Concentric Hyperspaces and Disk Allocation for Fast Parallel Range Searching

Hakan Ferhatosmanoğlu Divyakant Agrawal Amr El Abbadi
Department of Computer Science
University of California
Santa Barbara, CA 93106
hakan@cs.ucsb.edu

Abstract

Data partitioning and declustering have been extensively used in the past to parallelize the I/O that is incurred in range queries. Numerous declustering and disk allocation techniques have been proposed in the literature. However, most of these techniques were primarily designed for two-dimensional data and for balanced partitioning of the data space. As databases increasingly integrate multimedia information in the form of image, video, and audio data, it is necessary to extend the declustering techniques for multidimensional data. In this paper, we first establish that traditional declustering techniques do not scale for high-dimensional data. We then propose several new partitioning schemes based on concentric hyperspaces. We then develop disk allocation methods for each of the proposed schemes. We conclude with an evaluation of range queries based on these schemes and show that partitioning based on concentric hyperspaces has a significant advantage over balanced partitioning approach for declustering in multimedia databases.

keywords: multimedia databases, high dimensional data, declustering, parallel I/O.

1 Introduction

Efficient retrieval of data is a well-studied problem in traditional databases. Several index structures are proposed for efficient range, partial match, and similarity searching in traditional databases. As databases increasingly integrate multimedia information in the form of image, video, and audio data, it becomes necessary to support efficient retrieval of high dimensional data. It has been shown that the techniques for traditional databases do not perform well for high dimensional data [3, 5]. Several index structures have been proposed specifically for dealing with high dimensional data. Some examples of these include the TV-tree [15], the X-tree [5], and the Pyramid Technique [4]. An alternative technique to indexing in databases is to use I/O parallelism for efficient data retrieval. In this approach, the data space is partitioned into disjoint regions, and data is allocated to multiple disks. When users issue a query, data falling into disjoint partitions is retrieved in parallel from multiple disks. This technique is referred to as disk declustering and has been very effective in relational databases. It is very important to find a good way of distributing data to multiple disks. Earlier work on disk declustering [11, 12, 17] has established that good declustering of data on multiple disks results in significant improvements on response time for range queries.

Declustering or disk allocation has been extensively studied in the past especially in the context of relational data and partial match and range queries. Disk Modulo (DM) [9], Fieldwise Exclusive (FX) [14], Hilbert (HCAM) [10], Near Optimal Declustering (NoD) [2], General Multidimensional Data Allocation (GMDA) [13], Cyclic Allocation Schemes [17, 18] are the best known techniques. All of these approaches consider a data set that has been divided into regular partitions along each dimension. In these methods, the data space is first split into equi-sized partitions and then a declustering method is used to allocate these partitions to I/O devices such that the response time of the queries is minimized. We refer to such partitioning technique as *balanced partitioning* and the data space partitions stored in devices are referred to as buckets. The algorithms proposed for balanced partitioning is extended to nonuniform partitioning in [16] and [8]. Besides balanced partitioning, there are additional restrictions on most of the techniques. For example, FX requires that the number of disks is a power of 2. NoD was proposed only for similarity queries and requires binary partitioning in each dimension. There are schemes that have been proposed for only two dimensional data [7, 6, 17].

To process a hyperrectangular query, all buckets that intersect with the query need to be accessed. The cost of executing the query is proportional to the maximum number of buckets accessed from a single I/O device. The minimum possible cost when retrieving A buckets distributed over M devices is $\lceil \frac{A}{M} \rceil$. An allocation policy is said to be strictly optimal if no hyperrectangular area A has more than $\lceil \frac{A}{M} \rceil$ buckets allocated to the same device. The current declustering techniques try to achieve a performance close to the optimal performance for balanced partitioning. However, this optimality definition is valid only for balanced partitioning. For high dimensions, even the optimal allocation for balanced partitioning is not effective. In this paper, we establish that current declustering techniques do not scale for high dimensional data and perform poorly as dimensionality increases. By employing different partitioning strategies, better performance can be achieved. None of the proposed methods could reach the optimal allocation for balanced partitioning. However, even the optimal allocation for balanced partitioning has performance much worse than the proposed techniques in this paper.

Declustering techniques must enable fast parallel query processing. Even a perfect distribution of the buckets to different disks may not lead to fast query processing. As can be seen from the cost formula, there are two important parameters that must be considered for fast parallel searching: A, the number of buckets that is retrieved by the query, and M, the number of disks. These two parameters determine the two important goals of a declustering technique: minimizing the number of partitions that are intersected by the query, and developing an algorithm to distribute the buckets across multiple disks so that retrieval of any set of buckets intersecting a query is maximally parallelized.

Absence of one of these goals would lead to declustering techniques with poor performance. Reaching the optimality of one goal is not enough for fast parallel searching especially in high dimensions. The current approaches to the problem is focused only on the second goal. They try to reach a performance as much close as to the optimal case for the second parameter of the problem. None of the current approaches consider the first goal which plays an important role on the performance of parallel searching. On the other hand, considering only the first goal is also not enough. For a fast parallel search, a fair distribution of the buckets for different partitioning strategies must be found. In this paper, we propose techniques that consider both goals together and develop techniques for fast parallel query processing. A very high speedup over the current techniques is obtained by the proposed techniques in this paper.

The paper is organized as follows. In Section 2, we identify the problems associated with balanced partitioning of high dimensional data. In Section 3, we explore alternative partitioning schemes based on concentric hyperspaces for high dimensional data. In Section 4, we develop disk allocation techniques for data partitioned as the basis of concentric hyperspaces. An evaluation of these proposed partitioning and allocation techniques is performed in Section 5. Conclusions appear in Section 6.

2 High Dimensional Data and Balanced Partitioning

Balanced partitioning is a common assumption for most of the declustering techniques. The data space is divided into N_d parts in d^{th} dimension. In high dimensions, dividing each dimension once creates a huge number of buckets. For $d = 24$, dividing each dimension once creates 16,777,216 buckets. Therefore, for high dimensions, the data space is usually divided into two parts in selected dimensions. This fact is an important point for declustering techniques. There are declustering techniques, such as Near Optimal Declustering for similarity searching [2], which are designed specifically for only the data sets divided into two parts in each dimension.

