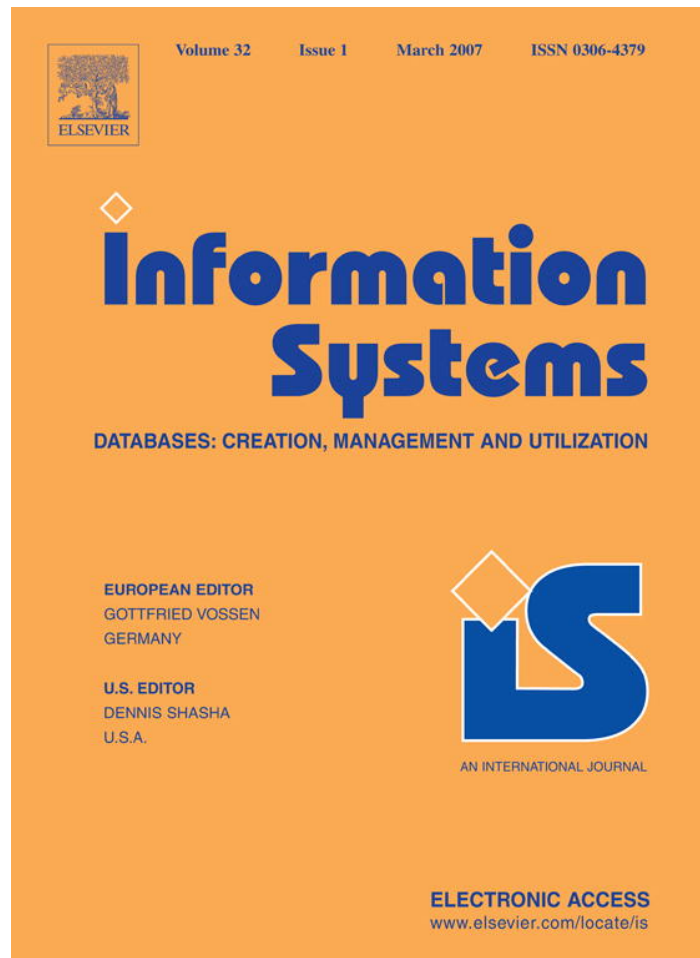


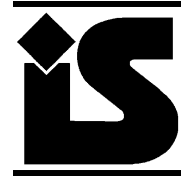
Provided for non-commercial research and educational use only.
Not for reproduction or distribution or commercial use.



This article was originally published in a journal published by Elsevier, and the attached copy is provided by Elsevier for the author's benefit and for the benefit of the author's institution, for non-commercial research and educational use including without limitation use in instruction at your institution, sending it to specific colleagues that you know, and providing a copy to your institution's administrator.

All other uses, reproduction and distribution, including without limitation commercial reprints, selling or licensing copies or access, or posting on open internet sites, your personal or institution's website or repository, are prohibited. For exceptions, permission may be sought for such use through Elsevier's permissions site at:

<http://www.elsevier.com/locate/permissionusematerial>



Data space mapping for efficient I/O in large multi-dimensional databases ^{☆, ☆ ☆}

Hakan Ferhatosmanoglu^{a,*}, Aravind Ramachandran^{b,1},
Divyakant Agrawal^c, Amr El Abbadi^c

^aComputer Science and Engineering, Ohio State University, USA

^bMicrosoft

^cComputer Science, University of California, Santa Barbara, USA

Received 10 March 2004; received in revised form 17 May 2005; accepted 7 June 2005

Abstract

In this paper, we propose data space mapping techniques for storage and retrieval in multi-dimensional databases on multi-disk architectures. We identify the important factors for an efficient multi-disk searching of multi-dimensional data and develop secondary storage organization and retrieval techniques that directly address these factors. We especially focus on high dimensional data, where none of the current approaches are effective. In contrast to the current declustering techniques, storage techniques in this paper consider both inter- and intra-disk organization of the data. The data space is first partitioned into buckets, then the buckets are declustered to multiple disks while they are clustered in each disk. The queries are executed through bucket identification techniques that locate the pages. One of the partitioning techniques we discuss is especially practical for high dimensional data, and our disk and page allocation techniques are optimal with respect to number of I/O accesses and seek times. We provide experimental results that support our claims on two real high dimensional datasets.

© 2005 Published by Elsevier B.V.

Keywords: Data space mapping; Space partitioning; Parallel I/O; Disk and page allocation; High dimensional data; Storage; Multi-disk architectures; Performance

[☆] Recommended by F. Carino Jr.

^{☆☆} This material was prepared with the support of the U.S. Department of Energy (DOE) Award No. DE-FG02-03ER25573 and National Science Foundation Award No. CNS-0403342. However, any opinions, findings, conclusions or recommendations expressed herein are those of the authors, and do not necessarily reflect the views of sponsors.

*Corresponding author. Tel.: +1 614 2926377.

E-mail addresses: hakan@cse.ohio-state.edu (H. Ferhatosmanoglu), araram@microsoft.com (A. Ramachandran), agrawal@cs.ucsb.edu (D. Agrawal), amr@cs.ucsb.edu (A. El Abbadi).

¹The work was completed when the author was with the Database Lab at the Department of Computer Science and Engineering, The Ohio State University.

1. Introduction

The advance of technology has seen increase in applications that integrate new kinds of information, such as multimedia and scientific data. The data in these systems are usually represented by multi-dimensional summarizations of the original data. Both the dimensionality and the amount of data that need to be processed have been increasing rapidly and it becomes necessary to support the efficient retrieval of large amounts of high dimensional data. *Multimedia databases* are a typical example of such applications. Multimedia data is inherently large and is usually represented by a feature vector which describes the original data usually with a high number of dimensions. Another example of large databases of high dimensionality is *scientific databases*. Both the dimensionality and the amount of data collected and generated by scientific and engineering simulations are increasing rapidly. Some examples include climate simulation and model development, computational biology, astrophysics, high energy and nuclear physics. Many large data sets in these domains consist of a large number of attributes that may be queried and analyzed, and therefore considered as high dimensional data [1]. Satellite data repositories will soon add one to two terabytes of data in a day [2]. If the current trends continue, large organizations will have petabytes of storage managed by thousand of processors [3]. Traditional retrieval methods based on index structures developed for single disk and single processor environments [4–9] are becoming ineffective for the storage and retrieval of high dimensional data in multiprocessor and multiple disk environments. A popular approach is to develop storage and retrieval techniques that exploit I/O device and processor parallelism.

1.1. Related work

In the past, storage redundancy has been successfully exploited in the context of data declustering on multiple disks. The basic idea of current declustering approach in database management systems can be summarized as follows. First, a data space is partitioned based on a given

criterion. Then the data partitions or buckets are allocated to multiple I/O devices such that neighboring partitions are allocated to different disks. Performance improvements for queries occur when the buckets involved in query processing are stored on different disks, and hence can be retrieved in parallel. Numerous methods have been proposed: disk modulo (DM) [10], fieldwise exclusive OR (FX) [11], Hilbert (HCAM) [12], near optimal declustering (NoD) [13], general multidimensional data allocation (GMDA) [14], cyclic allocation schemes [15,16], golden ratio sequences [17], Hierarchical [18], and discrepancy declustering [19] are some of the well-known disk allocation techniques. In [20–22], declustering techniques for multi-attribute databases are proposed for situations where there is some information about the query distribution. Latin squares [23] and Latin Cubes [24] have been discussed in detail for parallel access of arrays. Recently, declustering techniques have been proposed in [25] which are near-optimal for restricted cases. All of these techniques have been proposed for regular grid partitioning, where the data space is split into equi-sized partitions along each dimension. And most of them are originally proposed for two-dimensional data [26,27,15]. We have also proposed a technique for optimal declustering for two-dimensional data with a limited amount of replication [28]. There are several additional restrictions on each of them. For example, the technique in [25] requires that the number of disks is a power of a prime. 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. A performance evaluation of standard declustering schemes [20,29,30] and some theoretical bounds on the cost achieved by declustering schemes [31–33] have been discussed in the literature.

For non-uniform data, the algorithms proposed for regular grid partitioning can be extended using various greedy algorithms [34,35]. Parallel R-trees [36] have been proposed as technique for parallel processing of queries. X-Trees [37] have also been proposed for indexing high dimensional data. Graph partitioning based approaches [38–40] can also be used for non-uniform data declustering. In

this paper, we propose an orthogonal approach to the problem, where we implement a partitioning that would allow us to implement an optimal or near-optimal declustering.

1.2. Major contributions of the paper

This paper provides a scheme for storage of high dimensional data. What involves three steps: partitioning the data space, declustering the partitions among disks and then, clustering pages within each disk. We define the conditions for optimal declustering in a scenario where the data is uniformly distributed, and propose a scheme which comprises of the partitioning, declustering and page allocation that would satisfy the condition of optimality. Furthermore, we provide an adaptation of the technique for non-uniform data spaces. This is followed by a set of experiments, that illustrate that the theoretical optimality translates to efficient query processing in real applications.

To process a query, all buckets that intersect with the query region need to be accessed. The I/O cost of executing the query is typically assumed to be dominated by the maximum number of buckets accessed from a single I/O device. The optimal number of such page accesses in retrieving A buckets distributed over M devices is $\lceil A/M \rceil$. An allocation policy is said to be *strictly optimal* if no query region area A has more than $\lceil A/M \rceil$ buckets allocated to the same device. It has been proved that none of the techniques for regular grid partitioning can reach this optimal parallelism [31,32]. Furthermore, the current techniques do not scale for high-dimensional data and perform poorly as dimensionality increases. By employing a different partitioning strategy, the performance can be improved significantly. We develop a simple partitioning and declustering technique which achieves strictly optimal parallel I/O.

