by induction on t. The basis, $t = 1$, is trivial.

We assume inductively that the properties hold for t. To show that they hold for $t+1$, we observe that the coloring of a $2^{t+1} \times 2^{t+1}$ grid can be thought of as consisting of the following four steps (which we describe assuming a coloring that starts with an initial column and operates left-to-right – essentially the same argument can be made for a coloring process that starts with an initial row and operates row-wise).

1. (*Coalesce step*) For the initial column (of size 2^{t+1}), *coalesce* entry $2\ell - 1$ and entry 2ℓ, $1 \le \ell \le 2^t$. This "shrinks" the column into one of half the size ($= 2^t$), with 2^t distinct *new* colors; each new color c corresponds to an ordered pair (c', c'') of old colors.

2. (*Induction step*) Perform the iterative coloring process on a $2^t \times 2^t$ array A with the (shrunk) column of size 2^t as the initial column (and using the new colors). This results in a $2^t \times 2^t$ colored array A that (by the induction hypothesis) has the desired properties (relative to the new colors, of course).

3. (*Duplication step*) Duplicate the colored $2^t \times 2^t$ array A and, in the duplicate copy $\hat{A}$, replace every new color $c = (c', c'')$ by its *complement* $\hat{c} = (c'', c')$ (i.e., complementing c consists of interchanging the ordering of the two old colors c' an c'' that define it). Array $\hat{A}$ has the desired properties (relative to the complemented colors).

4. (*Expansion step*) Append $\hat{A}$ to the right of A, resulting in a $2^t \times 2^{t+1}$ array. This array is turned into a $2^{t+1} \times 2^{t+1}$ one by "expanding" each color c to the two old colors corresponding to it (thus doubling the size of each column).

Figure 2 gives an example of the above four steps for $t = 2$.

We must show that the last ("expansion") step results in an array that satisfies the claimed properties. We do so separately for each property we are trying to prove.

Group property 1: Because we assumed a coloring that is left-to-right legal, we must show that the final array is also right-to-left legal, top-to-bottom legal, and bottom-to-top legal (the proof would be very similar if we had assumed

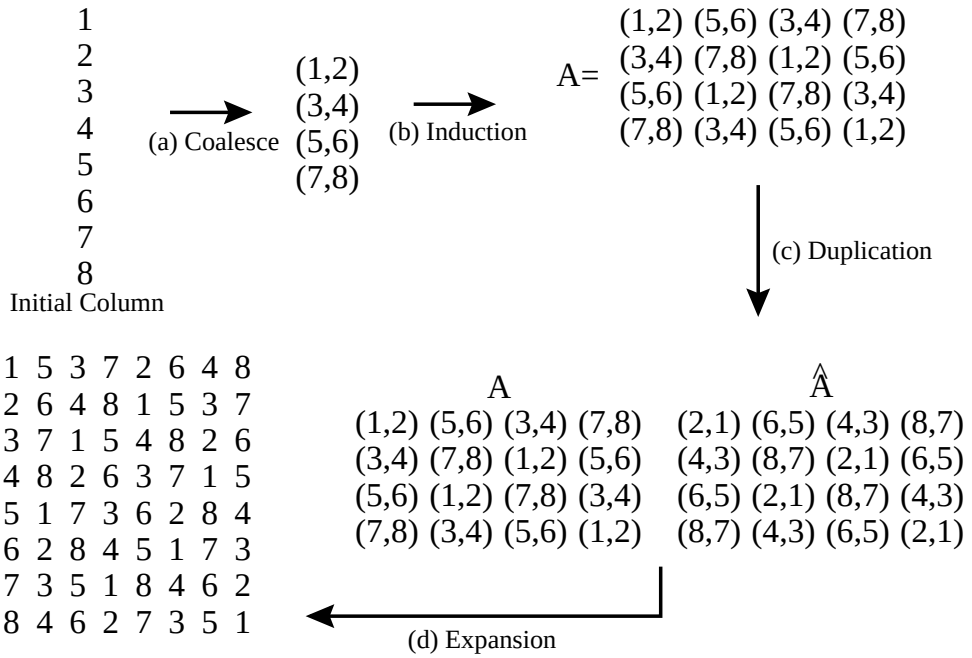

Figure 2: An example of the inductive step for $t = 2$

one of the other three legal colorings rather than left-to-right, so we avoid repeating this argument four times).

