Proc. Int. Conf. on Management of Data, 1997 ACM SIGMOD, Tuscon, AZ, 1997
SIGMOD Best Paper Award

Fast Parallel Similarity Search in Multimedia Databases

Stefan Berchtold
University of Munich
Germany
berchtol@informatik.uni-muenchen.de

Christian Böhm
University of Munich
Germany
boehm@informatik.uni-muenchen.de

Bernhard Braunmüller
University of Munich
Germany
braunmue@informatik.uni-muenchen.de

Daniel A. Keim
University of Munich
Germany
keim@informatik.uni-muenchen.de

Hans-Peter Kriegel
University of Munich
Germany
kriegel@informatik.uni-muenchen.de

Abstract

Most similarity search techniques map the data objects into some high-dimensional feature space. The similarity search then corresponds to a nearest-neighbor search in the feature space which is computationally very intensive. In this paper, we present a new parallel method for fast nearest-neighbor search in high-dimensional feature spaces. The core problem of designing a parallel nearest-neighbor algorithm is to find an adequate distribution of the data onto the disks. Unfortunately, the known declustering methods do not perform well for high-dimensional nearest-neighbor search. In contrast, our method has been optimized based on the special properties of high-dimensional spaces and therefore provides a near-optimal distribution of the data items among the disks. The basic idea of our data declustering technique is to assign the buckets corresponding to different quadrants of the data space to different disks. We show that our technique - in contrast to other declustering methods - guarantees that all buckets corresponding to neighboring quadrants are assigned to different disks. We evaluate our method using large amounts of real data (up to 40 MBytes) and compare it with the best known data declustering method, the Hilbert curve. Our experiments show that our method provides an almost linear speed-up and a constant scale-up. Additionally, it outperforms the Hilbert approach by a factor of up to 5.

1 Introduction

The most important query type in multimedia databases are similarity queries. A promising and widely used approach for fast similarity searching in multimedia databases is to map the multimedia objects into points in some d-dimensional feature space. In image databases, for example, the images are mapped into complex feature vectors consisting of color historgrams, shape descriptors, etc. and queries are processed against a database of those feature vectors [Fal 94]. Similarity of two images is defined as the proximity of their feature vectors in feature space and the similarity query corresponds to a nearest-neighbor query. Feature-based approaches are taken in many other areas including CAD [MG 93], molecular biology (for the docking of molecules) [SBK 92], string matching [AGMM 90], etc. Examples of feature vectors are color histograms [SH 94], shape descriptors [Jag 91, MG 95], Fourier vectors [WW 80], text descriptors [Kuk 92], etc. In many of the mentioned applications, the databases are very large and consist of millions of data objects with several tens to a few hundreds of dimensions. For querying these databases, it is essential to use appropriate indexing techniques which provide an efficient access to high-dimensional data. Data structures which have been specifically developed for indexing high-dimensional data include the TV-tree [LJF 94] and the X-tree [BKK 96]. Experiments with the TV-tree and the X-tree show significant performance improvements for point queries, but unfortunately only limited performance improvements for nearest-neighbor queries.

In this paper, we therefore propose a new parallel method for fast nearest-neighbor search in high-dimensional feature spaces. In section 2, we first define the nearest-neighbor search problem and briefly review the relevant literature. The core problem of designing a fast parallel nearest-neighbor algorithm is to find an adequate declustering algorithm which distributes the data onto the disks such that the data which has to be read in executing a query are distributed as equally as possible among the disks. Unfortunately, the known declustering methods such as the Disc Modulo [DS 82], FX [KP 88], and Hilbert [FB 93] have been designed to support different query types (range queries and partial match queries). Therefore, as we show in section 3, those techniques do not allow an optimal declustering for nearest-neighbor queries in high-dimensional spaces. In contrast, our new declustering method has been optimized based on the special properties of nearest-neighbor search in high-dimensional spaces (cf. subsection 3.1) and therefore provides a near-optimal distribution of the data items among the disks (cf. section 3.2). The basic idea of our data declustering technique is to assign the buckets which correspond to different quadrants of the data space to different disks. We show that this problem is equivalent to a graph coloring problem (cf. subsection 4.1). We then develop a simple but efficient algorithm which solves the graph coloring problem and show that our algorithm - in contrast to other declustering methods - guarantees that all buckets corresponding to neighboring quadrants are assigned to different disks (cf. subsec-

tion 4.2). A surprising result is that the number of disks necessary for the near-optimal declustering is a linearly bound staircase function which is optimal up to rounding (cf. subsection 4.2). We then provide extensions of our algorithm allowing for an arbitrary number of disks and highly clustered data distributions (cf. subsection 4.3). Finally, in section 5, we evaluate our method using large amounts of uniformly distributed and real data (up to 40 MBytes) with varying dimension, and compare it with the best known data declustering method, the Hilbert curve. Our experiments show that our method provides a near-linear speed-up and a constant scale-up, and it outperforms the Hilbert approach by a factor of up to 5.

2 Nearest-Neighbor Search in High-Dimensional Spaces

Nearest-neighbor search on high-dimensional feature vectors may be defined as follows:

Definition 1: *(nearest-neighbor search)*
Given a data set *DS* containing N *d*-dimensional points $v_0 \ldots v_{N-1}$, find the data point *NN* from the data set which is closer to the given query point *q* than any other point in the data set. More formally:

$$NN(q) = \{\bar{e} \in DS | \forall e \in DS: \|\bar{e} - q\| \le \|e - q\|\}.$$

Analogously, we can define a k-nearest-neighbor query as a query for the k-nearest-neighbors. For simplification, we assume without loss of generality that the extension of the data space is $[0..1]^d$. In the literature, various algorithms have been proposed to search a spatial database for points, which are closer to a given query point than any other point in the database. These algorithms for nearest-neighbor search may be divided into two major groups: partitioning algorithms and graph-based algorithms. Partitioning algorithms partition the data space (or the actual data set) recursively and store information about the partitions in the nodes. Graph-based algorithms precalculate some nearest-neighbors of points, store the distances in a graph, and use the precalculated information for a more efficient search. Examples for such algorithms are the RNG* algorithm [Ary 95] and algorithms using Voronoi diagrams [PS 85].

A rather simple partitioning algorithm is the bucketing algorithm of Welch [Wel 71]. The algorithm divides the data space into identical cells and stores the data items inside a cell in a list which is attached to the cell. During nearest-neighbor search the cells are visited in order of their distance to the query point. The search terminates if the nearest point which has been determined so far is nearer than any cell not visited yet. Unfortunately, the algorithm is not efficient for high-dimensional data. A more practical approach is the k-d-tree algorithm of Friedmann, Bentley and Finkel [FBF 77]. In contrast to Welch's algorithm, the order in which the k-d-algorithm visits the partitions of the data space is determined by the structure of the k-d-tree. Ramasubramanian and Paliwal [RP 92] propose an improvement of the algorithm by optimizing the structure of the k-d-tree.

Roussopoulos et.al. [RKV 95] propose a different approach for nearest-neighbor search based on the R*-tree [BKSS 90]. The algorithm traverses the R*-tree and stores for every visited partition a list of subpartitions ordered by their *minmaxdist*. The *minmaxdist* of a partition is the maximal possible distance from the query point to the nearest data point inside the partition. If a point is found having a distance smaller than the nearest point determined so far, all partition lists may be pruned because all nodes with a larger *minmaxdist* cannot contain the nearest-neighbor.