One of the most important query type in databases is range queries. A range query specifies a range of values for each dimension. The query result is the set of all data objects that have values within the specified range in each dimension. We define the selectivity of range query q, denoted $\mathcal{S}_q$ as follows:

$$\mathcal{S}_q = \frac{r}{D},$$

where r is the number of elements in the result set of q and D is the total size of the database. Note that the volume of the query rectangle will determine the selectivity of a query, since data is assumed to be uniformly distributed. We now would like to estimate the expected length of each side of range query q, for a given selectivity $\mathcal{S}_q$. This estimation is needed to determine the number of data partitions that will be included in the query rectangle inscribed by q. Without loss of generality, we assume that the data space is a *unit hypercube*, i.e., $[0,1]^d$, where d is the dimensionality of the data space.

The volume of a hypercube with d dimensions and edges with length a is a^d. Similarly, the data space, which is a hypercube with unit length edges, has a volume of 1. Under uniform distribution, the selectivity of the range query q, $\mathcal{S}_q$, is the ratio of the query volume V_q to the total volume of the entire data space, V, i.e.,, $\mathcal{S}_q = V_q/V$. Since $V = 1$, we have $V_q = \mathcal{S}_q$. Assuming d dimensional data space, we can compute the length of each edge, a_q, of a hypercubic query q as follows:

$$\begin{aligned} a_q &= (V_q)^{\frac{1}{d}} \\ &= (\mathcal{S}_q)^{\frac{1}{d}} \end{aligned}$$

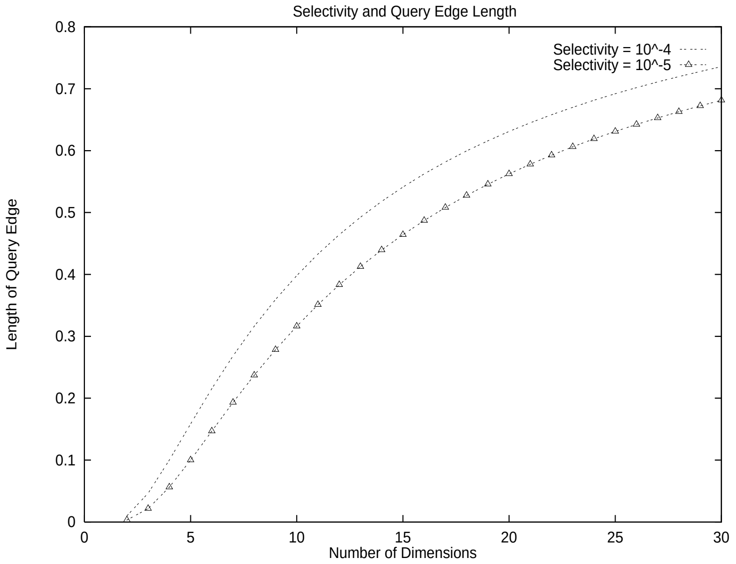


Figure 1: Query Edge and Selectivity

For $d = 16$, to have a selectivity of at least 10^{-4}, the side of a hypercube range query must be at least 0.56. Figure 1 shows the expected length of hypercube range queries with selectivities of 10^{-4}, and 10^{-5} for different dimensions. This figure illustrates that for high dimensions the hypercube range queries with desirable selectivities have edges larger than 0.5. For example, for 13 dimensions, if we specify a range query with sides of length 0.5, the volume which is covered is around 10^{-4} of the whole data space. Thus, the selectivity of the query is approximately 10^{-4}. Decreasing the length of the sides of range query will result in low selectivities. This problem becomes more acute in higher dimensions. In high dimensions, queries with small sizes generally will give result sets of size zero. Specifying a range query with sides of length 0.1 in d dimensions will give results with selectivity of 10^{-d}. For instance, for $d = 24$, specifying a range query with sides of length 0.1 will give a selectivity of 10^{-24} which is an unreasonable selectivity value, and most likely will not result in any answers.

The cost of the range query is proportional to the number of buckets that is read during the query processing. All the buckets that are intersected with the query rectangle must be retrieved since they may potentially contain data relevant to the query. We now examine the impact of range queries on high-dimensional data space that has been declustered on multiple disks.

Let us consider a range query with sides of length 0.5 and more. For any dimension, the difference between the maximum range and the minimum range given by the range query is greater than 0.5. The minimum point of the query in a particular dimension must be less than 0.5, and the maximum point is greater than 0.5. We have already argued that partitioning the high dimensional data space in more that two parts in each dimension is unrealistic, since that will give rise to a very high number of buckets. If the data space is divided into two parts, in each dimension, referred to as balanced binary partitioning, then the range query will intersect all the partitions in each dimension. Therefore, all of the buckets in the system must be read by the range query. As mentioned above, high dimensional range queries are expected to have edges large enough, usually more than 0.5, to have a reasonable selectivity. Hence, most of the high dimensional range queries intersect all the partitions of the data space which causes significant performance degradation. Even if the data space was divided into more than two partitions, which may happen very rarely, the number of partitions in each dimension would not be a large number to avoid huge number of buckets created at the end. Thus, even in this case, the number of intersected partitions is again very high.

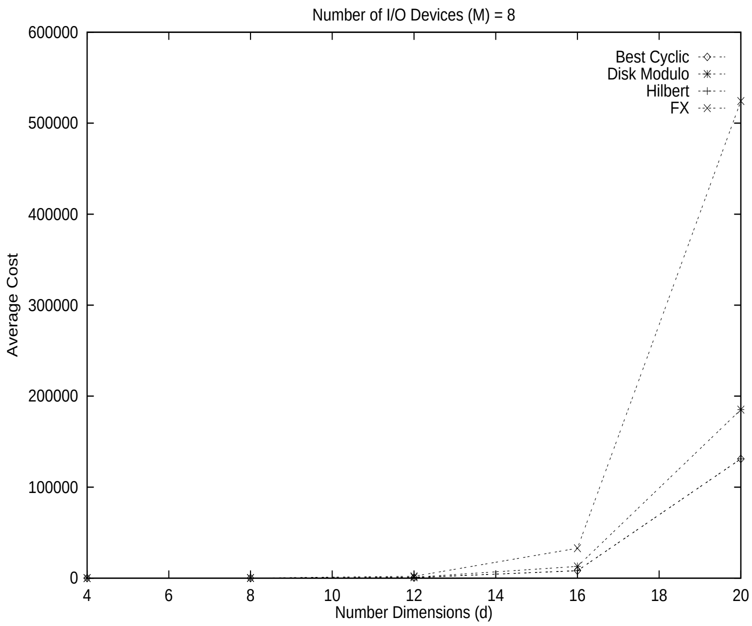
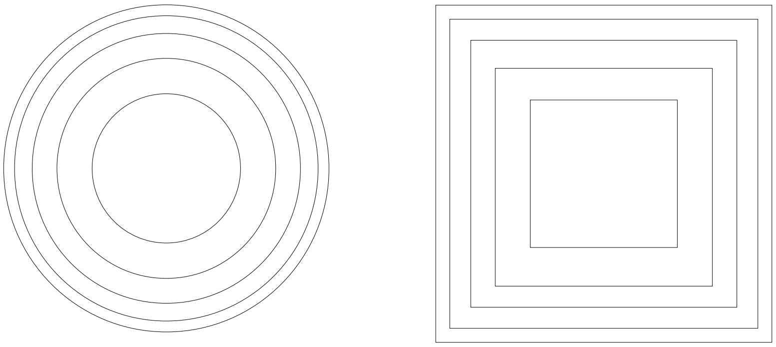


