Efficient Declustering of Non-Uniform Multidimensional Data using Shifted Hilbert Curves

Hak-Cheol Kim[1], Mario A. Lopez[2], Scott T. Leutenegger[2], and Ki-Joune Li[1]

[1] School of Electrical and Computer Engineering, Pusan National University
Jangjeon, Kumjeong, Pusan, 609-735, Korea
{hkckim, lik}@pusan.ac.kr,
[2] Department of Computer Science, University of Denver, 2360 S. Gaylord St.,
Denver, CO 80208-0183, U.S.A
{mlopez,leut}@cs.du.edu

Abstract. Data declustering speeds up large data set retrieval by partitioning the data across multiple disks or sites and performing retrievals in parallel. Performance is determined by how the data is broken into "buckets" and how the buckets are assigned to disks. While some work has been done for declustering uniformly distributed low dimensional data, little work has been done on declustering *non-uniform high dimensional data*. To decluster non-uniform data, a distribution sensitive bucketing algorithm is crucial for achieving good performance. In this paper we propose a simple and efficient data distribution sensitive bucketing algorithm. Our method employs a method based on shifted Hilbert curves to adapt to the underlying data distribution. Our proposed declustering algorithm gives good performance compared with previous work which have mostly focused on bucket-to-disk allocation scheme. Our experimental results show that the proposed declustering algorithm achieves a performance improvement up to 5 times relative to the two leading algorithms.

Keywords: Declustering, Parallel I/O, Non-uniform Multidimensional data

1 Introduction

Modern scientific and business applications such as geographic information systems(GISs), data warehouse, information retrieval systems, remote-sensing data bases, etc., store and handle a massive amounts of *high dimensional* data. While storing tera bytes of data is now possible, time-efficient retrieval methods remain a crucial challenge. One promising approach is to distribute or stripe the data across multiple parallel disks, thus speeding up retrievals. How to distribute data so as to minimize response time of queries is referred to as the *declustering* problem. When we distribute data across multiple disks and assume the disks can be accessed independently, the response time of a query is proportional to the maximum number of disk blocks retrieved from any one of the disks.

Most previous work has focused on evenly distributing the data blocks across the disks, and assumes uniformly distribute data that is partitioned into disjoint regular tiles[4, 5, 10, 7, 6, 15, 2, 3].

Although some of the previous algorithms have been show to be optimal under the uniform data assumption, most real data is far from uniformly distributed. Uniformly tiling non-uniform data results in low storage utilization. Furthermore, even for uniform data, tiling results in more tiles than needed and becomes worse as dimension increases. Therefore, most of the tiling and mapping function based declustering schemes *show a drop in efficiency for high dimensional data, whether it is uniformly distributed or not.* The phenomena is demonstrated in section 3.

Some initial work has done on how to decluster non-uniform data [8, 12, 13] focusing on the bucket-to-disk problem. While most previous work allocates a disk number based on an interval number of each dimension, these algorithms can be applied to an arbitrary shaped data block by graph theoretic approaches. Although the methods have improved allocation methods, the bucketing methods are lacking [8, 13] or use the Gridfile [14] method resulting in poor space utilization in high dimensions.

Most of the previous work, whether aimed at uniform data or not, focused on the bucket-to-disk allocation scheme and ignored the effect of a bucketing method on the performance of a declustering algorithm. Our results show that performance can be improved by reducing the number of data blocks to be retrieved, which is mainly affected by a bucketing algorithm, and hence bucketing must be considered as well as allocation.

In this paper, we propose an efficient and simple bucketing algorithm for *high-dimensional non-uniform or uniform data.* By using multiple orderings of the data set using the orderings from a set of shifted Hilbert curve we build an *approximate* nearest neighbor graph. Our experimental results, using real skewed and high-dimensional data sets, show that our proposed declustering algorithm significantly improves performance relative to previous work.

The rest of this paper is organized as follows: In the next section, we present preliminaries of the declustering problem and show related work and our motivations in section 3. In section 4, we propose an approximate nearest neighbor graph and present our proposed declustering algorithm in section 5. We present experimental results in section 6 and conclude this paper in section 7.

2 Preliminaries

In this section, we present assumptions and definitions relevant to the declustering problem. To decluster data across multiple disks two steps are needed: 1) grouping the data into buckets, and 2) allocating the buckets to disks.