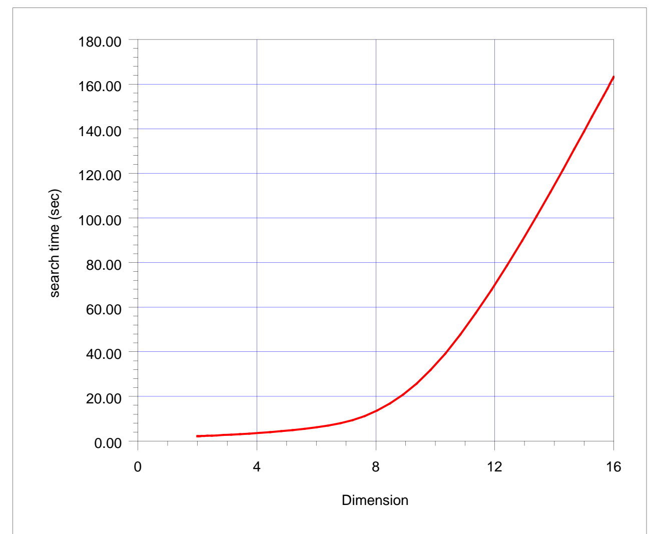


Figure 1: Nearest-Neighbor Queries in High Dimensions (X-tree)

In [HS 95], Hjaltason and Samet propose an algorithm using PMR-Quadtrees. In contrast to the algorithm of Roussopoulos et.al., partitions are visited ordered by their *mindist*. The *mindist* of a partition is the minimal distance from the query point to any point inside the partition P. The algorithmic principle of the method of Hjaltason and Samet can be applied to any hierarchical index structure which uses recursive partitioning.

In [BKK 96], we applied the algorithm of Roussopoulos et.al. [RKV 95] to the X-tree, an index structure for high-dimensional data. The X-tree is an R^*-tree-based index structure which avoids a degeneration of the directory in high-dimensions using a special split algorithm and variable sized directory nodes. In higher dimensions, the X-tree outperforms the R-tree and other index structures

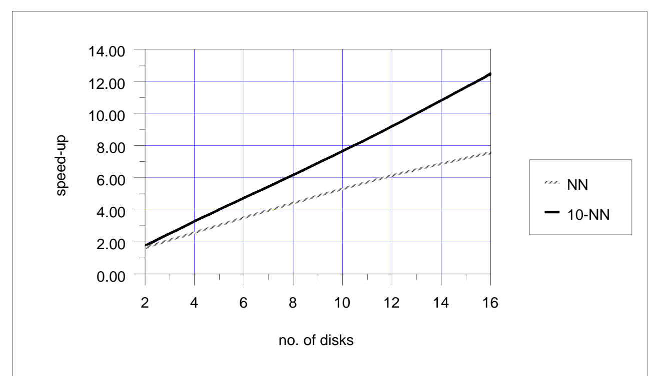


Figure 2: Speed-Up of Parallel Nearest-Neighbor Search

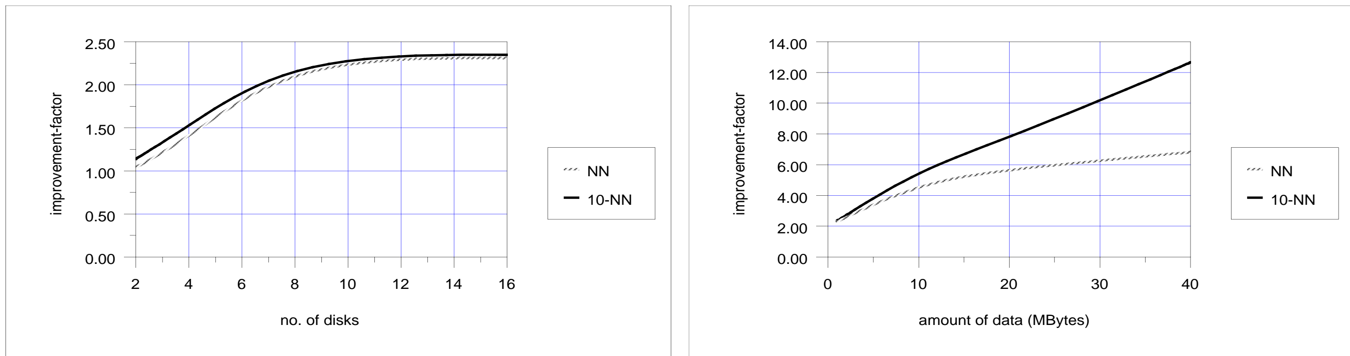


Figure 3: Improvement of Hilbert over Round Robin

for point queries by orders of magnitude. However, even the performance of the X-tree degenerates for nearest-neighbor queries in high-dimensions. Figure 1 shows the total search time of a 10-nearest-neighbor query on an X-tree containing 100 MB of uniformly distributed data. As further experiments showed, this observation also holds for real data.

Friedmann et.al showed in [FBF 77] that the nearest-neighbor search in high-dimensional data spaces is an inherently computationally intensive problem. In a recent paper, we refined the model of Friedmann et al. for high-dimensional data spaces [BBKK 97] and confirmed the inherent high complexity of high-dimensional nearest-neighbor search. We believe that the use of parallelism is crucial for improving the performance of nearest-neighbor queries in high-dimensional space.

3 Parallel Nearest-Neighbor Search

The core problem of parallel nearest-neighbor search is the distribution of data among the available disks which is usually called the *declustering problem*. In the following, we denote the number of disks by n and the i-th disk by d_i.

The simplest method for distributing data is *round robin* where each disk d_i gets the data items $\{v_j | j \ mod \ n = i\}$. Figure 2 shows the speed-up of a parallel nearest-neighbor search (referred to as NN in all subsequent figures) and a parallel search for 10 nearest neighbors (10-NN) using the round robin data distribution on 1 MByte of uniformly distributed 15-dimensional data and uniformly distributed query points. In our experiment, the speed-up increases nearly linear with the number of disks. This simple experiment shows that nearest-neighbor search can be improved considerably by using parallelism.

More complex algorithms solving the declustering problem have been proposed in the literature. Using an equi-distant grid, all these algorithms divide the data space into equi-sized buckets b which may be characterized by the position of the bucket in the d-dimensional grid $(c_0, c_1, ..., c_{d-1})$. A bucket characterized by $b[c_0, c_1, ..., c_{d-1}]$ describes a partition of the data space having the shape of a hyperrectangle and containing a certain number of data objects.[1] A declustering algorithm *DA* can then be described as a mapping from the bucket characterization to a disk number.

A rather simple declustering algorithm is the disk modulo method of Du and Sobolewski [DS 82]. The *disk modulo* method uses the mapping

$$DM(c_0, c_1, \ldots, c_{d-1}) = \left(\sum_{l=0}^{d-1} c_l\right) mod \ n \ .$$

Kim and Pramanik improved the disk modulo method and presented in KP 88] the *FX* distribution method which has been specifically designed to support partial match queries. Kim and Pramanik distribute the buckets using a bitwise XOR operation. Slightly simplified, the FX method can be defined as the mapping

$$FX(c_0, c_1, \ldots, c_{d-1}) = \underset{l=0}{\overset{d-1}{\mathrm{XOR}}} \ c_l \ mod \ n \ .$$

In [FB 93], Faloutsos and Bhagwat apply the Hilbert curve to the declustering problem. The Hilbert curve maps a d-dimensional space to a 1-dimensional space. For mapping a point in the data space to a disk, the Hilbert value of the point is determined and the data point is stored on the disk corresponding to the Hilbert value. More formally, the i-th disk gets the bucket

$$HI(c_0, c_1, \ldots, c_{d-1}) = Hilbert(c_0, c_1, \ldots, c_{d-1}) \ mod \ n$$

Since the Hilbert curve preserves spatial neighborhood as far as possible, the mapping provides a good declustering. Faloutsos and Bhagwat compared their method to various methods such as the disk modulo and the FX technique. The experimental results reported in [FB 93] show that the Hilbert approach clearly outperforms the other methods for range queries in two-dimensional spaces. However, to our knowledge, none of the methods has been designed or tested for high-dimensional feature spaces and for nearest-neighbor queries. Therefore, in our first experiments we used the most promising technique, the Hilbert curve. The experiments show that the Hilbert approach provides a much better declustering for nearest-neighbor queries in high-dimensional spaces than the round robin method. Figure 3 depicts the improvement of the Hilbert approach over the round robin declustering. Note that the improvement increases, both, with an increasing number of disks, and

1. Our notation is similar to the notation of Faloutsos and Bhagwat used in [FB 93].

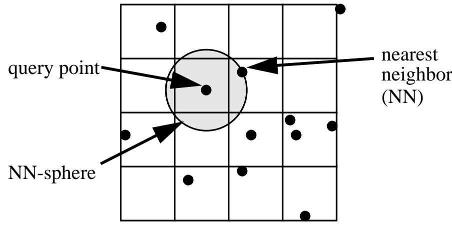


Figure 4: NN-Sphere

with an increasing amount of data. In section 3.2, however, we show that all the methods described in this section including the Hilbert method do not provide an adequate data distribution for nearest-neighbor queries in high-dimensional spaces.