That the left-to-right legal array is right-to-left legal can be seen by looking at the resulting rightmost colored column (after the expansion step) and trying to use it as the starting column for a right-to-left legal coloring: In this "right-to-left" process, we would first generate the expanded version of $\hat{A}$ because the unexpanded $\hat{A}$ itself has group property 1 (by the induction hypothesis). Next, the last step of the right-to-left process would duplicate the expanded version of $\hat{A}$ and append a 1-swapped copy of it to the left of the expanded $\hat{A}$: But a 1-swapped version of the expanded $\hat{A}$ is the same as the expanded version of A. Thus the right-to-left process would generate exactly the same array.

We now prove that the left-to-right array is bottom-to-top legal. Starting with the bottom row (after the expansion step), it is easy to see that the next-to-bottom row looks just like a copy of the bottom row but with the left and right halves interchanged. From that point on, the bottom-to-top process does not cause any interaction between the left half of the rows and their right half, and can be viewed as two separate bottom-up processes (one for each half). That the left-half process gives rise to the expanded version of A follows from the fact that A itself is bottom-to-top legal (by the induction hypothesis). Similarly, the right-half process gives rise to the expanded version of $\hat{A}$ because $\hat{A}$ is bottom-to-top legal.

To prove that the left-to-right array is top-to-bottom legal, we use the fact (just proved) that it is bottom-to-top legal, followed by an almost identical argument to the one we used for showing that left-to-right legal implies right-to-left legal (except that the roles of rows and columns are interchanged, "bottom" replaces "left" and "top" replaces "right").

This completes the proof of group property 1.

Superblock property 3: At the bottom of the recursive construction the additive deviation from optimal is zero (verify this – in fact the first deviation from optimality by an additive 1 occurs for $k = 64$). Because the superblocks in S can extend partly over A and partly over $\hat{A}$, each "expansion" step in the construction introduces one more additive unit deviation from optimality. To see that this is so, before expansion let m_1 (resp., m_2) be the number of cells of S in A (resp., $\hat{A}$), γ be the total number of parallel block accesses we use for those m_1 cells in A (independently of $\hat{A}$) and m_2 cells in $\hat{A}$ (independently of A), and let δ be the deviation of γ from the sum of the two individual OPT values of the two pieces of S (that is, the deviation from $\lceil 2m_1/k \rceil + \lceil 2m_2/k \rceil$ where we used the fact that the effective number of colors on each side before expansion is $k/2$). When we "expand" we effectively double m_1, m_2, and the number of colors (which becomes k). We can still process S, after expansion, with γ parallel block accesses, but the deviation of that γ from the new (combined) OPT value can increase by 1 unit because we could have $\lceil (2m_1 + 2m_2)/k \rceil = \lceil 2m_1/k \rceil + \lceil 2m_2/k \rceil - 1$. That is, the number of parallel block accesses needed has stayed same but the overall (combined) OPT has gone down by 1 compared to the sum of the two individuals OPT values (the OPT for the portion of S in A + the OPT for the portion of S in $\hat{A}$). Because a superblock has dimensions $\sqrt{k} \times \sqrt{k}$, the total deviation is $\log \sqrt{k} = 2^{-1} \log k$.

Superblock property 2: Immediately follows from group property 1 and superblock property 3 (because it is the "vertical" equivalent of superblock property 3).

Proof sketch of superblock property 1

For any $\alpha < t$, consider a partition of the leftmost column of a group into 2^α pieces of size $2^{t-\alpha}$ each. Call these pieces $1, \ldots, 2^\alpha$.

Claim. For any piece i ($1 \leq i \leq 2^\alpha$), the $2^\alpha \times 2^{t-\alpha}$ rectangle R of k blocks whose left side is piece i, contains all k colors (i.e., each color exactly once).

Before proving the above claim, we note that it would automatically imply superblock property 1 for $\sqrt{k} \times \sqrt{k}$ superblocks that are "left-adjusted" in the sense that their left side is on the leftmost column of their group (simply by choosing $\alpha = t$ in the claim). That the same is true for superblocks that are further to the right within the block, follows from the observation that the left-to-right legal coloring process maintains the same set of colors from one superblock R to the next superblock immediately to the right of R (it merely permutes the colors). Therefore it suffices to prove the above claim.

We prove the claim by induction on α. The basis ($\alpha = 0$) holds because of group property 3. Now, assume inductively that the claim holds for $\alpha - 1$. We partition the piece i into two halves U ("upper") and L ("lower"): By the induction hypothesis, the $2^{\alpha-1} \times 2^{t-\alpha+1}$ rectangle R_U (respectively, R_L) whose left side is U (respectively, L) satisfies the claim. Now, the colors in the right half of R_U (respectively, R_L) are the same as the colors in the left half of R_L (respectively, R_U) because the last step in the coloring of R consisted of a "swap" that copied the colors of the left half of R_L (respectively, R_U) into the right half of R_U (respectively, R_L). Therefore the set of colors that appear in R is the same as the set of colors that appear in R_U (or R_L), namely the full set of k colors (each color once). This completes the proof of the claim.