Formally, we define bucketing of data set P for a given disk blocking factor Bf_{max} as follows:

Definition 1. ***Bucketing of Data***
A bucketing of data π *is a collection of groups* $G_1^\pi, G_2^\pi, \cdots, G_{N_{page}}^\pi$ *where* $\mid G_i^\pi \mid \leq Bf_{max}$, $\bigcup_{i=1}^{N_{page}} G_i^\pi = P$ *and* $G_i^\pi \cap G_j^\pi = \varnothing$ *for* $i \neq j$ □

Based on this definition, we view a declustering algorithm as the following two steps for given M disks.

- **step 1. Bucketing:** $\{v \mid v \in P\} \rightarrow \{\ G_1^\pi, G_2^\pi, \cdots, G_{N_{page}}^\pi\}$
- **step 2. Allocation:** $\{\ G_1^\pi, G_2^\pi, \cdots, G_{N_{page}}^\pi\} \rightarrow \{0, 1, 2, \cdots, M\text{-}1\}$

When we distribute data across multiple disks, the response time of a query q is defined as the maximum number of disk blocks retrieved from any one of the disks. We formally define response time of a declustering scheme as follows:

Definition 2. ***Response time of a declustering algorithm***
For a given query q, the number of disk accesses $DA(q)$, *i.e, response time of a declustering algorithm, is determined as follows.*

$$DA(q) = \max_{i=1}^{M} DA_i(q)$$

where $DA_i(q)$ *is the i-th disk accesses to process a query* q □

Definition 2 means that a good declustering algorithm should access the same number of data blocks on any disk to process a query. Based on this notion, we define a strictly optimal declustering algorithm as follows:

Definition 3. ***Strictly optimal declustering algorithm***
A declustering algorithm is strictly optimal if

$$\forall q,\ DA(q) = \lceil \tfrac{N_{block}(q)}{M} \rceil$$

where $N_{block}(q)$ *is the number of data blocks touched by a query* q

Achieving the optimal response time in definition 3, for all queries without restrictions, has been show to be NP-complete[1]. Hence, any declustering algorithm has an additive error $\epsilon(> 0)$ and its actual response time is given as follows:

$$DA(q) = \lceil \frac{N_{block}(q)}{M} \rceil + \epsilon \tag{1}$$

Three factors affect the performance of a declustering algorithm in equation 1 : $N_{block}(q)$, the number data blocks touched by a query q, M, the number of parallel disks, and ϵ, an additive error determined by the allocation scheme. Since M is a constant, $N_{block}(q)$ and ϵ determine declustering performance.

3 Related Work and Motivations

3.1 Related Work

Several declustering schemes have been proposed to speed up retrieval for range queries or partial match queries from large data sets. We can classify them into two categories:

- **Allocation Oriented** This group of algorithms assume the data is first tiled into uniform region-sized buckets by splitting each dimension into disjoint intervals. Then, various bucket-to-disk allocation functions were proposed. The algorithms assume a uniform data distribution. Most previous work belongs to this category and includes Disk Modulo(DM)[5], Field-wise Xor(FX)[10], Error-Correcting Code(ECC)[7], Hilbert Curve Allocation Method(HCAM)[6], Cyclic Allocation Scheme[15], Golden-Ratio Sequence(GRS)[3], Coloring Scheme[2], and Discrepancy theory based declustering scheme[4].
- **Bucket Formation Oriented Using Graph Theory** Regular grid-shaped partitions (buckets) lead to low storage utilization with *non-uniform* data or even with uniform data in high dimensional space. A few previous algorithms have started to address this problem [8, 12, 13]. These works model declustering problem as a kind of graph. Their algorithm treats data items or data pages as a node of a graph and represents edge as distance between nodes or similarity between them.

3.2 Motivations

There are two factors in equation 1 that determine response time. The first factor, $N_{block}(q)$, is the number of data blocks touched by a query and it is mainly determined by a bucketing scheme. An additive error ϵ is the second factor and is determined by the bucket-to-disk allocation method.

Again, most previous work has focused on minimizing the additive error ϵ [4, 5, 10, 7, 6, 15, 2, 3]. Although these algorithms achieve good performance by applying an efficient disk allocation scheme, we can further improve performance by reducing $N_{block}(q)$ in equation 1.

When the data are non-uniformly distributed, $N_{block}(q)$ often increases considerably and the resultant performance degradation increases with data dimension. In addition to this weakness, regular grid-shaped partitioning schemes result in low storage utilization even with uniform data, especially in high dimensional space.

Thus, a good bucketing method as the first step of a declustering algorithm is necessary to improve declustering performance, especially with non-uniform data in high dimensional space. This is the main motivation of our work.