Figure 2: Declustering Techniques with Dimensionality

Since the number of buckets that must be read is very high, even the best declustering method



(a) (b)

Figure 3: Concentric Regions

for balanced partitioning does not have good performance. Figure 2 shows the performance of current techniques as the dimensionality increases. It is seen from the figure that there is a rapid increase in the cost of the queries as dimensionality increases. The performance degradation of the declustering methods can be avoided by using other kinds of partitioning and finding good declustering techniques for these partitioning.

3 Partitioning based on Concentric Hyperspaces

An important property of the hyperspaces is that the volume near the surface of the hyperspace dominates the inside volume as dimensionality increases. Therefore, most of the data will be very close to the surface of the hyperspace [2]. For example, for a dimensionality of 21, under uniform distribution, the probability that a data point is within 0.1 distance of the surface is more than 99%. Note that the cost of the parallel query processing depends on the first parameter A, the number of buckets intersected by the queries. For fast parallel searching it is desirable that this number is minimized. Since the data space near the surface of the data set is dominant in high dimensions, careful partitioning strategies must be applied for the volume near the surface so that it is finely divided. Balanced partitioning causes surface volume lumped with the inside volume, hence does not achieve the desired way of partitioning of the surface. The assumption of balanced partitioning causes performance degradation of declustering techniques because of the poor performance obtained for the first parameter.

A different partitioning approach would be to start from the center of the space and create hypershells of specified volume going inside out. These *concentric hyperspaces* will lead to partitions such that the inside partitions are thicker and the outer partitions (near the surface) become thinner and thinner as shown in Figure 3. There are different ways of concentric partitioning. We now explore different geometries for these concentric partitions.

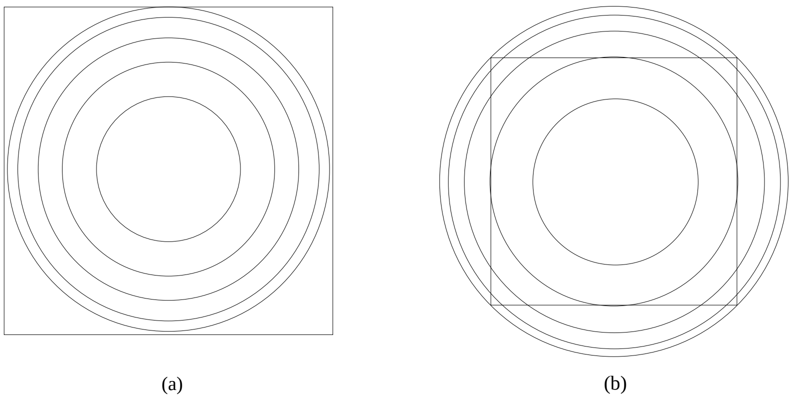


Figure 4: Concentric Hypersphere Partitioning

3.1 Concentric Hyperspheres

The first geometry we explore for creating concentric partitions is that of hyperspheres as shown in Figure 3(a). Note that we will continue to assume throughout the paper that the data space is a unit volume hypercube. The center of this data spaces as well as all the hyperspheres will be $[x_1, x_2, \ldots, x_d]$, such that each $x_i = 0.5$. The innermost partition is a solid hypersphere whereas the remaining partitions are hyperspherical shells between two hyperspheres as shown in Figure 3(a). Note that the partitions must encompass the entire data space which is a hypercube. Due to the mismatch between the two geometries, that of the data space and that of the partitions, there are two ways to encompass all the data points in the hyperspherical partitions. The first possible way of partitioning is illustrated in Figure 4(a). The last hypersphere is inscribed inside the hypercube such that the radius of the largest hypersphere is half the length of the edge of the data space, i.e., the radius is 0.5; the remaining data space that is not covered by the last hypersphere will be considered as different shaped partitions. The second way of partitioning, illustrated in Figure 4(b), is to inscribe the data space inside a hypersphere with the radius half the diagonal of the data space, i.e., the radius is $\sqrt{d}/2$. The second approach results in empty space being included in all hyerspheres with radius greater than 0.5.

In both approaches there are hyperspherical shells which are the partitions that are concentric. In the first approach (Figure 4(a)), the remainder data space that is left outside the last hypersphere inscribed in the data space, result in 2^d disjoint regions. One approach for declustering will be to use a round-robin allocation for both the hyperspherical shells and 2^d remainder regions. It is desirable that the volume of these different shaped partitions is small compared to hyperspherical shell partitions so that a regular and efficient declustering algorithm can be applied to the data space. However, we show that as d increases, the volume encompassed by these remainder regions becomes very large and requires further partitioning. This is because, it can be shown that for hypersphere inscribed inside a hypercube with d dimensions, the ratio of the volumes of the hypersphere to the hypercube, R, is[1]:

$$\begin{aligned} R &= \frac{\pi^{d/2}/\Gamma(d/2+1)}{2^d} \\ &\approx (\pi d)^{-1/2} \left(\frac{\pi e}{2d}\right)^{d/2} \end{aligned}$$

As d increase, the ratio R tends to zero. As the diagonal of the data space which is a hypercube increases with $d^{1/2}$, there is increasingly more data space between the cube and the inscribed sphere. The first proposed method for hypersphere partitioning will create a huge number of partitions with different shapes. The ratio of hyperspherical shell regions to these different shaped regions goes to zero as dimensionality increases. Therefore a declustering algorithm that exploits the idea of hyperspherical shells will lose performance as the number of dimensions increase. This indicates that the notion of concentric hyperspace will lose its dominance and therefore will not be effective.