This paper also discusses another important aspect of declustering: in-disk organization of data within a single disk. Current trends imply that besides exploiting the parallelism, a careful organization of each disk must be considered for fast searching. The following quote summarizes the

importance and the necessity of disk arm optimization:

While disk capacities are improving very quickly, seek times are improving relatively slowly. Hence, the amount of data that can be transferred to main memory during an average seek time is rising very quickly. Put differently, the cost of a seek relative to the transfer of a byte of data is rising quickly.

The Asilomar Report on Database Research [3]

Goals for declustering. We identify three important factors for efficient high dimensional retrieval in multi-disk environments: (i) minimizing the expected number of partitions that are intersected by the query, i.e. pages accessed during the query, (ii) maximizing the degree of parallelism, by distributing the buckets across multiple disks so that retrieval of any set of buckets intersecting a query is maximally parallelized, and (iii) minimizing the disk arm movement by clustering the data within each disk. The failure to achieve any of these goals would lead to poor performance. Most of the current approaches to the problem focus *only* on the *second goal*, i.e., the uniform distribution of buckets among the disks.

Organization of the paper. In this paper, we propose data space mapping techniques that consider the parameters mentioned earlier. In Section 2 we define the problem and define terms that we will use throughout the paper. Section 3 develops data space partitioning techniques for parallel and multi-disk architectures. Section 4 proposes disk and page allocation, and bucket identification techniques for the partitioning techniques. Section 5 contains experimental results that compare the proposed declustering techniques to those currently used. Section 6 concludes the paper.

2. Problem statement

In general, to support complete data space mapping to physical storage requires the following

basic functionalities.

- **Data space partitioning:** given a multi-dimensional data space, partition it into disjoint buckets.
- **Disk allocation:** given a partitioned data space in terms of buckets and a limited number of disks, decluster the buckets on different disks.
- **Page allocation:** given a set of buckets destined for a given disk, cluster these buckets within the disk that are expected to be retrieved together.
- **Bucket identification:** given a query, identify the buckets that contain data relevant for the query. Similarly, given a point, find the bucket that contains the point.

The first three functionality correspond to the three identified goals for an effective data space mapping: (i) minimizing the expected number of partitions that are intersected by the query, i.e. pages accessed during the query, (ii) maximizing the degree of parallelism, and (iii) minimizing the disk arm movement.

Current data declustering approaches mostly assume a regular grid as the underlying partitioning. While disk allocation functions have been studied extensively, the issue of intra-disk page allocation has not been addressed. In order to map the data space into physical storage, we introduce the notion of *page allocation* which additionally provides the location of a bucket within a disk. Therefore, the allocation strategy not only maps to disks but also pages inside the disks as well. These functions must be developed so that the retrieval of the buckets is optimized.

To have a complete set of mapping of data space and to process queries, bucket identification techniques for points and queries must also be developed for the underlying partitioning strategy. The process of bucket identification is similar to the general notion of indexing. A range query $Q = ([a_1, b_1], [a_2, b_2], [a_3, b_3], \dots, [a_d, b_d])$ specifies a range of values, $[a_j, b_j]$, for each dimension, $1 \leq j \leq d$. The result of the query is the set of all data objects that have values within the specified range in each dimension. The *retrieval set* is the set of all *BucketIds* and it represents the buckets that must be retrieved by the query.

Since we are extending declustering to integrate both inter and intra disk organizations, new optimality criteria for evaluating disk and page allocation schemes need to be established. Typically, the parallelism degree is considered to be proportional to the maximum number of buckets accessed from a single I/O device. Hence, the following is the optimality definition for inter-disk allocation [10].

Definition 1 (Optimal disk allocation). A disk allocation policy is said to be *optimal* if no hyperrectangular area A has more than $\lceil A/M \rceil$ buckets allocated to the same device, where M is the total number of disks.

Similarly, for intra-disk retrieval, the minimum cost is incurred when all the pages that need to be retrieved from a disk are clustered together such that a single seek is incurred per query.

Definition 2 (Optimal page allocation). A page allocation policy is said to be *optimal* if the k pages that need to be retrieved from any disk as a result of a query are stored sequentially on that disk.

Note that the above definition implies that under optimal page allocation, each disk incurs only one seek per query. Thus, if t_{seek} , t_{latency} , and t_{transfer} are the seek time, rotational latency, and data transfer time, respectively, then an optimal allocation incurs the following delay per query:

$$t_{\text{seek}} + t_{\text{latency}} + \left\lceil \frac{A}{M} \right\rceil \cdot t_{\text{transfer}}.$$

3. Data space partitioning

In this section, we briefly discuss the problems of regular grid partitioning approach and then develop partitioning techniques that are later used for data space mapping. 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.

3.1. High dimensional grid partitioning

In most declustering techniques, the underlying partitioning is assumed to be a regular grid. Grid

based partitioning is well-known to perform poor in high dimensions, and the number of splits in each dimension is typically limited. For example, Near-optimal declustering [13] assumes binary partitioning of the space.

A range query specifies a range of values for each dimension. We define the selectivity of range query q , denoted \mathcal{S}_q , as $\mathcal{S}_q = r/D$, where r is the number of elements in the result set of q and D is the total size of the database. Under uniform distribution \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 $V_q = \mathcal{S}_q$.

The expected length a_q of each side of range query q , for a given selectivity \mathcal{S}_q can be simply estimated as

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

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. For high dimensions, range queries with realistic selectivities will typically have large side lengths. Note that for non-uniform data sets, the edge width of a hypercubic query of fixed selectivity can be much less than the estimated value. This case will be discussed later in the paper. Fig. 1 shows the expected length of hypercube range queries for with selectivities of 10^{-4} , and 10^{-5} for different dimensions of uniform data. As a result, the number of buckets that must be

retrieved is very high and even the best declustering method for regular grid partitioning does not have good performance for high dimensions. This is reflected in the experiments in Section 5. The performance degradation can be avoided by developing declustering techniques integrated with effective partitioning techniques for high dimensional data.

3.2. Partitioning based on concentric hyperspaces

Concentric partitioning starts from the center of the space and creates hypershells of specified volume going inside out. Since the volume near the surface of the hyperspace dominates the inside volume as dimensionality increases, most of the data is very close to the surface of the hyperspace [13]. 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%. Therefore the *concentric hyperspaces* lead to partitions such that the inside partitions are thicker and the outer partitions (near the surface) become thinner and thinner as shown in Fig. 2. We now explore different geometries for these concentric partitions.

3.2.1. Concentric hyperspheres

The first geometry we explore for creating concentric partitions is that of hyperspheres as shown in Fig. 2(a). Note that we will continue to assume throughout the paper that the data space is a unit hypercube of unit volume. The center of this data space as well as all the hyperspheres will be $[x_1, x_2, \dots, x_d]$, such that each

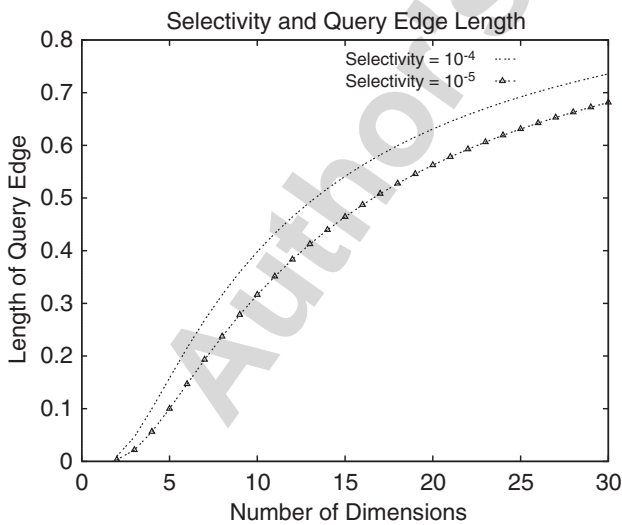


Fig. 1. Query edge and selectivity.

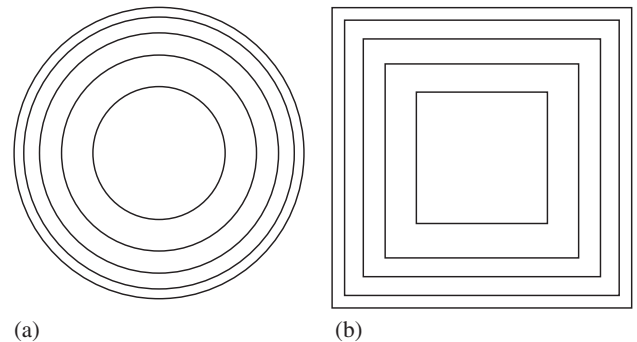


Fig. 2. Concentric regions.