4. PROOF OF PERFORMANCE BOUND

If the query is entirely contained in one superblock then our coloring implies a single parallel block access (because no color appears twice in a superblock), which is optimal. We henceforth assume that the query is not entirely contained in a superblock.

We distinguish two cases, depending on whether the query completely contains a superblock or not. We begin with the case where it contains one or more superblocks.

If the query does not completely contain any superblock, then it can be decomposed into at most six subqueries: Four that are each completely contained in a superblock, and two each of which spans (either horizontally or vertically) a set S of superblocks that are (vertically or horizontally) contiguous. The four subqueries that are completely contained in superblocks can each be done in one parallel block access. The other two subqueries are each done with a number of block accesses that is within an additive $2^{-1} \log k$ of the OPT value *for that individual subquery* (by superblock properties 2 and 3 of the previous section). Therefore the total number of parallel block accesses for such queries cannot exceed OPT by more than $3 + \log k$.

If the query completely contains one or more superblocks, then we can partition it into nine subqueries:

1. Four subqueries that are each completely contained in a superblock (these are the four "corners" of the rectangle defining the original query). These subqueries can each be done in one parallel block access.

2. One subquery that consists of all the superblocks that are completely contained in the original query, say, a rectangle of m' superblocks. These can be done in m' parallel block accesses with full disk utilization (i.e., no disk is idle during any of these m parallel steps). This follows from the fact that each color appears exactly once in a superblock.

3. Two subqueries each of which vertically spans a set of superblocks that are vertically contiguous (one of them is just below the top-left corner subquery, the other just below the top-right corner subquery). Each such subquery is done with a number of block accesses that is within an additive $2^{-1} \log k$ of optimal for that individual subquery (by superblock property 2 of the previous section).

4. Two subqueries each of which horizontally spans a set of superblocks that are horizontally contiguous (one of them is just to the right of the top-left corner subquery, the other just to the right of the bottom-left corner subquery). Each such subquery is done with a number of block accesses that is within an additive $2^{-1} \log k$ of optimal for that individual subquery (by superblock property 3 of the previous section).

The above implies that the number of subqueries that can (each) introduce a deviation of 1 from OPT are (at most) 4 "corner" subqueries, the number of subqueries that can (each) introduce an additive $2^{-1} \log k$ from optimal are 4 subqueries that span sets of superblocks that are contiguous along a dimension. The total possible deviation from OPT is then $4 + 4 * 2^{-1} \log k - 1 = 3 + 2 \log k$ (where we subtracted one because even in an optimal coloring at least one parallel block access is needed for these 8 subqueries).

This completes the proof. □

In practice, the deviation from OPT is much less than $3 + 2 \log k$, as shown in Section 6. In particular, experimentally, we found the deviation to be no more than 3, and the average deviation less than 0.5 from the (unachievable) OPT.

5. HIGHER DIMENSIONS

For d-dimensional range queries, we assume a $2^q \times 2^q \times \cdots \times 2^q$ grid of 2^{dq} blocks. We describe two schemes that are natural generalizations of the 2-dimensional scheme.

5.1 Scheme *NEW1*

In the first scheme, we assume $k = 2^{(d-1)t}$ for some integer t, and partition the grid into groups, where each group

is a d-dimensional square of volume $k^{d/(d-1)}$, i.e., it is a $k^{1/(d-1)} \times \cdots \times k^{1/(d-1)}$ grid. Note that a group consists of $k^{1/(d-1)}$ copies of a $(d-1)$-dimensional square of k blocks. Here we describe the coloring scheme for the blocks in a group (the scheme will vary from one group to the next, but for now we focus on how to solve the problem for a single group). We start with some definitions.

Let A be a $(d-1)$-dimensional square of k blocks whose blocks have been colored, and let A' be another similarly shaped square whose coloring is to be derived from that of A. We say that the coloring of A' is a $k^{1/(d-1)}/2$-*swap* of the coloring of A if we first copy the coloring of A into A' and then we do the following: We partition A' into 2^{d-1} $(d-1)$-dimensional sub-squares of size $k/2^{d-1}$ each, and for each of pair of diagonal sub-squares we "swap" the colorings of the pair. For example, if $d = 3$ then A and A' are two-dimensional squares, and the coloring of A' is obtained by first copying the coloring of A into it and then interchanging the colorings of the top-right (TR) and bottom-left (BL) quadrants, and of its top-left (TL) and bottom-right (BR) quadrants, as shown in Figure 3. For example, if A is 2×2 (hence $k = 4$) and the coloring of A is $1, 2$ for row 1 and $3, 4$ for row 2, then the coloring of A' is $4, 3$ for row 1 and $2, 1$ for row 2, as shown in Figure 4.