Let us discuss about $N_{block}(q)$ in more detail. Suppose that $s(q)$ is the selectivity of a query q. Then

$$N_{block}(q) = s(q) \cdot N_{total} \tag{2}$$

where N_{total} means the total number of blocks occupied by data objects. From this equation, we see that $N_{block}(q)$ is determined by both the total number of blocks, in other words the *storage utilization*, and *selectivity.* Since the selectivity $s(q)$ is given by a query, we will discuss only on storage utilization in the rest of this subsection. For high dimensional data, it is sufficient to perform binary partition along each dimension. Let Bf_{max} be the maximum disk blocking factor

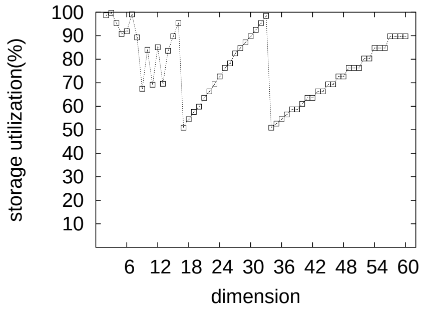


Fig. 1. Example of storage utilization for uniform data by simple tiling

and suppose we have decomposed a high dimensional data set with binary partition along p different dimensions thus resulting in 2^p tiles. If some of the tiles contain more than Bf_{max} objects we need to partition again along the $(p+1)^{th}$ dimension and this produces 2^p more tiles, i.e. a doubling of required disk space. These newly created tiles may cause a considerable drop of storage utilization for the following two reasons:

- case 1: uniform distribution
  25 percent of this newly allocated space is likely wasteful for uniformly distributed data. Assume the $(p+1)^{th}$ split results in each tile having Bf_{max} or fewer entries. If we assume before the last partition each tile had between Bf_{max} and $2 \cdot Bf_{max}$ entries, then after the split each tile has between $0.5 \cdot Bf_{max}$ and Bf_{max} entries, i.e. one fourth of the total space is being wasted. Furthermore, it is likely that before the split some of the tiles had less than Bf_{max} entries hence utilization would be even lower. Figure 1 shows this drop of storage utilization for uniform data. In this example, we assume 10^6 data items whose dimension is varied from 2 to 60 and page size is 4KByte. Although the performance of tiling uniform data depends on the number of data items and its dimension for the given disk page size, actual storage utilization is between 53% and 100%.
- case 2: non-uniform distribution
  While $p_{opt} = (\lceil \log_2 \frac{N}{Bf_{max}} \rceil)$ decompositions may be enough under uniform distribution for N objects, we need more decompositions than p_{opt} if the objects are not uniformly distributed. If we need $p'(> \lceil \log_2 \frac{N}{Bf_{max}} \rceil)$ decompositions, the ratio of the storage utilization for non-uniform distribution over uniform distribution is $2^{\lceil \log_2 \frac{N}{Bf_{max}} \rceil - p'}$ and results in a significant drop of storage utilization. Experimental results show that actual storage utilization for real high dimensional data by tiling algorithm is between 3.5% and 6.7%.

Besides the above shortcomings, a large query size may require all data blocks be accessed when using a tiling algorithm. If we store N uniformly distributed

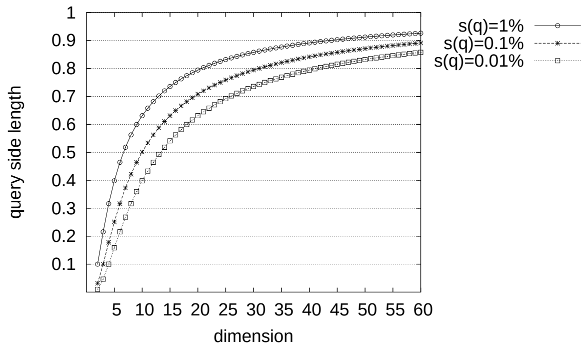


Fig. 2. Selectivity *vs* query side length

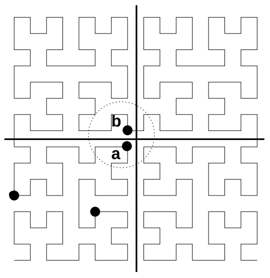
objects whose dimension is higher than $\log_2 \lceil \frac{N}{Bf_{max}} \rceil$, it is sufficient to perform binary partition along some of dimension. In this case, a query whose side length is greater than 0.5 may overlap all of the tiles.

The size of query region becomes extremely large as increases the dimensionality. For example, suppose that a range query, whose selectivity is 0.1%, is given in 10-dimensional space. Then the side length of this query should be more than 0.5. It means that this query touches every tile, and $N_{block}(q) = N_{total}$ for binary decomposition, which gives the worst performance of declustering. Figure 2 shows the side length of a query versus selectivity $s(q)$ as varies dimension.