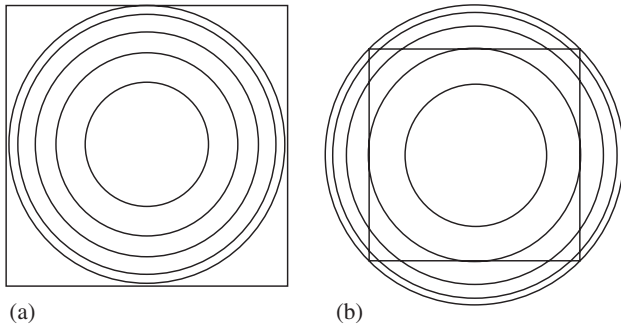


Fig. 3. Concentric hypersphere partitioning.

$x_i = 0.5$. The innermost partition is a solid hypersphere whereas the remaining partitions are hyperspherical shells between two hyperspheres as shown in Fig. 2(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 two possible ways of partitioning is illustrated in Fig. 3.

In Fig. 3(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. In Fig. 3(b), the data space is inscribed inside a hypersphere with a radius of 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 hyperspheres with radius greater than 0.5. On the other hand, in the first approach (Fig. 3(a)), the remainder data space, that is left outside the last hypersphere, results 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, 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:²

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

As d increases, 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 approach 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 increases. 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 (Fig. 3(b)). For a cube inscribed in a sphere the ratio of volume of the sphere to 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. Due to the mismatch between the two geometries, that of the data space and the partitions, concentric hyperspheres are not directly applicable.

3.2.2. 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 Fig. 2(b). One of the first problems we are confronted with, when dividing the data space in this way, is to determine the

²where the gamma function is defined as $\Gamma(n + 1) = n!$ and the volume of a hypersphere with dimension d and radius r is $r^d \cdot \pi^{d/2}/\Gamma(d/2 + 1)$ [41].

size of each of these shells. In regular grid partitioning, where each dimension i is divided into N_i parts, the total number of partitions is $N_1 \times N_2 \times \dots \times N_d$.

We partition the data space into p equally filled partitions with the Concentric Hypercube technique. First, p concentric hypercubes, $\square_1, \square_2, \dots, \square_p$ are created inside the unit hypercube. These hypercubes have a common center $(0.5, 0.5, \dots, 0.5)$. The first hypercube, \square_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 \square_1 and \square_2 . Similarly, $part_i$ is created by \square_{i-1} and \square_i . The hypercube that covers all the hypercubes, \square_p is the unit hypercube itself. The volume of each partition is $1/p$, and the volume of \square_1 which is also $part_1$ is

$$Volume(\square_1) = Volume(part_1) = 1/p.$$

In general, the volume of hypercube \square_i , for $i > 1$, is:

$$Volume(\square_i) = Volume(part_i) + Volume(\square_{i-1}).$$

By solving the above recurrence relation $Volume(\square_i)$ is i/p . From this we can compute the length of the edge, a_i of hypercube \square_i as $\sqrt[d]{i/p}$. Since $a_i/2$ is the distance of the surface of the hypercube \square_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 et al. [42] have proposed another geometry based on pyramids as being useful for reducing high dimensions into one-dimension and indexing the one-dimensional data using B-trees. Hyperpyramid partitioning can be considered as a special case of concentric partitioning. The partitions in the hyperpyramid partitioning are the hypertrapezoids created by dividing each hypercubic shells into $2d$ hypertrapezoids. The data space is divided into $2d$ major hyperpyramids and each hyperpyramid is divided into concentric

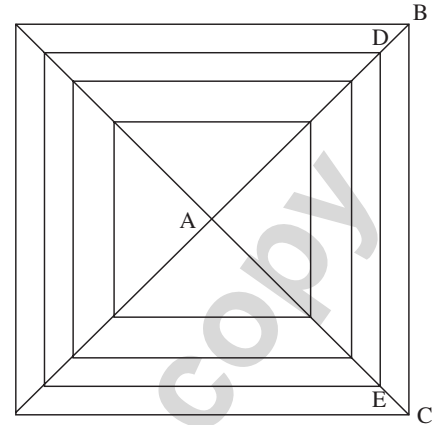


Fig. 4. Pyramids and trapezoids.

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 used as the actual partitions. In Fig. 4, for two-dimensional case, $\triangle ABC$ is a major hyperpyramid, $\triangle ADE$ is a hyperpyramid, and $\square DBCE$ is a hypertrapezoid.

There are totally $2d$ major hyperpyramids each with a volume of $1/2d$ and with the apex (x_1, x_2, \dots, x_n) , where $x_i = 0.5$. First, $p/2d$ hyperpyramids, $\Delta_1, \Delta_2, \dots, \Delta_{p/2d}$ with a common apex are created inside each major hyperpyramid. The distance between the base of the hyperpyramid Δ_i and the apex is defined as the height, h_i . Hyperpyramid Δ_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, Δ_1 , and the smallest hyperpyramid that covers the top one, Δ_2 . Similarly, the i th level is the one created by the hyperpyramids, Δ_{i-1} and Δ_i . All $2d$ partitions at the same level 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. In the base case, the volume of the top level pyramid is:

$$Volume(\Delta_1) = Volume(part_1) = 1/p.$$

The recurrence relation is as follows:

$$Volume(\Delta_i) = Volume(\Delta_{i-1}) + Volume(part_i) = i/p.$$

We will find the height values in hyperpyramid partitioning using concentric hypercube formulas. Let \square_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,

$$Volume(\square_i) = Volume(\Delta_i) \times 2d = (i/p) \times 2d.$$

We also know that, $Volume(\square_i)$ 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 Δ_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.

Lemma 1. *Consecutive hypertrapezoid based retrieval results in disk declustering that has a cost that is at most $2d$ more than the optimal declustering.*

Proof. In the case of hypertrapezoid based clustering, for an arbitrary query Q , let B be the total number of buckets intersected. Let the number of hyperpyramids intersected be k and the number of buckets intersected in each hyperpyramid i be b_i . From Lemma 3, we know that the declustering in each pyramid is optimal (i.e.) the buckets within each pyramid can be obtained with optimal parallelism if they are retrieved in the sequentially ranked order. However, this ranking property is not valid across various hyperpyramids. Let b_1, b_2, \dots, b_k be the number of buckets intersected by the query in each of the intersected hyperpyramids. \square

We have the result that the I/O cost is at most $\lceil b_1/M \rceil + \lceil b_2/M \rceil + \dots + \lceil b_k/M \rceil$. Now, since $\lceil b_i/M \rceil \leq \lfloor b_i/M \rfloor + 1$, we have the inequality, I/O cost \leq Optimal I/O cost + k , where k is the number of intersected hyperpyramids. Since $k \leq 2d$, we get the required result.

3.4. Data space partitioning for non-uniform databases

The hyperspace based partitions are based about the center of the data space. This is built upon the premise that the distribution is uniform about the center. This assumption is not valid in real databases, where the distribution is often skewed. For instance, in multimedia databases, the data is a vector of amplitude values. Typically, the mean of the amplitude values is much less than half the maximum amplitude. Thus, once the data is scaled to the canonical space of $[0, 1]^d$, we observe a skewed distribution of the data. Another property of real world data is that it is normally clustered. For instance, most of the data points are often concentrated in one corner of the data space. This phenomenon becomes well pronounced with increase in dimensionality—resulting in very large page sizes due to the non-uniform distribution. Thus, we need to extend the available techniques for partitioning uniform data to make them applicable to real world data sets; namely the multimedia and scientific databases.

In the case of uniform data, the property that partitions of equal volume contain equal amounts of data helps us find bounds on the performance of these techniques. However, in non-uniform data spaces, a partitioning technique that does not account for the distribution of the data is often sub-optimal. The non-uniform distribution of the data would result in many partitions either sparsely populated or completely empty. Furthermore, due to clustering, we would require large page sizes as well. Since we know that the available partitioning techniques are guaranteed to be optimal or near-optimal for uniform data, the logical step to extend these techniques would be to transform the data to something closer to a uniform distribution and then apply them to the transformed data.

We now identify the formal requirements for such a transformation and identify a transformation function that will achieve those requirements.

3.4.1. Data space transformation functions

Berchtold et al. [42] proposed a transformation function towards this purpose, which involves

mapping from the given data space to the canonical data space $[0, 1]^d$ such that the d -dimensional median of the data roughly coincides with the center point.

To obtain such a mapping, it is necessary that the transformed data space lies in $[0, 1]^d$ and that the median of the transformed data space is roughly the geometric center. It is also necessary that the transformation is defined for every point in the original data space. These conditions can be summarized as follows. For a d -dimensional data space, given the median $(mp_0, mp_1, \dots, mp_d - 1)$, we require a set of transformation functions $t_i, \forall i: 0 \leq i \leq (d - 1)$ such that the following conditions hold.

- $t_i(0) = 0$,
- $t_i(1) = 1$,
- $t_i(mp_i) = 0.5$,
- $t_i: [0, 1] \rightarrow [0, 1]$.

The transformation function $t_i(x) = x^{e_i}$ where $e_i = -1/\log_2(mp_i)$ satisfies the above stated constraints and can be applied for the transformations. However, we note that this transformation does not result in a transformation that is uniformly distributed across the hyperspace.

In addition to these conditions, we propose an additional constraint that ensures that the resultant data space is more uniformly populated. This can be expressed by the condition

$$\sigma(t_i(x)) = 1,$$

where σ denotes the variance of the distribution. This can be accomplished by the transformation function,

$$t_i(x) = (((c_i + x)^{e_i} - c_i^{e_i}) / ((1 + c_i)^{e_i} - c_i^{e_i})),$$

where e_i is given by $e_i = \sigma_i / (\sigma_i - 1)$. σ_i is in turn, the variance of the actual distribution in the i th dimension. The value of parameter c_i is the given by the approximate solution to the equation, $t_i(mp_i) = 0.5$.

