Clustering Declustered Data for Efficient Retrieval

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

Abstract

As databases increasingly integrate multimedia information in the form of image, video, and audio data, both the dimensionality and the amount of data that need to be processed is increasing rapidly. It becomes necessary to support the efficient retrieval of large amounts of multimedia data. Declustering techniques for multi-disk architectures have been effectively used for storage in relational databases. In this paper, we first establish that besides exploiting the parallelism, a careful organization of each disk must be considered for fast searching. We introduce the notion of page allocation and data space mapping which can be used to organize and retrieve multidimensional data. We work on these notions based on three different partitioning strategies: regular grid partitioning, concentric hypercubes and hyperpyramids. We develop techniques that satisfy efficient retrieval by optimizing the number of buckets retrieved by the query, disk arm movement and I/O parallelism. We prove that concentric hypercube based mapping satisfies the optimal clustering and optimal parallelism. We develop techniques based on hyperpyramid partitioning which reduces the number of buckets retrieved by the query and has very efficient inter and intra disk organizations. We evaluate the performance of proposed techniques by comparing it with the current best approaches. The new techniques lead to very significant improvement, up to 43 times, over the existing techniques, therefore resulting in fast retrieval of multimedia data.

1 Introduction

As databases increasingly integrate multimedia information in the form of image, video, and audio data, it becomes necessary to support the efficient retrieval of large amounts of high dimensional data. In fact, both the dimensionality and the amount of data that need to be processed has been increasing rapidly. Commercial data warehouses are doubling their size every 9-12 months and 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 [7]. Traditional retrieval methods based on index structures developed for single disk and single processor environments are becoming ineffective for the storage and retrieval of high dimensional data in multiprocessor and multiple disk environments. It has therefore become necessary to revisit these techniques in order to support scalable and fast retrieval of high dimensional data. An obvious approach is to develop new storage and retrieval techniques that exploit I/O device and processor parallelism.

In the past, storage redundancy has been successfully exploited in the context of data declustering on multiple disks. The basic idea of declustering 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) [8], Fieldwise Exclusive OR (FX) [17], Hilbert (HCAM) [10], Near Optimal Declustering (NoD) [4], General Multidimensional Data Allocation (GMDA) [16] and the Cyclic Allocation Schemes [20] are some of the well-known disk allocation techniques. Although declustering was originally proposed for range and partial match queries, recently it has been extended for similarity queries [4, 21] over multimedia databases.

Declustering techniques distribute data on different disks but do not specify the in-disk organization of data within a single disk. However, 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[7]

In the current declustering approach, it is generally assumed that an index structure is used to retrieve data from parallel disks. Hence, the declustering work views the parallel search of the data as a different problem. We first note that the current high-dimensional indexing techniques are still not efficient even for single disk and single processor environments. Numerous index structures exist that facilitate search and retrieval of two dimensional data or spatial databases [22, 13, 14, 3]. With the proliferation of multimedia data these index structures have been extended and new ones are being proposed to deal with the high dimensional nature of information [18, 6, 24, 19, 15]. In fact, Berchtold et al. [5] have argued that using multidimensional index structures for searching beyond a certain dimension becomes worse than a sequential scan. Besides being inefficient in high dimensions, it is hard to achieve optimal parallelism for these index structures. We therefore need techniques which are optimized for parallel architectures and results in efficient retrieval. If the underlying index structures use different ways for bucket creation and retrieval of the buckets, which is the current situation, the careful allocation on different disks may not result in efficient retrieval parallelism. Furthermore, these index structures may create a different bucket organization from the one created by the declustering partitioning. Since the initial partitioning and indexing are incompatible, even a perfect distribution of the data with respect to the partitioning used by declustering may not necessarily lead to fast parallel searching.

The requirement of efficient and compatible mapping lead us to investigate indexing of data space, data space mapping, optimized for parallel searching. Is it then feasible and effective to extend the declustering technique itself to parallel searching so that they are used as the overall data space indexing mechanism in the system? To reach this goal, besides exploiting the parallelism as much as possible, careful consideration must also be given to in-disk organizations.