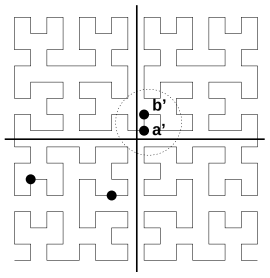
Due to above shortcomings, the number of data blocks touched by a query $N_{block}(q)$ unnecessarily grows for high dimensional data and this results in an overall low declustering performance with tiling scheme. To improve storage utilization and reduce the number of blocks touched by a query, we need a well designed bucketing algorithm producing a small number of data blocks and minimizing dead space within a bucket regardless of data distribution. We should employ a more elaborate bucketing method respecting the distribution of data rather than a simple decomposition of tiling scheme. In this paper, we propose a bucketing and declustering method for *high dimensional data* to increase storage utilization and reduce the number of blocks touched by a query region even for *non-uniform high dimensional data.*

4 Approximate Nearest Neighbor Graph by Hilbert Lists

In the previous section, we showed that one must take storage utilization and data distribution into account in order to improve the performance of a declustering algorithm. One approach is to adapt to the data distribution or spatial proximity between objects and we employed Delaunay Triangulation algorithm in [9]. However, it cannot be applied to *high-dimensional data* due to the algorithmic limitation of Delaunay triangulation. In this paper, we replace it with



(a) Before shifting objects



(b) After shifting objects

Fig. 3. An effect of shifting data objects

an *approximate* nearest neighbor graph based on multiple instances of the input points sorted by their position along a shifted Hilbert curve.

This structure, first proposed in [11], has been used to also encode information on spatial proximity, but is easy and efficient to compute nearest-neighbor graph for high dimensional data. As shown by Figure 3-(a), a space filling curve does not guarantee, in general, that a nearest neighbor is a neighbor in the Hilbert sorted list. The two objects a and b constituting the closest pair in Figure 3-(a), are not neighbors along the curve. We improve the information about spatial proximity if we create the second instance of the input where all of the data objects have been shifted by the vector $(1, 1)$, as illustrated in Figure 3-(b). After shifting by (1,1), a' and b' become neighbors and the detection of the closest pair can be guaranteed.

Formally, for d-dimensional point set $P = \{p_1, \ldots, p_n\}$, we define shifted Hilbert lists as follows:

Definition 4. ***Hilbert lists***
Let $\{H_j, j = 0, \ldots, d\}$ *be the* $d + 1$ *Hilbert ordered lists of points of* P *such that* $H_j[i] = \{p_i + v^{(j)}\}$, $i = 1, \ldots, n$ *and* $H_j[i] <_h H_j[i+1]$, *where* $v^{(j)} = (j/(d+1), \ldots, j/(d+1)) \in R^d$.

For more details as well as the mathematical justification for the use of shift vector $v^{(j)}$, see [11]. Once the lists have been built, an *approximate* nearest neighbor for a given point can be obtained by examining k predecessors and successors of the (shifted) given point in each of the Hilbert ordered lists. The more points we examine on either side of the point the more accurate our answer is, but even with $k = 1$ the difference between the approximation and the true answer is bounded by a constant factor that depends on d.

Based on Hilbert lists, we formally define an *approximate* nearest neighbor graph for the given point set $P = \{p_1, \ldots, p_n\}$ in $[0, 1)^d$ as follows:

Definition 5. ***Approximate Nearest Neighbor Graph*** G_H^k
$G_H^k(P) = (V, E)$, *where* $V = P$ *and for every pair of points* $p, q \in P$, $(p, q) \in E$ *iff* $p = H_j[i]$ *and* $q = H_j[h]$ *for some* $1 \le i \le n$, $0 \le j \le d$ *and* $|i - h| \le k$.

An *approximate* nearest neighbor graph $G_H^k(V, E)$ contains one vertex for each data point and an edge between every pair of points that are at most k positions apart in at least one of the $(d+1)$ Hilbert sorted lists. The proposed nearest neighbor graph has at most $k(d+1)n$ edges for d-dimensional n points when we examine k neighbor objects along each sorted list. So its maximum space requirement for building the graph grows linearly according to dimension of data. Experimental results show that the number of edges is around 70% of the expected upper-bound limit. The building time of a nearest neighbor graph is $O((d+1) \cdot n \log n)$ for n objects.

There are two factors that can affect the quality of the nearest neighbor graph. We are more likely to find an exact nearest neighbor object by shifting data several times and by examining more neighbors on every shifted list. However, experimental results show that the number of neighbors to be examined has a trivial effect on the performance of the proposed declustering algorithm while we can improve the performance by shifting data several times.