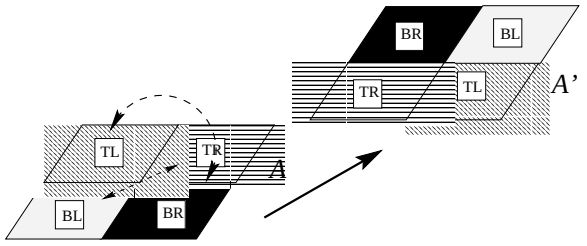

Figure 3: Example of a $k^{1/(d-1)}/2$-Swap

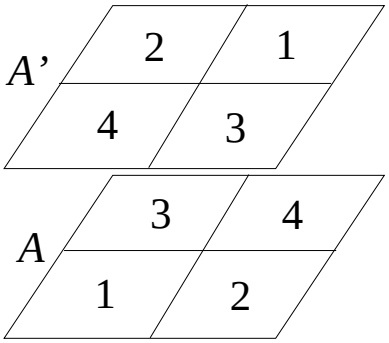

Figure 4: Example coloring for A and A'

More generally, a $k^{1/(d-1)}/2^i$-swap of the coloring of A, for an integer $i \leq t$, is defined by partitioning A into 2^{d-i-1} $(d-1)$-dimensional squares of size $k/2^{d-i-1}$ each and then, *within each square*, doing what we did above (that is, partitioning each square into 2^{d-1} sub-squares and swapping the colorings of the pairs of diagonal sub-squares within each square). Figure 5 shows how the coloring for layer 3 (layer 4) is derived from that of layer 1 (layer 2) by a $k^{1/(d-1)}/4$-swap. For clarity only some of the tiles are shaded to show the swapping.

We are now ready to describe the coloring of a group of $k^{1/(d-1)} \times \ldots \times k^{1/(d-1)}$ blocks. We view the group as consisting of the "stacking" on top of each other, along the "height" dimension, of $k^{1/(d-1)}$ $(d-1)$-dimensional squares of k blocks each (each of which is therefore perpendicular to the height dimension).

1. (`Initial coloring`)
   Assign the colors $1, \ldots, k$ (in any order) to the k cells of the "bottom" $(d-1)$-dimensional square of k blocks.
2. (`''Spreading'' the coloring along the height dimension`)
   For $\mu = 1, \ldots, t$ in turn, do the following: For $j = 1, \ldots, 2^{\mu-1}$ in turn, assign to the $(d-1)$-dimensional square of height $2^{\mu-1}+j$ a coloring that is a $k^{1/(d-1)}/2^{\mu}$-swap of the coloring of the $(d-1)$-dimensional square of height j.

Figure 6 shows an example of the copying and swapping operations for a $4 \times 4 \times 4$ cube beginning with the initial coloring of a 4×4 square (Layer 1). The colors are represented by the letters $a, b, \cdots, p$ to avoid confusion. The coloring obtained above is has interesting properties, one of which is that, for *any* $(d-1)$-dimensional square of k blocks (i.e., even one that is not perpendicular to the height dimension), a group contains the k distinct colors. More on this in the final version of the paper.

Extending the 2-dimensional performance bounds to this kind of higher-dimensional coloring, seems to bring up an unfortunate 2^d additive factor due to the fact that there are now 2^d of the "corners" mentioned in the proof we gave earlier for the 2-dimensional case (but this is a naive bound, however, and a sharper analysis of the worst-case and average-case performance of the scheme is an interesting area of future investigation).

5.2 Scheme *NEW2*

The second scheme for higher dimensions assumes that there are $k = 2^t$ disks or colors, for some integer t. The grid is first divided into sets of blocks of side k. We describe the coloring for each such $k \times k \times \cdots \times k$ blocks. This coloring is periodically repeated to obtain the coloring for the entire grid.

As with the 2-dimensional scheme, we begin by coloring a "column" along any edge of the $k \times k \times \cdots \times k$ block. Beginning with this 1-dimensional column we generate a 2-dimensional coloring by applying the 2-dimensional scheme described in Section 2. This results in a coloring for a 2-dimensional square of side k. Note that the "direction" (i.e. the dimension along which we apply the 2-dimensional coloring) can be any of the dimensions orthogonal to the "column" that was initially colored.

