RJ 10093 (91909) November 20, 1997
Math & Comp. Sci.

EFFICIENT NEAREST NEIGHBOR INDEXING BASED ON A COLLECTION OF SPACE FILLING CURVES

Nimrod Megiddo
Uri Shaft

IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099

ABSTRACT:

A database is populated with a set of points represented by n-tuples of real numbers. A query consists of a point q (not necessarily in the database) and an integer k, asking for the k "nearest" database points to the query point. The exact output for the query consists of the k nearest points, but if the database is large and a quick response is required, a good approximate output is sought. All currently known methods require at query time calculation of distances from the query point to many database points. The computational effort is dominated by the number of such distance calculations since points have to be fetched from random locations in the database and the high dimension implies that current database indexes cannot significantly restrict the number of points that have to be fetched. In many cases, a complete linear scan of the database beats the currently known methods. The number of such distance calculations performed by current methods grows with the number of points in the database.

The method described in this report has shown (in experiments on databases with tens of thousands of points with hundreds of dimensions, and asking for about 100 nearest neighbors) to provide very good approximate output sets while limiting the number of distance calculations to a few hundreds. Theoretical analysis predicts that the number of required distance calculations depends on the dimension and not on the number of points in the database. Therefore, when more points are added to a database the dominant factor in the query effort does not change.

Efficient Nearest Neighbor Indexing Based on a Collection of Space Filling Curves

Nimrod Megiddo and Uri Shaft
IBM Almaden Research Center

October 20, 1997

1 Introduction

1.1 The nearest neighbors problem

Imagine a database DB consisting of points from $\mathcal{S} = \mathcal{D}_1 \times \cdots \times \mathcal{D}_n$, where $\mathcal{D}_i \subseteq \mathbb{R}$. Each $\mathcal{D}_i$ usually consists of either integers or floats. Our results can be extended to this general case, but here we restrict attention to the case of $\mathcal{S} = \mathcal{D}^n$ where $\mathcal{D}$ is the set of rational numbers in $[0, 1)$ whose denominators are powers of 2.

A *k-nearest neighbors* query consists of a point $\mathbf{q} \in \mathcal{D}^n$ and a positive integer k. An (exact) output set O consists of k points from the database such that

$$\forall \mathbf{p}' \in O \text{ and } \forall \mathbf{p}'' \in DB \setminus O \qquad ||\mathbf{p}' - \mathbf{q}|| \le ||\mathbf{p}'' - \mathbf{q}|| \,.$$

The actual problem in some database management system (DBMS) applications is how to process such queries so that a good approximate output (the sense of the approximation will be discussed later) is returned within the desired response time. The goodness of an approximate output depends of course on the application, and the response time depends on the processing needs, such as disk I/O and CPU time. For example, in image DBMSs, features of images are sometimes mapped into $\mathcal{D}^n$ and the similarity of images is determined by the distance between the features in $\mathcal{D}^n$. Since this similarity metric is an approximation of the desired similarity, adding another small approximation error does not hurt the results much.

1.2 Motivation

Various approaches have been tried for near neighbor searching. It appears that most of the approaches are defeated by the dimensionality. The intuition offered by two- and three-dimensional observations can be quite misleading. We review some of these approaches briefly.

Bounding boxes or spheres

Many hierarchical structures have been proposed for indexing multidimensional data in database management systems. These structures collect data points into disk pages and compute bounds on the points in the disk page. Usually, the bound is the minimal bounding box that is parallel to the axes, but also spheres and convex polyhedra have been tried. The collection of disk pages that contain the points is stored at the bottom level of the hierarchical structure. The level above it is created by taking the collection of bounds (*e.g.*, the bounding boxes) and treating them as data points. These are collected into groups, each fitting on a disk page, and for each group a new bound is computed. This new bound is of the same type as the bounds of the lower level (*i.e.*, if we used boxes, than the upper levels use boxes as well). We continue to create levels until the top level fits on one disk page.

The most known structures are the R-tree family (see R. Guttman), the hB-tree (D. Lomet), the TV-tree (C. Faloutsos) and many others, all use bounding boxes. The sphere-tree (see ?) uses spheres. The cell-tree (Gunther) uses convex polyhedra. The only structures that are used in practice are those of the R-tree family.

A query to these structures amounts to specifying a region.Points in that region are sought by going down the structure. If the region overlaps the bounds of a page then the subtree under that page may contain points that are in the region. A nearest neighbors query is processed as a query about a small spherical region. If the result does not contain enough points, then the region is enlarged (*e.g.*, by doubling the radius or the sphere).

This method works well as long as the index structure behaves well. This means that the complexity of searching for points in a region should be proportional to the volume of that region (assuming it is convex and "nice"). This assumption may be justified only in very low dimension (*i.e.*, up to 4). Experiments show that careful construction of the structure may push this up a bit (to dimension 6 or 7). In higher dimension, however, the method behaves far worse than a sequential scan of the entire data set.

Projections

One-dimensional projections induce orderings on the set of database points, and the projection of a query point can be quickly located within these orderings. Projections are continuous, so close points have close projections. On the other hand, distant points may also have close projections. The higher the dimension, the more severe this problem becomes. When candidates or nearest neighbors are picked from among the points whose projections are close to the projection of the query, the number of candidates gets too large. Good candidates for nearest neighbors should be those that have many close projections. However, the problem of determining such candidates is very closely related to the nearest neighbors problem itself. An interesting theoretical result is reported in [4]. The difference between orderings based on one-dimensional projections and ones based on space filling curves is that in the latter, proximity in the ordering implies proximity in the space, so if there are sufficiently many orderings, it suffices to consider candidates who are close in at least one of the orderings.

Clustering

Placing the database points in clusters reflecting proximity is believed to help in the search for near neighbors. Each cluster is represented either by a database point in it or by the centroid of the cluster. The cluster whose representative point is closest to the query point is searched first. Other clusters are searched in order of proximity of their representatives to the query point or based on bounds that are derived in various ways. The search is expected to end without checking all the database points. As with other approaches, in high dimension this approach may break down since it may not be possible to identify sufficiently many clusters as distant.

2 The one-dimensional encoding

Our near neighbors algorithm is based on locating neighbors of the query point with respect to several linear orderings over the database points. In this section we introduce a family of suitable orderings of the data space $\mathcal{D}^n$ that can be used for ordering the database points. We start by defining a canonical ordering based on a one-dimensional code $C : \mathcal{D}^n \to \mathcal{D}$ and then explain how to generate from it the other orderings. In order to describe the calculation of $C(\mathbf{v})$ for $\mathbf{v} \in \mathcal{D}^n$, we use a convenient representation of $\mathbf{v}$ as follows. Suppose $\mathbf{v} = (v_n, \ldots, v_1)$ where $v_j = \sum_{i=1}^{m} v_{ij} 2^{-i}$ ($v_{ij} \in \{0,1\}, j = 1, \ldots, n, i = 1, \ldots, m$). Then we can represent $\mathbf{v} = \sum_{i=1}^{m} 2^{-i}\mathbf{v}^i$, where $\mathbf{v}^i = (v_{i1}, v_{i2}, \ldots, v_{in})$ ($i = 1, \ldots, m$). The encoding can be defined, recursively in terms of m, using the following objects which will be defined later:

1. A code $J : \{0,1\}^n \to \{0, \ldots, 2^n - 1\}$ (which is essentially the inverse of a Gray code),
2. Affine transformations A_I of $\mathbb{R}^n$, for $I = 0, \ldots, 2^n - 1$.

2.1 Gray code and its inverse

For every $\mathbf{x} = (x_n, \ldots, x_1)$ ($x_i \in \{0,1\}, i = 1, \ldots, n$), the code $J(\mathbf{x})$ is represented as

$$J(\mathbf{x}) = \sum_{i=1}^{n} y_i 2^{i-1}$$

where $y_i \in \{0,1\}$, so $0 \le J(\mathbf{x}) < 2^n$. It is calculated by the following algorithm:

Algorithm `J; input:` $(x_n, \ldots, x_1)$; `output:` $(y_n, \ldots, y_1)$

```
1. $y_n := x_n$
2. for $i = n-1$ downto 1 do
       $y_i := x_i \otimes x_{i+1}$
```

Intuitively, the code J defines a linear ordering on the vertices of the unit cube in $\mathbb{R}^n$ so that consecutive vertices in the linear ordering are also adjacent in the cube. Note that $J(0,\dots,0) = 0$ so $(0,\dots,0)$ is the first in this ordering. Similarly, $J(1,0,\dots,0) = 2^n - 1$, so $(1,0,\dots,0)$ is the last. These two vertices are also adjacent.

The following fact follows from the definition of J:

Fact 2.1

$$J(1, x_{n-1}, \dots, x_1) = 2^n - 1 - J(0, x_{n-1}, \dots, x_1) .$$

We also use the Gray code itself (*i.e.*, the inverse of J) $G(I)$ which maps an integer $I = \sum_{i=1}^{n} y_i 2^{i-1}$ to a vector $\mathbf{x} = (x_n, \dots, x_1)$ $(x_i \in \{0,1\}, i = 1, \dots, n)$ according to the following algorithm:

Algorithm `G; input:` $(y_n, \dots, y_1)$`; output:` $(x_n, \dots, x_1)$

```
1. $x_n := y_n$
2. for $i = n-1$ downto 1 do
       $x_i := y_i \otimes x_{i+1}$
```

The following fact from the definition of G:

Fact 2.2 *For $I \geq 2^{n-1}$,*

$$G(I) = 2^{n-1} + G(2^n - 1 - I) .$$

Using these recursive formulas given in Facts 2.1 and 2.2, we can prove by induction:

Fact 2.3 *The mappings J and G are the inverse functions of each other.*

When we wish to determine the order relation between any two points $\mathbf{p} = (p_n, \dots, p_1)$ and $\mathbf{q} = (q_n, \dots, q_1)$ in $[0,1)^n$, the first phase is based on the code J. In order to have a unique binary representation for any number, we always choose between the first of the two equivalent tail expansions: 1000... and 0111... Let $p_j^1 = \lfloor 2p_j \rfloor$ and $q_j^1 = \lfloor 2q_j \rfloor$ (*i.e.*, these are obtained by discarding all the bits except for the first one). To compare $\mathbf{p}$ and $\mathbf{q}$, we first compare $J(p_n^1, \dots, p_1^1)$ and $J(q_n^1, \dots, q_1^1)$. If the latter are distinct, the comparison has been decided. Otherwise[1], we proceed to perform a comparison based on the second

[1]It is interesting to observe that if the points are picked independently from a uniform distribution then with probability $1 - 2^{-n}$ the comparison will be resolved in the first phase, so the expected number of phases to resolve the comparison is $1 + 1/(2^n - 1)$.

bits, and so on. Here, however, we do not use the function J directly. Since the ordering is expected to reflect distance, we have to transform the points as explained below.

The code J induces an ordering on the 2^n sub-cubes of the unit cube, obtained by cutting the latter with the hyperplanes $\{x_i = 0.5\}$ $(i = 1, \dots, n)$. More precisely, for every $\mathbf{a} = (a_n, \dots, a_1) \in \{0,1\}^n$, let

$$S(\mathbf{a}) = \{\mathbf{x} = (x_n, \dots, x_1) \mid 0 \le x_i < \tfrac{1}{2} \text{ if } a_i = 0,\ \tfrac{1}{2} \le x_i < 1 \text{ if } a_i = 1\} .$$

The sub-cubes $S(\mathbf{a})$ are disjoint and ordered by the function J (*i.e.*, $S(\mathbf{a})$ comes before $S(\mathbf{a}')$ if and only if $J(\mathbf{a}) < J(\mathbf{a}')$). If $\mathbf{p}$ and $\mathbf{q}$ belong to distinct $S(\mathbf{a})$'s, they inherit their ordering relation from the ordering on these sub-cubes. Otherwise, they are in the same $S(\mathbf{a})$ and the sub-cubes of $S(\mathbf{a})$ itself have to be considered. It is crucial though that the orderings on the sub-cubes of the $S(\mathbf{a})$'s will be such that for every $\mathbf{a}$ (except the last sub-cube) the last sub-cube of $S(\mathbf{a})$ would be adjacent to the first sub-cube of $S(\mathbf{a}')$, where $S(\mathbf{a}')$ is the sub-cube that succeeds $S(\mathbf{a})$ in the ordering over the sub cubes of the unit cube. If this condition is not satisfied, then there would be two distant points in the cube that are arbitrarily close in the linear order, and in order for our indexing scheme to be efficient this should never happen. Thus, the ordering on the sub cubes of each $S(\mathbf{a})$ will be determined by first applying a suitable affine transformation, and then applying the function J.