Using this technique, we can transform any data space to a data space that is close to uniform. As we have shown the partitioning techniques to be efficient in the uniform case, they preserve some of their properties in a scenario where the distribu-

tion is near uniform. Later, in our experimental results, we observe that on applying these transformations, hyperspace partitioning techniques are substantially better than the regular partitioning techniques on non-uniform data as well.

4. Data space mapping with clustering and declustering

In this section, we discuss the mapping of the data space, for storage and retrieval, based on various partitioning techniques discussed in this paper. In Section 4.1 we discuss these issues under regular grid partitioning and state the problems. We then develop a complete set of mapping based on concentric partitioning: hypercubes in Section 4.2, and hyperpyramids in Section 4.3.

4.1. Regular grid partitioning

We first discuss the page allocation and bucket identification issues for regular grid partitioning. Note that, we consider a d -dimensional space where the data points are normalized in $[0..1]$. Assuming regular grid partitioning, that the i th dimension is partitioned into equal sized N_i regions and the bucket corresponding to a point (x_1, x_2, \dots, x_d) can be determined as follows:

$$BucketId = \left(\left\lfloor \frac{x_1}{N_1} \right\rfloor, \left\lfloor \frac{x_2}{N_2} \right\rfloor, \dots, \left\lfloor \frac{x_d}{N_d} \right\rfloor \right)$$

To be useful in range query processing, efficient techniques must be developed to determine the buckets that need to be retrieved by a query. In two dimensions with N partitions in each dimension, a query region bounded by points (x_0, y_0) and (x_1, y_1) is the enumeration of all buckets in the region $(x_0/N, y_0/N)$ and $(x_1/N, y_1/N)$.

After identifying the buckets, disk and page allocation techniques are needed both for storage and retrieval. The overall goal is to distribute the neighboring buckets across different disks while clustering within a disk for reducing seek latency on the same disk. Fig. 5 illustrates a two-dimensional data space that is partitioned into five regions in each dimension. The labels in the figure specify the disks as well as the page

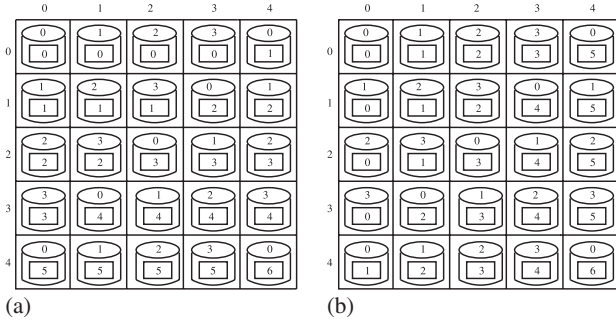


Fig. 5. Disk and page allocation for regular grid partitioning.

assignments of each buckets on that disk, assuming that there are four disks in the system. The disk assignment in both figures is based on the DM declustering [10]. Several other single disk allocation mechanisms have been proposed to linearize a multidimensional space on a single disk [43]. In Fig. 5(a), we assign the pages in the row major order for each disk. For example, bucket (3,0) is assigned to page 3 of disk 3 whereas bucket (0,3) is assigned to page 0 of disk 3. In Fig. 5(b), the page allocation is done in a column-major round robin way. Different other techniques can be examined for page allocation for regular grid partitioning. Here, we introduce one of the possible formal page allocation functions, which uses row-major order, to the declustering mechanism based on DM and regular grid partitioning. Given a bucket, $BucketId = (i, j)$, the physical location of the bucket, i.e. the corresponding disk and the page within that disk, can be computed as follows:

$$disk(BucketId) = (i + j) \bmod M,$$

$$page(BucketId) = \left\lfloor \frac{iN + j}{M} \right\rfloor,$$

where N is the number of partitions in each dimension and M is the number of disks. These functions are used both in the storage and the query processing steps. The bucket is stored to the appropriate location according to its $BucketId$ and the above formula. Queries retrieve buckets by first identifying their $BucketIds$ and then finding the locations.

More effective but complex disk allocation techniques have been developed [14,15]. The page

assignment functions under Cyclic Allocation Techniques, for example, is more complicated. In the cyclic allocation technique under regular grid 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 $j \bmod M$. Each partition in row 1, (1, j) is assigned to device $(H + j) \bmod M$, 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 $(Hi + j) \bmod M$. Next, we define general page allocation formula, besides disk allocation, of $BucketId = (i, j)$ for two-dimensional Cyclic Allocation Techniques by the following functions.

$$disk(BucketId) = d = (Hi + j) \bmod M$$

$$page(BucketId)$$

$$= \left\lfloor \frac{j + 1 - ((d - i.H) \bmod M)}{M} \right\rfloor + \sum_{k=0}^{i-1} \left\lfloor \frac{N - ((d - k.H) \bmod M)}{M} \right\rfloor - 1.$$

The disk allocation function is the general function for Cyclic Allocation Techniques. Here, we introduce the page allocation function which is needed to locate the data within a disk. The summation in the page allocation formula computes the number of buckets assigned to disk d for each row starting from the 0th row to $(i - 1)$ st row. The first expression computes the same value for the i th row but not considering the buckets come after the bucket (i, j). Then we subtract 1, because the page numbering starts with a zero as initial page number. As an example, let's compute the disk and page number for bucket (3, 2) under DM, which is a Cyclic Scheme with a skip value H of 1 (see Fig. 5(a)). $disk(3, 2) = (1 \cdot 3 + 2) \bmod 4 = 1$. The loop in the page allocation function counts the number of assignments to disk 1 in the first three rows, row 0, row 1, and row 2, which is found to be 1, 2, and 1 respectively. The first part computes the number of pages on row 3, which comes before (3, 2), including itself, and is assigned to disk 1. (3, 2) is the only such bucket, so it gives 1. The following is the formulation of this

explanation by using the function.

$$\begin{aligned}
 & \text{page}(3, 2) \\
 &= \left\lceil \frac{4 - ((1 - 3) \bmod 4)}{4} \right\rceil + \left(\left\lceil \frac{5 - ((1 - 0) \bmod 4)}{4} \right\rceil \right. \\
 &\quad + \left\lceil \frac{5 - ((1 - 1) \bmod 4)}{4} \right\rceil \\
 &\quad \left. + \left\lceil \frac{5 - ((1 - 2) \bmod 4)}{4} \right\rceil \right) - 1 \\
 &= 1 + (1 + 2 + 1) - 1 = 4.
 \end{aligned}$$

Therefore, the disk number of (3, 2) is found to be 1 and the page number inside the disk 1 is found to be 4. The high-dimensional page allocation techniques will be more complicated and finding a general formula is non-trivial.

4.2. Concentric partitioning with optimal clustering and declustering

Except under very restricted conditions it is not possible to find an optimal allocation for regular grid partitioning [31]. Similarly, achieving optimality for page addressing necessary for in-disk organization is also hard under regular grid partitioning. Furthermore, this way of partitioning creates an exponential number of partitions which causes a very large number of buckets in high dimensions. Besides the large number of partitions, the number of neighboring buckets increases rapidly as the dimensionality increases, which makes the allocation more difficult. If the k th-level neighbor of a partition is defined as the partitions that differ in k dimensions, then an algorithm considering up to k levels of neighboring must assure that $1 + \sum_{i=1}^k \binom{d}{i}$ buckets are equally distributed over the disks [13].

We propose using concentric hyperspaces, as an alternative to regular partitioning in disk allocation for range queries. Concentric partitioning approach is appropriate to be used in a complete mapping of multi-dimensional data space to physical storage. It does not suffer from the problems that exist for regular partitioning, i.e., impossibility of optimal disk and page allocation, exponential number of partitions, large number of buckets retrieved as a result of a query, large

number of neighboring buckets, and large number of buckets in the same level of neighborhood.

The bucket identification of a point (x_1, x_2, \dots, x_d) in a concentric hypercube partitioning is carried out as follows. First a single value y such that $y = \max(|x_1 - 0.5|, |x_2 - 0.5|, \dots, |x_d - 0.5|)$ is computed. This value illustrates the relative position of the point from the center of the data space. From the table T , the first entry i , that has the value $T[i]$, greater than or equal to y gives the corresponding $BucketId = i$ for the point. Since the partitions are concentric, this value will be the hypercubic shell that contains the points that have this point and points with similar positions relative to the center.

The bucket identification of a range query, $Q = ([a_1, b_1], [a_2, b_2], [a_3, b_3], \dots, [a_d, b_d])$ is carried out as follows. We must find the $BucketIds$ in the retrieval set, which is defined to be the set of buckets to be retrieved. By using the properties of concentric hypercubes, we can compute the maximum and minimum $BucketIds$ in the retrieval set, which means that everything in between these values needs to be retrieved. This is in contrast to regular grid partitioning, in which bucket identification requires enumeration through each bucket in the intersection.