A similar problem arises with the second approach (Figure 4(b)) for hypersphere partitioning. For a cube inscribed in a sphere the ratio of volume of the sphere to the the volume of the cube is given as:

$$R' = \frac{\pi^{d/2}/\Gamma(d/2+1)}{2^d} \cdot d^{d/2}$$

which increases indefinitely as d increases. This indicates that the hyperspherical shells with radius greater than 0.5 will be mostly empty spaces. From here, we conclude that due to the mismatch between the two geometries, that of the data space and the partitions, concentric hyperspheres may not be useful for declustering.

3.2 Concentric Hypercubes

The next geometry for concentric hyperspaces we examine is that consisting of a hypercube at the center of the data space followed a series of hypercubic shells leading up to the entire data space as shown in Figure 3(b). One of the first problems we confront when dividing the data space in this way is to determine the size of each of these shells. Determining the size of each bucket or partitions under balanced partitioning is rather straightforward. Each dimension i is divided into N_i parts. Then, the total number of partitions in the data space is $N_1 \times N_2 \times \ldots \times N_d$.

Now, we will explain how to partition the data space into p equally filled partitions with Concentric Hypercube technique. First, p concentric hypercubes, $\Box_1, \Box_2, ..., \Box_p$ are created inside the unit hypercube. These hypercubes have a common center (0.5, 0.5, ..., 0.5). The first hypercube, $\Box_1$, is a partition itself, $part_1$. The other partitions are the hypercubic shells between two consecutive hypercubes. The second partition, $part_2$, is the hypercubic shell created by $\Box_1$ and $\Box_2$. Similarly, $part_i$ is created by $\Box_{i-1}$ and $\Box_i$. The hypercube that covers all the hypercubes,$\Box_p$ is the unit hypercube itself. The hypercubes inside the data set have sides of length in the range of [0..1].

[1] where the gamma function is defined as $\Gamma(n+1) = n!$ and the volume of a hypersphere with dimension d and radius r is $r^d \cdot \pi^{d/2}/\Gamma(d/2+1)$ [19].

Since the volume of the data space is 1 and there are p partitions, the volume of each partition is $1/p$. Therefore, the volume of $\Box_1$ which is also $part_1$ is:

$$\begin{aligned} Volume(\Box_1) &= Volume(part_1) \\ &= 1/p \end{aligned}$$

Since each partition has the volume of $1/p$, we can find the volume of concentric hypercubes. In general, the volume of hypercube $\Box_i$, for $i > 1$, is:

$$Volume(\Box_i) = Volume(part_i) + Volume(\Box_{i-1})$$

By solving the above recurrence relation $Volume(\Box_i)$ is i/p. From this we can compute the length of the edge, a_i of hypercube $\Box_i$ as $\sqrt[d]{i/p}$. Since $a_i/2$ is the distance of the surface of the hypercube $\Box_i$ from the center, we can use this value to partition the data space in the desired manner. Note that this information (the size of each hypercubic shell) will also be used to compute the intersection of any rectilinear range query with concentric hypercubic shells.

3.3 Hyperpyramids and Hypertrapezoids

Berchtold, Böhm, and Kriegel [4] have recently proposed another geometry based on pyramids as being useful for indexing high-dimensional data. We explore this geometry in the context of concentric hyperspace based partitioning. Hyperpyramid partitioning is a special case of concentric partitioning. The partitions in the hyperpyramid partitioning are the hypertrapezoids created by dividing each hypercubic shells into $2d$ hypertrapezoids.

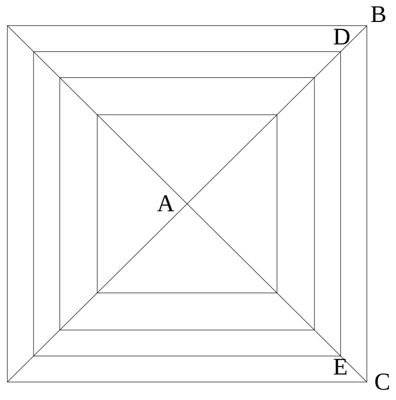


Figure 5: Pyramids and Trapezoids

Instead of creating a hypercube in the center of the data space, the data space is divided into $2d$ *major* hyperpyramids and each hyperpyramid is divided into concentric hyperpyramids such that the apex of each pyramid is the same, i.e., the center of the data space. The hypertrapezoids created in this way are the actual partitions. In Figure 5, for two-dimensional case, $\triangle ABC$ is a *major* hyperpyramid, $\triangle ADE$ is a hyperpyramid, and $\Box DBCE$ is a hypertrapezoid.

We now develop the hyperpyramid partitioning strategy to create p partitions with equal volumes. As before, the volume of the data space is 1. Hence, the volume of each partition must be $1/p$. The point $(x_1, x_2, ..., x_n)$, where $x_i = 0.5$, is the apex for each hyperpyramid. There are totally $2d$ *major* hyperpyramids in the data space, where a major hyperpyramid is the one whose base corresponds to the surface of the unit hypercube. Since each major hyperpyramid has the same size, the volume of a major hyperpyramid is $1/2d$.

First, $p/2d$ hyperpyramids, $\triangle_1, \triangle_2, ..., \triangle_{p/2d}$ with a common apex are created inside each major hyperpyramid. The distance between the base of the hyperpyramid $\triangle_i$ and the apex is defined as the height, h_i. Hyperpyramid $\triangle_1$ inside a major hyperpyramid is an actual partition. Since the volume of each partition is $1/p$, the volume of the top hyperpyramid, inside each major hyperpyramid is $1/p$. There are $2d$ hyperpyramids in the first level partitions, $part_1$, which are the top hyperpyramids of major hyperpyramids. These $2d$ partitions are the only partitions which are hyperpyramids. The partitions in the other levels are hypertrapezoids. The second level partition in a major hyperpyramid, $part_2$, is the hypertrapezoid which is created by the top hyperpyramid, $\triangle_1$, and the smallest hyperpyramid that covers the top one, $\triangle_2$. Similarly, the i^{th} level is the one created by the hperpyramids, $\triangle_{i-1}$ and $\triangle_i$. An interesting property of these partitions is that all $2d$ partitions at the same level together corresponds to a hypercubic shell partition, and the levels together create the concentric hypercube partitioning.