To summarize the discussion so far, the comparisons are based on a one-dimensional code $C : [0,1)^n \to [0,1)$ that has a fractal nature which can be described as follows. For $\mathbf{v} = (v_n, \dots, v_1) \in [0,1)^n$, denote

$$\mathbf{v}^1 = (\lfloor 2v_1 \rfloor, \dots, \lfloor 2v_n \rfloor).$$

For $I = 0, \dots, 2^n - 1$ we define below an affine transformation A_I of $\mathbb{R}^n$ that maps boolean vectors to boolean vectors. An accurate statement requires that we define C with respect to all kinds of cubes $\mathcal{I}_1 \times \cdots \times \mathcal{I}_n$, where each $\mathcal{I}_i$ is either $[0,1)$ or $(0,1]$, and with respect to all the permutations of the coordinates. For simplicity of notation, however, we prefer to state it as follows.

$$C(\mathbf{v}) = 2^{-n} \left[J(\mathbf{v}^1) + C\left(A_{J(\mathbf{v}^1)}(2\mathbf{v} - \mathbf{v}^1)\right)\right] .$$

2.2 The affine transformations

We now define for each $I \in \{0, \dots, 2^n - 1\}$ an affine transformation A_I of $\mathbb{R}^n$ that maps the set $\{0,1\}^n$ onto itself. It is essential for our application that the transformations could be computed in time proportional to the dimension of the space. The transformation A_I is composed of a "reflection" component represented by a vector $\mathbf{s} = \mathbf{s}(I) \in \{0,1\}^n$, and a "swap" component represented by a permutation matrix $\boldsymbol{P} = \boldsymbol{P}(I)$ of order $n \times n$ that swaps coordinate n with some coordinate $i = i(I)$ (*i.e.*, if $i \ne n$ then the only difference between $\boldsymbol{P}$ and the $n \times n$ identity matrix is that $P_{in} = P_{ni} = 1$ and $P_{ii} = P_{nn} = 0$). The composition is:

$$A_I(\mathbf{v}) = \boldsymbol{P}(I) \cdot (\mathbf{s}(I) \otimes \mathbf{v}) .$$

2.2.1 The reflection

The reflection component, $\mathbf{s}(I)$, of A_I is calculated by the following algorithm, where $\mathbf{e}_1 = (0, \ldots, 0, 1)$:

```
Algorithm s; input: $I$; output: $\mathbf{s} = \mathbf{s}(I)$
    if $I = 0$ then $\mathbf{s} := 0$
    else if $I \equiv 1 \pmod 2$ then $\mathbf{s} := G(I-1)$
    else if $I \equiv 2 \pmod 4$ then $\mathbf{s} := G(I-1) - \mathbf{e}_1$
    else if $I \equiv 0 \pmod 4$ then $\mathbf{s} := G(I-1) + \mathbf{e}_1$
```

Fact 2.4 *For odd k, $\mathbf{s}(k+1) = G(k-1)$ while for even k, $\mathbf{s}(k+1) = G(k)$. Thus, for odd k, $\mathbf{s}(k+1) = \mathbf{s}(k)$.*

Proof: It follows from the definition of the Gray code that if $k \equiv 1 \pmod 4$, then $G(k) = G(k-1) + \mathbf{e}_1$, in which case

$$\mathbf{s}(k+1) = G(k) - \mathbf{e}_1 = G(k-1) \ ,$$

and if $k \equiv 3 \pmod 4$ then $G(k) = G(k-1) - \mathbf{e}_1$, in which case again

$$\mathbf{s}(k+1) = G(k) + \mathbf{e}_1 = G(k-1) \, .$$

For even k, by definition $\mathbf{s}(k+1) = G(k)$. ∎

2.2.2 The swap

The swap component $\boldsymbol{P} = \boldsymbol{P}(I)$ is determined by the index $i = i(I)$ of the coordinate that is swapped under $\boldsymbol{P}$ with coordinate n. We define $i(0) = i(2^n - 1) = 1$, and for $I = \sum_{i=1}^{n} x_i 2^{i-1}$ such that $0 < I < 2^n - 1$, $i(I)$ is the largest index $j \geq 2$ such that $x_k = 0$ for all $2 \leq k < j$.

This index is calculated by the following algorithm:

```
Algorithm i; input: $I$; output: $i = i(I)$
    if $I = 0$ or $I = 2^n - 1$ then $i := 1$
    else
        1. $i := 2$
        2. $I := \lfloor (I+1)/2 \rfloor$
        3. while $I \equiv 0 \pmod 2$ do
            (a) $i := i + 1$
            (b) $I := \lfloor I/2 \rfloor$
```

Fact 2.5 *For even $k > 0$, $i(k) = i(k-1)$.*

Proof: In Step 2 of the algorithm we convert I to $\lfloor (k+1)/2 \rfloor$ (for input k), and to $\lfloor k/2 \rfloor$ (for input $k-1$). If $k > 0$ is even then $\lfloor (k+1)/2 \rfloor = \lfloor k/2 \rfloor$, so the result of the algorithm is the same for k and $k-1$. ∎

Fact 2.6 *For* $0 < k < 2^n - 1$, $G(k+1)$ *and* $G(k-1)$ *differ in coordinates* 1 *and* $i(k)$.

The transformations in the case $n = 2$ are summarized in the following table:

I	$G(I-1)$	Δ	$s(I)$	$i(I)$	$A_I(x_2, x_1)$
00			00	1	(x_1, x_2)
01	00		00	2	(x_2, x_1)
10	01	−01	00	2	(x_2, x_1)
11	11		11	1	$(1 - x_1, 1 - x_2)$

The transformations in the case $n = 3$ are summarized in the following table:

I	$G(I-1)$	Δ	$s(I)$	$i(I)$	$A_I(x_3, x_2, x_1)$
000			000	1	(x_1, x_2, x_3)
001	000		000	2	(x_2, x_3, x_1)
010	001	−001	000	2	(x_2, x_3, x_1)
011	011		011	3	$(x_3, 1 - x_2, 1 - x_1)$
100	010	+001	011	3	$(x_3, 1 - x_2, 1 - x_1)$
101	110		110	2	$(1 - x_2, 1 - x_3, x_1)$
110	111	−001	110	2	$(1 - x_2, 1 - x_3, x_1)$
111	101		101	1	$(1 - x_1, x_2, 1 - x_3)$

The transformations in the case $n = 4$ are summarized in the following table:

I	$G(I-1)$	Δ	$s(I)$	$i(I)$	$A_I(x_4, x_3, x_2, x_1)$
0000			0000	1	(x_1, x_3, x_2, x_4)
0001	0000		0000	2	(x_2, x_3, x_4, x_1)
0010	0001	−0001	0000	2	(x_2, x_3, x_4, x_1)
0011	0011		0011	3	$(x_3, x_4, 1 - x_2, 1 - x_1)$
0100	0010	+0001	0011	3	$(x_3, x_4, 1 - x_2, 1 - x_1)$
0101	0110		0110	2	$(1 - x_2, 1 - x_3, x_4, x_1)$
0110	0111	−0001	0110	2	$(1 - x_2, 1 - x_3, x_4, x_1)$
0111	0101		0101	4	$(x_4, 1 - x_3, x_2, 1 - x_1)$
1000	0100	+0001	0101	4	$(x_4, 1 - x_3, x_2, 1 - x_1)$
1001	1100		1100	2	$(x_2, 1 - x_3, 1 - x_4 x_1)$
1010	1101	−0001	1100	2	$(x_2, 1 - x_3, 1 - x_4, x_1)$
1011	1111		1111	3	$(1 - x_2, 1 - x_3, 1 - x_4, 1 - x_1)$
1100	1110	+0001	1111	3	$(1 - x_2, 1 - x_3, 1 - x_4, 1 - x_1)$
1101	1010		1010	2	$(1 - x_2, x_3, 1 - x_4, x_1)$
1110	1011	−0001	1010	2	$(1 - x_2, x_3, 1 - x_4, x_1)$
1111	1001		1001	1	$(1 - x_1, x_3, x_2, 1 - x_4)$

2.3 The algorithm for computing the mapping C

We are now ready to describe the algorithm that calculates the code $C(\mathbf{v})$ of a given point $\mathbf{v} = (v_n, \dots, v_1)$, where each v_j is an m-bit number, $v_j = \sum_{i=1}^{m} v_{ij}2^{-i}$ $(v_{ij} \in \{0,1\}$, $j = 1, \dots, n$, $i = 1, \dots, m)$. The algorithm produces the representation $C(\mathbf{v}) = \sum_{i=1}^{m} I_i 2^{-in}$ $(0 \leq I_i < 2^n$ $(i = 1, \dots, m)$. It also produces the representation $\mathbf{v} = \sum_{i=1}^{m} 2^{-i}\mathbf{v}^i$, where $\mathbf{v}^i = (v_{i1}, v_{i2}, \dots, v_{in})$ $(i = 1, \dots, m)$. All the operations on vectors are component-wise.

Algorithm C; input: $\mathbf{v} = (v_n, \dots, v_1)$; output: $C = C(\mathbf{v}) = \sum_{i=1}^{m} I_i 2^{-in}$

```
1. $i := 1$;  $\mathbf{s} := \mathbf{0}$;  $\boldsymbol{P} = \boldsymbol{I}$
2. while $\mathbf{v} \neq \mathbf{0}$ do
   2.1 Convert $\mathbf{v}$ into a bit vector and a remainder :
       (a) $\mathbf{v}^i := \lfloor 2\mathbf{v} \rfloor$
       (b) $\mathbf{v} := 2\mathbf{v} - \mathbf{v}^i$
   2.2 Compute $I_i$ :
          $I_i := J(\boldsymbol{P} \cdot (\mathbf{s} \otimes \mathbf{v}^i))$
   2.3 Compute the new transformation :
       (a) $\mathbf{s} := \mathbf{s} \otimes (\boldsymbol{P} \cdot \mathbf{s}(I_i))$
       (b) $\boldsymbol{P} := \boldsymbol{P} \cdot \boldsymbol{P}(I_i)$
   2.4 Increment $i$ :
          $i := i + 1$
```

2.4 Properties of the mapping C

We first prove that the mapping C is adequate for unambiguous coding of n-dimensional data.

Theorem 2.7 *The mapping C is one-to-one.*

Proof: Suppose $\mathbf{p}$ and $\mathbf{q}$ are distinct points in $[0,1)^n$. There exist unique representations $\mathbf{p} = \sum_{i=1}^{\infty} \mathbf{p}^i 2^{-i}$ and $\mathbf{q} = \sum_{i=1}^{\infty} \mathbf{q}^i 2^{-i}$, where $\{\mathbf{p}^i, \mathbf{q}^i\} \subset \{0,1\}^n$ $(i = 1, 2, \dots)$, and for every natural N and every j $(1 \leq j \leq n)$, there exist $k, \ell > N$ such that $p^j_k = q^j_\ell = 0$. Let i $(i \geq 1)$ be the smallest index such that $\mathbf{p}^i \neq \mathbf{q}^i$. It follows that in the hierarchy of the sub-cubes of the unit cube (obtained by repeatedly halving all the dimensions), there exists a sub-cube S_0 of the unit cube whose edges are of length 2^{-i+1}, such that $\{\mathbf{p}, \mathbf{q}\} \subset S_0$, whereas there exist two disjoint sub-cubes S_1, S_2 of S_0, whose edges are of length 2^{-i}, such that $\mathbf{p} \in S_1$ and $\mathbf{q} \in S_2$. The points of S_1 and S_2 are mapped under J, into two disjoint intervals of $[0,1)$ (each of length 2^{-in}), hence $J(\mathbf{p}) \neq J(\mathbf{q})$. ∎

Before stating the main claims, we introduce notation as follows. For $m = 1, 2, \dots,$

1. For $k_1, \ldots, k_n$ such that $0 \le k_j \le 2^m - 1$ $(j = 1, \ldots, n)$, define the cube
$$S(k_1, \ldots, k_n; m) = \{\mathbf{x} \in \mathbb{R}^n \mid k_j 2^{-m} \le x_j < (k_j + 1) 2^{-m},\ j = 1, \ldots, n\} .$$
2. Denote by $\mathcal{S}(m)$ the collection of all the cubes $S(k_1, \ldots, k_n; m)$.
3. For every $S \in \mathcal{S}(m)$, denote by $\mathcal{S}(m+1; S)$ those members of $\mathcal{S}(m+1)$ that are contained in S. Also, if $S \in \mathcal{S}(m)$ is not the sub-cube that is mapped under C into $[1 - 2^{-mn}, 1)$, then denote by S' the member of $\mathcal{S}(m)$ that succeeds S in the ordering induced by C, *i.e.*, if S is mapped into $[k 2^{-mn}, (k+1) 2^{-mn})$, then S' is mapped into $[(k+1)2^{-mn}, (k+2)2^{-mn})$.

Note that for each m, the members of $\mathcal{S}(m)$ are mapped under C into 2^{mn} disjoint (half open) subintervals of $[0, 1)$, each of length 2^{-mn}.

Lemma 2.8 *For $m = 1, 2, \ldots$, and for every $S \in \mathcal{S}(m)$ except for the last one (relative to the ordering induced by C), the last member of $\mathcal{S}(m+1; S)$ and the first member of $\mathcal{S}(m+1; S')$ are adjacent sub-cubes (i.e., they have a common facet).*

Proof: The proof goes by induction on m. Let us first consider the case $m = 1$. Suppose $S \in \mathcal{S}(1)$ is mapped into the interval
$$T(1, k) \equiv [k 2^{-n}\ ,\ (k+1) 2^{-n})$$
where $0 \le k \le 2^n - 2$. Thus, the last member of $\mathcal{S}(2; S)$ is mapped into the interval
$$T_L \equiv T(2, 2^n(k+1) - 1) = [(k+1) 2^{-n} - 2^{-2n}\ ,\ (k+1) 2^{-n})\ ,$$
while the first member of $\mathcal{S}(2; S')$ is mapped into
$$T_R \equiv T(2, 2^n(k+1)) = [(k+1) 2^{-n}\ ,\ (k+1) 2^{-n} + 2^{-2n})\ .$$
If a point $\mathbf{v}_R = \frac{1}{2}\mathbf{v}_R^1 + \frac{1}{4}\mathbf{v}_R^2 + \varepsilon_R$ (where $\mathbf{v}_R^i \in \{0, 1\}^n$, $i = 1, 2$, and $\varepsilon_R \in [0, \frac{1}{4})^n$) has $C(\mathbf{v}_R) \in T_R$, then it means that the first two values that Algorithm C calculates for it are $I_1 = k + 1$ and $I_2 = 0$. Thus, necessarily,
$$J(\mathbf{v}_R^1) = k + 1$$
which is the same as
$$\mathbf{v}_R^1 = G(k+1)\ .$$
Now, the first values of the other objects computed by Algorithm C are
$$\mathbf{s}_R^1 = \mathbf{s}_R^0 \otimes (\boldsymbol{P}_R^0 \cdot \mathbf{s}(k+1)) = \mathbf{s}(k+1)$$
$$\text{and}\quad \boldsymbol{P}_R^1 = \boldsymbol{P}_R^0 \cdot \boldsymbol{P}(k+1) = \boldsymbol{P}(k+1)\ .$$