3.1 Effects in High-Dimensional Spaces

To find a good declustering algorithm, we have to consider several special effects occurring in high-dimensional spaces and their consequences for nearest-neighbor queries. In this section, we therefore analyze nearest-neighbor query processing in high-dimensional space and derive the requirements for an optimal declustering. For the following considerations, we assume uniformly distributed data and uniformly distributed query points.

During nearest-neighbor search, any NN-algorithm has to examine all data pages intersecting the so-called *NN-sphere* (cf. Figure 4). The NN-sphere is a d-dimensional hypersphere having the query point as the centre and a radius equal to the distance from the query point to the nearest-neighbor. Unfortunately, according to [BBKK 97], the radius of the NN-sphere increases rapidly with increasing dimension of the data space[1], and therefore, the number of partitions any sequential algorithm has to access also increases rapidly.

Another important property (cf. section 3.2) of high-dimensional data spaces is that most data items are located near the (d-1)-dimensional surface of the data space. An example clarifies this effect: Figure 5 (right partition) depicts the probability that a point in a d-dimensional space is located near the surface where "near" means that the distance of the point to the surface is less than 0.1:

$$p_{surface}(d) = 1.0 - (1.0 - (2 \cdot 0.1))^d .$$

As the figure shows, the probability grows rapidly with increasing dimension and reaches more than 97% for a dimensionality of 16.

Declustering algorithms such as the disk modulo method or the FX method assume a partitioning of the data space into buckets. In the 2-dimensional case, the data space is partitioned many times in each direction, for example to obtain 10,000 buckets, the space is divided 100 times in x-direction and 100 times in y-direction. If we consider a 16-dimensional space, a complete *binary* partitioning of the space would already produce 65,536 partitions. Thus, in high-dimensional spaces it is not possible to consider more than a binary partitioning. In addition, the usage of a finer partitioning would produce many underfilled buckets. For the following considerations, we therefore assume each dimension of the space to be split exactly once. Thus, from our point of view, the buckets are the quadrants of the data space. The bucket coordinates $(c_0, c_1, ..., c_{d-1})$ can then be seen as binary values and $(c_0, ..., c_{d-1})$ may be represented as a bitstring. Note that $(c_0, c_1, ..., c_{d-1})$ with $c_i \in \{0, 1\}$ corresponds to the binary representation of the corresponding grid partition stored in the bucket. We use this property to define an unambiguous bucket number, *bn*, which will be the basis for our algorithm presented in section 4.2.

1. The increase of the radius depends on the bucket size, the number of data items and the dimension. However, the dimension is the most important parameter.

Definition 2: *(bucket number)*
Given a bucket b characterized by $(c_0, c_1, ..., c_{d-1})$ with $c_i \in \{0, 1\}$, $0 \le i < d$. The bucket number *bn* is defined as

$$bn(b) = \sum_{i=0}^{d-1} c_i \cdot 2^i .$$

3.2 Declustering for Nearest-Neighbor Search

The goal of each declustering algorithm is to distribute the buckets which are involved in an *arbitrary* search to different disks. For the parallel nearest-neighbor search, this means that the partitions intersecting the NN-sphere should be distributed to different disks. If all disks are equally involved in the search, the speed-up is maximal.

Figure 6 illustrates the effects of an increasing NN-sphere using a two-dimensional example. Let us assume that the query point is located in the upper left corner of the data space. If the radius of the NN-sphere is less than 0.5, only the bucket containing the query point has to be accessed (the upper left bucket in Figure 6). Thus, only the disk which stores the bucket is involved in the search process and any declustering technique provides the same result. If the radius of the NN-sphere is 0.6, however, two other buckets are involved in the search (the lower left and the upper right bucket in Figure 6). Obviously, for obtaining a good speed-up, the three buckets involved in the search should be distributed to different disks. Note that in high-dimensional space, this observation holds for most queries even if the query point is not located exactly in a

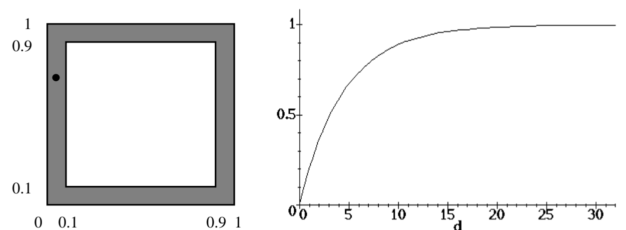


Figure 5: Data Points on the (d-1)-Dimensional Surface of the Data Space

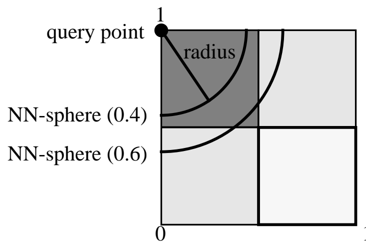


Figure 6: Partitions Affected by the Nearest-Neighbor Search when Increasing the NN-sphere

corner of the data space but on a lower-dimensional surface, e.g. a two-dimensional surface (cf. Figure 5).

Generalizing this result to the *d*-dimensional case, a good declustering technique must assure that adjacent buckets are assigned to different disks. From the example in Figure 6, we can derive that not only directly adjacent buckets (such as the upper left and upper right bucket) have to be considered, but also indirectly neighboring buckets (such as the lower left and the upper right bucket). This can be formalized as follows**:**

Definition 3: *(direct and indirect neighbors)*
Given two buckets *b* and *c*.
b and *c* are *direct neighbors*, $b \sim_d c$, if and only if

$$\exists i: \begin{cases} b_k \neq c_k, & \text{iff } k = i \\ b_k = c_k, & \text{otherwise} \end{cases}, \text{ where } (0 \le i, k \le (d-1)).$$

b and c are *indirect neighbors*, $b \sim_i c$, if and only if

$$\exists (i,j), i \neq j: \begin{cases} b_k \neq c_k, & \text{iff } k = i \text{ or } k = j \\ b_k = c_k, & \text{otherwise} \end{cases}$$

$$\text{where } (0 \le i, j, k \le (d-1)).$$

Intuitively, two buckets *b* and *c* are direct neighbors, if their coordinates differ in one dimension, and the remaining *(d-1)* coordinates are identical. Note that this definition of neighborhood implies that applying the binary exclusive-or-function (XOR) to direct neighboring buckets *b* and *c* results in a bitstring of the form 0^*10^*. Analogously, applying the XOR function to indirectly neighboring buckets results in a bitstring of the form $0^*10^*10^*$. Note further that considering more than one level of indirection would produce a huge amount of neighboring buckets. An algorithm considering *i* levels of indirection in *d*-dimensional space would have to assure that

$$1 + \sum_{k=1}^{i} \binom{d}{k}$$

buckets are equally distributed over the disks. For two levels of indirection in a 16-dimensional space, for example, the number of buckets would be

$$1 + \sum_{k=1}^{2} \binom{16}{k} = 1 + 16 + 120 = 137.$$

We therefore restriced our definition of neighboring buckets to direct and indirect neighbors. Another important observation is the following. From the point of view of the surface of the data space, direct neighbors share a common 1-dimensional surface of the data space, whereas indirect neighbors share a 2-dimensional surface.