The height of each concentric hyperpyramid is needed for partitioning the data space and finding intersections between the queries and the partitions. Therefore, we must find a general formula to compute the height of the pyramids such that the volume of each partition is $1/p$. In the base case, the volume of the top level pyramid is:

$$\begin{aligned} Volume(\triangle_1) &= Volume(part_1) \\ &= 1/p \end{aligned}$$

The recurrence relation is as follows:

$$\begin{aligned} Volume(\triangle_i) &= Volume(\triangle_{i-1}) + Volume(part_i) \\ &= i/p \end{aligned}$$

As mentioned above, hyperpyramid partitioning is a special case of concentric hypercube partitioning. We will find the height values with the help of concentric hypercube formulas. Let $\Box_i$ be the hypercube that has the apex of the pyramid as its center and the base of the pyramid as one of its hypersurfaces and let a_i be the length of each side of the hypercube. Then,

$$\begin{aligned} Volume(\Box_i) &= Volume(\triangle_i) \times 2d \\ &= (i/p) \times 2d. \end{aligned}$$

We also know that, $Volume(\Box_i)$ is a_i^d. From, equation above, we can compute $a_i = \sqrt[d]{2 \cdot i \cdot d/p}$. The height of the hyperpyramid is the half of the length of the hypercube. Hence, the height of pyramid $\triangle_i$ is $(\sqrt[d]{2 \cdot i \cdot d/p})/2$. By using this general height formula for hyperpyramids, we can divide the data space, and can determine the intersections of the queries with the partitions.

4 Allocation Methods

We have proposed the concentric hyperspace based partitioning techniques to reduce the number of buckets intersected by the queries. After the reduction, the next step is to develop allocation methods for the placement of the buckets to multiple I/O devices such that the retrieval of any set of buckets is maximally parallelized. In this section we propose several declustering methods for the different partitioning strategies developed in this paper. In contrast to existing techniques, our declustering methods use special partitioning strategies and allocation techniques for these strategies to optimize the two parameters for the general declustering problem. We first describe a disk allocation method for concentric hypercubes. Next, we describe allocation methods for hyperpyramids and hypertrapezoids.

4.1 Declustering of Concentric HyperCubes

In [1], it is shown that strictly optimal allocation for balanced partitioning data space exists under very restrictive conditions. For high dimensions, these conditions are further constrained. Now, we will show that there exists a strictly optimal allocation method for concentric hypercube partitioning. By using this allocation technique, optimality for the second parameter is achieved which is the only parameter that is worked on by the current techniques. Since none of the techniques achieved strict optimality, the dominant improvement achieved by this declustering technique is on the second parameter, parallelization.

The allocation method is: $disk(part_i) = i \ mod \ M$, where M is the number of disks and I/O devices available to allocation. The method assigns the buckets to disks by starting from the innermost partition and sequentially going through edges, the remaining partitions, in a round robin fashion. This allocation is strictly optimal. Since the partitioning is done in a concentric manner, if a range query intersects two hypercubic shells $part_i$ and $part_j$, such that $i < j$, then it also intersects all the hypercubic shells between $part_i$ and $part_j$, i.e., $part_{i+1}, ..., part_{j-1}$. Assume that the range query intersects only the set of partitions $part_i$, $part_{i+1}$, .., $part_{j-1}$, $part_j$. With this allocation method, the maximum number of buckets that is assigned to a single disk is $\lceil \frac{j-i+1}{M} \rceil$ which is also the optimal cost for retrieving these buckets from M disks. The performance impacts of this method will be examined in the next section.

4.2 Declustering of Hyperpyramids and Hypertrapezoids

We now develop declustering techniques based on the hyperpyramids and hypertrapezoids. The techniques that we propose for hyperpyramid partitioning is based on the idea of allocating hyperpyramid and hypertrapezoid buckets in a cyclic manner with extra shifting if necessary. It has been shown that many existing declustering methods are based on the idea of cyclic allocation [17]. Although the basic idea of these declustering schemes are similar, resulting performance differs in a dramatic way [17, 18].

We first explain the idea of two-dimensional cyclic allocation techniques. Given a value of M, each cyclic declustering method is distinguished by the value of the skip, H, that it uses. Under balanced partitioning, the partition (0,0) is assigned to device 0. Next, the partitions along the row 0 are assigned to consecutive devices, i.e., partition (0, j) is assigned to device $j \ mod \ M$. Each partition in row 1, (1, j) is assigned to device $(H + j) \ mod \ M$, H devices away from the device on which the partition from the same column in row 0 was assigned. Similarly, each partition on row $i, (i, j)$ is assigned to device $(Hi + j) \ mod \ M$. This procedure can be easily extended to high dimensional data. The partition given by $(x_0, x_1, ..., x_{d-1})$ in d-dimensions is mapped to $(x_0 * H_0 + x_1 * H_1 + .. + x_{d-1} * H_{d-1}) \ mod \ M$. The core problem of cyclic allocation is to find appropriate skip values. Different skip values perform very differently and using appropriate skip values becomes a complex problem that can only be solved by trial and error.

In order to use the idea of two dimensional cyclic schemes, we will use a special transformation of the high dimensional data space to two dimensions. By this transformation, we can extend the current techniques to the ones for hyperpyramid partitioning. It is also easier to see the allocation techniques in two dimensional tables, rather than actual hyperpyramid partitions. In the hyperpyramid partitioning, there are $2d$ major hyperpyramids in the dataset. Each major hyperpyramid has $p/2n$ hyperpyramid shells, hypertrapezoids and a hyperpyramid, which are the actual partitions. These shells are the buckets that have to distributed among multiple I/Os. The major hyperpyramids correspond to the rows of the two dimensional table. The 1^{st} row represents $\triangle_1$, the 2^{nd} row represents $\triangle_2$, and so on. The columns represent the hyperpyramid shell levels. The 1^{st} column in each row represents the top hyperpyramid, $part_1$, in each major hyperpyramid. The 2^{nd} column represents the hypertrapezoid $part_2$, and so on.

Hyperpyramid partitioning is not as regular as the balanced partitioning and the concentric hypercube partitioning. However, with the help of this transformation, the complexity of the hyperpyramid partitioning in high dimensions does not effect the declustering techniques that we propose. Moreover, this transformation allows us to use all of the existing techniques for balanced partitioning. The structure of two dimensional balanced partitioned table and our transformed table are similar to each other. However, the entries in these tables have different meanings in terms of bucket representation. The entries in transformed table represent corresponding buckets in concentric partitioning. For example, in our transformed table, the first column of each row is the bucket that is located in the center of the data space. Hence, all entries in the first column, i.e. $(1,1),(2,1),...,(2d,1)$, are adjacent in the real data space. However, the existing techniques, that use the balanced partitioning, consider the buckets $(2d,1)$ and $(1,1)$ as being far away from each other. Therefore, the current techniques need some extensions to be applicable for our two-dimensional table. We use the extra shifting idea that was proposed in [13]. We apply the general cyclic allocation algorithm to our transformed table and if a row is allocated with exactly the same I/O devices with the previous checked row, we shift the current row by one and mark it as new check row.