Thus, the other necessary condition for $\mathbf{v}_R = \frac{1}{2}\mathbf{v}_R^1 + \frac{1}{4}\mathbf{v}_R^2 + \boldsymbol{\varepsilon}$ to be mapped into T_R is:

$$0 = I_2 = J(\boldsymbol{P}_R^1 \cdot (\mathbf{s}_R^1 \otimes \mathbf{v}_R^2)) = J(\boldsymbol{P}(k+1) \cdot (\mathbf{s}(k+1) \otimes \mathbf{v}_R^2)) ,$$

which, since $G(0) = \mathbf{0}$, is the same as $\mathbf{s}(k+1) \otimes \mathbf{v}_R^2 = \mathbf{0}$, or simply

$$\mathbf{v}_R^2 = \mathbf{s}(k+1) .$$

Similarly, if a point $\mathbf{v}_L = \frac{1}{2}\mathbf{v}_L^1 + \frac{1}{4}\mathbf{v}_L^2 + \boldsymbol{\varepsilon}_L$ has $C(\mathbf{v}_L) \in T_L$, then it means that the first two values that Algorithm C calculates for it are $I_1 = k$ and $I_2 = 2^n - 1$. Thus, necessarily,

$$J(\mathbf{v}_R^1) = k$$

which is the same as

$$\mathbf{v}_R^1 = G(k) .$$

Since the first values of the other objects computed by Algorithm C are $\mathbf{s}_L^1 = \mathbf{s}(k)$ and $\boldsymbol{P}_L^1 = \boldsymbol{P}(k)$, the other necessary condition for $\mathbf{v}_L = \frac{1}{2}\mathbf{v}_L^1 + \frac{1}{4}\mathbf{v}_L^2 + \boldsymbol{\varepsilon}$ to be mapped into T_L is:

$$2^n - 1 = I_2 = J(\boldsymbol{P}_L^1 \cdot (\mathbf{s}_L^1 \otimes \mathbf{v}_L^2)) = J(\boldsymbol{P}(k) \cdot (\mathbf{s}(k) \otimes \mathbf{v}_L^2)) .$$

Since $G(2^n - 1) = \mathbf{e}_n$, this is equivalent to

$$\mathbf{v}_L^2 = \mathbf{s}(k) \otimes (\boldsymbol{P}(k) \cdot \mathbf{e}_n) = \mathbf{s}(k) \otimes \mathbf{e}_{i(k)}$$

(where $\mathbf{e}_j$ denotes a unit vector with 1 in coordinate j). Obviously, the fact that $\mathbf{v}_L^1 = G(k)$ and $\mathbf{v}_R^1 = G(k+1)$ implies that points mapped into T_L and T_R must belong, respectively, to adjacent members of $\mathcal{S}(1)$, but in order to prove that they belong to adjacent members of $\mathcal{S}(2)$ we have to rely on $\mathbf{v}_L^2$ and $\mathbf{v}_R^2$. In fact, it suffices to prove that the latter always differ in one coordinate. By Fact 2.4,

1. if k is odd then

$$\mathbf{v}_R^2 = \mathbf{s}(k+1) = \mathbf{s}(k)$$

and

$$\mathbf{v}_L^2 = \mathbf{s}(k) \otimes \mathbf{e}_{i(k)} ,$$

so $\mathbf{v}_R^2$ and $\mathbf{v}_L^2$ differ in precisely one coordinate, namely, $i(k)$;

2. if k is even, then

$$\mathbf{v}_R^2 = \mathbf{s}(k+1) = G(k)$$

and

$$\mathbf{v}_L^2 = \mathbf{s}(k) \otimes \mathbf{e}_{i(k)}$$

so

(a) if $k = 0$, then $\mathbf{v}_R^2 = \mathbf{0}$ and $\mathbf{v}_L^2 = \mathbf{e}_1$, so $\mathbf{v}_R^2$ and $\mathbf{v}_L^2$ differ in precisely one coordinate;

(b) if $k \equiv 0 \pmod 4$ and $k > 0$, then $\mathbf{s}(k) = G(k-1) + \mathbf{e}_1$, so

$$\mathbf{v}_L^2 = (G(k-1) + \mathbf{e}_1) \otimes \mathbf{e}_{i(k)} = G(k-2) \otimes \mathbf{e}_{i(k)} ,$$

whereas $\mathbf{v}_R^2 = G(k)$. But $G(k)$ and $G(k-2)$ differ only in coordinates 1 and $i(k-1)$ (Fact 2.6). Also $i(k) = i(k-1)$ (Fact 2.5). Thus, $\mathbf{v}_R^2$ and $\mathbf{v}_L^2$ differ in only one coordinate, namely, coordinate 1;

(c) if $k \equiv 2 \pmod 4$ and $k > 0$, then $\mathbf{s}(k) = G(k-1) - \mathbf{e}_1$, so

$$\mathbf{v}_L^2 = (G(k-1) - \mathbf{e}_1) \otimes \mathbf{e}_2 = G(k-2) \otimes \mathbf{e}_{i(k)}$$

hence, as in the previous case, $\mathbf{v}_R^2$ and $\mathbf{v}_L^2$ differ in only one coordinate, $i(k)$.

Finally, consider the inductive step. Given $S \in \mathcal{S}(m)$, where $m > 1$, denote by R the member of $\mathcal{S}(1)$ that contains S. By the fractal formula, the induction hypothesis applies to the hierarchical structure of sub-cubes of R, and that implies our claim. ∎

Theorem 2.9 *For every two points* $\mathbf{p}, \mathbf{q}$ *in* $[0,1)^n$,

$$\|\mathbf{p} - \mathbf{q}\|_\infty \leq 2\,|C(\mathbf{p}) - C(\mathbf{q})|^{1/n} .$$

Proof: Given $\mathbf{p}, \mathbf{q} \in [0,1)^n$, denote by m the number such that

$$2^{-mn} \leq |C(\mathbf{p}) - C(\mathbf{q})| < 2^{-(m-1)n} .$$

Without loss of generality, suppose $C(\mathbf{q}) < C(\mathbf{p})$. Since the $2^{-mn} \leq C(\mathbf{p}) - C(\mathbf{q})$, and since each member of $\mathcal{S}(m)$ is mapped by C into a half open interval of length 2^{-mn}, it follows that the points $\mathbf{p}$ and $\mathbf{q}$ cannot belong to the same member of $\mathcal{S}(m)$. On the other hand, since $C(\mathbf{p}) - C(\mathbf{q}) < 2^{-(m-1)n}$, there exists a k, $1 \leq k \leq 2^{(m-1)n} - 1$, such that

$$(k-1)\,2^{-(m-1)n} \;\leq\; C(\mathbf{q}) \;<\; C(\mathbf{p}) \;<\; (k+1)\,2^{-(m-1)n}) .$$

If either $C(\mathbf{p}) < k\,2^{-(m-1)n}$ or $C(\mathbf{q}) \geq k\,2^{-(m-1)n}$, then both $\mathbf{p}$ and $\mathbf{q}$ belong to the same member of $\mathcal{S}(m)$ and hence

$$\|\mathbf{p} - \mathbf{q}\|_\infty < 2^{-m} = (2^{-mn})^{1/n} \leq |C(\mathbf{p}) - C(\mathbf{q})|^{1/n} .$$

Otherwise,

$$C(\mathbf{q}) < k\,2^{-(m-1)n} \leq C(\mathbf{p}) .$$

In this case $\mathbf{p}$ and $\mathbf{q}$ belong to distinct members of $\mathcal{S}(m)$ which are mapped under C into consecutive intervals $[(k-1)\,2^{-mn}, k\,2^{-mn})$ and $[k\,2^{-mn}, (k+1)\,2^{-mn})$. By Lemma 2.8, these members of $\mathcal{S}(m)$ must be adjacent. Thus,

$$\|\mathbf{p} - \mathbf{q}\|_\infty < 2 \cdot 2^{-m} \leq 2\,|C(\mathbf{p}) - C(\mathbf{q})|^{1/n} .$$

∎

3 Implementation of multi-orderings in a relational database

3.1 A search based on a single one-dimensional encoding

Suppose $C : [0,1)^n \to [0,1)$ is a mapping with the following characteristics: (i) C is one-to-one, (ii) the inverse mapping $C^{-1} : \mathcal{I} \to [0,1)^n$ (where $\mathcal{I} \subseteq [0,1)$ is the image of C) is continuous. Since C is one-to-one into an interval, it defines an order on $[0,1)^n$:

$$\forall \mathbf{x}, \mathbf{y} \in [0,1)^n \qquad \mathbf{x} \preceq_C \mathbf{y} \Leftrightarrow C(\mathbf{x}) \le C(\mathbf{y})$$

Let

$$\mathcal{P} = \{\mathbf{p}_1, \dots, \mathbf{p}_N\} \subset [0,1)^n \,.$$

The linear ordering can be implemented as a simple relation R with only two attributes: *point-id* and *value*. Each $\mathbf{p}_i$ has one entry in R where *point-id* equals i and *value* equals $C(\mathbf{p}_i)$.

A query consists of two components: a point $\mathbf{q} \in [0,1)^n$ and a number $k \in \mathbb{N}$. An exact answer to the query consists a set $\mathcal{O} \subset \mathcal{P}$ of size k so that

$$\forall \mathbf{x} \in \mathcal{O} \;\; \forall \mathbf{y} \in (\mathcal{P} \setminus \mathcal{O}) \qquad \|\mathbf{x} - \mathbf{q}\| \le \|\mathbf{y} - \mathbf{q}\|$$

We employ the relation R for getting a set of candidates from $\mathcal{P}$. For every $\delta > 0$ define $R(\delta; \mathbf{q}) \subseteq \mathcal{P}$ as

$$R(\delta; \mathbf{q}) = \{\mathbf{p} \in \mathcal{P} \mid |C(\mathbf{p}) - C(\mathbf{q}| \le \delta\}$$

Since C^{-1} is continuous, the points in $R(\delta)$ are likely to be close in the space $[0,1)^n$. A good implementation for one such index in a conventional DBMS is to build a B-tree index in attribute *value*.

3.2 Employing a multitude of one-dimensional encodings

The definition of C lends itself to a multitude of other possible orderings. First, note that the dimensions $1, \dots, n$ can be arbitrarily permuted and each permutation defines a different ordering. Sub-cubes that are distant relative to one permutation may be close relative to another one. However, the sub-cubes hierarchy itself is the same for all the permutations. In order to obtain further orderings that do not share the same hierarchy, we define a new ordering based on C by using a translation vector $\boldsymbol{\varepsilon} \in [0, \frac{1}{3}]^n$. Define a mapping $C_{\boldsymbol{\varepsilon}} : [0,1)^n \to [0,1)$ by

$$C_{\boldsymbol{\varepsilon}}(\mathbf{x}) = C\left(\tfrac{3}{4}(\mathbf{x} + \boldsymbol{\varepsilon})\right) \,.$$

For each ε, define a candidate set by

$$R(\delta, \varepsilon; \mathbf{q}) = \{\mathbf{p} \in \mathcal{P} \mid |C_{\boldsymbol{\varepsilon}}(\mathbf{p}) - C_{\boldsymbol{\varepsilon}}(\mathbf{q}| \leq \delta\}$$

We employ a fixed number of orderings, using a positive parameter δ as follows:

1. Pick ℓ vectors $\boldsymbol{\varepsilon}_1, \ldots, \boldsymbol{\varepsilon}_\ell$.
2. For each ordering define a relation R_i similar to R, based on $C_{\boldsymbol{\varepsilon}_i}$ instead of C.
3. For a query $(\mathbf{q}, k)$ return a candidate set

$$\mathcal{R}(\delta; \mathbf{q}) = \bigcup_{i=1}^{l} R(\delta, \boldsymbol{\varepsilon}_i; \mathbf{q})$$

4. For each point in $\mathcal{R}(\delta; \mathbf{q})$, calculate its distance from $\mathbf{q}$ and finally return the k closest points.

3.3 Implementation of the query in a conventional DBMS

Suppose we work with data set $\mathcal{P}$ and have defined orderings based on ℓ one-dimensional codes $C_1, \ldots, C_\ell$. We create a *single* relation[2] R with attributes *index-id, point-id, value.* For each $\mathbf{p}_i \in \mathcal{P}$ and for each $1 \leq j \leq l$ we include an entry $(j, i, C_{\boldsymbol{\varepsilon}_j}(\mathbf{p}_i))$ in R (*i.e.*, *index-id* equals j, *point-id* equals i, and *value* equals $C_{\boldsymbol{\varepsilon}_j}(\mathbf{p}_i)$. We create a B-tree index on the combination of attributes < *index-id,value* >. This means that two entries in the tree are compared by their *index-id* and only if their *index-id*'s are equal, the comparison is resolved based on *value*.

A query is implemented as follows. Suppose $\mathbf{q}$ is a query point. We create a relation Q with attributes *index-id, value.* For each j, $j = 1, \ldots, \ell$, we put one tuple in the relation Q, where *index-id* equals j and *value* equals $C_{\boldsymbol{\varepsilon}_j}(\mathbf{q})$. We choose a small d and perform a query in SQL as follows:

```
SELECT DISTINCT point-id
FROM   R,Q
WHERE  R.index-id = Q.index-id
  AND  R.value <= Q.value + d
  AND  R.value >= Q.value - d
```

This SQL query returns $\mathcal{R}(d; \mathbf{q})$. If we are satisfied with this result, we can find the k nearest neighbors from this candidate set. Otherwise, we can get a larger candidate set by increasing d. The new candidate set will be a superset of $\mathcal{R}(d; \mathbf{q})$.

[2]We are indebted to Bruce Lindsay for this idea.