Using the above definitions, we can define a *near-optimal* declustering as a declustering which guarantees that all direct and indirect neighboring buckets are assigned to different disks. We use the term *near-optimal* because an optimal declustering technique would have to guarantee that arbitrary queries are handled by different disks. This however would require to consider arbitrary neighbors – not only direct and indirect neighbors.

Definition 4: *(near-optimal declustering)*
A declustering algorithm DA is near-optimal, if and only if for any two buckets b and c and for any dimension d of the data space:

$$b \sim_d c \rightarrow DA(b) \neq DA(c) \quad \text{and} \quad b \sim_i c \rightarrow DA(b) \neq DA(c).$$

As we show in our experimental evaluation, our definition of a near-optimal declustering algorithm is close to the optimum, i.e. it provides a high speed-up and a nearly constant scale-up. The following lemma shows that the known declustering techniques do not provide a near-optimal declustering.

Lemma 1:
The disk modulo, the FX, and the Hilbert declustering techniques are not near-optimal declustering algorithms.

The validity of lemma 1 can be shown by a simple three-dimensional counter-example (cf. Figure 7). The numbers in the corner of each cube denote the disk number the corresponding bucket is assigned to. The thick line in each cube shows indirect (c.f. 6) neighbors which are assigned to the same disk. The right most portion of Figure 7 demonstrates the existence of a near-optimal declustering. Note that there exist more than one colliding pair of indirect neighbors, which however are not shown in Figure 7.

4 Near-optimal Declustering for Nearest-Neighbor Queries

In this section, we present a new declustering technique which is near-optimal according to definition 4. The basic idea of our technique is to transform the declustering problem into an equivalent graph-coloring problem so that buckets correspond to vertices, neighborhood-relations to edges, and disks to colors. We then propose a simple but efficient algorithm for solving the graph-coloring problem. To show that our declustering technique is near-optimal, we prove that our graph-based algorithm assigns different colors to connected vertices in the graph. The number of colors (disks) required by our algorithm is a linearly bounded staircase function which is optimal up to rounding. Furthermore, we describe some extensions of our method, allowing the method to be used in a wide range of real applications, i.e. on data with various data distributions and dimensionalities, and an arbitrary number of disks.

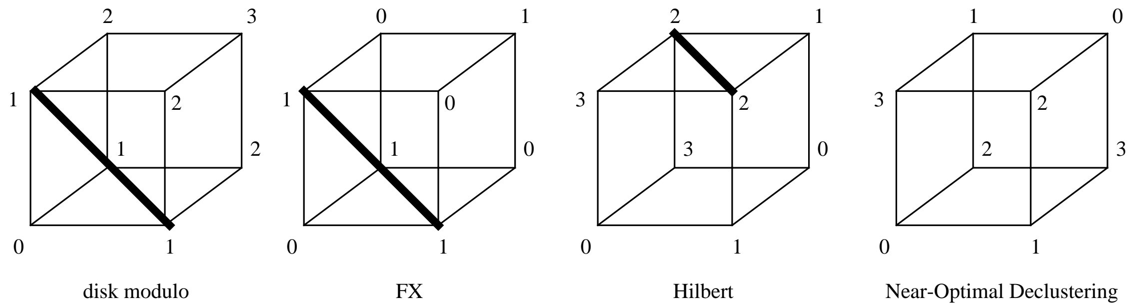


Figure 7: Disk Modulo, FX and Hilbert are not Near-Optimal Declustering Techniques

4.1 Declustering as a Graph Coloring Problem

In order to transform the declustering problem into a graph coloring problem, we first define the disk assignment graph. The disk assignment graph is an undirected graph in which buckets correspond to vertices and neighborhood relationships between buckets to edges.

Definition 5: *(disk assignment graph)*:
The disk assignment graph G_d = (V, E) for a d-dimensional data space is an undirected graph where V={0, ..., 2^d-1} is the set of bucket numbers and E = {(b, c) | b, c ∈ V and b $\sim_d$ c or b $\sim_i$ c} is the set of direct and indirect neighborhood relationships.

Since our definition of the edges includes both direct and indirect neighbors, it is obvious that an algorithm which assigns different colors to connected vertices, provides a near-optimal declustering. Thus, we reduce the declustering problem to an equivalent graph coloring problem.

Figure 8 shows the disk assignment graph G_3 for a three-dimensional data space. In the left partition of the figure, the data space with the corresponding buckets is depicted. In the middle of the figure, the corresponding disk assignment graph is shown with thick lines denoting direct neighbors and thin lines denoting indirect neighbors. The disk assignment graph G_3 may be colored using 4 colors. Transforming the graph back, we get a near-optimal declustering of the space (cf. right part of figure 8). Obviously, a lower bound of d+1 colors is required to color a graph G_d because each vertex has d directly neighboring vertices and at least all directly neighboring vertices must have pairwise different colors. It is a well-known fact from graph theory [Big 89] that the graph coloring problem for arbitrary graphs (including the determination of the required number of colors) is a hard problem which has not been solved in polynomial time yet and therefore, it is believed that the problem is NP-complete. Nevertheless, we are able to exploit some regularities in our graph to develop a simple but efficient coloring algorithm.

4.2 The Vertex Coloring Algorithm

In this section, we introduce an algorithm to determine the vertex color (i.e., the disk number) for a given vertex (i.e., bucket number). After describing the algorithm, we prove that our algorithm assigns different colors to connected vertices and we provide a formula for the number of colors required by our algorithm.

The basic idea of our algorithm is to determine for a vertex b all positions in its binary representation which equal to 1. Incrementing these positions by 1, each position can again be interpreted as a binary number, and the positions are combined by the XOR function. Interpreting the resulting binary number as a decimal number, we finally obtain the corresponding vertex color [1].

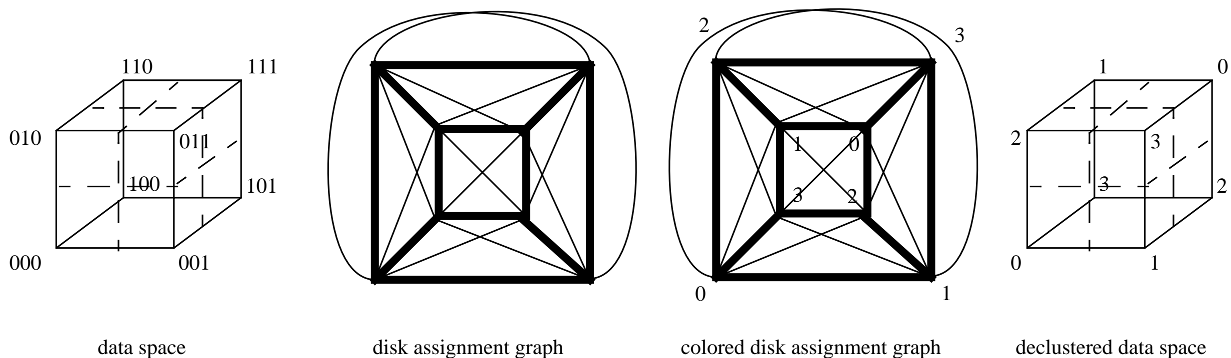


Figure 8: disk assignment graph

```
function col (c: integer): integer
var i: integer ;
begin
      col := 0;
      for i := 0 to dimension-1 do
              if bit_set (i, c) then
                      col := col XOR (i+1);
              endif
      endfor
end
```

Figure 9: vertex coloring algorithm

Let us, for example, assume a given vertex $c = 5 = 101_2$ in a disk assignment graph G_3 (representing a 3-dimensional data space). As the bits c_0 and c_2 are set, the positions to be considered are 0 and 2. Incrementing the positions by one, we obtain (2+1)=3 and (0+1)=1. We then combine the binary representations 011_2 (=3) and 001_2 (=1) by the XOR function and obtain 011_2 XOR $001_2 = 010_2$. Interpreting this binary number as a decimal number, we get $010_2 = 2_{10}$. The color of vertex 5 is therefore 2. Figure 9 shows the vertex coloring algorithm in algorithmic pseudocode. It is obvious from the algorithm that the color of an arbitrary vertex may be determined in $O(d)$ time. The following formal definition provides a very compact form of the algorithm.