In Section 2 we define the problem, the major steps needed for the extension, and the necessary optimality definitions in the extended system. In Section 3 we work on the problem in the context of various partitioning strategies and develop new techniques for each of them. Section 3.1 considers the problem under regular grid partitioning and proposes a set of possible solutions. In Section 3.2, we consider concentric hypercube partitioning and develop a complete data space mapping which is shown to be optimal for both in-disk and intra-disk retrieval. In Section 3.3 we define a third very important parameter which must be minimized for efficient retrieval. We propose an efficient technique, based on hyperpyramid partitioning, which is not theoretically optimal but achieves very significant speedups in practice. Finally, in Section 4 we show the significant improvement achieved by new techniques technique using a simulation system.

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, find and retrieve all buckets that contain data relevant for the query. Similarly, given a point, find the bucket that contains the point.

In the context of current data declustering, the partitioning and disk allocation steps are done as a preliminary step so that the results can be used during the query execution. We note that only the disk identity is specified by the allocation technique, i.e., the bucket identification process only provides information as to the identity of the disk where the bucket resides. It is for this reason that an additional index structure or directory is required to locate the particular pages on which the data is stored. For instance, in [4], the declustering of the buckets is done by using regular grid partitioning, and the data retrieval is handled by a parallel version of X-tree. An appropriate partitioning strategy must be used, which efficiently supports the allocation and identification steps.

Current allocation methods are just used for the storage of data into appropriate disks and do not consider the in-disk organization of the data. 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 $Q = ([a_1, b_1], [a_2, b_2], [a_3, b_3], \ldots, [a_d, b_d])$ 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 *BucketId*s and it represents the buckets that must be retrieved by the query. For example, given a query region we need to develop methods that will allow us to identify all the intersecting buckets. After identifying the buckets, the allocation functions will be used to retrieve the buckets by identifying the disk and the page inside the disk.

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 [8].

Definition 1 Optimal Disk Allocation. *A disk allocation policy is said to be* optimal *if no hyper-rectangular area A has more than $\lceil \frac{A}{M} \rceil$ 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:

$$t_{seek} + t_{latency} + \lceil \frac{A}{M} \rceil \cdot t_{transfer}$$

3 Data Space Mapping with Clustering and Declustering

We discuss the data space mapping for efficient retrieval based on various partitioning strategies. Without loss of generality, throughout the paper, we consider a d-dimensional space where the data points are normalized in [0..1]. In Section 3.1 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 3.2, and hyperpyramids in Section 3.3.

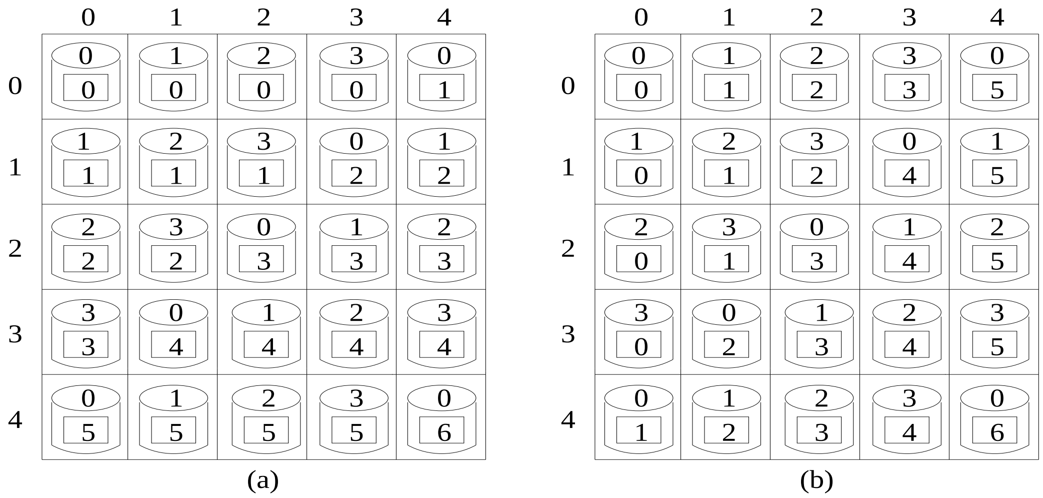

Figure 1: Disk and Page Allocation for Regular Grid Partitioning

3.1 Regular Grid Partitioning

Most of the current techniques use regular grid partitioning as the underlying partitioning strategy. In this section, we discuss the page allocation and bucket identification issues for regular grid partitioning. Assuming regular grid partitioning, that the i^{th} dimension is partitioned into equal sized N_i regions and the bucket corresponding to a point $(x_1, x_2, ..., x_d)$ can be determined as follows:

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

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. Figure 1 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 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 [8]. Several other single disk allocation mechanisms have been proposed to linearize a multidimensional space on a single disk [9]. In Figure 1(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 Figure 1(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:

$$disk(BucketId) = (i + j) \bmod M$$
$$page(BucketId) = \lfloor \tfrac{iN+j}{M} \rfloor$$

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 $BucketId$s and than finding the locations.

More effective but complex disk allocation techniques have been developed [16, 20]. 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 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. Next, we define general page allocation formula, besides disk allocation, of $BucketId = (i, j)$ for two dimensional Cyclic Allocation Techniques by the following functions.

$$disk(BucketId) = d = (Hi + j) \bmod M$$
$$page(BucketId) = \lceil \tfrac{j+1-((d-i.H) \bmod M)}{M} \rceil + \textstyle\sum_{k=0}^{i-1} \lceil \tfrac{N-((d-k.H) \bmod M)}{M} \rceil - 1.$$

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 0^{th} 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 Figure 1(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.

$$page(3, 2) = \lceil \tfrac{4-((1-3) \bmod 4)}{4} \rceil + (\lceil \tfrac{5-((1-0) \bmod 4)}{4} \rceil + \lceil \tfrac{5-((1-1) \bmod 4)}{4} \rceil + \lceil \tfrac{5-((1-2) \bmod 4)}{4} \rceil) - 1$$
$$= 1 + (1 + 2 + 1) - 1 = 4$$

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.

3.2 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 [1]. 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 [4].

We have proposed using concentric hyperspaces, as an alternative to regular partitioning in disk allocation for range queries [12]. In this section, we establish that the concentric partitioning approach is also appropriate to be used in a complete mapping of multidimensional data space to physical storage for further retrieval. 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. We now develop a complete set of extensions based on concentric partitioning for retrieval and in-disk organization with bucket identification and page allocation. First we discuss these issues for hypercubes, then for hyperpyramids.

The first partitioning strategy that we consider under the concentric partitioning is concentric hypercubes. Concentric hypercubes are created by starting from the center, $(0.5, 0.5, .., 0.5)$, of the space and creating expanding concentric hypercubic shells. If the volumes of these partitions are equal, inside partitions will be thicker and the outer partitions (near the surface) become thinner and thinner as shown in Figure 2. The innermost partition, the 0^{th} partition, is a hypercube itself. Other than that, each i^{th} shell partition is created by two adjacent concentric hypercubes, i.e., i^{th} and $(i-1)^{th}$ hypercubes. We define a table, $T[0..(p-1)]$, where p is the total number of partitions. Each $T[i]$ consists of a single number reflecting the distance of the center from any surface of the corresponding hypercube, i. These numbers will be used to represent the hypercubes, and hence the hypercubic shell partitions. To create equi-volumed hypercubic shell partitions, the following formula can be used [12]:

$$T[i] = \frac{\sqrt[d]{(i+1)/p}}{2}.$$

The bucket identification of a point $(x_1, x_2, ..., x_d)$ is carried out as follows. First a single value y such that $y = max(|x_1 - 0.5|, |x_2 - 0.5|, .., |x_d - 0.5|)$ 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, $Q = ([a_1, b_1], [a_2, b_2], [a_3, b_3], \ldots, [a_d, b_d])$ is carried out as follows. We must find the $BucketId$s in the retrieval set, which is defined to be the set of buckets