We combine the idea of cyclic allocation and extra shift and try various allocation methods with different skip values with and without extra shifting. Our experiments show that the Extended Cyclic Allocation Schemes which use extra shift performs better than the ones that do not use extra shift for hyperpyramid partitioning. In particular, Extended Cyclic Allocation with skip value of $\lfloor\sqrt{M}\rfloor$ gives the most promising results. Hence, we use this method in our performance evaluation for *Hyperpyramid based declustering.* We note that, Cyclic Allocations with skip values of $\lceil\frac{M}{2d}\rceil$ and 1 give similar results with this technique.

5 Performance Evaluation

We now evaluate the performance of newly proposed high dimensional declustering techniques based on different partitioning strategies. The comparisons are made with the current best known techniques. We also compare our techniques with the special technique, *Best Cyclic Scheme* [17, 18], which is proposed by determining the best skip value in each dimension. Each cyclic scheme uses predetermined skip values. Instead of these skip values, one can choose the best skip value for each dimension. Clearly, this will perform better. Hence, *Best Cyclic Scheme* is the current limit on the performance of cyclic allocation methods. We note that the computation of these best skip values is very expensive, so it is not realistic to compute them in real-life applications especially when such applications involve high dimensional data. Although *Best Cyclic Scheme* is the current best technique, it is not strictly optimal. A lower bound on the cost of the query is given by $\lceil\frac{A}{M}\rceil$ where A is the number of buckets retrieved by the query and M is the number of disks. However, our experiments show that *Best Cyclic Scheme* is close to the optimal for *binary* balanced partitioning. Hilbert (HCAM) also has a similar performance for balanced binary partitioning.

Our simulation system consists of the implementation of the proposed techniques and the current best known declustering techniques. First of all, the data space is partitioned by using the corresponding technique for each declustering method. The formulas given in Section 3 is used for this part of the system. DM, FX, HCAM, and all Cyclic Schemes use balanced partitioning. Our techniques are based on Concentric Hyperspace Partitioning (concentric hypercubes and hyperpyramids). In order to compare the existing techniques with ours, we perform the concentric partitioning such that that the number of partitions, p, and the number of data points in each partition is same as in balanced partitioning.

After partitioning the data space, we evaluate the techniques with high dimensional range queries. To process a range query, first, the set of buckets that will be retrieved by the query is computed. This set consists of the buckets intersected by the query. The intersection of the buckets with queries are computed by the partitioning formulas discussed in Section 3 and the intersection formulas of a pyramid and a rectangle given in [4]. The buckets that are retrieved during the query are read from the I/O devices which are identified by different declustering techniques. The cost of a particular query is computed to be the maximum number of buckets retrieved from the same I/O device. Hypercubic and hyperrectangular range queries are used in different experiments. These queries are created with realistic selectivities, such as 10^{-4} and 10^{-5}. All of the edges of a hypercubic query have equal length. A hyperrectangular query may have edges with different lengths. We ran 500 random queries for each experiment and the average cost of these queries is computed. This average cost is used as a measure of performance for the declustering technique.

The experiments were done for several dimensions. As dimensionality increases, the performance improvement gained by our techniques becomes more significant.

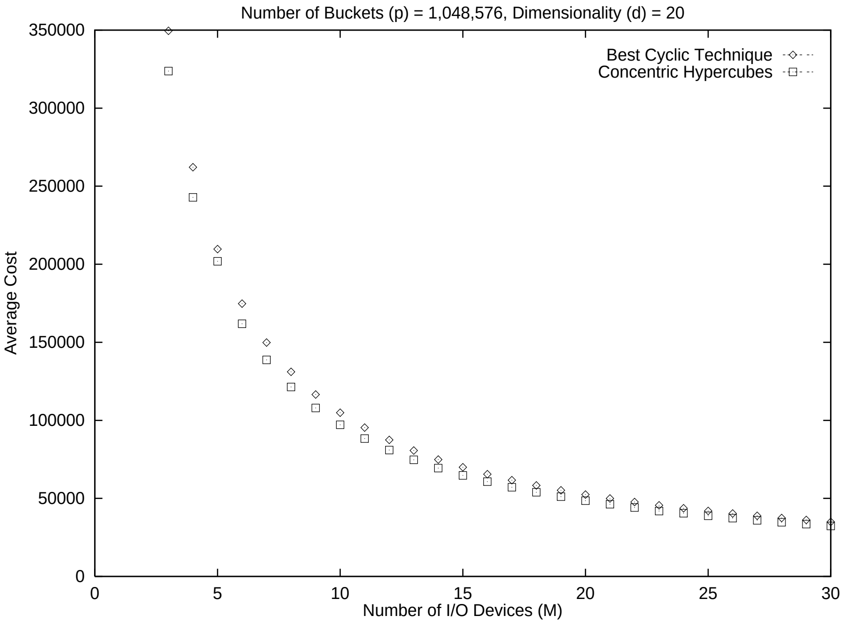


Figure 6: Hyperrectangular Range Queries

In our first set of experiments, we fix the number of dimensions and vary the number of I/O devices. First we evaluate the performance of *Hypercube based declustering.* Figure 6 illustrates the performance comparison of *Hypercube based declustering* and *Best Cyclic Scheme* with hyperrectangular queries. For all values of M, *Hypercube based declustering* performs better. We repeat the same experiment by using hypercubic range queries. In this setup, the queries are chosen to