Definition 6: *(vertex coloring function)*:
Given a vertex number c in binary representation $c_{d-1}, ..., c_0$. The corresponding vertex color is

$$\text{col}(c) = \left(\underset{i=0}{\overset{d-1}{\text{XOR}}} \left(\begin{cases} i+1 & \text{if } c_i = 1 \\ 0 & \text{otherwise} \end{cases} \right) \right)_{10}$$

In the following, we show that our vertex coloring function *col* guarantees that vertices which are connected in the disk assignment graph are colored differently. Our proof is divided into three lemmata. First, we prove the distributivity of col and XOR. Then, we prove that vertices which are connected by an edge representing direct neighborhood are colored differently, and finally we prove the same for edges representing indirect neighborhood.

Lemma 2 (*distributivity of col and XOR*):

$$\forall b\ \forall c: \ col(b) \text{ XOR } col(c) = col(b \text{ XOR } c)$$

Proof: cf. appendix.

Using Lemma 2, we now prove that vertices which are connected by an edge representing direct neighborhood are colored differently. We make use of some algebraic laws which are valid for the XOR function, especially the associativity, commutativity and the following equivalences:

$$a \text{ XOR } b = 0 \Leftrightarrow a = b$$
$$a \text{ XOR } b = a \Leftrightarrow b = 0$$

Lemma 3: *(coloring of direct neighbors)*
Two vertices b and c which are connected by an edge representing a direct neighborhood are colored differently.

Proof:
As the vertices b and c are differing in exactly one bit, say bit j, b XOR c is of the form 0^*10^* with only bit j set (cf. definition 3). Therefore, using the definition of the vertex coloring function, we may derive that $col(b \text{ XOR } c) = j + 1 \neq 0$. Thus,

$$\begin{aligned} col(c) &= \\ &= col(b \text{ XOR } b \text{ XOR } c) = \\ &\qquad \text{(since } b \text{ XOR } b = 0 \text{ and } 0 \text{ XOR } c = c\text{)} \\ &= col(b) \text{ XOR } col(b \text{ XOR } c) = \\ &\qquad \text{(according to lemma 2)} \\ &= col(b) \text{ XOR } (j+1) = \\ &\qquad \text{(since only bit } j \text{ is set in } b \text{ XOR } c\text{)} \\ &\neq col(b) \\ &\qquad \text{(since otherwise, } (j+1) \text{ would have to be 0)} \end{aligned}$$ ❑

Lemma 4: *(coloring of indirect neighbors)*
Two vertices b and c which are connected by an edge representing an indirect neighborhood are colored differently.

Proof:
According to definition 3, b XOR c has the form $0^*10^*10^*$ with a bit set at the positions i and j, $i \neq j$ and $col(b \text{ XOR } c) = (i+1) \text{ XOR } (j+1)$, which cannot be zero, since $i+1 \neq j+1$. Thus,

$$\begin{aligned} col(c) &= col(b) \text{ XOR } col(b \text{ XOR } c) \\ &= col(b) \text{ XOR } (i+1) \text{ XOR } (j+1) \neq col(b) \end{aligned}$$ ❑

Lemma 5: *(col provides a near-optimal declustering)*
The vertex coloring function *col* for the declustering of a d-dimensional data space is near-optimal.

Proof:
According to definition 4, a declustering algorithm DA is near-optimal, if and only if

$$b \sim_d c \rightarrow DA(b) \neq DA(c)$$

and

$$b \sim_i c \rightarrow DA(b) \neq DA(c) .$$

We proved that our algorithm *col* assigns different colors to connected vertices in the disk assignment graph. As vertices are connected if the corresponding buckets are direct or indirect neighbors, the function *col* guarantees that neighboring buckets are assigned to different disks. ❑

So far, we have shown that our algorithm computing the vertex color assigns pairwise different colors to all neighbors of any given vertex and therefore provides a near-optimal declustering.

Now, we want to determine how many colors are necessary for a d-dimensional data space. It seems to be obvious, that any vertex coloring algorithm solving the disk assignment problem must use at least d+1 colors, since each vertex and its d direct neighbors have to be colored differently. This means that no algorithm exists which

1. We will motivate later why we have to increment the positions before combining them using XOR. Intuitively, the reason is that otherwise the information about dimension '0' would not be considered by the vertex coloring function.

is better than linear in the number of dimensions. We show in our next lemma, that the number of colors provided by our algorithm is a linearly bounded staircase function which is optimal up to rounding.

Lemma 6: (*number of colors required by the color assignment function*)

The number of colors required by the color assignment function is $\|\lceil d+1 \rceil\|$, where $\|\lceil ... \rceil\|$ denotes the rounding to the next-higher power of two, in other words

$$\|\lceil a \rceil\| = 2^{\lceil \log_2 a \rceil}.$$

Proof:

First, we prove, that our algorithm never generates a vertex corresponding to a color greater or equal to:

$$2^{\lceil \log_2(d+1) \rceil}$$

According to *col*, the color of a vertex is a XOR-combination of some numbers from the set {1, ..., *d*} (cf. definition 6). The binary representation of *d* has exactly $\lceil \log_2(d+1) \rceil$ bits. Therefore, the XOR-combination cannot create a number with more bits, and the highest-possible number with $\lceil \log_2(d+1) \rceil$ bits is

$$2^{\lceil \log_2(d+1) \rceil} - 1 .$$

Next, we prove that all color numbers in the interval

$$[0,\ 2^{\lceil \log_2(d+1) \rceil} - 1\ ]$$

are generated by the color assignment function. According to *col*, the vertex of the origin (0, 0, ..., 0) has color number zero (*col*(0) = 0). For any other vertex color *c*, bounded by the interval above, an appropriate bucket number *b* can easily be constructed, such that *col*(*b*)=*c* by the following algorithm: If bit *j* is set in *c*, then set also bit 2^j-1 in *b* and reset all other bits in *b*. The result is a valid bucket number for the *d*-dimensional hypercube, as can be seen from the following argumentation: We know that

$$j < \lceil \log_2(d+1) \rceil$$

and therefore, *b* has less than $2^{\lceil \log_2(d+1) \rceil - 1}$ bits,

and thus $b < 2^{2^{\lceil \log_2 d+1 \rceil - 1}} = 2^{\frac{1}{2} \cdot \|\lceil d+1 \rceil\|}$.

As *b* has to be smaller than 2^d in order to be a legal vertex number for a *d*-dimensional hypercube,

$$2^{\frac{1}{2} \cdot \|\lceil d+1 \rceil\|} \le 2^d \Leftrightarrow \|\lceil d+1 \rceil\| \le 2d .$$

This is guaranteed, since a power of two is always between a number and its double:

$$\forall d \in \mathbb{N} :\ \exists k \in \mathbb{N}_0 :\ d < 2^k \le 2d .$$

$\|\lceil d+1 \rceil\|$ cannot be rounded up to anything above 2*d*. If the bits with the numbers $2^{j_i} - 1$ for some j_i are set in *b*, then according to definition 6, the color number *col*(*b*) is