The maximum $BucketId$, \max_b , in the retrieval set is the id of the innermost hypercube that includes both of the extreme points of the query hyperrectangle. Formally, this can be computed as

$$\begin{aligned}
 \max_b &= \max(BucketId(a_1, a_2, a_3, \dots, a_d), \\
 &\quad BucketId(b_1, b_2, b_3, \dots, b_d)).
 \end{aligned}$$

The minimum $BucketId$, \min_b , is the id of the smallest hypercube that intersects the query hyperrectangle. Note that, two hyperrectangles intersect if they intersect in each dimension. Since we eliminated all the hypercubes greater than \max_b , \min_b can be computed performing binary search over the entries in the table $T[1..\max_b]$. The \min_b^{th} hypercube intersects the query, but the $(\min_b - 1)^{\text{st}}$ does not. Since, we have computed the minimum and the maximum $BucketIds$, the retrieval set is the set of $BucketIds$ within the range \min_b and \max_b .

Lemma 3. *Concentric hypercube based retrieval uses an optimal disk declustering technique for range queries.*

Proof. Assume $R = i, i + 1, \dots, i + k - 1, i + k$. These buckets are allocated to disks $i \bmod M, (i + 1) \bmod M, \dots, (i + k - 1) \bmod M, (i + k) \bmod M$. The maximum number of repetition of a single number in this is set, i.e., the maximum number of buckets that is assigned to a single disk $\lceil k + 1/M \rceil$, which is the optimal cost. (Note that $A = k + 1$). \square

We now examine the in-disk organizations and the disk arm optimization in concentric hypercube based searching.

Lemma 4. *Concentric hypercube based retrieval uses an optimal page allocation technique for range queries.*

Proof. Let us consider a query that retrieves the pages i and j , $i < j$, from a single disk, D . There are two corresponding global shell partitions intersected by the query, S_i and S_j , s.t. $disk(S_i) = disk(S_j) = D$ and $page(S_i) = i < j = page(S_j)$. Assume that there is a page, k , in D , such that $i < k < j$ and k is not in the query result. The corresponding global shell partition of page k is S_k with $disk(S_k) = D$ and $i < page(S_k) = k < j$. Since the round robin method is used for allocation, the partition S_k must be between S_i and S_j . From Lemma 2, if the query intersects S_i and S_j it must also intersect S_k . This implies that the page k is in the query result which contradicts the initial assumption. Therefore, there cannot be any gaps between any two pages retrieved from a disk as a result of a query. Hence, all buckets that are retrieved from each disk are sequentially stored in that disk. \square

4.3. Hyperpyramid based data space mapping

Concentric hypercubes satisfy optimality for the multi-disk parallelism and intra-disk organizations. However, another very important factor for fast parallel searching is the number of buckets retrieved by the queries. Reaching optimality in the first two factors may not lead to fast searching if this third factor is not minimized. It has been

shown that hyperpyramid based partitioning reduces the number of partitions retrieved by queries [44,42].

The bucket identification of a point (x_1, x_2, \dots, x_d) in pyramid partitioning is carried out as follows. First the major hyperpyramid P_i in which that point lies is computed by using the following function.

$$i = \begin{cases} j_{\max} & \text{if } (x_{j_{\max}} < 0.5), \\ (j_{\max} + d) & \text{if } (x_{j_{\max}} \geq 0.5), \end{cases}$$

where $j_{\max} = (j | (\forall k, 0 \leq (j, k) < d, j \neq k : |0.5 - x_j| \geq |0.5 - x_k|))$.

Then, the height h of the point from the center is computed by $h = |0.5 - x_{i \bmod d}|$. From the table H , the first entry j , that has the value $T[j]$, greater than or equal to h gives the corresponding $BucketId = j$ for the point.

The bucket identification of a range query, $Q = ([a_1, b_1], [a_2, b_2], [a_3, b_3], \dots, [a_d, b_d])$ is carried out as follows. First of all, the major hyperpyramids that intersect with the range query are identified. Then within each major hyperpyramid, the sequence of partitions that are intersected by the query is computed. These functions are used in [42] to create a single one-dimensional index value for each multi-dimensional point. This single value is used to create a B-tree like index structure. We use the same functions for bucket and query identification in hyperpyramid based mapping. The details of these functions can be found in [42].

To develop an efficient disk and page allocation technique, we transform the high-dimensional hyperpyramid partitioning into a simple two-dimensional table. In the hyperpyramid partitioning, there are $2d$ major hyperpyramids in the data set. Each major hyperpyramid has $p/2n$ hyperpyramid shells, hypertrapezoids and a hyperpyramid, which are the actual partitions. By using this transformation, the major hyperpyramids correspond to the rows of the two-dimensional table. The 1st row represents P_1 , the 2nd row represents P_2 , and so on. The columns represent the hyperpyramid shell levels. The 1st column in each row represents the top hyperpyramid, $part_1$, in each major hyperpyramid. The 2nd column represents the hypertrapezoid $part_2$, and so on.

By using this transformation, the general two-dimensional disk and page formulas for Cyclic Allocation Techniques can also be used for hyperpyramid partitioning. This technique can be intuitively summarized as round robin allocation in each major hyperpyramid. Each major hyperpyramid is declustered to different disks and assigned to pages in a round robin fashion, starting from the partition closest to the center going through to the ones near the edge. The initial mapping value is determined by the skip values defined in Cyclic Allocation Techniques.

Since concentric hypercube based retrieval guarantees sequential access, only 1 seek was needed for each data transfer from each disk. By using the page allocation technique for hyperpyramid partitioning, pages in each major hyperpyramid are read sequentially. An initial seek is enough to retrieve each major hyperpyramid. Since there are $2 \cdot d$ major hyperpyramids, in the worst case scenario of a query one seek will be needed for each of them (Worst case is the case that the query intersects all the major hyperpyramids). Therefore, in the worst case $2 \cdot d$ seeks is enough to retrieve any query. Hyperpyramid based partitioning guarantees at most $2 \cdot d$. However, there is no such guarantee in the current declustering techniques. Without using a specific page allocation technique, the retrieval of the buckets may include arbitrary number of seeks.

5. Performance evaluation

We first evaluate the quality of the proposed partitioning and disk allocation techniques. We then evaluate the overall performance of the proposed data space mapping techniques by considering also the page allocation. Our experiments were carried out on projections of the LANDSAT and the STOCK datasets. The LANDSAT data set contains satellite image texture vectors extracted from satellite images. This data set is highly in use for high-dimensional indexing and similarity search research. Each data point in the data set consists of 60-dimensional floating point numbers. The STOCK data set contains time series information about the stock

prices of companies. While this data set is smaller, it is very high in dimensionality. We perform experiments to compare the performance of various approaches for higher dimensions.

5.1. Evaluation of partitioning and disk allocation techniques

We now evaluate the performance of newly proposed high-dimensional declustering techniques based on different partitioning strategies. The comparisons are made with the current techniques, DM, FX, RPHM, GDMA, and Cyclic Schemes. We also included *Best Cyclic Scheme* [15,16], which is proposed by exhaustively computing the best parameters for cyclic allocation. First, the data space is partitioned by using the corresponding technique for each declustering method. The formulas given in Section 4.2 is used to partition the data space. The competing methods DM, FX, RPHM, GDMA and all Cyclic Schemes use balanced partitioning, whereas our techniques are based on concentric hyperspaces. The number of partitions, p , and the number of data points in each partition is same in both cases.

Evaluation is performed using hypercubic and hyperrectangular range queries. The query processing is performed using the formulas given in Section 4.2. The queries are generated with realistic selectivities, such as 10^{-4} and 10^{-5} . We ran 500 random queries for each experiment and the average cost of these queries is computed. The experiments were done for several different dimensionalities. As dimensionality increases, the performance improvement gained by our techniques becomes more significant. We evaluate the techniques in two different architectures. One with the fastest disks available and one with average speed disk. Both of the specifications have been taken from [45]. Table 2 illustrates the disk architectures. We varied the number of disks available in the system from 5 to 50.

In our first set of experiments, we fix the number of dimensions and vary the number of I/O devices for hypercubic range queries. Fig. 7 illustrates the performance comparison of several declustering techniques with newly proposed concentric hyperspace based disk allocation techniques, hypercubes

and hyperpyramids. These figures illustrate the performance of *Best Cyclic*, *DM*, *RPHM*, *FX*, *GDMA*, *Hyperpyramid based declustering*, and *Hypercube based declustering* with hypercubic range queries. Fig. 7 demonstrates the disk access times based on these techniques on an average speed disk architecture as well as a fast disk architecture. We observe that the advantage in performance for hypercubic and hyperpyramid partitioning is more pronounced in faster architectures.

The dimensionality of the data is 32, and each dimension is divided into two parts, creating 65,536 buckets in the system. It can be seen that *FX*, *DM* and *RPHM* do not perform well for high dimensional data. We observe from Fig. 7 that *Hyperpyramid based declustering* provides significant performance improvement over all the existing techniques. As dimensionality increases this improvement becomes more apparent. We note

Table 2
Disk specification

Time	Fast disk	Average disk
Average seek time(ms)	3.6	8.5
Latency(ms)	2.00	4.16
Transfer(MByte/s)	86	57

that in each graph, *Hypercube based declustering* performs better than the existing techniques but worse than the *Hyperpyramid based declustering*.

We now compare the performance of hypercube based partitioning and the best cyclic allocation techniques for hyperrectangular queries i.e., where the queries do not have equal width in each dimension. Fig. 8 shows the difference in I/O time between the two declustering techniques for a dimensionality of 16 and a selectivity of 10^{-5} for hypercubic queries and hyper-rectangular queries.