4 Obtaining an exact result

As indicated above, the set of candidates (obtained as the union of the sets of candidates provided by the various orderings) is not guaranteed to contain all the k nearest neighbors. Our experiments show that in many cases a very good approximation is achieved, either in terms of the overlap with the set of the true k nearest neighbors, or in terms of the actual distances between the query point and the reported neighbors, compared to the distances between the query point and the true k nearest neighbors. In this section we propose a method of finding the true k nearest neighbors. This extended feature would be used only if it is essential to get the exact result. If an exact output is desired, the search has to be extended and a stopping rule has to be developed so that when the algorithm stops it is guaranteed to have found the exact result. The method works as follows. Candidates are obtained by enlarging the scanned intervals within each of the orderings. At the same time, lower bounds are calculated for various cubes, giving the minimum distance between any point in the cube and the query point. The algorithm stops the search as soon as the available bounds imply that no unchecked database point is closer to the query than any of the current k nearest neighbors. The sub-cubes are stored in the same trees that store the linear orders.

4.1 Distances to boxes

If $\mathbf{q} = (q_1, \ldots, q_n)$ is a query point and

$$B = B(\mathbf{a}, \mathbf{b}) = \{\mathbf{x} \in \mathbb{R}^n \mid a_i \le x_i \le b_i,\ i = 1, \ldots, n\}$$

$(\mathbf{a}, \mathbf{b} \in \mathbb{R}^n)$ is a rectangular box, then the Euclidean distance between $\mathbf{q}$ and B can be computed as follows. The nearest point of the cube is obtained by minimizing $\sum_{i=1}^n (q_i - x_i)^2$ subject to $a_i \le x_i \le b_i$ $(i = 1, \ldots, n)$. The objective function is separable, so we can describe the optimal solution as follows. (i) If $q_i \le a_i$, then let $d_i = a_i - q_i$. (ii) If $a_i \le q_i \le b_i$, then let $d_i = 0$. (iii) If $q_i \ge b_i$, then let $d_i = q_i - b_i$. It is easy to see that the distance between $\mathbf{q}$ and B is given by $\delta(\mathbf{q}, B) = ||\mathbf{d}||$, where $\mathbf{d} = (d_1, \ldots, d_n)$.

Suppose for a given $\mathbf{q}$ we are seeking the k points in the database that are nearest to $\mathbf{q}$. Suppose we have already located the database points $\mathbf{p}^1, \ldots, \mathbf{p}^k$ as candidates for the k nearest neighbors, and we have

$$||\mathbf{q} - \mathbf{p}^1|| < \cdots < ||\mathbf{q} - \mathbf{p}^k|| \, .$$

Obviously, $||\mathbf{q} - \mathbf{p}_k||$ is an upper bound on the distance to the kth nearest neighbor. Thus, if some box B contains database points, and $\delta(\mathbf{q}, B) > ||\mathbf{q} - \mathbf{p}_k||$, then there is no need to search B since none of its points can be among the k nearest neighbors. We show below that such bounds become useful in case an exact output is desired. We maintain each of our linear orderings in a tree where nodes store information about bounding boxes, and the ordering lends itself to creating good boxes.

4.2 Trees

Let $\mathbf{v}^1, \ldots, \mathbf{v}^N$ be all the database points. Here we consider one ordering induced by a code C, so let us assume without loss of generality that

$$C(\mathbf{v}^1) < \cdots < C(\mathbf{v}^N) .$$

For each i ($i = 1, \ldots, N$), let us use the representation

$$\mathbf{v}^i = \sum_{j=1}^{m} 2^{-j} \mathbf{v}^{ij} .$$

Thus, the components $\mathbf{v}^{i1}, \ldots, \mathbf{v}^{N1}$ determine the first level of the ordering. Recall that $\mathcal{S}(1)$ denotes the collection of the sub-cubes at the first level of the hierarchy, so it has 2^n members, each with a volume of 2^{-n}. Since N is expected to be much smaller than 2^n, most of the members of $\mathcal{S}(1)$ will not contain any database point. Those members of $\mathcal{S}(1)$ that do contain such points can be stored at the leaves of a balanced tree according to the order which is induced on them by the codes $J(\mathbf{v}i1)$ (the inverse of the Gray code). Due to the nature of this code, if two sub-cubes are close in the ordering, then there exists a relatively small box that contains their union. For example, any two consecutive members form a box of volume 2^{-n+1}, and any two members that are at distance 2 in the ordering are contained in a box of volume 2^{-n+2}. In general, if the two children of a node in the tree hold the boxes $B(\mathbf{a}^1, \mathbf{b}^1)$ and $B(\mathbf{a}^2, \mathbf{b}^2)$, then the parent holds the box $B(\mathbf{a}, \mathbf{b})$ where $a_i = \min(a_i^1, a_i^2)$ and $b_i = \max(b_i^1, b_i^2)$. Thus, relying on the currently known upper bounds on the distance to the kth nearest neighbor, the algorithm can decide not to proceed into the subtree rooted at the node, if the box associated with it is sufficiently distant from the query point.

4.3 The stopping criterion

During the search for the exact k nearest neighbors, the algorithm maintains the following information. First, there is a globla upper bound on the distance between the query point and the kth nearest neighbor. Second, in each tree there is the current interval in the corresponding ordering all of whose points have been checked. Moreover, in each such tree some of the nodes are marked as ones whose subtrees should not be checked. The intervals are expanded repeatedly, and the upper bound is updated. At the same time, lower bounds are updated for a collection of nodes whose subtrees cover the unchecked data points. In each tree, the number of such subtrees at any time does not exceed $O(\log n)$ where n is the number of data points. The search may stop when in *one* tree all the lower bounds of subtrees that cover the unchecked points are greater than the global upper bound.

5 Experimental Setup

The experminents were conducted with five data sets, in the same way for all the sets. For each data set the following steps were performed:

1. Preparation.
2. Creating index structures.
3. Performing queries and analyzing the results (For each index structure).

5.1 Preparation

During the preparation phase, the data set is converted into an easy to use format, and everything that does not depend on the indexing scheme is computed. This consists of the following steps:

1. Transform the data set. The input is converted into unsigned integers so that the Euclidian metric still applies for finding neighbors.
2. Find minimal and maximal values in the dataset.
3. Choose 100 random points in the data set as query points.
4. For each query point: sort all the points in the data set according to the distance to the query point. Make a table that consists of the id of each point and its distance to the query point.
5. Make a principle component analysis of the data set. Identify a transformation that reduces dimensionality and keeps as much variance as possible.
6. Make a transformed data set by projecting the points into the subspace defined by the first n eigenvectors of the principle component analysis. n is chosen according to the cumulative variance of the data set projected into the subspace.

5.2 Creating Index Structures

The results presented in this document are about two index structures that were created for each data set. Each structure uses n ordering mappings (where n is the dimensionality of the reduced data set described in the preparation phase). In one structure, the difference among the mappings is a permutation of the dimensions. We choose an initial random permutation and make n permutations from it using a round robin approach. We call this *RR* for short. The other structure uses a random permutation of dimensions and a random shift of the data set for each different mapping. We call this structure *RS* for short.

A permutation means that the data set is transformed before each ordering is created. The transformation is a permutation of the dimensions. A shift means that a vector is chosen (one per ordering) and the data set is moved by adding to each point the chosen random vector. Both transformations do not change the distances between points.

Each of the two structures consist of n orderings of the data set. Each such order is a list of point id's. Since the query is a point from the data set, we can find it in the list, and return m points to the left of it and m points to the right. This way, we do not need to save the key (the mapped point).

5.3 Queries and Results

For each of the two index structures we perform queries with the following parameters:

k. **The number of nearest neighbors requested for the query.** The final result of a query is a set of points. This parameter is the number of points in that set.

c. **The number of candidates returned by the index structure.** An index probe finds the query point in each of the n orderings. After that we return $2m$ points from each ordering by taking the m points to the left of the query point in the ordering, and the m points to its right. We have a multiset of size $2mn$. However, we remove duplicates from this set. After duplicate elimination we want a specific number of points (that is the parameter we use). To do this, we start with $m = 1$ and increment m until we have enough points (we may need to discard some of them). These points are the candidates. From these candidates we choose the ones that are the closest to the query point in Euclidian space. This is done by using the nearest neighbors information calculated for the query in the preparation phase.

The above process is repeated for all the 100 random queries. The number of nearest neighbors we request varies from 5 to 50. The number of candidates we allow vary from 400 to 1200. There are two main measurements of success that we tried:

Precent found. This is the percentage of true nearest neighbors that were found by the index.

Average distance. For each query we know the distance of the query point from all the points in the database (preparation phase). We find the median, and approximate a standard deviation from the median by the following method: if the data points are sorted according to distance, and the distances are $d(1) \leq d(2) \leq \cdots \leq d(N)$ then we define

$$stddev = \frac{1}{2}\left(d(5N/6) - d(N/6)\right)$$

Note that for a normal distribution this is a good estimator of the standard deviation. We chose this method because we want to eliminate the effect of outliers. Once these were found we measure the distance of a point by its distance from the median in multiples of *stddev*. We measure the average distance for the k nearest neighbors we found. We divide this by the average distance for the true k nearest neighbors. We present these numbers in percentages.

5.4 Presentation of Results

For each data set we present the following:

- A histogram of distances of all the data points from the first query, and a graph of those distances for the first 100 nearest neighbors. This gives us some idea about the kind of data set we have.
- Graphs that show the percent of nearest nighbors found vs. the distance ratio explained above. We present several graphs for different number of nearest neighbors k, for different number of candidates c, and for the two different index structures.
- Graphs showing for each structure and a specific c the five quartiles of percentage of points found over 100 queries. The X axis is k. For each k we have five points representing the quartiles for the results. The top curve represents the maximum (4'th quartile), the middle curve represents the median (2'nd quartile) and the bottom represents the minimum.
- Graphs showing for each structure and a specific c the 1'st and 3'rd quartiles of distance measurement, compared to the 1'st and 3'rd quartiles taken from the true nearest neighbors. The X axis varies over k. For each k we perform 100 queries, and the quartiles are taken from these 100 results. The top curve belongs to the 3'rd quartile of the true nearest neighbors. The one below it belongs to the 3'rd quartile for the index structure. The two bottom curves are similar for the 1'st quartile.

6 Color Data Set

Each point in the color data set is obtained from a histogram of colors in one image. The histogram is transformed using a linear transformation s.t. the Euclidian distance can be used on the transformed points. The properties of the data set are as follows:

- The number of points (images) is 13536.
- The number of dimensions (arity) is 256.
- The values of data points in each dimension are integers ranging from 0 to 9759006 (after using a shift transformation so the numbers are not negative).
- 94% of the variance is explained by a subspace of 64 dimensions. The dataset is projected into that subspace for the reduced dimensionality experiments.

Figure 1 shows the distribution of distances of database points from some fixed query point. Figure 2 shows the same case for the first 100 nearest neighbors. We see that the difference in distance among the 100 nearest neighbors is relatively small compared to their distance to the query point. This makes us believe that approximation schemes would be more accurate in terms of average distance than in terms of number of actual nearest neighbors found.

Figure 3 shows what happened with the reduced dimensionality data for 100 random queries, where the indexing scheme is RR. We have $c = 400$ and $k = 25$ (i.e., we look for 25 nearest neighbors and we check 400 candidates from the index structure). Figure 4 shows the same thing with the indexing scheme RS. Note that the queries are the same through all the figures in this section. It is very clear that shifting the data for each space ordering creates a much better approximation. We see that the average percent of points found jumps from about 50% for random permutations, to about 85% for random shifts. Also, the average distance ratio jumps from about 95% to about 99.5%.

Figure 5 shows the same thing for RS when the number of candidates returned by the index is c = 800. We see some improvement over the $c = 400$ case. However, we anticipate that as we increase c the result converges to the true nearest neighbors. Since we were very close to the optimal result, a large increase in c results in a small improvement only. Figure 6 shows the same thing when the number of candidates returned by the index is $c = 400$, and the number of nearest neighbors is $k = 10$.

Figure 7 is about the RR index with $c = 800$ and 100 random queries. For each k there are five points for the five quartiles of the percent found measurement. Figure 8 shows the same for the RS index. Note how much improvement there is in the bottom curve (minimal performance) compared to the RR scheme.

Figure 9 is about the RR index with $c = 400$ and 100 random queries. We see the the 1'st and 3'rd quartiles of distance measure for the index and the same for teh true nearest neighbors. Figure 10 shows the same for the RS index. Note that even when we check only

400 points from the index, the curves for the index and for the true nearest neighbors are almost identical.

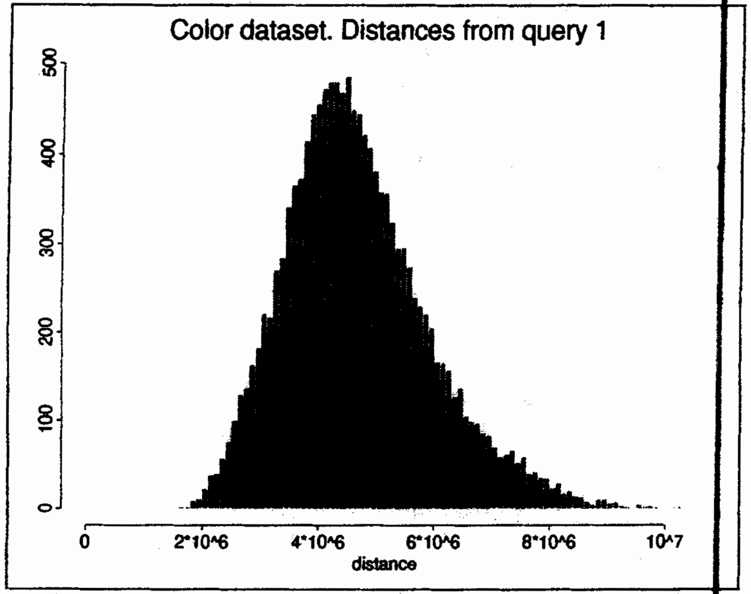

Figure 1: Distribution of distances of data points from query point number 1 (Color data set).