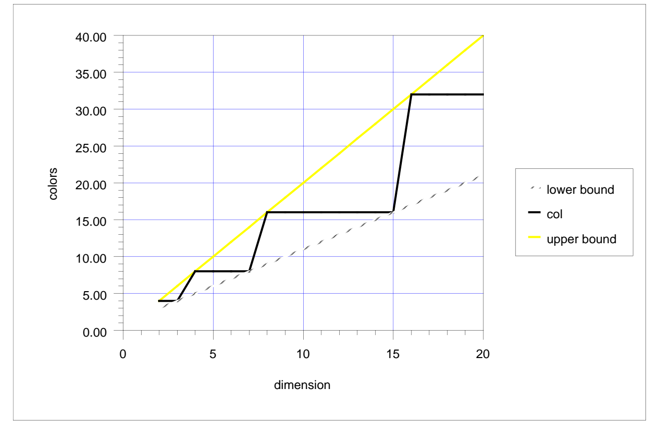


Figure 10: Number of Colors Required by *col*

$$col(b) = \underset{i}{\mathrm{XOR}}\ (2^{j_i}) ,$$

which combines to *c*. Altogether, we have proven that our algorithm uses exactly the colors with the numbers

$$0 \le c < \|\lceil d+1 \rceil\| .$$ ❑

The number of colors required to solve the vertex coloring problem is a staircase-function (cf. figure 10) above the line (d + 1) which has already been identified to be a lower bound for the number of colors. For lower dimensions, we have verified by enumerating all possible color assignments, that there is no method which uses fewer colors than our staircase function. We conjecture that this is also true for higher dimensions. In any case, we are able to give the linear upper and lower bounds for the staircase function. As already mentioned, the lower bound is *d*+1. The upper bound is 2*d*, as may be seen with the same argument already used in lemma 6: There is always a number corresponding to power of two between a number *d* and its double 2*d*. Therefore, $\|\lceil d+1 \rceil\|$ cannot be higher than 2*d* for $d \in \mathbb{N}$.

4.3 Extensions of our Declustering Technique

In this section, we propose two extensions of our declustering technique. First, we describe an adaptation of our method for supporting an arbitrary number of disks and second, we describe an extension of our method for highly clustered data.

An important requirement for any parallel approach is to support an arbitrary number of processing units (disks). For our problem, this means that we have to adapt our algorithm to work with an arbitrary number of disks, since our vertex coloring function *col* requires the optimal number of 2^i disks. We now describe a simple method for reducing the number of disks required; in a first step by a factor of 2 (preserving that direct neighbors are assigned to different disks), and in a second step to an arbitrary number.

As we can easily derive from the 3-dimensional example in figure 8, there exists no near-optimal declustering algorithm using less than 4 disks for the 3-dimensional case. As a consequence, reducing the number of colors generated by our function *col* may induce that indirectly neighboring buckets are assigned to the same

disk. Our extension of the function *col,* however, guarantees that most directly neighboring buckets are still assigned to different disks. The extension reduces the number of required disks by a factor of 2. The basic idea of our extension is to map one half of the colors to their binary-complementary color. For example, to decluster an 8-dimensional data space, the function *col* requires $C = 16$ disks numbered from 0 to 15. In our first reduction step, we map the colors 8..15 to the colors 0..7 such that 8 is mapped to 7, 9 is mapped to 6, ..., and 15 is mapped to 0. Obviously, our extended algorithm requires a total number of $(C/2)$ disks. Note that, this mapping guarantees that most directly neighboring buckets are still assigned to different disks. Intuitively, we map the colors to their complement because complementary colors have the maximal Hamming distance, i.e. differ in a maximum number of bits.

In the general case, let us assume that we have *n* disks available, where $n < C$. If $n \le C/2$, we map each color *c*, which is larger than $C/2$ to its binary complement. Thus, we have only $C/2$ colors left. Note that the most significant bit of these $C/2$ colors is the bit 0. If *n* is smaller than $C/4$, we again map the colors greater than $C/4$ to their complement, while, however, ignoring the most significant bit. This process is repeated until $n \le C/2^k$. The number of colors required by the algorithm is now $C/2^{k-1}$. In order to obtain exactly *n* colors, we again map the highest $C/2^{k-1} - n$ colors to their complement. Recording the mappings in a table, we are able to determine the disk number from the color number *col* by a single table look-up.

Another extension of our declustering techniques focuses on highly clustered data. In real applications, high-dimensional data is usually not distributed uniformly. If the data points are highly clustered, i.e. most data points are located in one quadrant of the hypercube, our technique as described so far would assign most data points to a single disk. Although in most applications such an extreme case will not occur, we have to consider data distributions where many points are assigned to a few disks, i.e. the amount of data stored on the disks differs largely.

A first solution to this problem is to use a statistical measure, the α-quantile, to divide the buckets. Instead of splitting each dimension in the middle, we determine the 0.5-quantile of each dimension and use the values as split values for determining the bucket boundaries. One may argue that we do not know the data distribution a priori and are therefore not able to determine the correct 0.5-quantile in advance. To solve the problem, we dynamically adapt the 0.5-quantile by recording the distribution according to the previous 0.5-quantile, i.e. counting the number of data points below and above the split value. If the ratio of these two numbers extends a certain threshold, we reorganize our data distribution using the new 0.5-quantile for each dimension.

If the data points are highly correlated, the usage of a one dimensional quantile is not sufficient. This situation is detected if the one-dimensional α-quantile does not change but the disks are loaded unbalanced, nonetheless. Our strategy for this case is to recursively decluster the overloaded buckets of the data space. The optimal declustering means to decluster all overloaded buckets. This, however, would require an amount of $O(2^d)$ of storage space which cannot be handled for higher dimensions. Our approach therefore re-

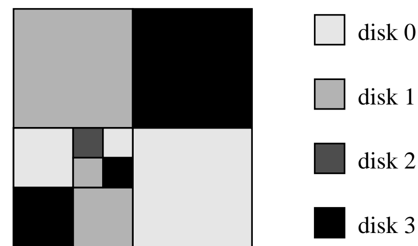


Figure 11: Recursive Declustering

cursively declusters all buckets of a single disk in one step using our *col* declustering function (cf. Figure 11), which means a transfer of the affected data to another disk. Note that we may have to apply the recursive declustering more than once if necessary. As first experiments show, permuting the colors using a simple heuristic when going to the next level of recursion provides good speed-ups (cf. Figure 16).

Note that our parallel nearest-neighbor search is completely dynamical. This means, that we are able to support insertions, updates, and deletions without any a priori knowledge of the data. However, for highly clustered or correlated data a reorganization of the data may be necessary.

5 Experimental Results

In order to show the efficiency and practical relevance of our declustering technique, we performed an extensive experimental evaluation of our technique and compared it to the Hilbert declustering which is the most promising declustering method designed for low-dimensional data spaces. All experiments have been computed on a workstation cluster of 16 HP710 workstations, each having 32 MBytes of main memory and several hundred MBytes of secondary storage. All programs have been implemented in C++ as templates to support different types of data objects. In order to analyse our method, we integrated our declustering technique and the Hilbert declustering into a parallel version of the X-tree [BKK 96].

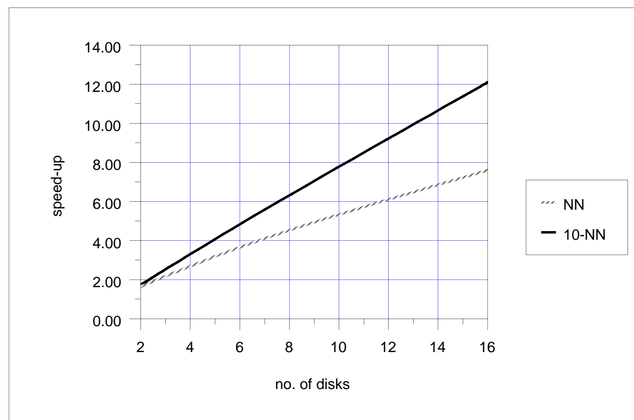
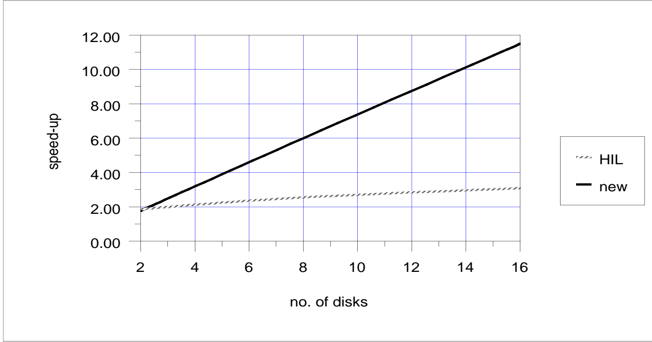
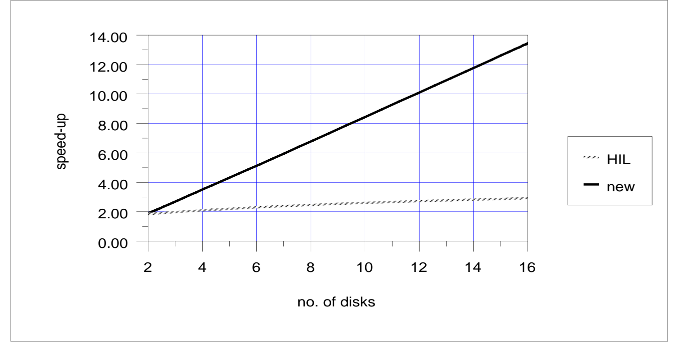