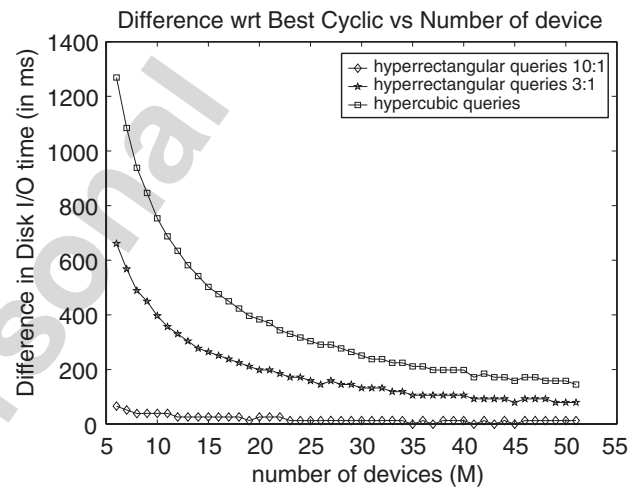


Fig. 8. Hyperrectangular and hypercubic queries.

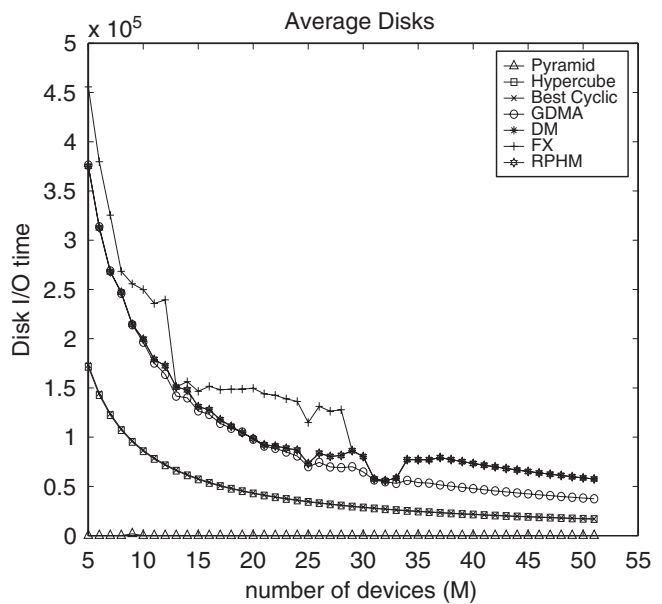
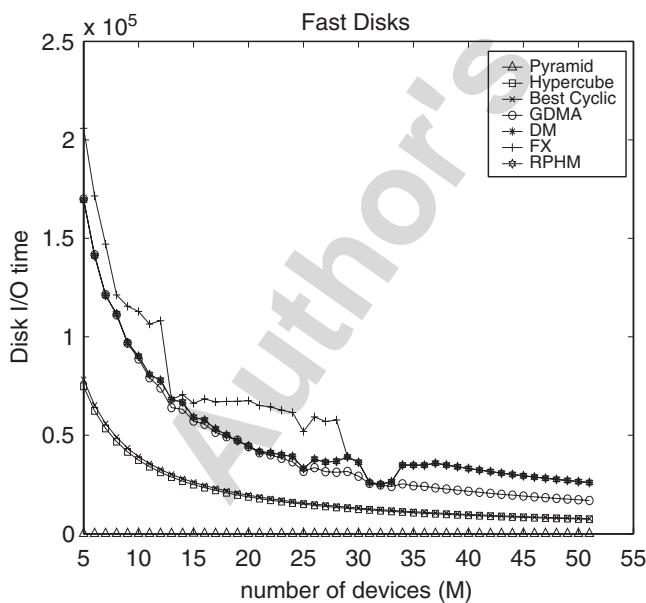


Fig. 7. Hypercubic range queries.

We provide results for hyperrectangular queries where the ratio of the maximum width in a dimension to the minimum width is 3:1 and 10:1. Again, the I/O times are those corresponding to the fast disk specification in Table 2.

We observe that *Hypercube based declustering* provides significantly better performance with hypercubic range queries rather than hyperrectangular range queries. (Fig. 8). This difference in performance is more evident when the query is more uniform across the dimensions. This difference can be explained by the probability of having query edge near to the surface i 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 4.2.

In the third set of the experiments, which are on the Stock Dataset, we fix the number of I/O

devices to 20, and change the number of dimensions to see the effect of dimensionality on the techniques. The number of buckets is computed as 2^d for the dimensionality of d . Fig. 9 illustrates the result of this experiment. The *concentric hyperspace based declustering*, especially the *Hyperpyramid based declustering*, performs significantly better than *best cyclic declustering*, which in turn is better than the other existing cyclic allocation techniques. Again, the difference is more evident in the faster architecture.

5.2. Data space mapping with page allocation

The page allocation and bucket identification techniques are implemented over the three partitioning strategies. For regular grid partitioning, we created the best cyclic scheme by evaluating all the possible values for cyclic schemes and choose the best performed values in each dimension. The techniques are evaluated for partitioning based on 16- and 32-dimensional projections on the Landsat data set. In both cases, the data set is partitioned into 65,536 buckets. The bucket size in each setup is 32 KBytes.

First, the buckets that are needed to be retrieved by the query are identified. Then by using the disk and page allocation functions, the location of the

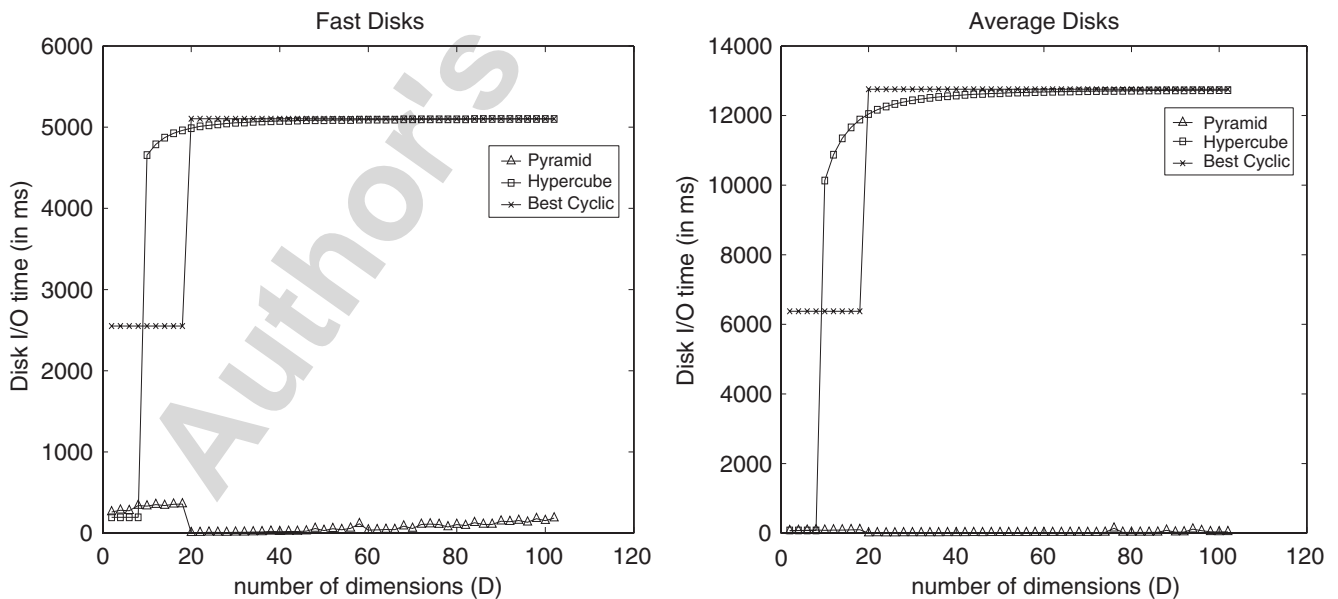


Fig. 9. Declustering with dimension.

buckets are determined and retrieved by processing appropriate seeks and transfers. Since we are using a multi-disk simulation the overall time of a query is identified by the maximum retrieval time spent from a single disk. Our performance metrics is average time spent for 1000 random range queries. The queries have realistic selectivities that vary between 10^{-4} and 10^{-5} for each data set. Results are similar for both data sets [46], here we only present the results for the larger set.

For regular grid partitioning, without using a specific page allocation technique, the retrieval of the buckets may include arbitrary number of seeks. For example, if we define a range query with two corner points (1,2) and (4,3), then from disk 3 the query retrieves the pages 1 and 5, which needs two seek operations. Therefore, first we evaluate the current regular partitioning techniques with the expected numbers of seeks. Retrieval starts with an initial seek in each disk and a seek each time the accessed page is not contiguously stored in the accessed disk.

In the first set of experiments, we compared concentric partitioning based data space mapping with the best cyclic declustering based on regular partitioning along with the page allocation techniques that we have proposed. We implemented the concentric hypercubes, and hyperpyramids

with page allocation and bucket identification schemes discussed in this paper. In this setup, we estimate the number of seeks based on the pages accessed by the intersected query. On the basis of the page allocation techniques discussed, we calculate the number of seeks as the number of times, we need to access non-contiguous pages in each disk. This technique clearly improves upon a random page allocation scheme which would require almost a seek for each page access Fig. 10.