Next we obtain a coloring for a $k \times k \times k$ cube using the coloring for the $k \times k$ square obtained above. To achieve

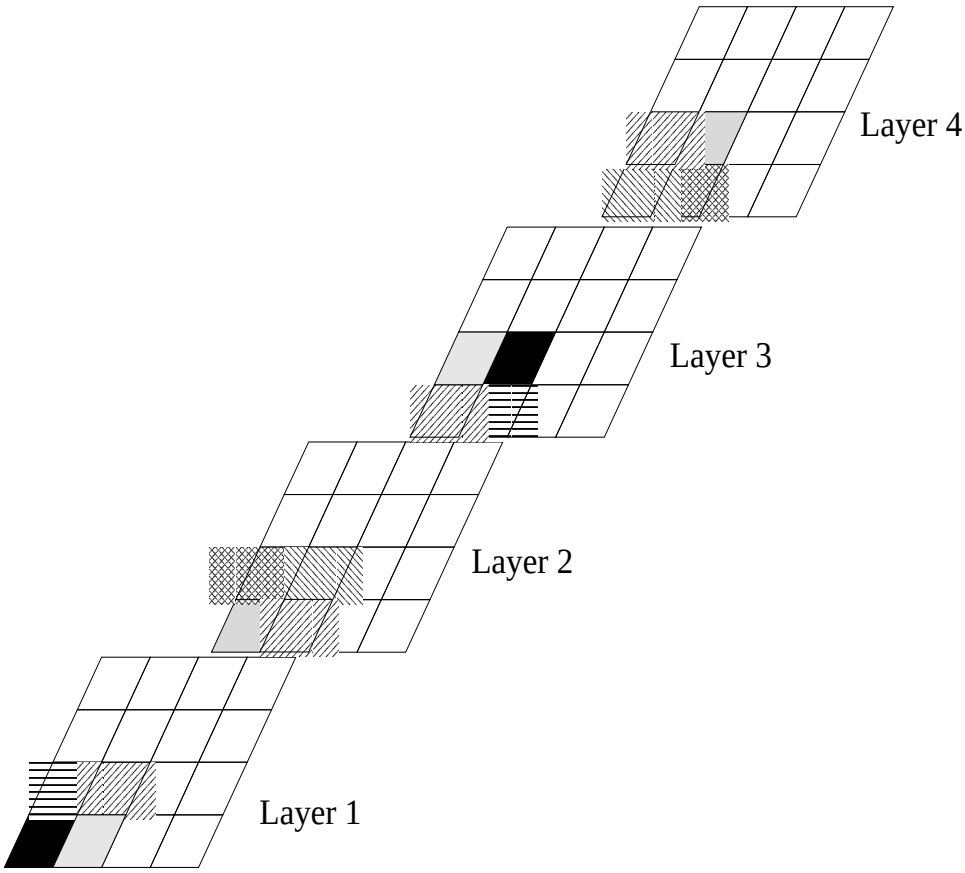

Figure 5: Example of a $k^{1/(d-1)}/4$-Swap

this, the 2-dimensional scheme is reapplied to k columns that were produced by the previous step. There are two sets of "columns" that make up the $k \times k$ square that has been colored in the previous step: those that are parallel to the initial "column" used to obtain the $k \times k$ coloring, and those that run perpendicular to this "column", i.e. the "rows". We apply the 2-dimensional coloring using the set of k "rows" by spreading in a new dimension (direction).

In this manner, we can generate a j-dimensional coloring using the "columns" produced by the $(j-1)$-dimensional coloring and applying the 2-dimensional scheme on each such "column". The orientation of the column is chosen to be different for each application of this step.

1. The coloring obtained above is such that, for *any* $(d-1)$-dimensional square of k blocks (i.e., even one that is not perpendicular to the height dimension), a group contains the k distinct colors.

2. Suppose we are given any coloring obtained in the way we described. Then by starting with *any* already colored "column" of k blocks as the initial one and applying the rest of the coloring steps, we still obtain the same coloring as before.

Figure 7 shows an example of the coloring for a $4 \times 4 \times 4$ cube using this scheme, which we call *NEW2*. The initial column is along dimension 0, the first application of the 2-dimensional coloring is applied to this single column along dimension 1, to generate the coloring for the bottom layer. The next step is to take the 4 "columns" that are parallel to dimension 1 and apply the 2-dimensional coloring to each of these columns in the direction of dimension 2 to yield the 3-dimensional coloring shown in the figure. Proofs similar to the ones given in the previous section for the case $d = 2$, give the following.