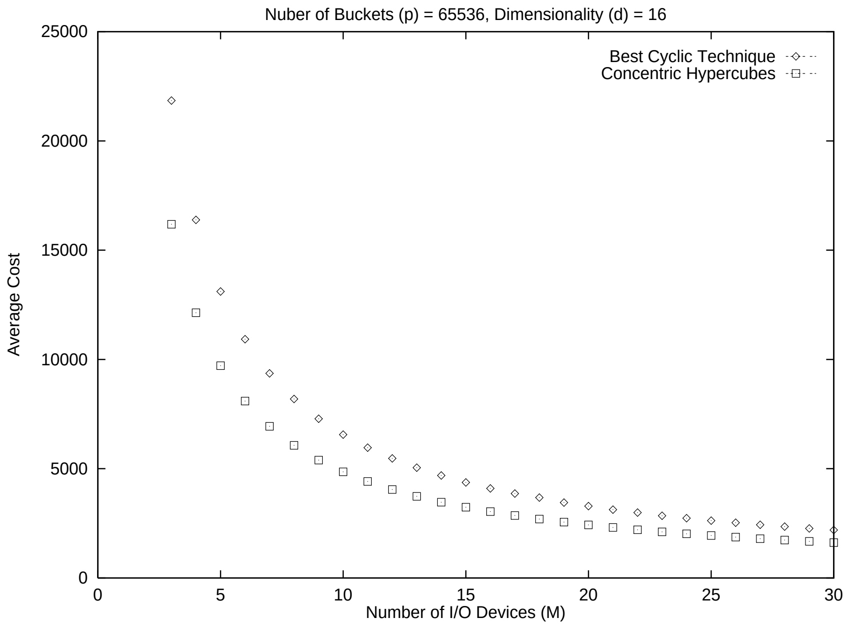


Figure 7: Hypercubic Range Queries

have equal size edges in each dimension. We observe that *Hypercube based declustering* provides significantly better performance with hypercubic range queries rather than hyperrectangular range queries. (Figure 7). This difference can be explained by the probability of having query edge near to the surface in hyperrectangular queries. Even if the query does not cover much space, having long ranges in some dimensions may cause the query to intersect with high number of buckets. This probability is lower with hypercubic queries. This fact leads us to have a geometry where the dimensions can be considered separately, i.e., extreme range values in some dimensions do not effect the others. Dividing the hypercubes into several parts w.r.t each $d-1$ dimensional surface is a possible solution to the problem. A way to do is the employ the hyperpyramid partitioning as mentioned in Section 3 .

Figure 8 and Figure 9 illustrate the performance comparison of several declustering techniques with newly proposed Concentric Hyperspace based Disk Allocation Techniques, hypercubes and hyperpyramids. Figure 8 illustrates the performance of *Best Cyclic, DM, Hilbert, FX, Hyperpyramid based declustering, and Hypercube based declustering* with hypercubic range queries. The dimensionality of the data is 16, and each dimension is divided into two parts, creating 65,536 buckets in the system. It can be seen that *FX* and *DM* do not perform well for high dimensional data. It is interesting to note that although *HCAM* does not perform well in general ([18]), it performs well in binary balanced partitioning and hypercubic queries. *HCAM* was originally proposed to perform well for square queries. We observe from Figure 8 that *Hyperpyramid based declustering* provides very significant improvement over all the existing technique. As dimensionality increases

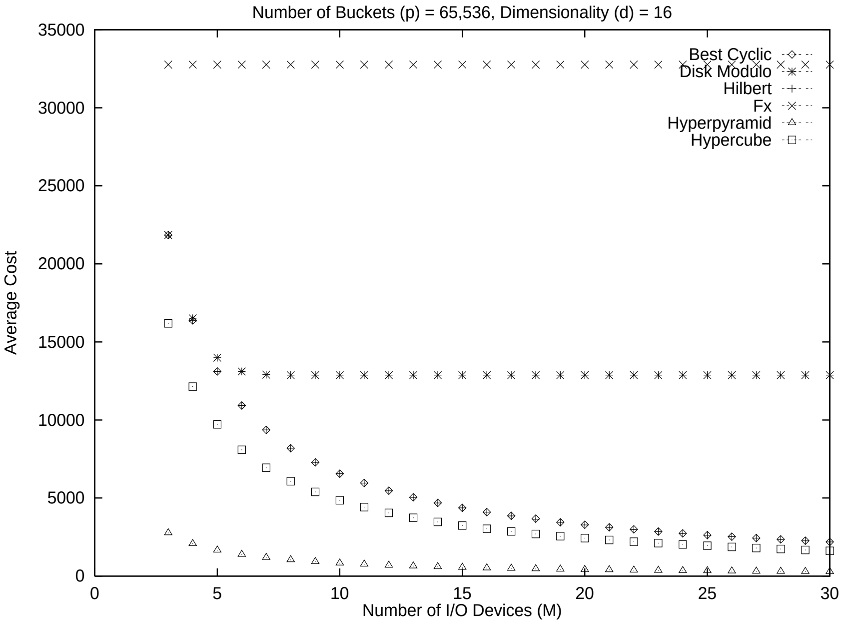


Figure 8: Hypercubic Range Queries

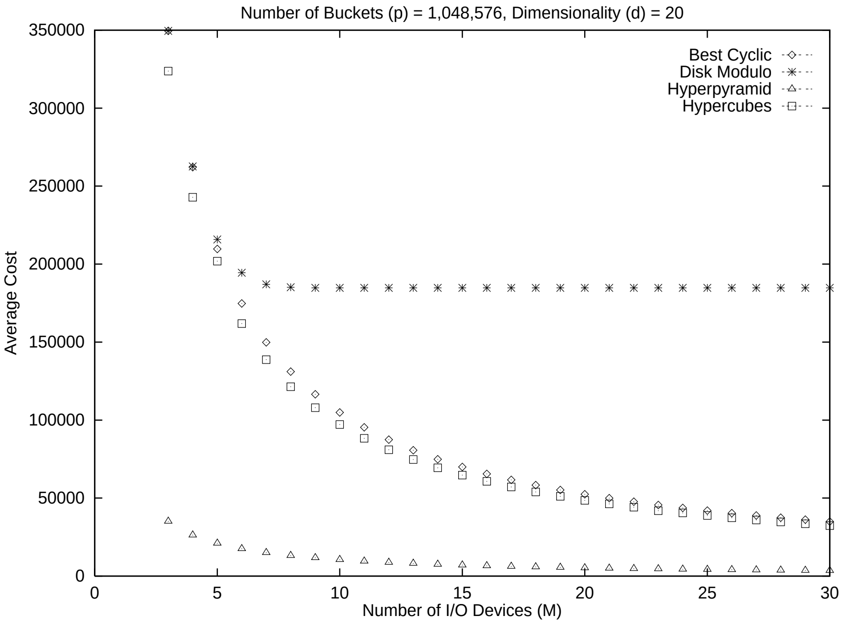


Figure 9: Hyperrectangular Range Queries

this improvement becomes more apparent. Figure 9 illustrates a similar experiment this time with hyperrectangular queries. The dimensionality is 20 and 1,048,576 buckets are created. Again hyperpyramid based declustering performs significantly better than the others. We note that in each graph, *Hypercube based declustering* performs better than the existing techniques but worse than the *Hyperpyramid based declustering.*