In the next set of experiments, on the Stock data set, we compare the best possible performance of the regular grid partitioning approach with concentric partitioning based data space mapping. We fix the selectivity at 10^{-5} and the number of buckets to be 65536. Fig. 11 illustrates the results in the fast architecture for two different values for the number of disks, M . The concentric partitioning has a significant improvement over the current techniques. Especially hyperpyramid based approach performs much better than current techniques. The speedups in the faster disk setup are greater than the ones in the average disk setup. We also observe that with increase in the number of disks (from $M = 20$ to 100 in Fig. 11), the difference in performance between the hypercubic and hyperpyramid based partitioning is decreased with increase in the number of disks.

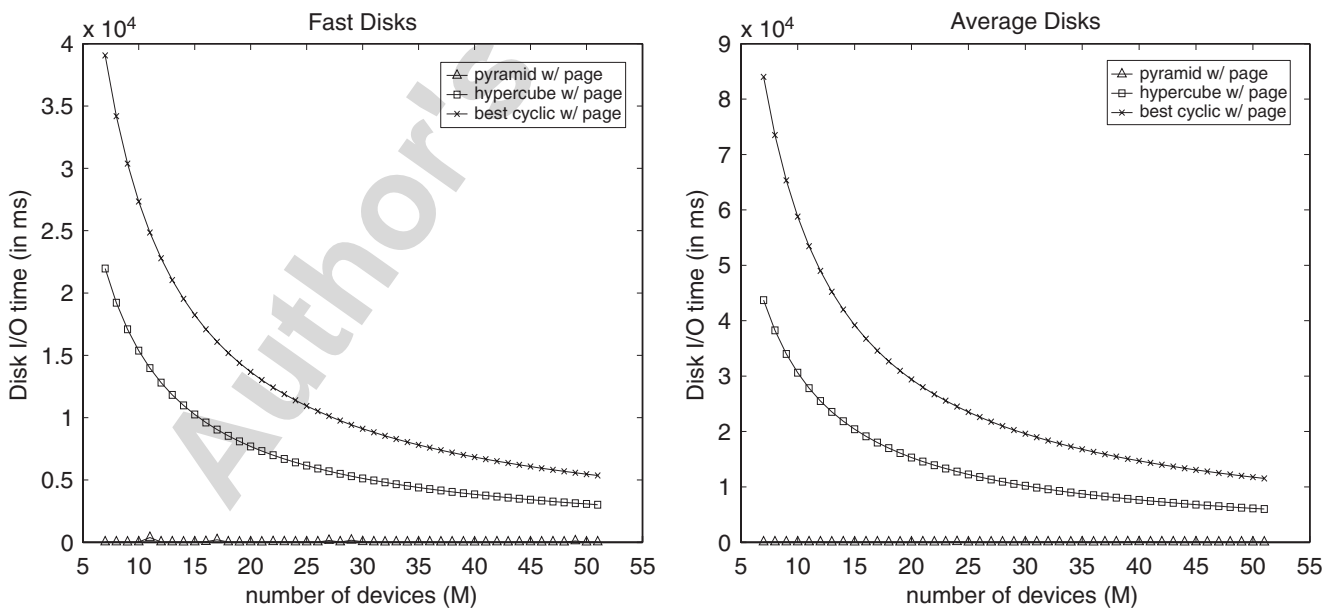


Fig. 10. Declustering techniques with page allocation.

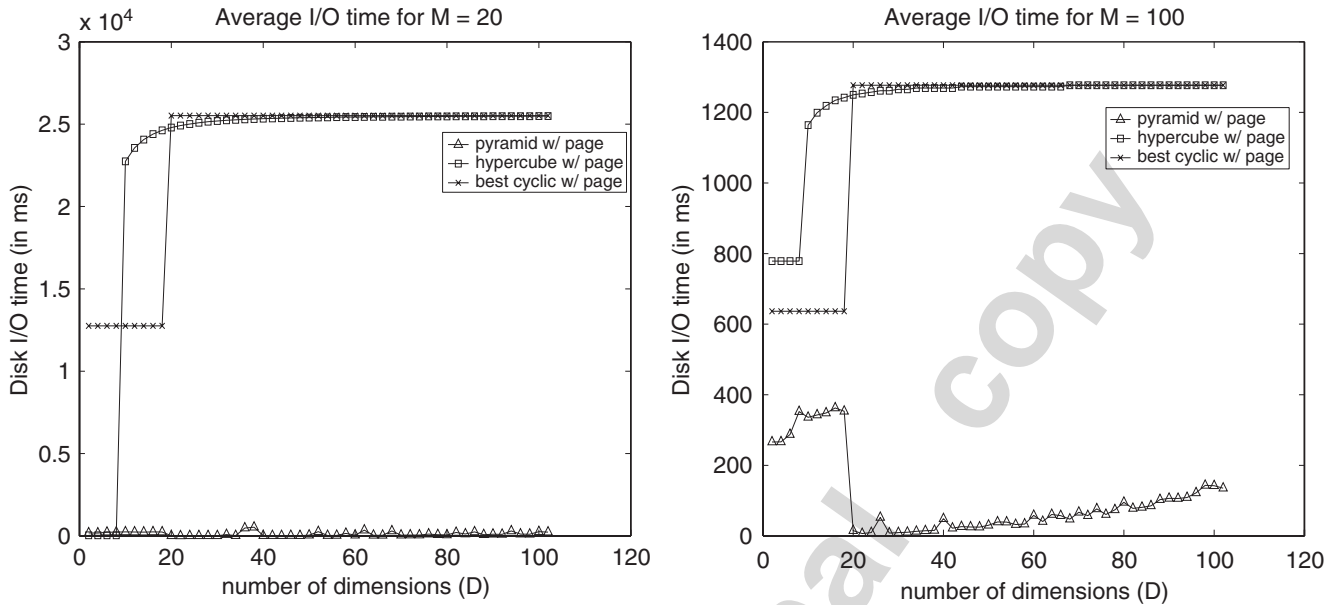


Fig. 11. Page allocation in hyperpyramid based techniques.

The results show that concentric hypercube partitioning based data space mapping is 1.8–5.4 times faster than the current regular grid partitioning approach. The hyperpyramid based mapping results 19–37 times faster queries than the ones in regular grid partitioning approach, and 9 times faster than the hypercube approach for average disks. The speedups in the fast disk setup are greater than the ones in the average disk setup. The new technique based on hyperpyramid partitioning has a speedup of 21–43 over the regular grid and 9.2–9.4 over the hypercube based approach.

In the second set of experiments, we analyze the effect of page allocation techniques on hyperpyramid based partitioning. Using page allocation reduces the query time significantly. Fig. 12 illustrates the comparison of hyperpyramid, hypercube and best cyclic allocation techniques using page allocation to the current approach which just declusters data proposed in [44] on a fast disk architecture.

5.3. Performance implications

Our experimental results validated the analytical cost formulas we have discussed. Each of the three

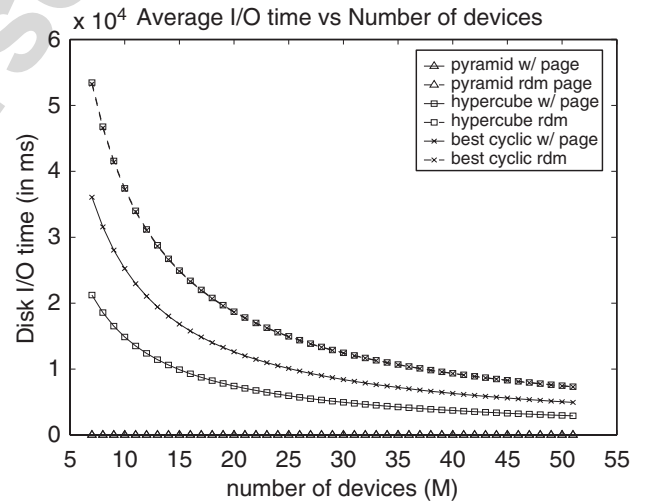


Fig. 12. Declustering with page allocation.

steps, i.e., partitioning, disk, and page allocation, has a variable effect in the overall performance of the system. The partitioning step, which aims to minimize the number of pages accessed, is identified as the most dominant factor in performance. Our experiments suggest that hyperpyramid partitioning performs best, while regular grid is the worst with a factor of 15–40 more pages accessed. This result is consistent with the known

problems of regular grid in high dimensions. Current declustering techniques traditionally focus on grid partitioning. They perform well especially when the data is uniform and two-dimensional. The performance drops as dimensionality increases, or as the data becomes non-uniform.

In terms of parallelism, hypercube technique achieves the best performance among all. However, since, hyperpyramids result in less buckets intersected by a typical query than the hypercubes, the resultant disk I/O cost in hyperpyramid based declustering is less than that of the hypercube in spite of the fact that the concentric declustering is optimal for hypercubes.

Once the data is declustered, page allocation plays an important role to make sure that similar data within each disk is still stored sequentially on disk. This third factor is becoming more important as the seek times, not the transfer times, become the bottleneck in many applications. In other words, the cost of a seek relative to the transfer cost increasing. Our performance evaluation establishes that a good page allocation scheme can speed up disk access time significantly by reducing the number of seeks. In fact, page allocation of concentric cubes guarantees sequential access. Hyperpyramid based partitioning guarantees $2 \cdot d$ seeks in the worst case.

The proposed data space mapping techniques are based on non-overlapping structures, which enable more effective parallelism and intra-disk organization. As an alternative, one could utilize tree-based structures (such as parallel R-trees), however their overlapping partitions does not permit the use of an efficient declustering approach.