6. EXPERIMENTAL RESULTS

In this section, we briefly discuss the performance of our new allocation scheme on sample datasets. The objective of the experiments is to observe by how much the new allocation scheme deviates from the optimal, OPT. Note that we are considering the *difference* between the optimal and the other schemes, and not the *ratio* to the optimal, as has been considered in other work. We also tested the major existing allocation schemes described in Section 1, viz. Disk Modulo (DM) [5], Fieldwise eXclusive (FX) [7], Hilbert Curve Allocation Method (HCAM) [6], and the Generalized Fibonacci (GFIB) scheme from the Cyclic al-

Tiling	*Maximum*					*Average*				
	HCAM	DM	FX	GFIB	NEW	HCAM	DM	FX	GFIB	NEW
4×4	1	1	1	1	1	0.22	0.09	0.05	0.09	0.01
16×16	6	1	1	1	1	0.637	0.070	0.035	0.070	0.014
32×32	12	1	1	1	1	0.998	0.066	0.033	0.066	0.015
33×29	14	1	1	1	1	1.049	0.066	0.033	0.066	0.017

Table 1: Results for various tilings of 2-dimensional data with 4 disks

Tiling	*Maximum*					*Average*				
	HCAM	DM	FX	GFIB	NEW	HCAM	DM	FX	GFIB	NEW
4×4	0	3	3	1	0	0.0	0.46	0.42	0.06	0.0
16×16	5	4	4	2	2	0.697	1.091	0.876	0.300	0.181
32×32	10	4	4	2	2	1.430	0.994	0.795	0.276	0.179
64×64	23	4	4	2	2	2.658	0.954	0.763	0.267	0.178
61×28	15	4	4	2	2	1.797	0.971	0.774	0.271	0.180

Table 2: Results for various tilings of 2-dimensional data with 16 disks

Tiling	*Maximum*					*Average*				
	HCAM	DM	FX	GFIB	NEW	HCAM	DM	FX	GFIB	NEW
16×16	2	12	12	1	1	0.350	2.608	2.392	0.230	0.127
32×32	6	16	16	2	2	0.881	4.464	4.040	0.360	0.336
64×64	12	16	16	2	3	1.850	5.347	4.515	0.447	0.468

Table 3: Results for various tilings of 2-dimensional data with 64 disks

Tiling	*Maximum*						*Average*					
	HCAM	DM	FX	GFIB	NEW1	NEW2	HCAM	DM	FX	GFIB	NEW1	NEW2
$4 \times 4 \times 4$	3	4	8	2	2	2	0.532	0.955	0.911	0.319	0.179	0.463
$8 \times 8 \times 8$	10	4	8	2	4	4	1.434	1.128	0.705	0.371	0.254	0.254
$16 \times 16 \times 16$	40	4	8	2	4	16	4.106	0.970	0.587	0.320	0.231	0.611