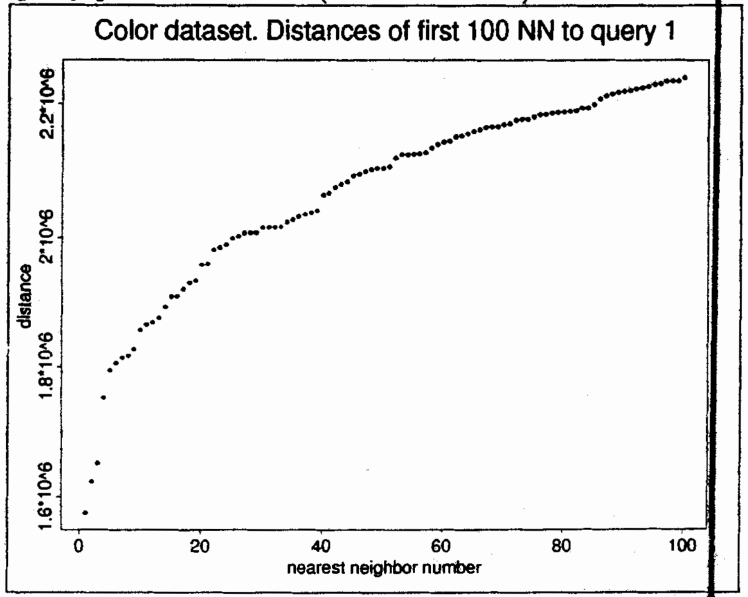

Figure 2: Distances of first 100 nearest neighbors from query point number 1 (Color data set).

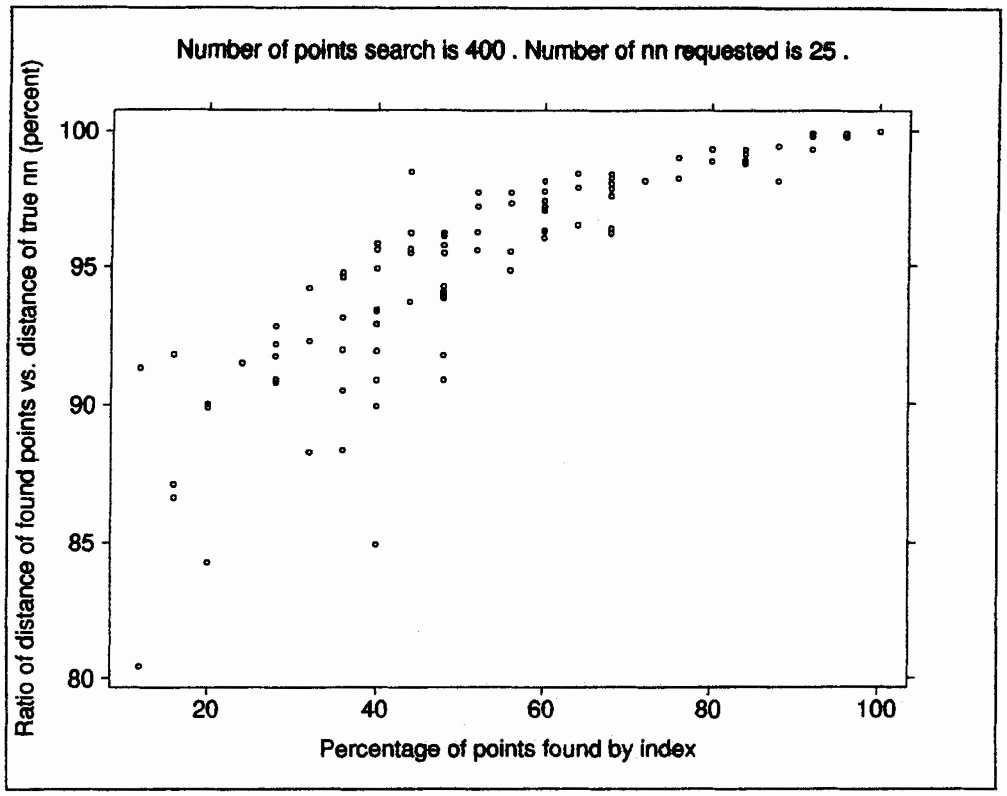

Figure 3: Distance and found ratio for 100 random queries. *RR* scheme.

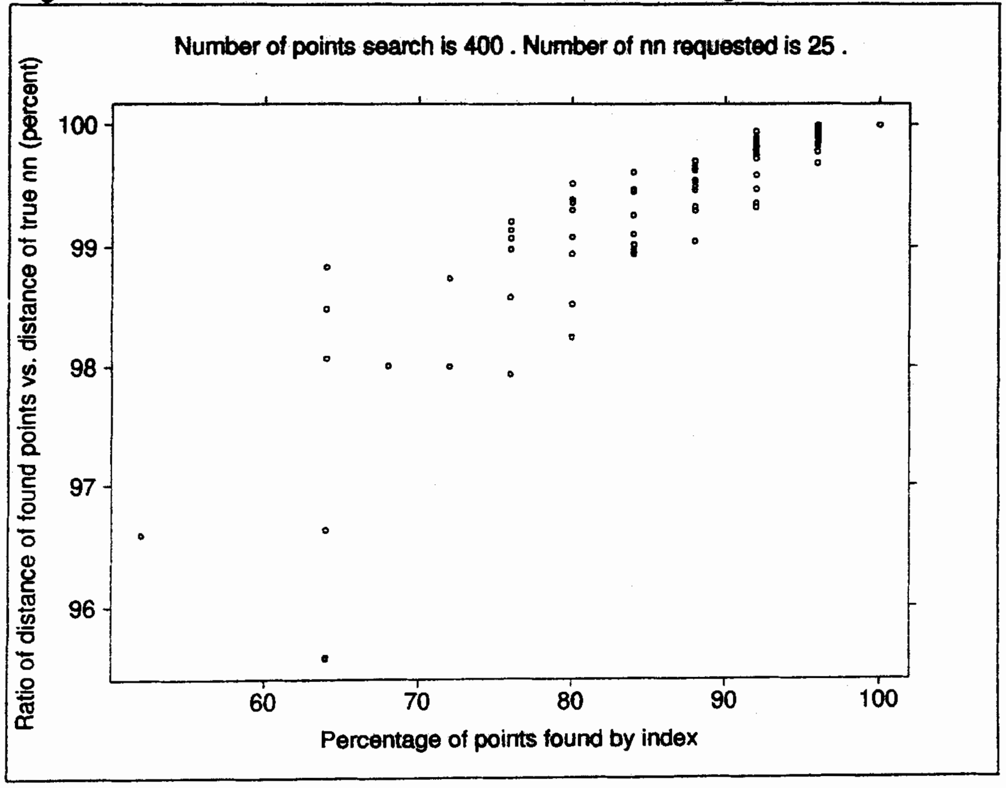

Figure 4: Distance and found ratio for 100 random queries. *RS* scheme.

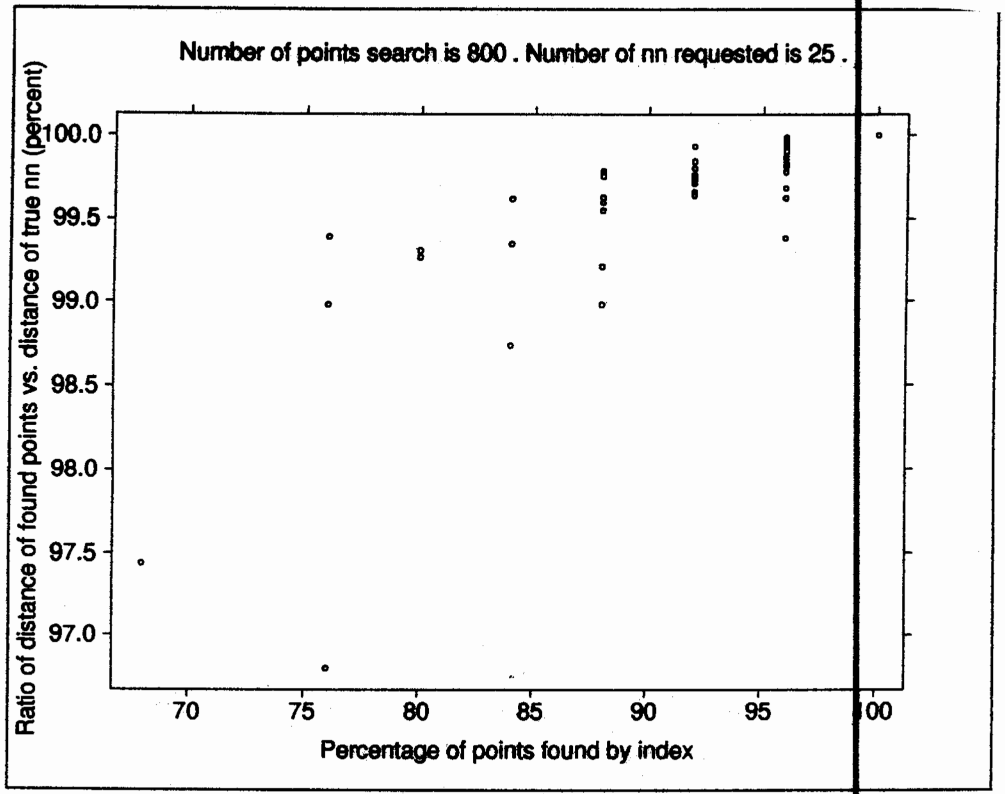

Figure 5: Distance and found ratio for 100 random queries. *RS* scheme.

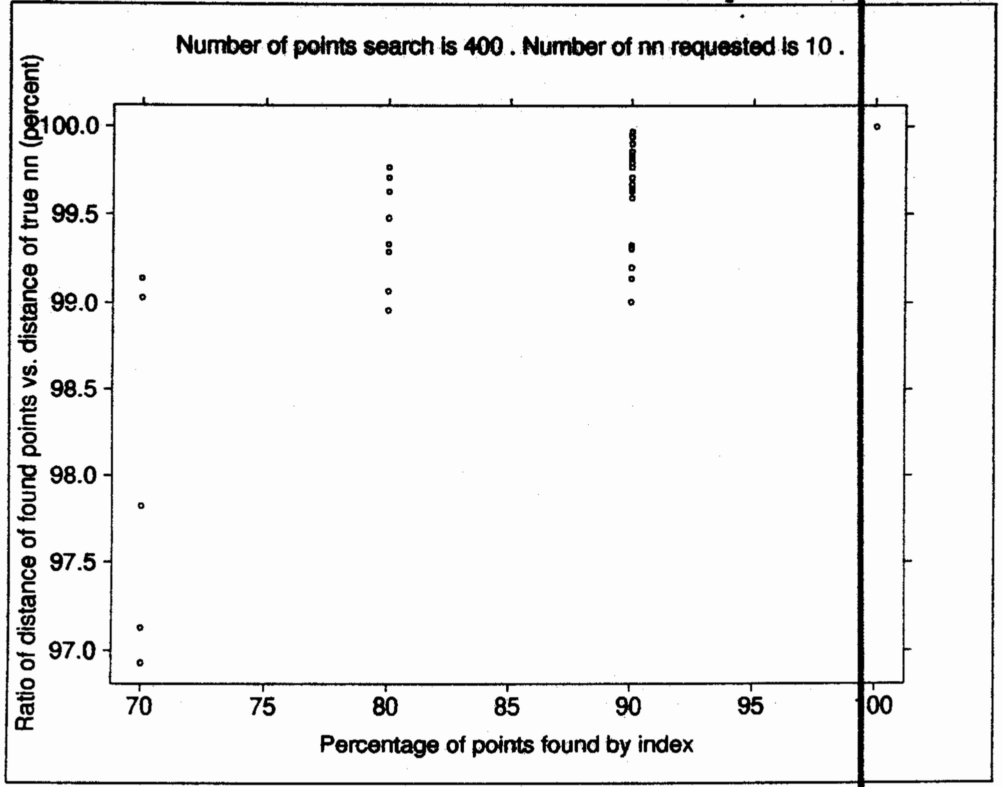

Figure 6: Distance and found ratio for 100 random queries. *RS* scheme.

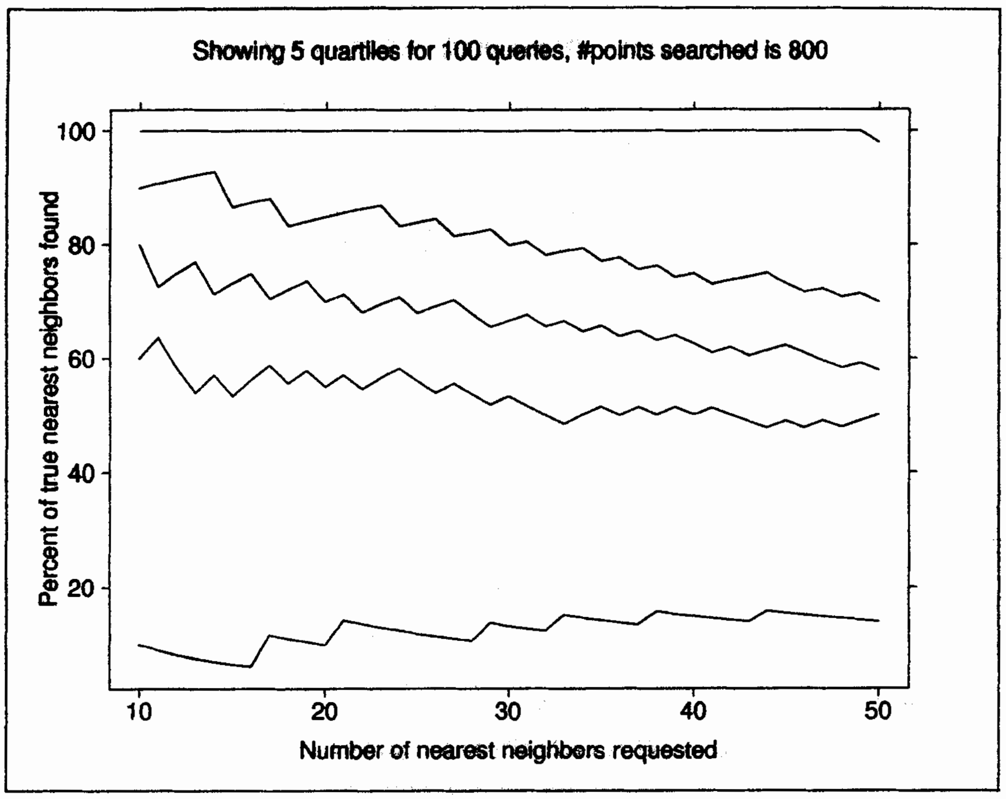

Figure 7: Five quartiles of percentage of points found over 100 queries. *RR* scheme.