In the next section, we present our proposed declustering algorithm using an *approximate* nearest neighbor graph G_H^k.

5 DC-SH(Declustering Clusters by Shifted Hilbert lists): Algorithm Description

Once we detect spatial proximity information between data as described in the previous section, we can organize objects by the unit of bucket in a way that reduces dead space and achieves high storage utilization. Our proposed declustering algorithm consists of the following five steps:

– **step 1.** Build an *approximate* nearest neighbor graph G_H^k
– **step 2.** Find an initial bucket set
– **step 3.** Split overfilled buckets
– **step 4.** Assign a disk number
– **step 5.** Place buckets onto physical disk page

In the rest of this section, we present each step in more detail.

Step 1: Build an *approximate* nearest neighbor graph G_H^k
First, we build an *approximate* nearest neighbor graph for input points as explained in the previous section. Unless commented, we examined only one neighbor object and use the full set of d+1 shifted curves in order to find more accurate proximity information.

Step 2: Find an initial bucket set
After building an *approximate* nearest neighbor graph G_H^k, we obtain an initial bucket set by cutting edges whose distance is greater than a predefined threshold value as shown in Figure 4-(c). The threshold value determines total number of buckets and their size. A small value may produce many buckets and vice versa. Determining a proper threshold value may affect goodness of resultant bucket

set, but we leave this issue for future research. In this work, we iteratively applied a threshold value and selected the best among them.

Step 3: Split overfilled buckets
After obtaining an initial bucket set, there may be buckets whose size is too large to fit on one disk page. As queries are likely to be centralized around data areas, there is a good chance that most of the objects within an overfilled bucket can be accessed at the same time. Therefore, an overfilled bucket should be split into several disk blocks for parallel disk I/0. For this, we sort data objects within an overfilled bucket by Hilbert order and pack them by the unit of maximum disk blocking factor.

Step 4: Assign a disk number
After splitting overfilled buckets into several small buckets, all of the buckets can be stored on one physical disk page. At the allocation step, those buckets that are likely to be touched by a query should be distributed onto *different physical disks* for parallel access. To do this, we sort buckets by the Hilbert order of their center point of bounding cube and assign a disk number in round robin fashion. We may apply alternative disk allocation methods, for example max cut graph partitioning scheme used in [12] by assigning buckets as nodes of a graph as long as similarity measure between buckets is given.

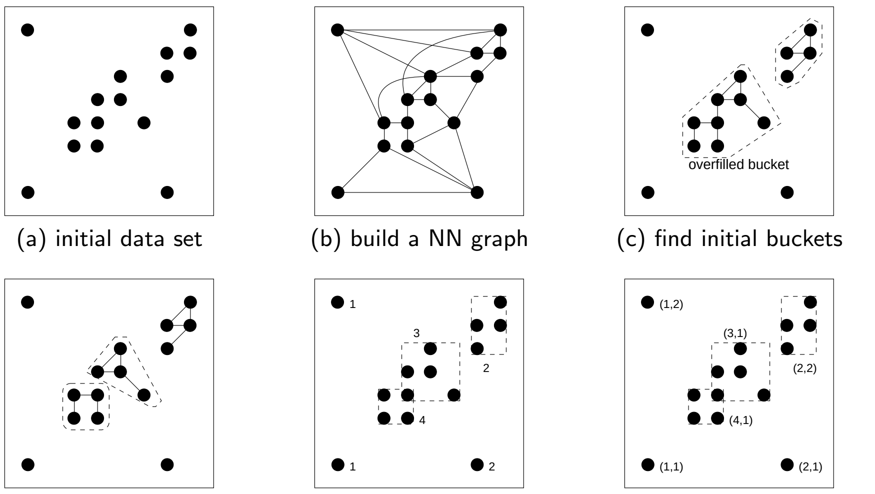
Step 5: Place buckets onto physical disk page
As we see in Figure 4-(e), there are buckets whose size is smaller than the physical disk page size. If we store only one small bucket onto a physical disk page, most of the rest space within that page is wasteful and it results in poor declustering performance. For high storage utilization, we sort buckets allocated the same disk number by Hilbert order of their center point of bounding cube and place consecutive buckets whose cumulative sum of size is within maximum disk blocking factor onto the same physical page within that disk.

Figure 4 illustrates five steps of our algorithm with small examples.

6 Experiments

We have conducted several experiments to show the performance of the proposed declustering algorithm.