Table 4: Results for various tilings of 3-dimensional data with 8 disks

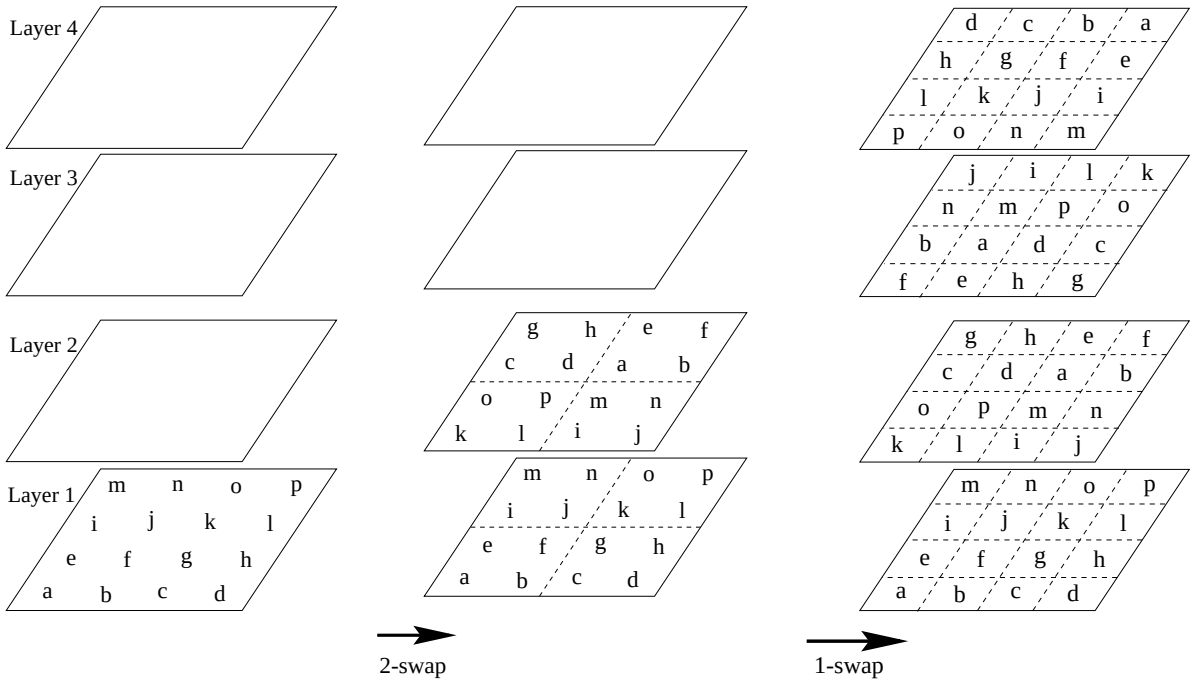

Figure 6: An example of *NEW1* for a $4 \times 4 \times 4$ cube

location schemes [9, 11]. Experiments were conducted for 2 and 3 dimensions. For 2 dimensions, tests were conducted with 4, 16, and 64 (recall that we require $k = 2^{2t}$) disks. In the case of 3 dimensions, tests were conducted with 8 disks. For each combination of number of dimensions and disks, several tilings were tested. In each test, we measured the maximum and average deviation from OPT over all queries. In all experiments all possible range queries were considered.

The results for 2 dimensions are shown in Tables 1 through 3. Table 1 gives the results for 4 disks, Table 2 gives the results for 16 disks, and Table 3 shows the results for 64 disks. The values for the newly developed scheme are shown under the "NEW" columns. These tables give the maximum and average values of the difference between the OPT and the cost for the given scheme over all possible queries for the grid. Thus, from Table 2, we can see that the HCAM scheme made 14 more disk accesses than an optimal scheme for some query, and made 1.049 more disk accesses on the average. The values for the newly developed scheme are shown under the "NEW" columns. As can be seen, the maximum value of the deviation for the new scheme is no more than 2 for 2 dimensions, and the average deviation is less than 0.5 from the optimal OPT. It should be noted that even tilings that are not multiples of 2^t have similar performance for most schemes (only the HCAM scheme is affected significantly).

For the tests with three dimensional data, there are two columns for the two alternative new schemes proposed: *NEW1* and *NEW2*. For high dimensionality compared to the number of disks, the GFIB performed better than the new schemes. We believe a heuristic that incorporates ideas from both schemes will do better than either, and more experimentation along these lines is needed.

7. CONCLUSION

Range queries are an important class of queries for several applications including relational databases, spatial databases, visualization, and GIS applications. For large datasets, the performance of range queries is limited by disk I/O. Performance improvements are typically achieved through parallel I/O by tiling the data set and distributing it among multiple disks or processing nodes. Therefore, in order to process a range query, it is necessary to access only those tiles or blocks that intersect with the query. Though several schemes for the allocation of tiles to disks have been developed, prior to our work this worst-case performance was known only for one scheme with two-dimensional data. Moreover, this bound was a *multiplicative* factor of 3 from OPT. In this paper we developed a novel allocation scheme with guaranteed worst-case performance for any number of dimensions. We showed that any range query on a $2^q \times 2^q$-block grid of blocks can be performed using $k = 2^t$ disks ($t \leq q$), in at most $OPT + O(\log k)$ parallel block accesses. The scheme is generalized to higher dimensions. Experimental data show that the algorithm achieves very close to OPT performance (on average less than 0.5 away from OPT, with a worst-case of 3). Since the new scheme is within a (small) additive deviation from OPT, it gives very good performance independent of the number of disks, the grid size, and the query size. Also, its performance relative to the other schemes (which all tend to give poorer performance as the problem size increases) will be even better for larger data sets.