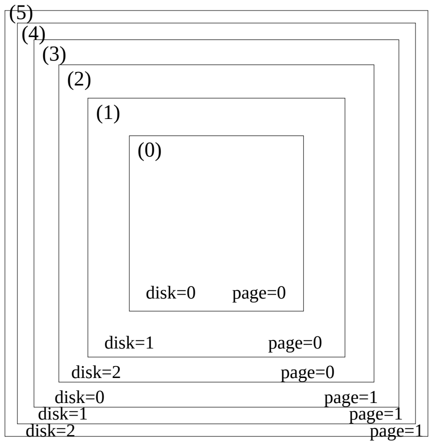

Figure 2: Disk and Page Allocation for Concentric Hypercube Partitioning

to be retrieved. By using the properties of concentric hypercubes, we can compute the maximum and minimum $BucketId$s 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

$$max_b = max(BucketId(a_1, a_2, a_3, \ldots, a_d), BucketId(b_1, b_2, b_3, \ldots, b_d)).$$

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 $(min_b - 1)^{st}$ does not. Since, we have computed the minimum and the maximum $BucketId$s, the retrieval set is the set of $BucketId$s within the range min_b and max_b.

Given a $BucketId$, the *disk allocation* and *page allocation* functions are as follows.

$$disk(BucketId) = BucketId \bmod M$$
$$page(BucketId) = \lfloor \frac{BucketId}{M} \rfloor$$

where M is the number of disks and I/O devices available. The disk allocation method assigns the buckets to disks by starting from the innermost partition and then sequentially allocating the hypercubic shells in a round robin fashion. The page allocation assigns the page numbers in the same way within each disk separately.

An important property of this schema is that we do not need to apply the allocation functions to all buckets in the bucket retrieval set individually. The number of buckets in the retrieval set and min_b are enough to locate all buckets. The procedure for this, $d = disk(min_b), p = page(min_b)$ is as follows. If B is the number of elements in the retrieval set, then the following is the location of the pages that need

BucketId	24	23	22	21	20	19	18	17	16
DiskId	0	5	4	3	2	1	0	5	4
PageId	4	3	3	3	3	3	3	2	2

Table 1: Allocation Table

to be retrieved: Read $\lceil \frac{(B-i)}{M} \rceil$ pages sequentially from each disk, D, where i is the minimum integer s.t. $D = (d+i) \bmod M$. The starting page P in each disk is $p+1$ if $D < d$, and p if $D >= d$.

Let us explain this procedure by an example. Assume $M = 6$ disks are available in the system and as a result of a query the number of elements in the retrieval set is $B = 9$, and min_b is 16. By the above technique, since $p = page(min_b) = \lfloor \frac{16}{6} \rfloor = 2$ and $d = disk(min_b) = 16 \bmod 6 = 4$, the locations of the buckets in the result are: pages 2,3 from disks 4 and 5, page 2 from disks 1,2, and 3, and pages 3,4 from disk 0, which is shown in the allocation table 1.

Next, we examine the parallelism and in-disk organization result under the concentric hypercube based retrieval. We first define two terms, the global retrieval set and the local retrieval set.

Definition 3 **Global retrieval set**, *R, of a query Q, is the set of BucketIds that represent the buckets that must be retrieved as the result of Q.*

Definition 4 **Local retrieval set**, *R_D of a query Q, for a disk D, is the set of local PageIds that represents the pages that must be retrieved from disk D as the result of Q.*

Lemma 1 *The global retrieval set R of any hyperrectangular range query is exactly a set of consecutive numbers, i.e., i, $i+1$, .., $i+k$. (There is no gap between any two numbers)*

Proof Range query defines a hyperrectangular query as discussed above. The query must retrieve all the points located within the ranges of this hyperrectangle. Since the partitioning is done in a concentric manner, if a hyperrectangular range query intersects any two buckets, i and $i+k$, then it also intersects all the buckets between i and $i+k$. This is also true for the maximum, max_b, and minimum, min_b, numbers in the retrieval set, R. Therefore, the retrieval set, R, of the range query is the set of consecutive numbers between min_b and max_b. □

Lemma 2 *Concentric hypercube based retrieval uses an optimal disk declustering technique for range queries.*