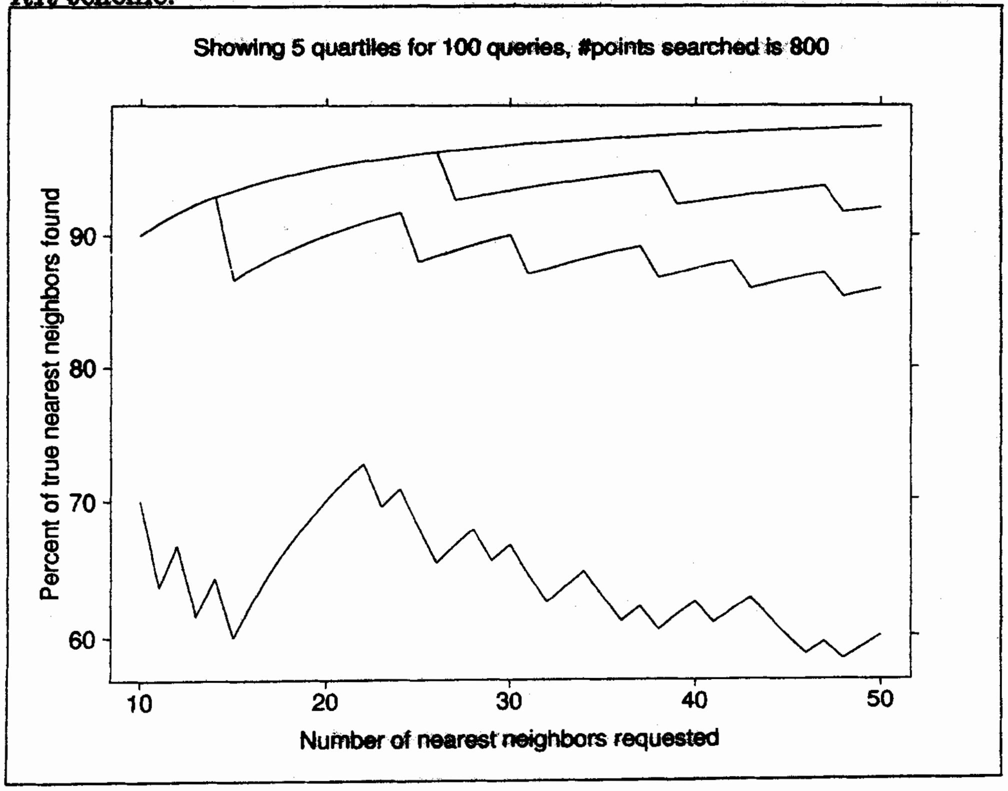

Figure 8: Same as Figure 7 for *RS* scheme.

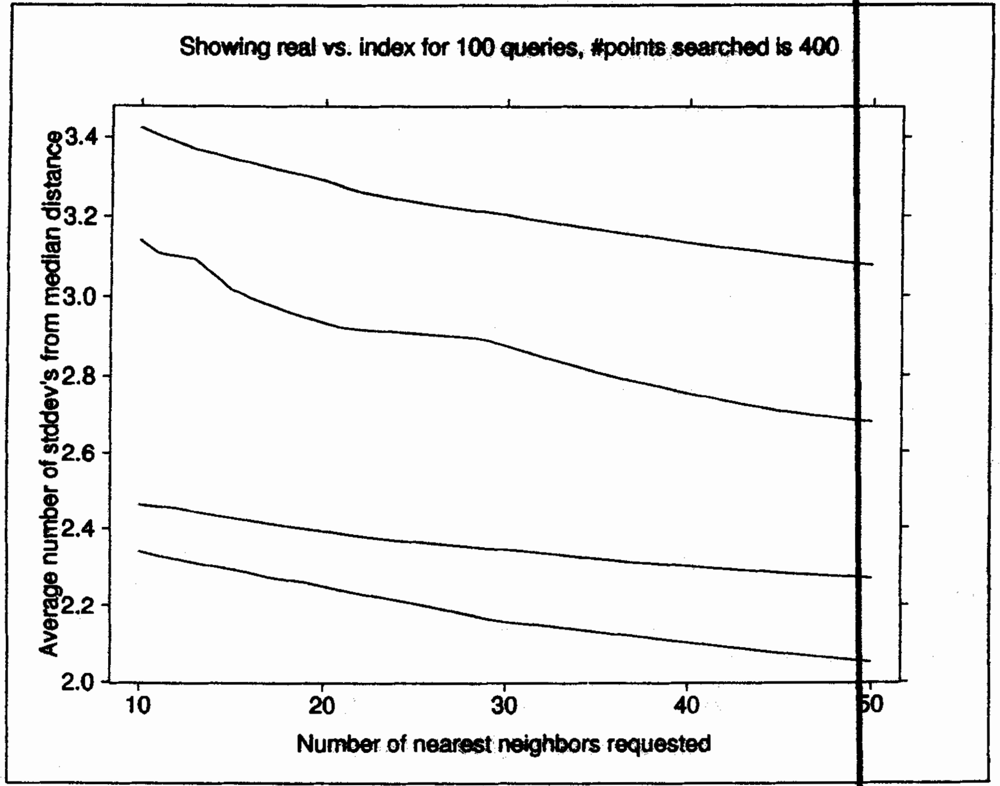

Figure 9: 1'st and 3'rd quartiles for true nearest neighbors and index structures. *RR* scheme. $c = 400$.

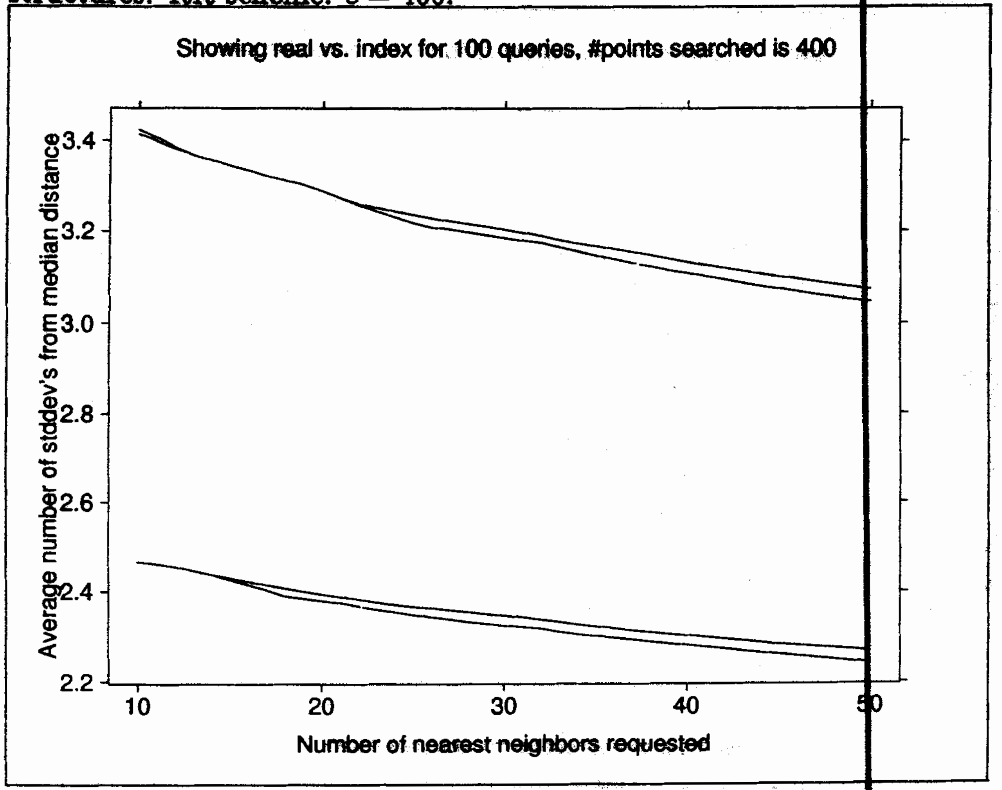

Figure 10: Same as Figure 9 for *RS* scheme.

7 Random Data Set

The data set is constructed from a random number generator. Each element in each tuple is a sum of 100 uniformly distributed numbers in $[0, 100]$. Each element is close to a normal distribution.

- The number of points is 20000.
- The number of dimensions (arity) is 64.
- The values of data points in each dimension are integers ranging from 1977 to 8289.
- The dimensionality is not reduced by principle components.

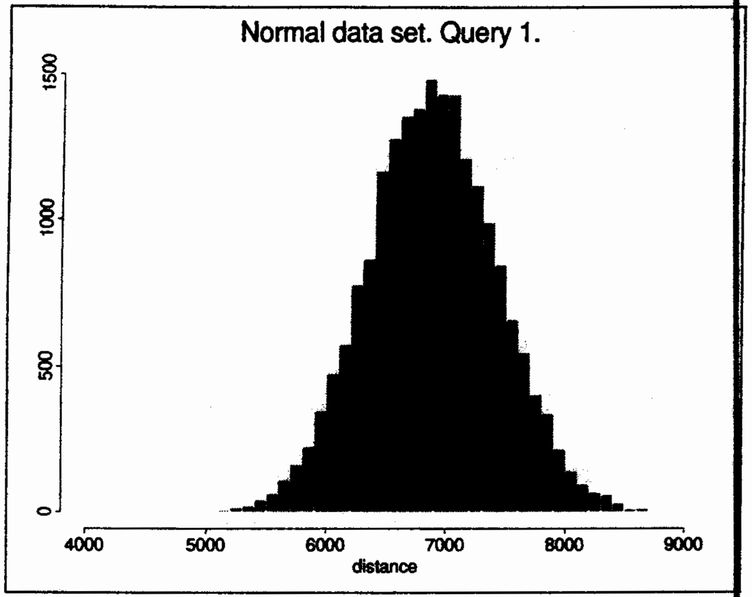

Figure 11: Distribution of distances of data points from query point number 1 (Normal data set).

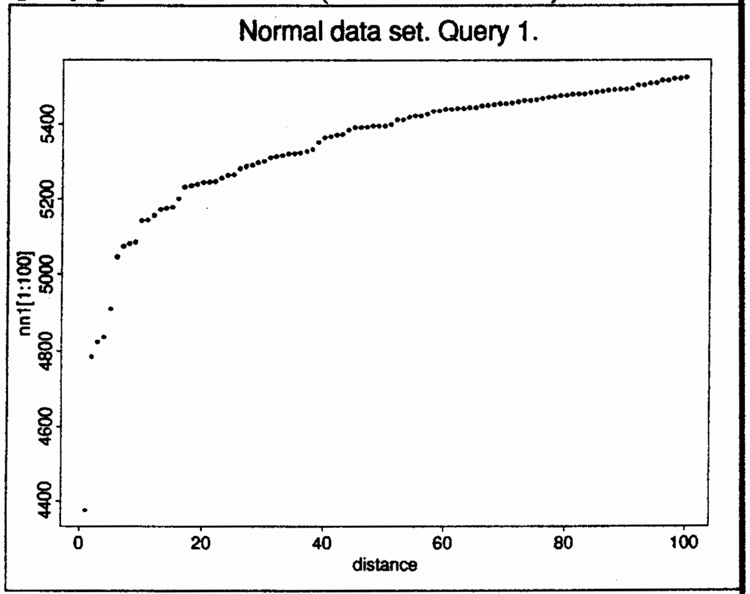

Figure 12: Distances of first 100 nearest neighbors from query point number 1 (Normal data set).

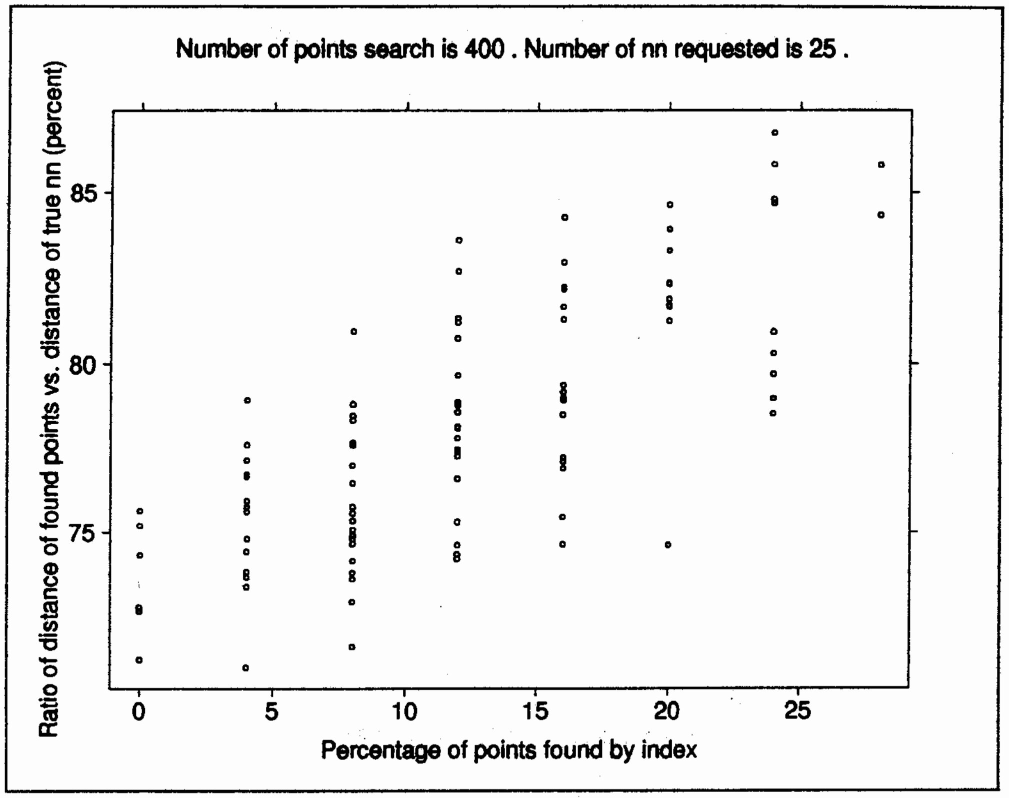

Figure 13: Distance and found ratio for 100 random queries. *RR* scheme.