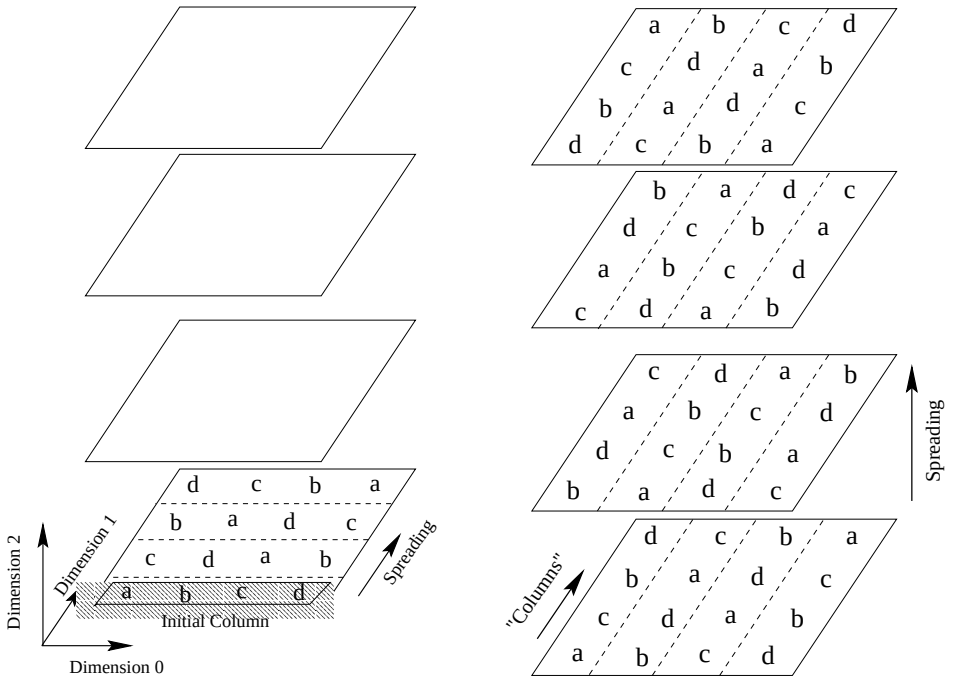

Figure 7: An example of *NEW2* for a $4 \times 4 \times 4$ cube with $k = 4$

Acknowledgment. We are grateful to R. Bhatia, R.K. Sinha, and C-M. Chen for alerting us to the existence of an error in the first draft of this paper (we tracked down the error to our earlier proof of superblock properties 2 and 3). We have recently learned that they have extended our results (in their forthcoming paper [4]) to the case where the number of disks is a product of the form $k_1 * k_2 * \cdots * k_t$ (whereas we assume in this paper that every k_i is 2).

8. REFERENCES

[1] K. A. S. Abdel-Ghaffar and A. El Abbadi. Optimal allocation of two-dimensional data. In *Int. Conf. on Database Theory*, pages 409–418, Delphi, Greece, Jan. 1997.

[2] S. Berchtold, C. Bohm, B. Braunmuller, D. A. Keim, and H-P. Kriegel. Fast parallel similarity search in multimedia databases. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 1–12, Arizona, U.S.A., 1997.

[3] R. Bhatia, R. K. Sinha, and C.-M. Chen. Declutering using golden ratio sequences. In *Proc. of Int'l. Conference on Data Engineering (ICDE)*, San Diego, California, March 2000.

[4] R. Bhatia, R. K. Sinha, and C.-M. Chen. Hierarchical declustering schemes for range queries. In *(To Appear in) Proc. of Intl. Conference on Database Theory*, 2000.

[5] H. C. Du and J. S. Sobolewski. Disk allocation for cartesian product files on multiple-disk systems. *ACM Transactions of Database Systems*, 7(1):82–101, March 1982.

[6] C. Faloutsos and P. Bhagwat. Declustering using fractals. In *Proc. of the 2nd Int. Conf. on Parallel and Distributed Information Systems*, pages 18 – 25, San Diego, CA, Jan 1993.

[7] M. H. Kim and S. Pramanik. Optimal file distribution for partial match retrieval. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 173–182, Chicago, 1988.

[8] J. Li, J. Srivastava, and D. Rotem. CMD: a multidimensional declustering method for parallel database systems. In *Proceedings of the Int. Conf. on Very Large Data Bases*, pages 3–14, Vancouver, Canada, August 1992.

[9] S. Prabhakar, K. Abdel-Ghaffar, D. Agrawal, and A. El Abbadi. Cyclic allocation of two-dimensional data. In *Proc. of the International Conference on Data Engineering (ICDE'98)*, pages 94–101, Orlando, Florida, Feb 1998.

[10] S. Prabhakar, D. Agrawal, and A. El Abbadi. Efficient disk allocation for fast similarity searching. In *Proc. of the 10th Int. Sym. on Parallel Algorithms and Architectures (SPAA'98)*, pages 78–87, Puerto Vallarta, Mexico, June 1998.

[11] S. Prabhakar, D. Agrawal, and A. El Abbadi. Efficient retrieval of multidimensional datasets through parallel I/O. In *Proc. of the 5th International Conference on High Performance Computing, (HiPC'98)*, Chennai, India, December 1998.