We base our experimental studies on three real high-dimensional data sets: ColorMoments, CoocTexture and ColorHistogram, whose dimension are 9, 16 and 32 respectively. These data sets are extracted from feature vectors of Corel images, which are available at http://ics.uci.edu/databases/CorelFeatures. For the reason of simplicity, we normalized these data sets in $[0,1)^d$. For query sets we generated various sized range queries to show the effect of query size on declustering performance. We vary query size from a point query to a hypersquare region with side length of 0.9 on each dimension. Region queries fully specify the range on each dimension relevant to the data set use. The lower left



(a) initial data set (b) build a NN graph (c) find initial buckets

(d) split overfilled buckets (e) assign a disk number (f) place buckets onto page

Fig. 4. An example of the proposed declustering algorithm when Bf_{max} is 4(In this example, (x, y) in figure (f) means (*disk number, block number within a disk*))

hand corner of queries is uniformly distributed over the unit hyper-cube thus allowing parts of large queries to extend beyond the data space returning no data for that portion of the query. We assume a physical disk page size as 4 KByte.

6.1 Performance Test on Bucketing Scheme($N_{block}(q)$)

In section 3.2, we stated that declustering performance for non-uniform high dimensional data is mainly determined by $N_{block}(q)$ in equation 1, which means the number of blocks touched by a query q. For this reason, we investigate $N_{block}(q)$ by experiments with real data sets. Hence, we compare the two previous bucketing algorithms with the bucketing step of our DC-SH algorithm. We compare our method with 'tiling', which is a simple uniform tiling of the data space, and 'gridfile', whose bucketing is determined by the buckets resulting from the creation of a gridfile when inserting one data item at a time [14]. For this section the bucket-to-disk allocation is not relevant and hence not considered.

In figure 5 we plot the average number of data blocks retrieved versus query size for the three data sets. Query region side length is varied from 0.1 to 0.9.

We see that the number of data blocks retrieved by the tiling algorithm increases rapidly with query size. This result is due to low storage utilization of a tiling algorithm as illustrated in section 3. The Gridfile based organizing scheme, requires fewer data blocks than tiling because it adapts better to the data distribution and merges under-filled blocks with neighboring grid cells. However, the Gridfile method still retrieves significantly more data blocks than our method.

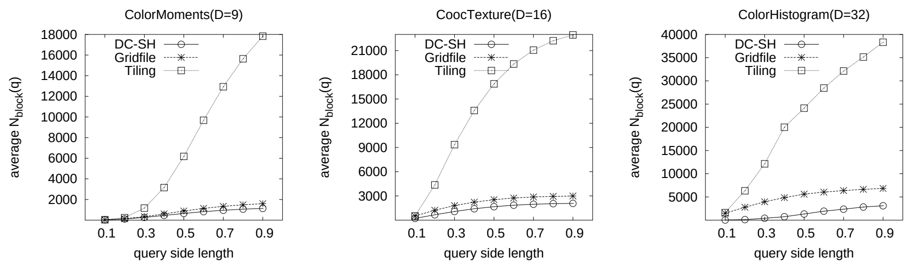


Fig. 5. Average number of data blocks touched by a query *vs* query size

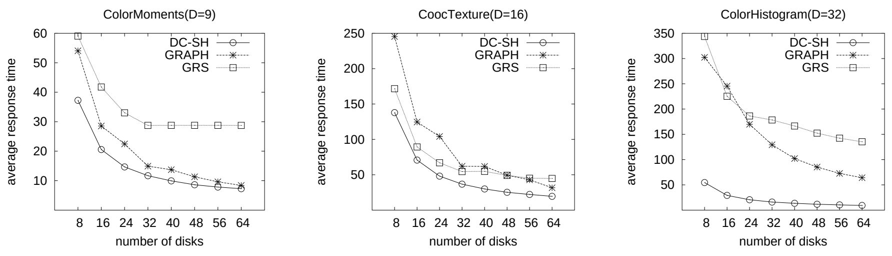


Fig. 6. Response time *vs* number of disks: query side length is 0.3

These results show that for the data sets used, our proposed approximate nearest neighbor graph method is better able to adapt to the data distribution than both of the previous methods.

6.2 Performance Test on Response Time($\max_{i=1}^{M} DA_i(q)$)

In this section we compare our new method with both GRAPH[12] and GRS[3]. The GRS algorithm first create buckets by using tiling whereas Graph using the gridfile bucketing method. They also differ in how they allocate the buckets to disk as specified in the original papers.

When storing data blocks onto a physical disk page, we pack them within that disk for high storage utilization. For this, we sort data blocks assigned the same disk id according to Hilbert value of center point of bounding cube and place consecutive blocks whose sum of data is within maximum disk blocking factor into the same page. We compare these algorithms to DC-SH.