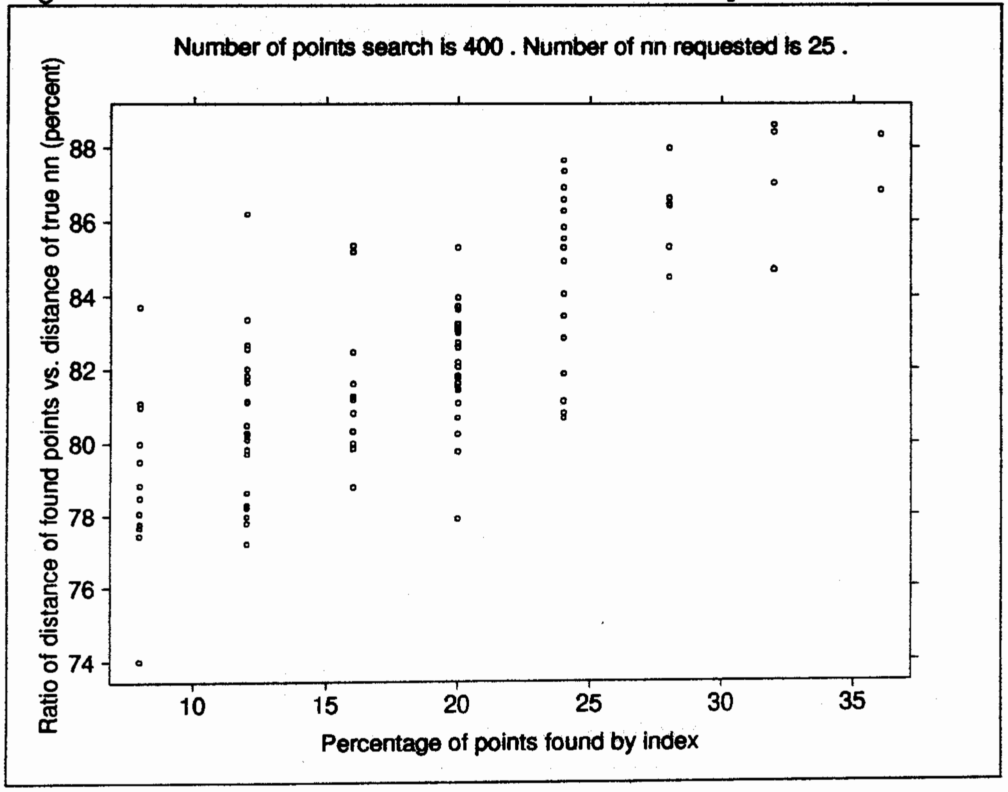

Figure 14: Distance and found ratio for 100 random queries. *RS* scheme.

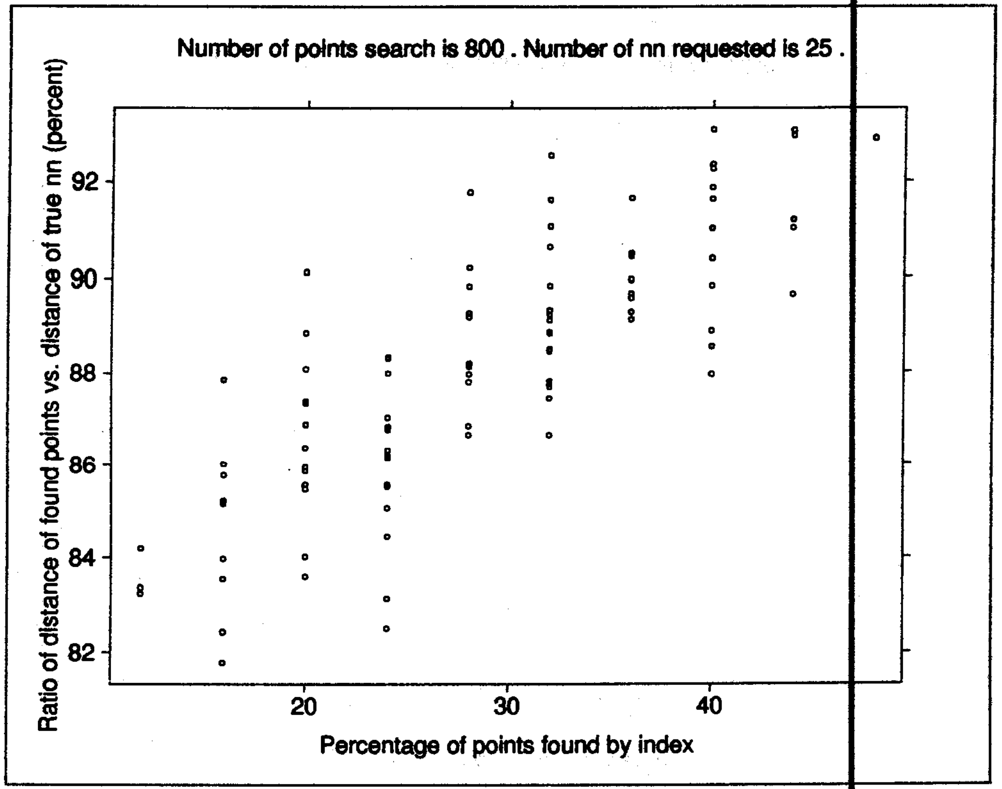

Figure 15: Distance and found ratio for 100 random queries. *RS* scheme.

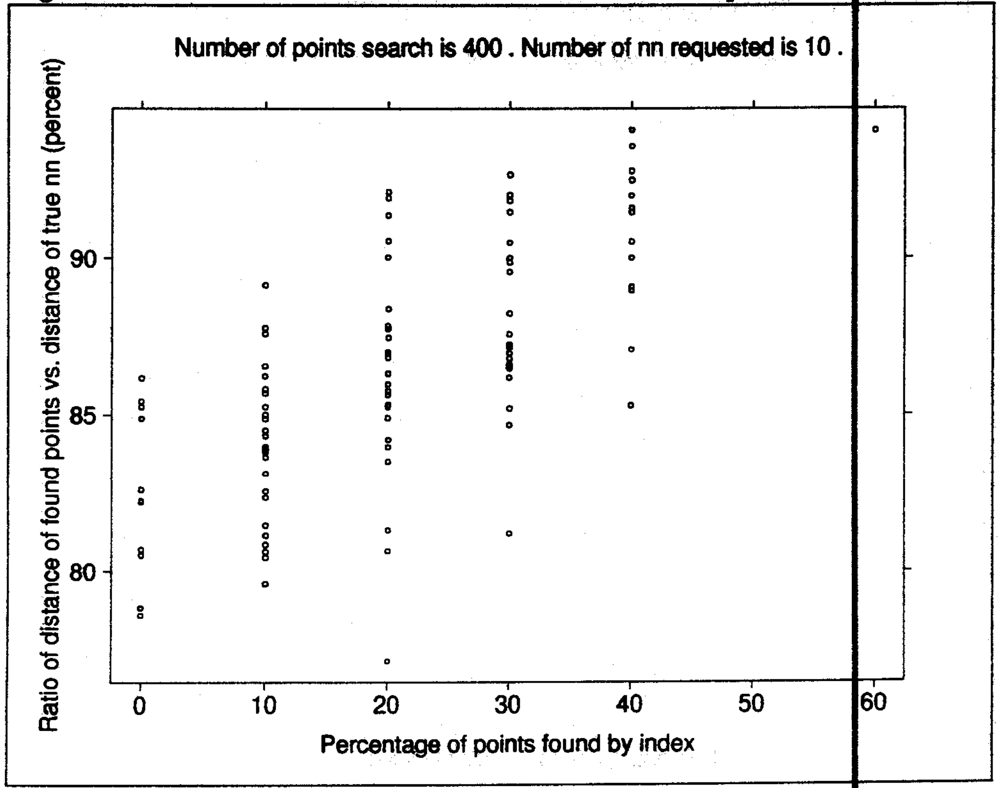

Figure 16: Distance and found ratio for 100 random queries. *RS* scheme.

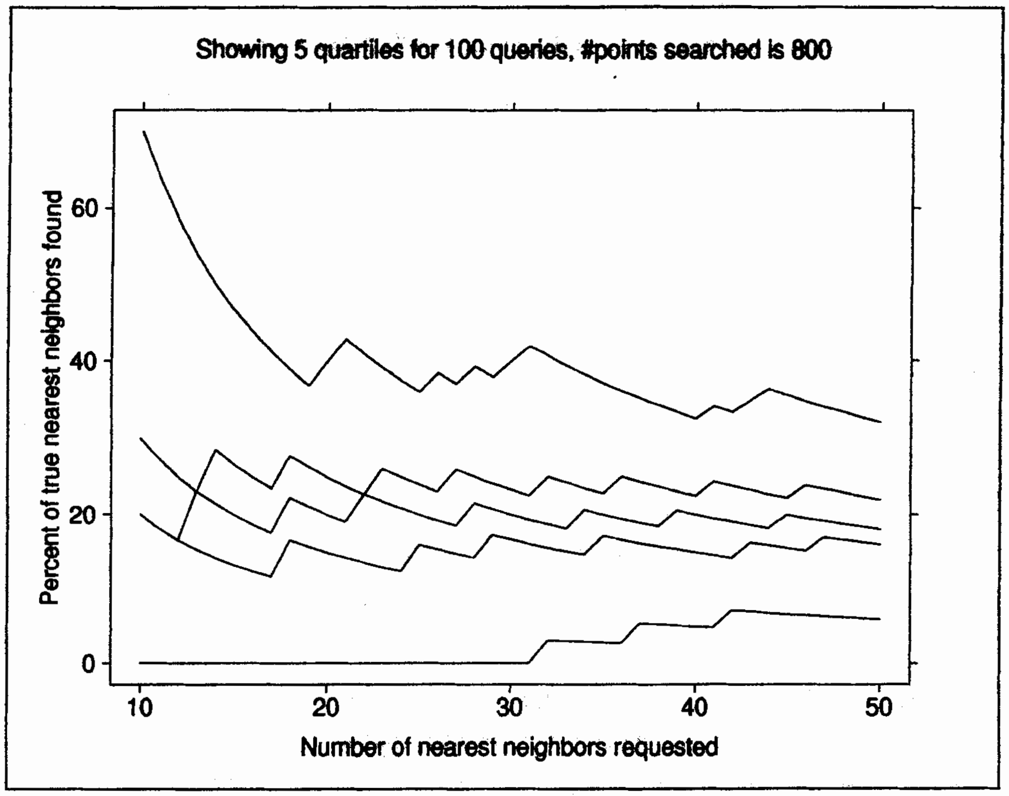

Figure 17: Five quartiles of percentage of points found over 100 queries. *RR* scheme.

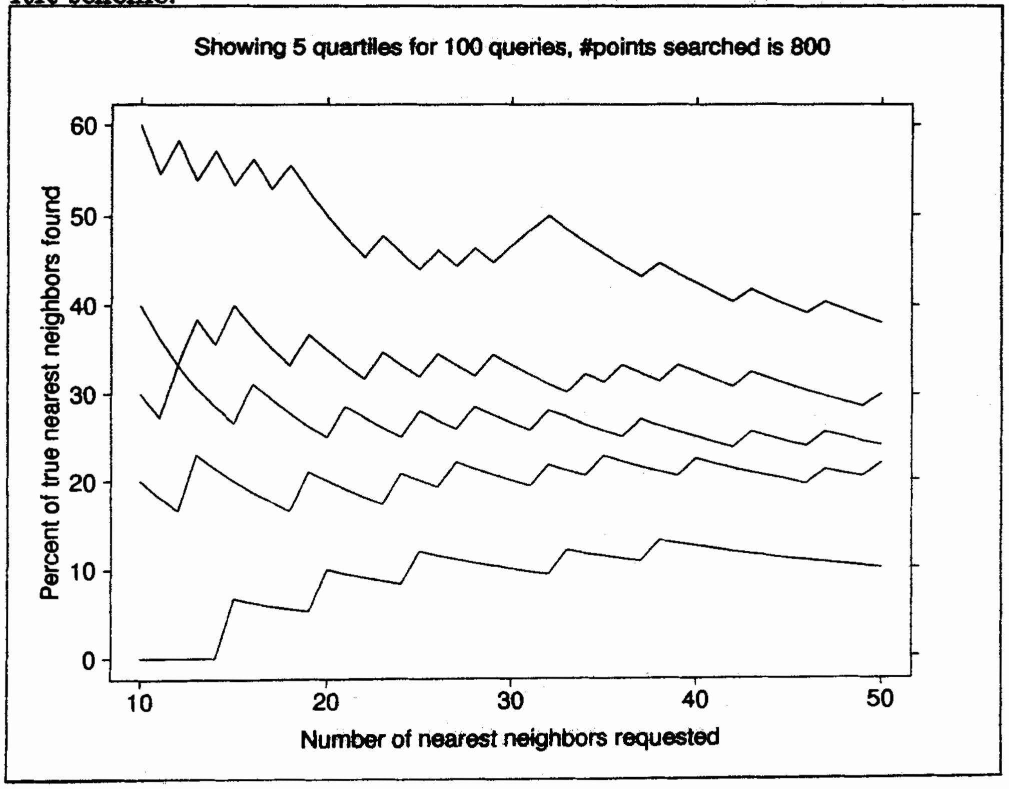

Figure 18: Same as Figure 17 for *RS* scheme.