Proof Assume $R = i, i+1, .., i+k-1, i+k$. These buckets are allocated to disks $i \bmod M, (i+1) \bmod M, ..., (i+k-1) \bmod M, (i+k) \bmod M$. 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 $\lceil \frac{k+1}{M} \rceil$, which is the optimal cost. (Note that $A = k+1$) □

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

Lemma 3 *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 1, 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. □

3.3 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 [12, 5]. Therefore, it is important to develop techniques for hyperpyramid based partitioning. In this partitioning technique, the data space is first divided into $2d$ major hyperpyramids, each has the center point as its apex, as shown in Figure 3. Then, $p/2d$ hyperpyramids, $\triangle_1, \triangle_2, ..., \triangle_{p/2d}$ with a common apex are created inside each major hyperpyramid. The hypertrapezoids created in this way are the partitions that represent buckets. Figure ref/pyramids illustrates a two-dimensional example. In this figure, $\triangle ABC$ is a *major* hyperpyramid, $\triangle ADE$ is a hyperpyramid, and $\square DBCE$ is a hypertrapezoid. We compute the distance of each hyperpyramid from the center point which will be used in bucket identification and query processing. The distance between the base of the hyperpyramid $\triangle_i$ and the apex is defined as the height, h_i. Similar to the hypercube case, we define a table H[0..(p/2d-1)] and store these distance values in this table. To create equi-volumed hypertrapezoid partitions, the height of pyramid $\triangle_i$ is $(\sqrt[d]{2 \cdot i \cdot d/p})/2$ [12].

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

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

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.

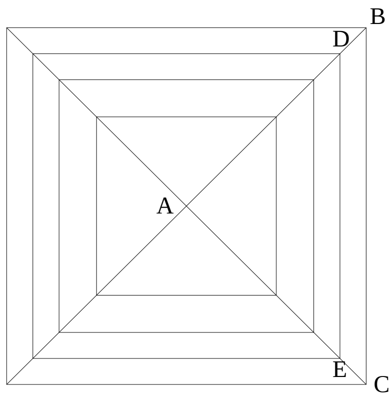

Figure 3: Pyramids and Trapezoids

The bucket identification of a range query, $Q = ([a_1, b_1], [a_2, b_2], [a_3, b_3], \ldots, [a_d, b_d])$ 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 [5] to create a single one-dimensional index value for each multidimensional 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 [5].

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 dataset. 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 1^{st} row represents P_1, the 2^{nd} row represents P_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. 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

TIME	Fast Disk	Average Disk
Average Seek Time(msec)	5.2	7.0
Latency(msec)	2.99	2.99
Transfer(MByte/sec)	21	14

Table 2: Disk Specification

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.

4 Performance Evaluation

We have developed a simulation system for evaluating the data space mapping techniques discussed in this paper. We implemented the disk and page allocation and retrieval mechanisms for different partitioning strategies. Regular grid partitioning and concentric hypercubes and hyperpyramids based partitioning are implemented. The page allocation and retrieval techniques, i.e. bucket and query identification, introduced in this paper are implemented on top of 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. In this way, we are able to compare our technique with the best possible cyclic allocation technique under regular partitioning.

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 [23]. Table 2 illustrates the disk architectures. We varied the number of disks available in the system from 4 to 30.

The techniques are evaluated for two sets of data. In both data sets, each dimension is divided into two for regular partitioning. The dimensionality is 16 in the first set and 20 in the second set. There are 65,536 buckets in the first dataset and 1,048,576 buckets in the second dataset. The bucket size in each setup is $8KBytes$.

The processing of a query is carried out as follows. First, the buckets that are needed to be retrieved by the query are identified by using the bucket identification techniques discussed in this paper. Note that, for range queries, these buckets are the ones whose bounding regions intersect the query. Then by using the disk and page allocation functions, the location of the 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 500 random range queries. The queries have realistic selectivities that vary between 10^{-4} and 10^{-5} for the small dataset and between 10^{-5} and 10^{-6} for the large dataset. For lack of space, we only present the results for the larger set (see [11] for both set of experiments).