We use our comparison metric response time as defined in definition 2. In Figure 6 we plot the response time versus the number of disks for each data set, and in Figure 7 we plot response time versus query size for each data set.

In all cases, the DC-SH algorithm has a smaller response time, ranging from 1.5 to 5 times improvement relative to the other algorithms. In general, the GRS algorithm does not perform as well as GRAPH, presumably due to the inferior bucket utilization of GRS relative to GRAPH. Note, again, we have improved the utilization of GRS and GRAPH by packing under-filled buckets into the

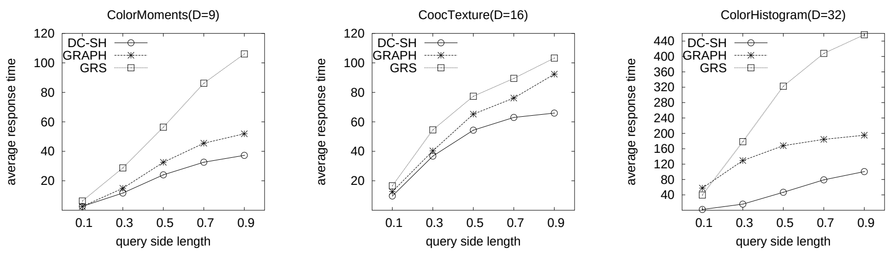


Fig. 7. Response time *vs* query side length: number of disks is 32

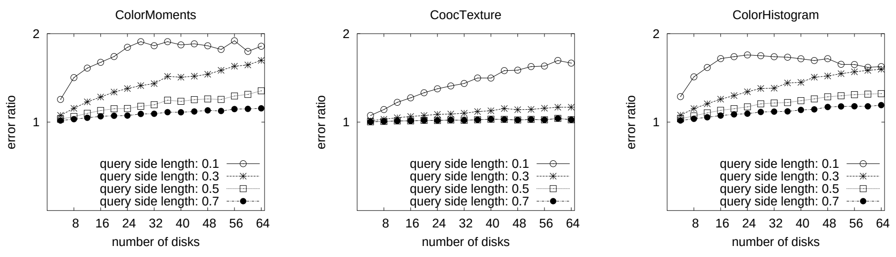


Fig. 8. Actual/optimal response time ratio *vs* query size

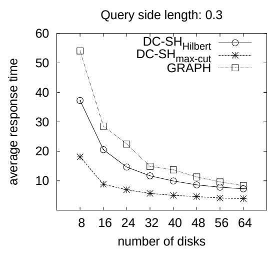
same physical page contingent on Hilbert ordering of the buckets as specified above.

6.3 Performance Study of our Bucket-To-Disk Allocation Scheme

In this section we isolate and quantify the quality of our bucket-to-disk allocation scheme. For a fixed bucketing, and hence a fixed $N_{block}(q)$, a perfect allocation scheme would result in the strict optimal response time as found in definition 3. Thus, to determine the quality of our allocation scheme we plot the ratio of the measured response time over the optimal response time in Figure 8. We plot results for each data set and for query sizes of 0.1, 0.3, 0.5, and 0.7. Note that the proposed allocation scheme results in poor performance as query size is small. As stated before, we allocate buckets to disks by Hilbert sorting buckets according to their bounding cube center point and allocating in a round robin fashion. This performance degradation is possible the result of the space filling curve failing to preserve spatial proximity between data blocks. However, our algorithm does gives nearly optimal performance as query size increases. In all cases, our method gives performance within 200% of strict optimal declustering algorithm in allocation aspect.

6.4 Improving DC-SH Through Better Allocation

The exhibited degradation for small queries presented in the previous section lead us to consider a different, and better, allocation method. We combined



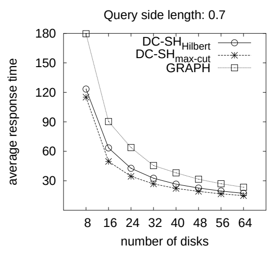


Fig. 9. Performance test on an alternative allocation scheme(Max-cut graph partitioning algorithm)

the DC-SH bucketing algorithm with the max cut graph partitioning bucket-to-disk allocation scheme proposed in [12]. We have done an initial experiment with 9 dimensional ColorMoments data to show the effect. In Figure 9 we plot average response time versus number of disks for two query sizes. In this figure, DC-SH$_{Hilbert}$ represents our proposed allocation scheme, DC-SH$_{max-cut}$ is our bucketing algorithm with max cut graph partitioning algorithm as an allocation scheme, and GRAPH is the original max cut graph partitioning algorithm[12], which was applied to data pages of a Gridfile[14]. As we see in this example, one can further improve declustering performance by combining our improved bucketing algorithm with a better bucket-to-disk allocation scheme.