Figure 12: Speed-Up of Our Technique on Uniformly Distributed Data(1 MByte)



a: nearest-neighbor-query



b: 10-nearest-neighbors-query

Figure 13: Speed-Up of our technique and the Hilbert Curve (Fourier Points)

In our experiments, we used three types of data: Fourier points corresponding to contours of industrial parts (*d*=8..15), text data corresponding to substrings of a large set of texts (*d*=15), and uniformly distributed points (*d*=8..15). The total amount of data used in our experiments was about 800 MBytes. The block size used is 4 KBytes. In order to measure the performance of our technique, we determined the disk which accesses most pages during query processing. We used the search time of this disk as the search time of the whole parallel X-tree. Each experiment has been performed 10 times and the average of the 10 experiments is used as the reported search time . In order to compute the speed-up, we compared the search time of the parallel X-tree with a sequential X-tree using the original implementation of [BKK 96]. In the following figures, "new" denotes our technique, whereas "HIL" denotes the Hilbert approach.

Our first objective was to show the linear speed-up of our new method. We therefore performed an experiment on 1 MByte of uniformly distributed data (*d*=15) with varying numbers of disks (c.f. Figure 12). In performing a nearest-neighbor query, the speed-up reaches a value of 8 for 16 disks for a nearest-neighbor query. For 10-nearest-neighbors queries, the speed-up increases up to a value of 12 for 16 disks. In both experiments, the speed-up was nearly linear.

Since one cannot assume a uniform data distribution for real life applications, we used real data for our further experiments. Again, we investigated the speed-up of our technique and compared it to the Hilbert declustering for a nearest-neighbor query and a 10-nearest-neighbor query. Figure 13 shows the speed-up of our technique and the Hilbert curve on 40 MBytes of 15-dimensional Fourier points. Obviously, both techniques achieve a near-linear speed-up for both query types. However, our technique clearly outperforms the Hilbert curve which reaches only 19% of the optimal speed-up using 16 disks. Figure 14 shows the improvement of our technique over the Hilbert approach in the same experiment. The factor linearly increases with the number of disks and approaches a value of 5 for 16 disks. Note that this is due to the fact that the Hilbert curve does not provide a near-optimal declustering.

Next, we made experiments to measure the scale-up of our technique, i.e. we increased the number of disks and proportionally increased the total amount of data. In particular, we increased the number of disks from 2 to 16 while increasing the amount of data from 1 to 8 MBytes. Figure 16 depicts the result of this experiment. The total search time is nearly constant for both, nearest-neighbor queries and 10-nearest-neighbor queries. The experiment shows that our technique scales well when increasing the problem size.

In addition to the Fourier data, we also used text descriptors for our experiments. The text descriptors are feature vectors characterizing substrings of large sets of various documents given in ASCII format. Again, we compared our technique to the Hilbert approach. Figure 17 shows a total search time of 771 ms for our technique in contrast to 1683 ms for the Hilbert approach, for a nearest-neighbor query (improvement of 2.18) on 1MByte of 15-dimensional text descriptors. For the 10-nearest-neighbor query the improvement of our technique increased to 2.99.

In section 4.3, we proposed several extensions of our technique. The first extension, the adaption to an arbitrary number of disks, has been used for all experiments presented in this chapter which use a varying number of disks. The second extension of our technique has also been implemented and tested. Figure 16 depicts the results of these experiments. The experiments have been performed using 40 MBytes of 15-dimensional Fourier points. The Fourier points represent a set of variants of CAD-parts and are therefore highly clustered. The original technique yielded a total search time of 537.6 ms for a nearest-neighbor query, whereas the extension re-

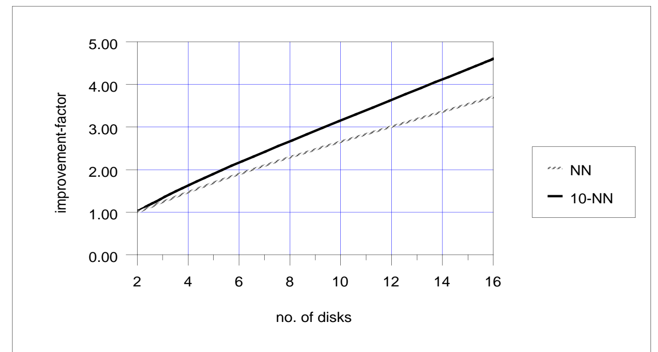


Figure 14: Improvement Factor over the Hilbert Curve (Fourier Points)

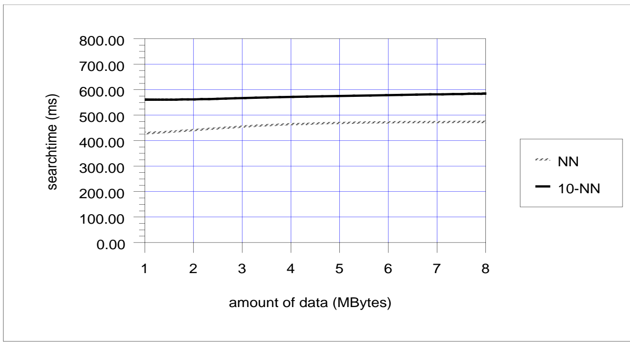


Figure 15: Scale-Up of our Technique on NN Queries and 10-NN Queries (Fourier Points)

duced the total search time to 137.7 ms. The large improvement factor of 3.9 is due to the fact that a large amount of data items is located in the same quadrant of the data space and therefore assigned to a single disk. Note that only one recursive declustering step was necessary in the experiments.

6 Conclusions

In this paper, we proposed a new method for parallel nearest-neighbor search in high-dimensional data spaces. High-dimensional data frequently occur in multimedia databases as the basis for a similarity retrieval. The core problem of designing a parallel nearest-neighbor algorithm is to determine an adequate distribution of the data to the disks which is called the declustering problem. The basic idea of our new declustering technique is to assign the buckets which correspond to different quadrants of the data space to different disks. We proved that our technique - in contrast to other declustering methods - guarantees that all buckets corresponding to neighboring quadrants are assigned to different disks. We evaluated our method using large amounts of real data and compared it with the Hilbert declustering. As the experiments show, our method provides a near-linear speed-up and a constant scale-up. Additionally, it outperforms the Hilbert approach by a factor of up to 5.

Our future work will include the optimization of the reorganization process which occurs if the data distribution changes during the insertion. Another topic which we will address in the future are declustering techniques which optimize the throughput instead of the search time for a single query.

References

[AGMM 90] Altschul S. F., Gish W., Miller W., Myers E. W., Lipman D. J.: *'A Basic Local Alignment Search Tool',* Journal of Molecular Biology, Vol. 215, No. 3, 1990, pp. 403-410.

[Ary 95] Arya S.: '*Nearest Neighbor Searching and Applications*', Ph.D. thesis, University of Maryland, College Park, MD, 1995.