As discussed earlier, since concentric hypercubes, with page allocation, guarantees sequential access, only 1 seek is needed for each data transfer from each disk. Similarly, hyperpyramid based partitioning

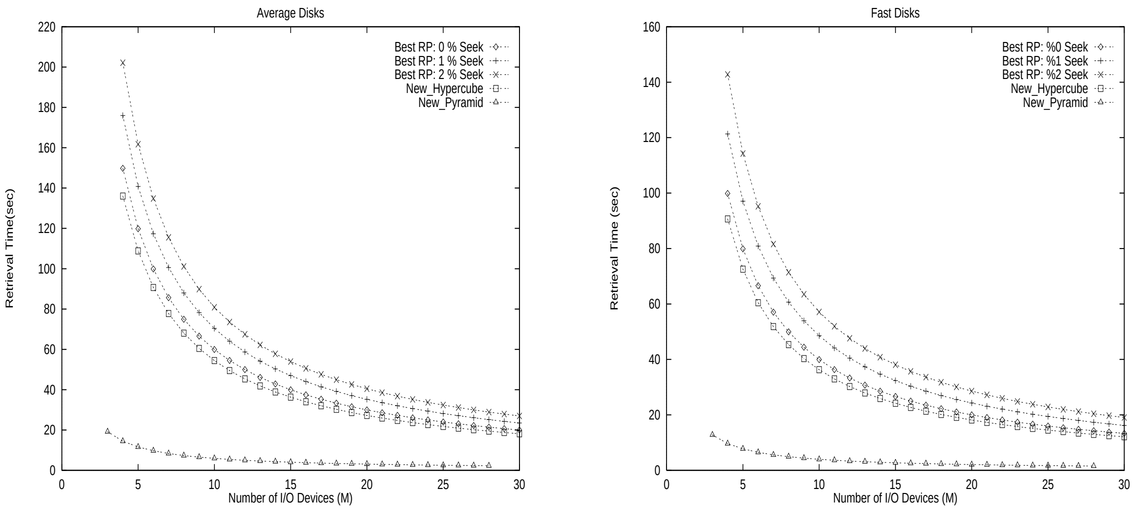

Figure 4: Small Amount of Seeks

guarantees $2 \cdot d$ in the worst case. 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. For example, in Figure 1(a), 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 various numbers of seeks. Retrieval starts with an initial seek in each disk and then $s\%$ seeks, $0 < s < 100$, where s denotes the number of seeks needed for transferring 100 buckets from disk. At the end of this section, we also discuss the effect of page allocation schemes for hyperpyramid based techniques.

In the first set of experiments, we compared concentric partitioning based data space mapping with the best cyclic declustering based on regular partitioning, with small number of seeks. We implemented the concentric hypercubes, and hyperpyramids with page allocation and bucket identification schemes discussed in this paper. In this setup, the current techniques are assumed to have 0%, 1%, and 2% seeks. Although current approaches do not guarantee the number of seeks, we also include the 0% seek case in this experiment to compare the best possible performance of the regular grid partitioning approach with concentric partitioning based data space mapping. The number of I/O devices in the system is varied from 4 to 30. Figure 4 illustrates the results in two different architectures: average and fast disks. Even in the 0% seek case, the concentric partitioning has a significant improvement over the current techniques. Especially hyperpyramid based approach performs much better than current techniques. For 1% and 2% seeks this difference increases. The speedups in the faster disk setup are greater than the ones in the average disk setup.

In the second set of experiments, we varied the seek percentage as 5%, 10%, and 15% to see the difference in the performance. The number of seeks may differ with the query. Figure 5 illustrates the results of these experiments. The results show that concentric hypercube partitioning based data space mapping is 2 to 4.6 times faster than the current regular grid partitioning approach. Similar to