7 Conclusions

In this paper, we proposed an efficient bucketing algorithm for *non-uniform* low and high dimensional data. Our method builds an *approximate* nearest neighbor graph by employing multiple shifted Hilbert curves. Our proposed nearest neighbor graph can be built at low time and space cost. After building the nearest neighbor graph, we place data objects onto a disk block in a way that preserve spatial proximity among them so that minimize dead space within a block and maximize storage utilization. Our method appears to adapt better to non-uniform and high dimensional data sets than previous methods.

Our experimental results show that, for the data sets considered, our proposed algorithm is highly stable to the number of disks and considerably outperforms both GRS, currently the best tiling based declustering method, and also the weighted similarity graph partitioning method, explicitly proposed for declustering non-uniform data.

The contributions of our study are summarized as follows:

– We explored the effects of a bucketing scheme on the performance of a declustering algorithm by analysis and experiments.
– We proposed a simple and efficient bucketing algorithm which for the data considered is adaptable to different data distributions regardless of dimension.

– Our proposed declustering algorithm, bucketing plus allocation, for the real skewed and high-dimensional data sets tested, results in a response time improvement of 1.5 to 5.0 times relative to the existing algorithms.
– Our proposed bucketing algorithm can be combined with previous bucket-to-disk allocation schemes, such as [12], to further enhance performance.

Acknowledgments

This work was supported by KOSEF grants number 01-2002-000-10014-0, by KOSEF grants number R05-2002-000-01288-0, by program for cultivating graduate students in regional strategic industry and by NSF grants number IRI-9610240 and ACR-9733658.

References

1. K. Abdel-Ghaffar and A. E. Abbadi. Optimal Allocation of Two-Dimensional Data. In *Proc. ICDT Conf*, pages 409–418, 1997.
2. M.J. Atallah and S. Prabhakar. (Almost) Optimal Parallel Block Access for Range Queries. In *Proc. PODS Conf*, pages 205–215, 2000.
3. R. Bhatia, R.K. Sinha, and C.-M. Chen. Declustering Using Golden Ratio Sequences. In *Proc. ICDE Conf*, pages 271–280, 2000.
4. C.M. Chen and C. T. Cheng. From Discrepancy to Declustering: Near optimal multidimensional declustering strategies for range queries. In *Proc. PODS Conf*, pages 29–38, 2002.
5. H.C Du and J.S. Sobolewski. Disk Allocation for Cartisian Files on Multiple-Disk Systems. *ACM Trans. Database Systems*, 7(1):82–102, 1982.
6. C. Faloutsos and P. Bhagwat. Declustering Using Fractals. In *Proc. Parallel and Distributed Information Systems Conf*, pages 18–25, 1993.
7. C. Faloutsos and D. Metaxas. Disk Allocation Methods Using Error Correcting Codes. *IEEE Trans on Computers*, 40(8):907–914, 1991.
8. M.T. Fang, R.C.T. Lee, and C.C. Chang. The Idea of De-Clustering and Its applications. In *Proc. VLDB Conf*, pages 181–188, 1986.
9. H.C. Kim and K.J. Li. Declustering Spatial Objects by Clustering for Parallel Disks. In *Proc. DEXA Conf*, pages 450–459, 2001.
10. M.H. Kim and S. Pramanik. Optimal File Distribution For Partial Match Retrieval. In *Proc. SIGMOD Conf*, pages 173–182, 1988.
11. S. Liao, M.A. Lopez, and S.T. Leutenegger. High Dimensional Similarity Search With Space Filling Curves. In *Proc. ICDE Conf*, pages 615–622, 2001.
12. D.R. Liu and S. Shekhar. Partitioning Similarity Graphs: A Framework for Declustering Problems. *International Journal Information System*, 21(6):475–496, 1996.
13. D.R. Liu and M.Y. Wu. A Hypergraph Based Approach to Declustering Problems. *Distributed and Parallel Databases*, 10(3):269–288, 2001.
14. J. Nievergelt, H. Hinteberger, and K.D. Sevcik. The Grid file: An Adaptable, Symmetric Multi-Key File Structure. *ACM Trans. on Database Systems*, 9(1):38–71, 1984.
15. S. Prabhakar, K. Abdel-Ghaffar, and A. El Abbadi. Cyclic Allocation of Two-Dimensional Data. In *Proc. ICDE Conf*, pages 94–101, 1998.