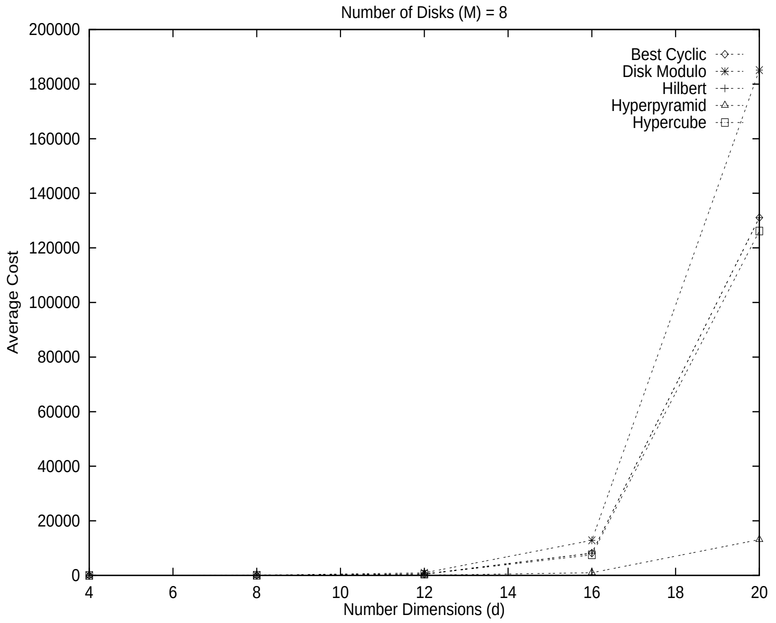


Figure 10: Declustering Techniques with Dimensionality

In the second set of the experiments, we fix the number of I/O devices to 8, and change the number of dimensions (4,8,12,16, and 20) to see the effect of dimensionality on the techniques. The number of buckets is computed as 2^d for the dimensionality of d. Figure 10 illustrates the result of this experiment. The *Concentric HyperSpace based declustering*, especially the *Hyperpyramid based declustering*, performs significantly better than the existing techniques.

6 Conclusion

In this paper, we propose several declustering techniques based on *Concentric Hyperspaces.* We first establish that traditional declustering techniques do not perform well for high dimensional data. The major problem with the existing techniques is the assumption of balanced partitioning or the index structures that do not perform well for high dimensional data. We define two important goals for fast parallel searching: minimizing the number of partitions that are intersected by the query, and developing an algorithm to distribute the buckets across multiple disks so that retrieval of any set of buckets intersecting a query is maximally parallelized. In contrast to the traditional techniques, we develop declustering techniques which achieve both goals. First we propose the idea of *Concentric Hyperspaces* for partitioning the data space. Then, we propose several disk allocation techniques for *Concentric Hypercubes and Hyperpyramids*. Our disk allocation technique based on *Concentric Hypercubes* optimizes the second goal, therefore is strictly optimal in terms of the second parameter, parallelization. We show that *Hypercube based declustering* outperforms the existing techniques. Finally, we show that *Hyperpyramid based declustering* performs much better than all the existing declustering techniques. In our experiments, we observe that *Hyperpyramid based declustering* achieves a very significant speed up, therefore leads to fast parallel range searching in multimedia databases.

References

[1] K. A. S. Abdel-Ghaffar and A. El Abbadi. Optimal allocation of two-dimensional data. In *International Conference on Database Theory*, pages 409–418, Delphi, Greece, January 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] S. Berchtold, C. Bohm, D. Keim, and H. Kriegel. A cost model for nearest neighbor search in high-dimensional data space. In *Proc. ACM Symp. on Principles of Database Systems*, Tuscon, Arizona, 1997.

[4] S. Berchtold, C. Bohm, and H.-P. Kriegel. The pyramid-technique: Towards breaking the curse of dimensionality. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 142–153, Seattle, Washington, USA, June 1998.

[5] S. Berchtold, D. Keim, and H. Kriegel. The x-tree: An index structure for high-dimensional data. In *Proceedings of the Int. Conf. on Very Large Data Bases*, pages 28–39, Bombay, India, 1996.

[6] L. T. Chen and D. Rotem. Declustering objects for visualization. In *Proceedings of the Int. Conf. on Very Large Data Bases*, pages 85–96, Dublin, Ireland, August 1993.

[7] B. Chor, C. E. Leiserson, R. L. Rivest, and J. B. Shearer. An application of number theory to the organization of raster-graphics memory. *Journal of the Association for Computing Machinery*, 33(1):86–104, January 1986.

[8] P. Ciaccia and A.Veronesi. Dynamic declustering methods for parallel grid files. In *Proceedings of Third International ACPC Conference with Special Emphasis on Parallel Databases and Parallel I/O*, pages 110–123, Berlin, Germany, September 1996.

[9] 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.

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

[11] S. Ghandeharizadeh and D. J. DeWitt. Hybrid-range partitioning strategy: A new declustering strategy for multiprocessor database machines. In *Proceedings of 16th International Conference on Very Large Data Bases*, pages 481–492, August 1990.

[12] S. Ghandeharizadeh and D. J. DeWitt. A performance analysis of alternative multi-attribute declustering strategies. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 29–38, San Diego, 1992.

[13] K. A. Hua and H. C. Young. A general multidimensional data allocation method for multicomputer database systems. In *Database and Expert System Applications*, pages 401–409, Toulouse, France, September 1997.

[14] 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.

[15] K. Lin, H. V. Jagadish, and C. Faloutsos. The TV-tree: An index structure for high-dimensional data. *VLDB Journal*, 3:517–542, 1995.

[16] B. Moon, A. Acharya, and J. Saltz. Study of scalable declustering algorithms for parallel grid files. In *Proc. of the Parallel Processing Symposium*, April 1996.

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

[18] S. Prabhakar, D. Agrawal, and A. El Abbadi. Efficient disk allocation for fast similarity searching. In *10th International Symposium on Parallel Algorithms and Architectures, SPAA'98*, Puerto Vallarta, Mexico, June 1998.

[19] J. C. Simon. *Patterns and Operators*. McGraw-Hill, 1986.