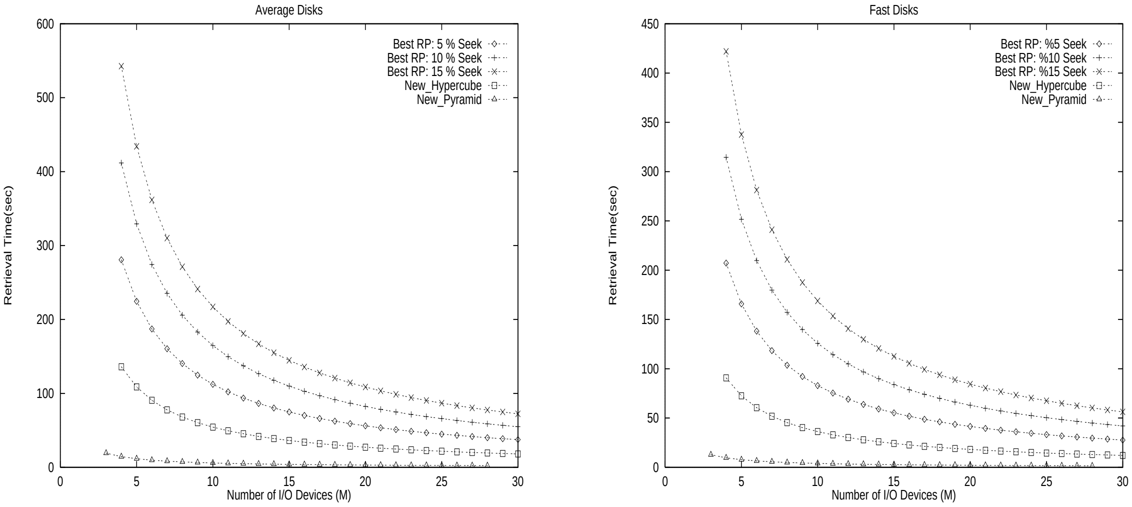

Figure 5: Average Amount of Seeks

the previous results for small seeks, here hyperpyramid based mapping is much faster than the current ones and hypercube technique. The hyperpyramid based mapping results 19 to 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 to 43 over the regular grid and 9.2 to 9.4 over the hypercube based approach. As the seek percentage increases, the speedup of the concentric partitioning based mapping increase.

Although the hyperpyramid based data space mapping may have larger number of seeks than the hypercubes, its resulting query time is more efficient than hypercube based techniques. The hyperpyramid technique is an efficient way to partition the high-dimensional data spaces. By adding disk and page allocation to the hyperpyramid technique, the parallelism is maximized and the number of seeks is minimized. Therefore in the first two sets of experiments, we get a very significant speedup by using techniques using hyperpyramid based partitioning.

In the third set of experiments, we analyze the effect of page allocation techniques on hyperpyramid based partitioning. Using page allocation reduces the query time significantly. Figure 6 illustrates the comparison of hyperpyramid techniques using page allocation to the current approach which just declusters data proposed in [12]. Similar to the previous experiments, we use various numbers of seeks for the current techniques to see different scenarios. Using page allocation technique results a speedup of 2.7 to 4.45 for average disks and 3.07 to 5.2 for fast disks over the current techniques.

5 Conclusion

In this paper, we argued that in order for declustering techniques to be effective, they must integrate both inter-disk and intra-disk allocations. We introduced the notion of page allocation and data space mapping, then develop solutions in the context of various partitioning strategies. We first discussed