6. Conclusion

A physical mapping of multi-dimensional data to multiple servers is a key factor for high performance large-scale systems. The current data placement research has mostly focused on distributing data under strong assumptions such as the existence of a uniform grid index, hence stayed largely theoretical. The proposed mapping investigates alternative partitioning schemes, introduces

the intra-disk organization coupled with declustering, and has been shown to work effectively for non-uniform real-life data sets.

We explored two partitioning schemes. Concentric hyperspaces has the advantage of being optimal for parallel I/O and for minimizing the seek time. Pyramid partitioning is effective in reducing the number of pages accessed as a result of a query. We developed bucket identification techniques for these partitioning strategies which are needed to process queries. Experiments on high dimensional datasets demonstrate the improvement in performance of range queries with realistic selectivities.

References

- [1] SciDAC. Scientific data management center. <http://sdm.lbl.gov/sdmcenter/>, 2002.
- [2] A. Acharya, M. Uysal, J. Saltz, Active disks: programming model, algorithms and evaluation, in: ASPLOS-VIII, September 1998, pp. 81–91.
- [3] P. Bernstein, M. Brodie, S. Ceri, D. DeWitt, M. Franklin, H. Garcia-Molina, J. Gray, J. Held, J. Hellerstein, H. Jagadish, M. Lesk, D. Maier, J. Naughton, H. Pirahesh, M. Stonebraker, J. Ullman, The Asilomar Report on database research, ACM Sigmod Record 27 (4) (1998).
- [4] H. Samet, The Design and Analysis of Spatial Structures, Addison Wesley Publishing Company, Inc., MA, 1989.
- [5] V. Gaede, O. Gunther, Multidimensional access methods, ACM Comput. Surveys 30 (1998) 170–231.
- [6] J. Nievergelt, H. Hinterberger, K.C. Sevcik, The grid file: an adaptable, symmetric multikey file structure, ACM Trans. Database Systems 9 (1) (1984) 38–71.
- [7] J.T. Robinson, The kdb-tree: a search structure for large multi-dimensional dynamic indexes, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, 1981, pp. 10–18.
- [8] A. Guttman, R-trees: a dynamic index structure for spatial searching, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, 1984, pp. 47–57.
- [9] N. Beckmann, H. Kriegel, R. Schneider, B. Seeger, The R* tree: an efficient and robust access method for points and rectangles, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, May 23–25, 1990, pp. 322–331.
- [10] H.C. Du, J.S. Sobolewski, Disk allocation for cartesian product files on multiple-disk systems, ACM Trans. Database Systems 7 (1) (1982) 82–101.
- [11] M.H. Kim, S. Pramanik, Optimal file distribution for partial match retrieval, in: Proceedings of the ACM

- SIGMOD International Conference on Management of Data, Chicago, 1988, pp. 173–182.
- [12] C. Faloutsos, P. Bhagwat, Declustering using fractals, in: Proceedings of the second International Conference on Parallel and Distributed Information Systems, San Diego, CA, Jan 1993, pp. 18–25.
- [13] S. Berchtold, C. Bohm, B. Braunmuller, D.A. Keim, H.-P. Kriegel, Fast parallel similarity search in multimedia databases, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, Arizona, USA, 1997, pp. 1–12.
- [14] K.A. Hua, H.C. Young, A general multidimensional data allocation method for multicomputer database systems, in: Database and Expert System Applications, Toulouse, France, September 1997, pp. 401–409.
- [15] S. Prabhakar, K. Abdel-Ghaffar, D. Agrawal, A. El Abbadi, Cyclic allocation of two-dimensional data, in: International Conference on Data Engineering, Orlando, FL, February 1998, pp. 94–101.
- [16] S. Prabhakar, D. Agrawal, 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, pp. 78–87.
- [17] C.-M. Chen, R. Bhatia, R. Sinha, Declustering using golden ratio sequences, in: International Conference on Data Engineering, San Diego, CA, February 2000, pp. 271–280.
- [18] R. Bhatia, R.K. Sinha, C.-M. Chen, Hierarchical declustering schemes for range queries, in: Advances in Database Technology—EDBT 2000, Seventh International Conference on Extending Database Technology, Lecture Notes in Computer Science, Konstanz, Germany, March 2000, pp. 525–537.
- [19] C.-M. Chen, C.T. Cheng, From discrepancy to declustering: near optimal multidimensional declustering strategies for range queries, in: Proceedings of the ACM Symposium on Principles of Database Systems, Wisconsin, Madison, 2002, pp. 29–38.
- [20] S. Ghandeharizadeh, 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, August, 1990, pp. 481–492.
- [21] S. Ghandeharizadeh, D.J. DeWitt, W. Qureshi, A performance analysis of alternative multi-attribute declustering strategies, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, June 1992, pp. 29–38.
- [22] S. Ghandeharizadeh, 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, August 1990, pp. 481–492.
- [23] K. Kim, V.K. Prasanna, Latin squares and parallel array access, *IEEE Trans. Parallel Distr. Systems* (1993) 361–370.
- [24] C. Fan, A. Gupta, J. Liu, Latin cubes and parallel array access, in: International Parallel Processing Symposium, April 1994, pp. 128–132.
- [25] M.J. Atallah, S. Prabhakar, Almost optimal parallel block access for range queries, in: Proceedings of the ACM Symposium on Principles of Database Systems, Dallas, TX, May 2000, pp. 205–215.
- [26] B. Chor, C.E. Leiserson, R.L. Rivest, J.B. Shearer, An application of number theory to the organization of raster-graphics memory, *J. Assoc. Comput. Mach.* 33 (1) (1986) 86–104.
- [27] L.T. Chen, D. Rotem, Declustering objects for visualization, in: Proceedings of the International Conference on Very Large Data Bases, Dublin, Ireland, August 1993, pp. 85–96.
- [28] H. Ferhatosmanoglu, A. Tosun, A. Ramachandran, Replicated declustering for parallel i/o, in: Proceedings of the ACM Symposium on Principles of Database Systems, Paris, France, June 2004, pp. 125–135.
- [29] S. Ghandeharizadeh, D.J. DeWitt, A multiuser performance analysis of alternative declustering strategies, in: Proceedings of the International Conference on Data Engineering, Los Angeles, CA, February 1990, pp. 466–475.
- [30] S. Ghandeharizadeh, D.J. DeWitt, A performance analysis of alternative multi-attribute declustering strategies, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, San Diego, 1992, pp. 29–38.
- [31] K.A.S. Abdel-Ghaffar, A. El Abbadi, Optimal disk allocation for partial match queries, *ACM Trans. Database Systems* 18 (1) (1993) 132–156.
- [32] K.A.S. Abdel-Ghaffar, A. El Abbadi, Optimal allocation of two-dimensional data, in: International Conference on Database Theory, Delphi, Greece, January 1997, pp. 409–418.
- [33] R.K. Sinha, R. Bhatia, C.-M. Chen, Asymptotically optimal declustering schemes for range queries, in: Eighth International Conference on Database Theory, Lecture Notes in Computer Science, Springer, London, UK, January 2001, pp. 144–158.
- [34] B. Moon, A. Acharya, J. Saltz, Study of scalable declustering algorithms for parallel grid files, in: Proceedings of the Parallel Processing Symposium, April 1996.
- [35] P. Ciaccia, 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, Berlin, Germany, September 1996, pp. 110–123.
- [36] I. Kamel, C. Faloutsos, Parallel R-trees, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, San Diego, CA, June 1992, pp. 195–204.
- [37] S. Berchtold, D.A. Keim, H.P. Kriegel, The X-tree: an index structure for high-dimensional data, in: 22nd Conference on Very Large Databases, Bombay, India, 1996, pp. 28–39.

- [38] M. Coyle, S. Shekhar, Y. Zhou, Evaluation of disk allocation methods for parallelizing spatial queries on grid files, *J. Comput. Software Eng.* 1995.
- [39] S. Shekhar, D.R. Liu, Partitioning similarity graphs: a framework for declustering problems, *Inform. Systems* 21 (4) (1996) 475–496.
- [40] S. Shekhar, S. Ravada, V. Kumar, D. Chubb, G. Turner, Declustering and load balancing methods for parallelizing geographical information systems, *IEEE Trans. Knowledge Data Eng.* 10 (4) (1998) 632–655.
- [41] J.C. Simon, *Patterns and Operators*, McGraw-Hill, New York, 1986.
- [42] S. Berchtold, C. Bohm, H.-P. Kriegel, The pyramid-technique: towards breaking the curse of dimensionality, in: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Seattle, Washington, USA, June 1998, pp. 142–153.
- [43] C. Faloutsos, Gray codes for partial match and range queries, *IEEE Trans. Software Eng.* 14 (10) (1998) 1381–1393.
- [44] H. Ferhatosmanoglu, D. Agrawal, A. El Abbadi, Concentric hyperspaces and disk allocation for fast parallel range searching, in: *Proceedings of the International Conference Data Engineering*, Sydney, Australia, March 1999, pp. 608–615.
- [45] Cheetah Specifications, <http://www.seagate.com/disc/cheetah/cheetah.shtml>, November 1998.
- [46] H. Ferhatosmanoglu, D. Agrawal, A. El Abbadi, Clustering declustered data for efficient retrieval, in: *Proceedings of the Conference on Information and Knowledge Management*, Kansas City, Missouri, November 1999, pp. 343–350.

Author's personal