[Big 89] Biggs N.L.: '*Discrete Mathematics*', Oxford Science Publications, Clarendon Press-Oxford, 1989, pp. 172-176.

[BBKK 97] Berchtold S., Böhm C., Keim D., Kriegel H.-P.: '*A Cost Model For Nearest Neighbor Search in High-Dimensional Data Space*', ACM PODS Symposium on Pricinples of Database Systems, 1997, Tucson, Arizona.

[BKK 96] Berchtold S., Keim D., Kriegel H.-P.: '*The X-tree: An Index Structure for High-Dimensional Data*', 22nd Conf. on Very Large Databases, 1996, Bombay, India, pp. 28-39.

[BKSS 90] Beckmann N., Kriegel H.-P., Schneider R., Seeger B.: *'The R*-tree: An Efficient and Robust Access Method for Points and Rectangles',* Proc. ACM SIGMOD Int. Conf. on Management of Data, Atlantic City, NJ, 1990, pp. 322-331.

[DS 82] Du H.C., Sobolewski J.S.: '*Disk allocation for cartesian product files on multiple Disk systems*', ACM TODS, Journal of Transactions on Database Systems, 1982, pp. 82-101.

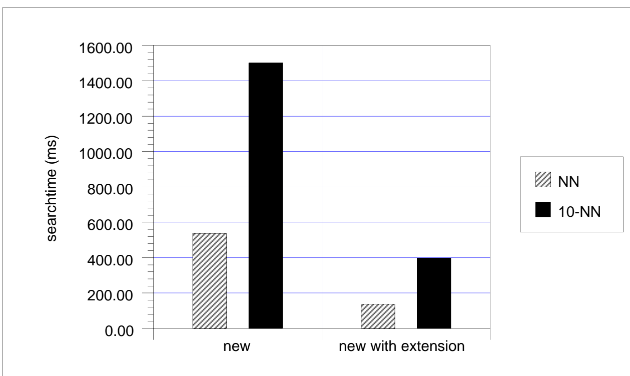


Figure 16: Effect of Recursive Declustering

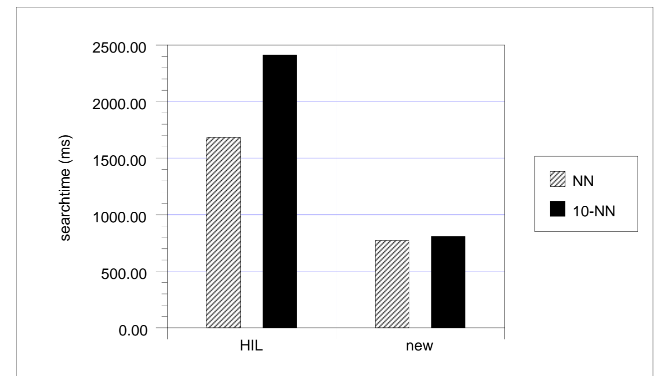


Figure 17: Total search time of our technique and the Hilbert curve (Text Data)

[Fal 94] Faloutsos C., Barber R., Flickner M., Hafner J., et al.: *'Efficient and Effective Querying by Image Content',* Journal of Intelligent Information Systems, 1994, Vol. 3, pp. 231-262.

[FB 93] Faloutsos C., Bhagwat P.: '*Declustering Using Fractals*', PDIS Journal of Parallel and Distributed Information Systems, 1993, pp. 18-25.

[FBF 77] Friedman J. H., Bentley J. L., Finkel R. A.: '*An Algorithm for Finding Best Matches in Logarithmic Expected Time*', ACM Transactions on Mathematical Software, Vol. 3, No. 3, September 1977, pp. 209-226.

[HS 95] Hjaltason G. R., Samet H.: '*Ranking in Spatial Databases*', Proc. 4th Int. Symp. on Large Spatial Databases, Portland, ME, 1995, pp. 83-95.

[Jag 91] Jagadish H. V.: *'A Retrieval Technique for Similar Shapes',* Proc. ACM SIGMOD Int. Conf. on Management of Data, 1991, pp. 208-217.

[Kuk 92] Kukich K.: '*Techniques for Automatically Correcting Words in Text*', ACM Computing Surveys, Vol. 24, No. 4, 1992, pp. 377-440.

[KP 88] Kim M.H., Pramanik S.: '*Optimal file distribution for partial match retrieval*', Proc. ACM SIGMOD Int. Conf. on Management of Data, 1988, pp. 173-182.

[LJF 94] Lin K., Jagadish H. V., Faloutsos C.: *'The TV-tree: An Index Structure for High-Dimensional Data',* VLDB Journal, Vol. 3, pp. 517-542, 1995.

[MG 93] Mehrotra R., Gary J.: *'Feature-Based Retrieval of Similar Shapes',* Proc. 9th Int. Conf. on Data Engeneering, April 1993

[MG 95] Mehrotra R., Gary J.: *'Feature-Index-Based Sililar Shape retrieval',* Proc. of the 3rd Working Conf. on Visual Database Systems, March 1995

[PS 85] Preparata F.P., Shamos M. I.: '*Computational Geometry*', Chapter 5 ('Proximity: Fundamental Algorithms'), Springer Verlag New York, 1985, pp. 185-225.

[RKV 95] Roussopoulos N., Kelley S., Vincent F.: '*Nearest Neighbor Queries*', Proc. ACM SIGMOD Int. Conf. on Management of Data, 1995, pp. 71-79.

[RP 92] Ramasubramanian V., Paliwal K. K.: '*Fast k-Dimensional Tree Algorithms for Nearest Neighbor Search with Application to Vector Quantization Encoding*', IEEE Transactions on Signal Processing, Vol. 40, No. 3, March 1992, pp. 518-531.

[SBK 92] Shoichet B. K., Bodian D. L., Kuntz I. D.: '*Molecular Docking Using Shape Descriptors*', Journal of Computational Chemistry, Vol. 13, No. 3, 1992, pp. 380-397.

[SH 94] Shawney H., Hafner J.: '*Efficient Color Histogram Indexing*', Proc. Int. Conf. on Image Processing, 1994, pp. 66-70.

[Wel 71] Welch T.: '*Bounds on the Information Retrieval Efficiency of Static File Structures*', Technical Report 88, MIT, 1971.

[WW 80] Wallace T., Wintz P.: *'An Efficient Three-Dimensional Aircraft Recognition Algorithm Using Normalized Fourier Descriptors',* Computer Graphics and Image Processing, Vol. 13, pp. 99-126, 1980

Appendix

Lemma 2 (*distributivity of col and XOR*):

$$\forall b\ \forall c:\quad col(b)\ \text{XOR}\ col(c) = col(b\ \text{XOR}\ c)$$

Proof:

$col\,(b)$ XOR $col\,(c)$ =

$$= \operatorname*{XOR}_{i=0}^{d-1}\left(\begin{cases} i+1 & \text{if } b_i = 1 \\ 0 & \text{otherwise}\end{cases}\right)\ \text{XOR}\ \operatorname*{XOR}_{i=0}^{d-1}\left(\begin{cases} i+1 & \text{if } c_i = 1 \\ 0 & \text{otherwise}\end{cases}\right)$$

$$= \operatorname*{XOR}_{i=0}^{d-1}\left(\begin{cases} 0\ \text{XOR}\ 0 & \text{if } b_i = 0 \text{ and } c_i = 0 \\ i+1\ \text{XOR}\ 0 & \text{if } b_i = 1 \text{ and } c_i = 0 \\ 0\ \text{XOR}\ i+1 & \text{if } b_i = 0 \text{ and } c_i = 1 \\ i+1\ \text{XOR}\ i+1 & \text{if } b_i = 1 \text{ and } c_i = 1 \end{cases}\right)$$

$$= \operatorname*{XOR}_{i=0}^{d-1}\left(\begin{cases} i+1 & \text{if } b_i\ \text{XOR}\ c_i = 1 \\ 0 & \text{otherwise}\end{cases}\right)$$

$= col\,(b$ XOR $c)$ ❑