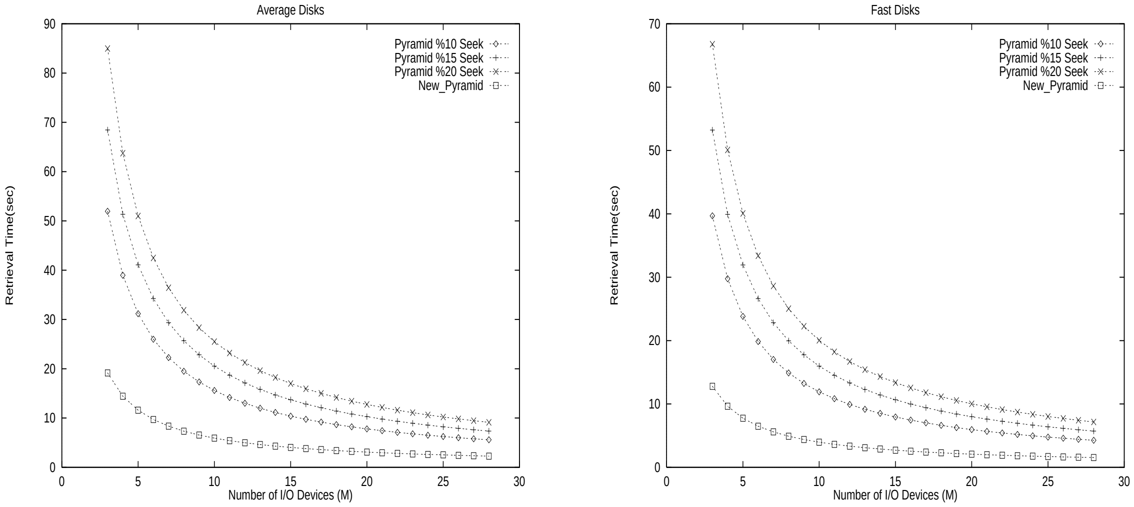

Figure 6: Page Allocation in Hyperpyramid based Techniques

some possible extensions to the most common declustering approaches based on regular grid partitioning and stated some of its problems in the context of data space mapping. We then introduced concentric hypercube partitioning which has been shown to be more amenable for disk allocation for range queries [12]. We developed page allocation and bucket identification techniques for both point and range queries. We illustrated that concentric hypercube partitioning allows both optimal disk and optimal page allocations. Similarly, we developed these operations for hyperpyramid partitioning. We established that hyperpyramid based partitioning is very efficient for range queries, since it reduces the expected number of buckets that are retrieved by the query. We developed page allocation for hyperpyramid partitions and established that in the worst case $2 \cdot d$ seek operations are enough for any query. We establish that concentric partitioning based data mapping results in very efficient query times. In fact, by using hyperpyramid based technique, we reach a speedup up to 43 times compared to the current techniques.

References

[1] K. A. S. Abdel-Ghaffar and A. El Abbadi. Optimal disk allocation for partial match queries. *ACM Transactions on Database Systems*, 18(1):132–156, March 1993.

[2] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active disks: Programming model, algorithms and evaluation. In *ASPLOS-VIII*, September 1998.

[3] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R* tree: An efficient and robust access method for points and rectangl es. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 322–331, May 23-25 1990.

[4] 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.

[5] 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.

[6] S. Berchtold, D. A. Keim, and H. P. Kreigel. The X-tree: An index structure for high-dimensional data. In *22nd. Conference on Very Large Databases*, pages 28–39, Bombay, India, 1996.

[7] 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, and J. Ullman. The asilomar report on database research, December 1998.

[8] 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.

[9] C. Faloutsos. Gray codes for partial match and range queries. *IEEE Transactions on Software Engineering*, 14(10):1381–1393, October 1988.

[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] H. Ferhatosmanoglu, D. Agrawal, and A. El Abbadi. Clustering declustered data for efficient retrieval. Technical report, Comp. Sci. Dept., UC, Santa Barbara, 1998.

[12] H. Ferhatosmanoglu, D. Agrawal, and A. El Abbadi. Concentric hyperspaces and disk allocation for fast parallel range searching. In *Proc. Int. Conf. Data Engineering*, Sydney, Australia, March 1999.

[13] V. Gaede and O. Gunther. Multidimensional access methods. *ACM Computing Surveys*, 30:170–231, 1998.

[14] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 47–57, 1984.

[15] J. Hellerstein, J. Naughton, and A. Pfeffer. Generalized search trees for database systems. In *Proceedings of the Int. Conf. on Very Large Data Bases*, pages 562–573, September 1995.

[16] 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.

[17] 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.

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

[19] D. B. Lomet and B. Salzberg. The hb-tree: A multi-attribute indexing method with good guaranteed performance. *ACM Transactions on Database Systems*, 15(4):625–658, December 1990.

[20] 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.

[21] 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.

[22] H. Samet. *The Design and Analysis of Spatial Structures.* Addison Wesley Publishing Company, Inc., Massachusetts, 1989.

[23] Cheetah Specifications. http://www.seagate.com/disc/cheetah/cheetah.shtml, November 1998.

[24] D. White and R. Jain. Similarity indexing with the SS-tree. In *Proc. Int. Conf. Data Engineering*, pages 516–523, 1996.