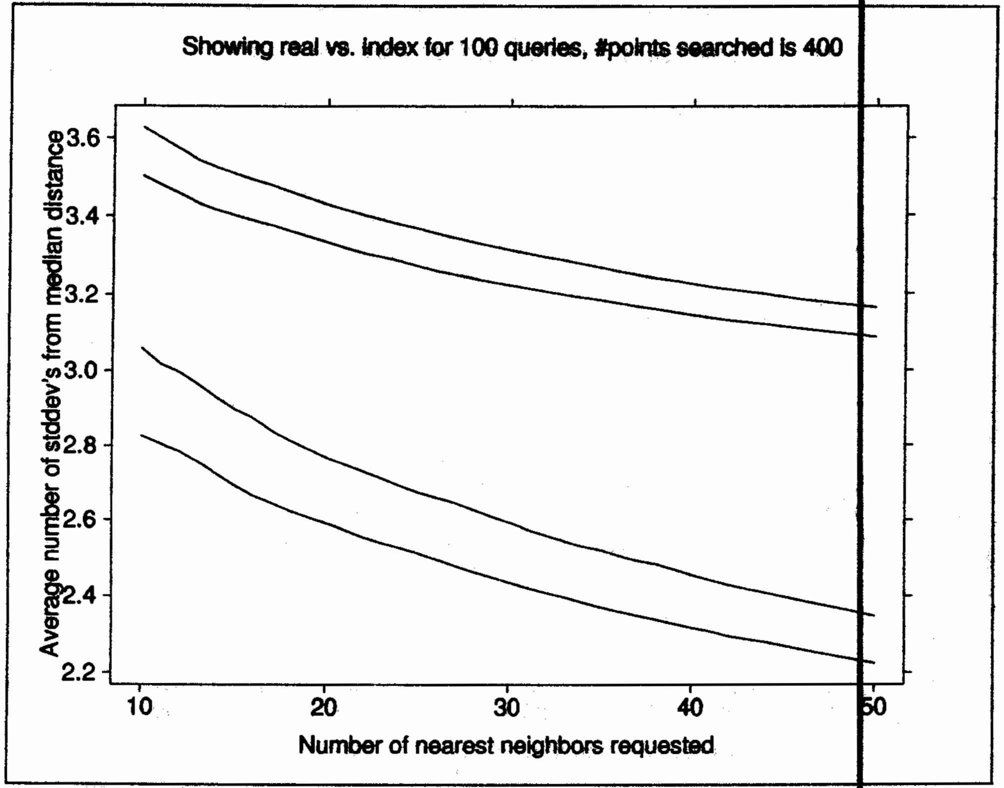

Figure 19: 1'st and 3'rd quartiles for true nearest neighbors and index structures. *RR* scheme. $c = 400$.

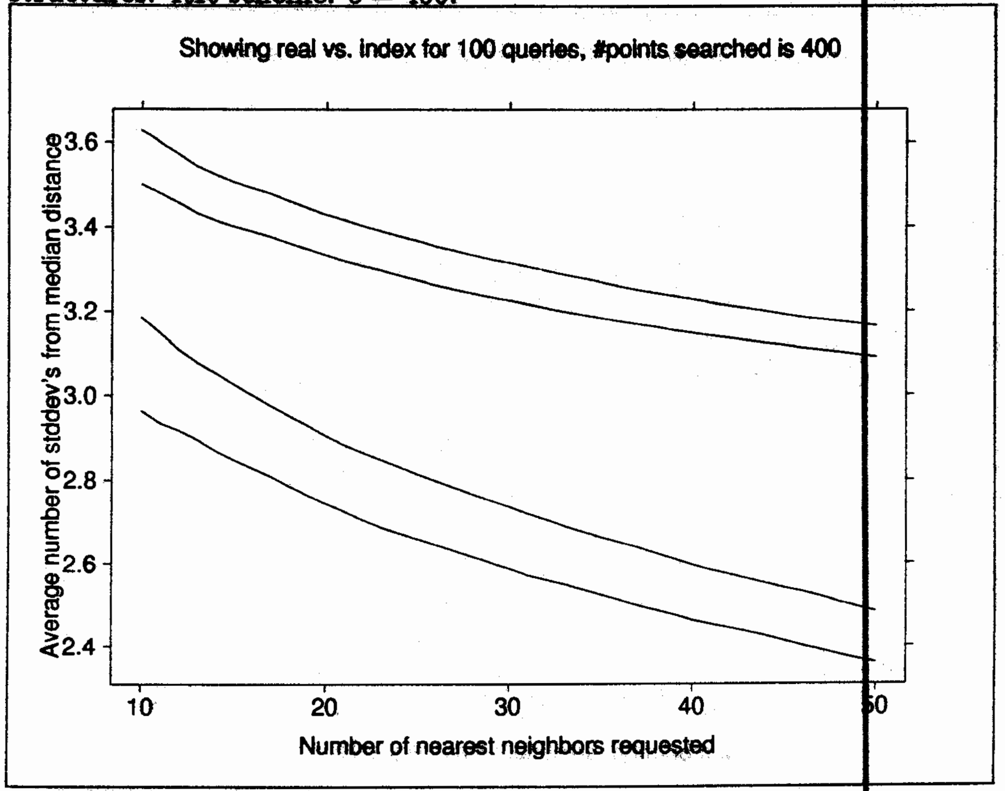

Figure 20: Same as Figure 19 for *RS* scheme.

8 Draw Data Set

The Draw data set is was taken from QBIC. It was shifted and scaled so it will be unsigned integers.

- The number of points is 13536.
- The number of dimensions (arity) is 324.
- The values of data points in each dimension are integers ranging from 0 to 85556.
- The dimensionality is reduced to 128 while keeping 95% of the variance.

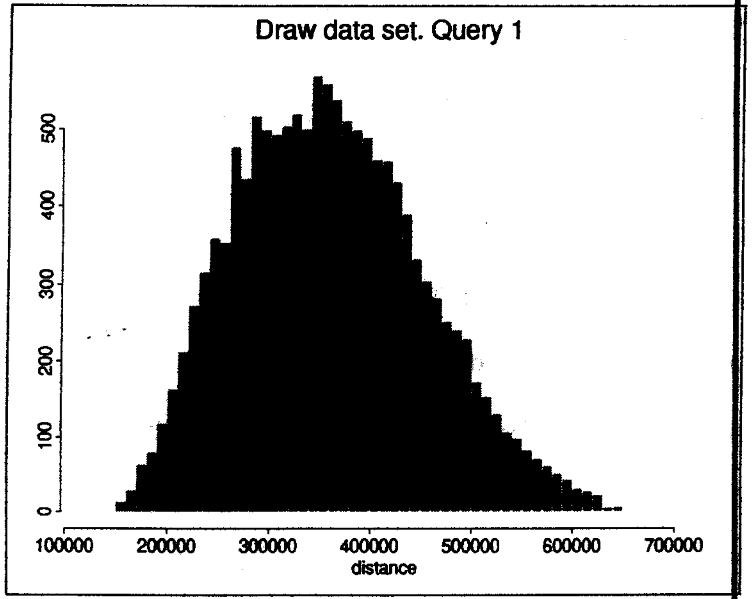

Figure 21: Distribution of distances of data points fro
query point number 1 (Draw data set).

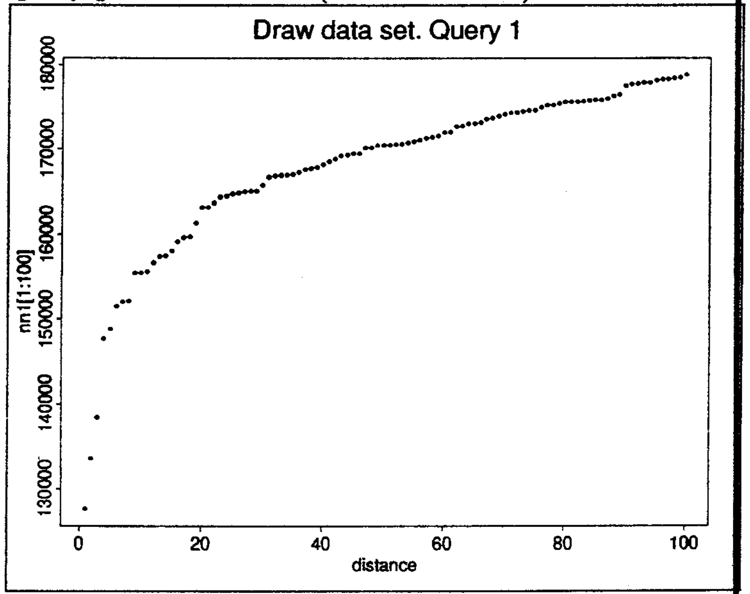

Figure 22: Distances of first 100 nearest neighbors fro
query point number 1 (Draw data set).

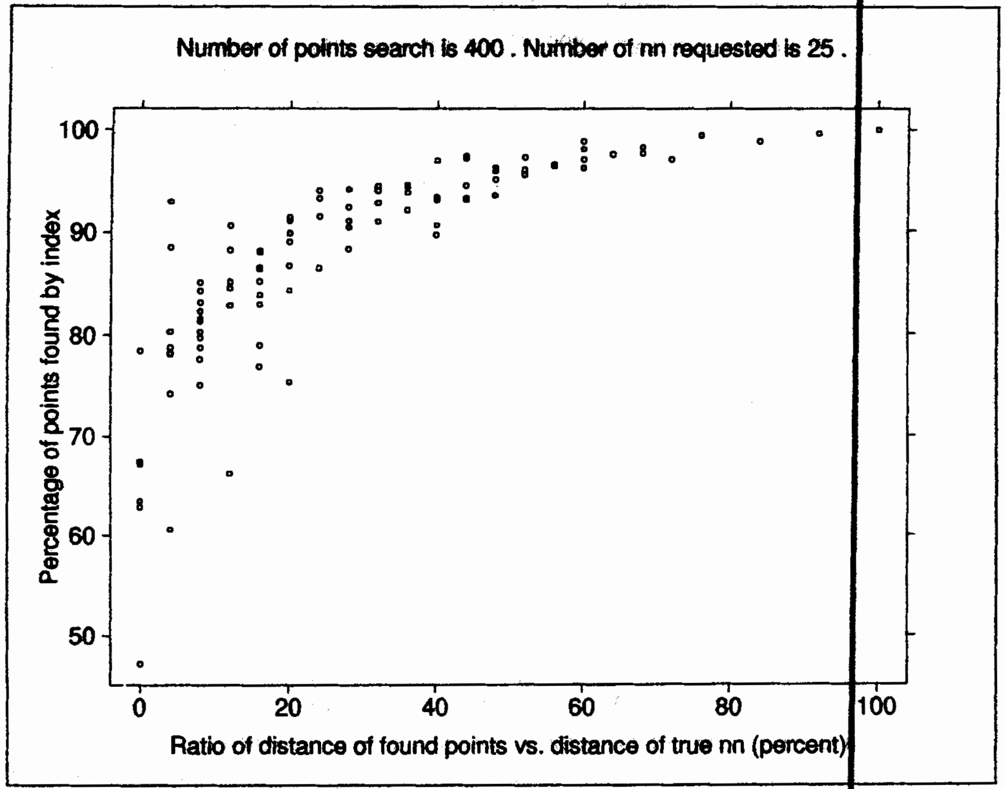

Figure 23: Distance and found ratio for 100 random queries. *RR* scheme.

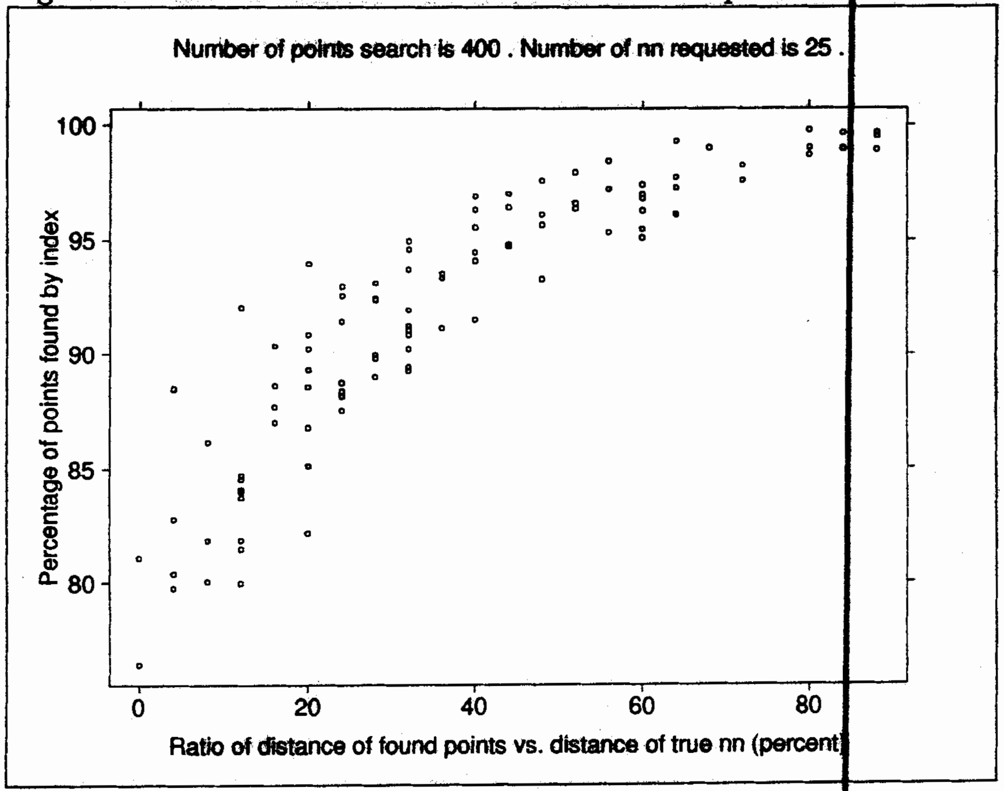

Figure 24: Distance and found ratio for 100 random queries. *RS* scheme.

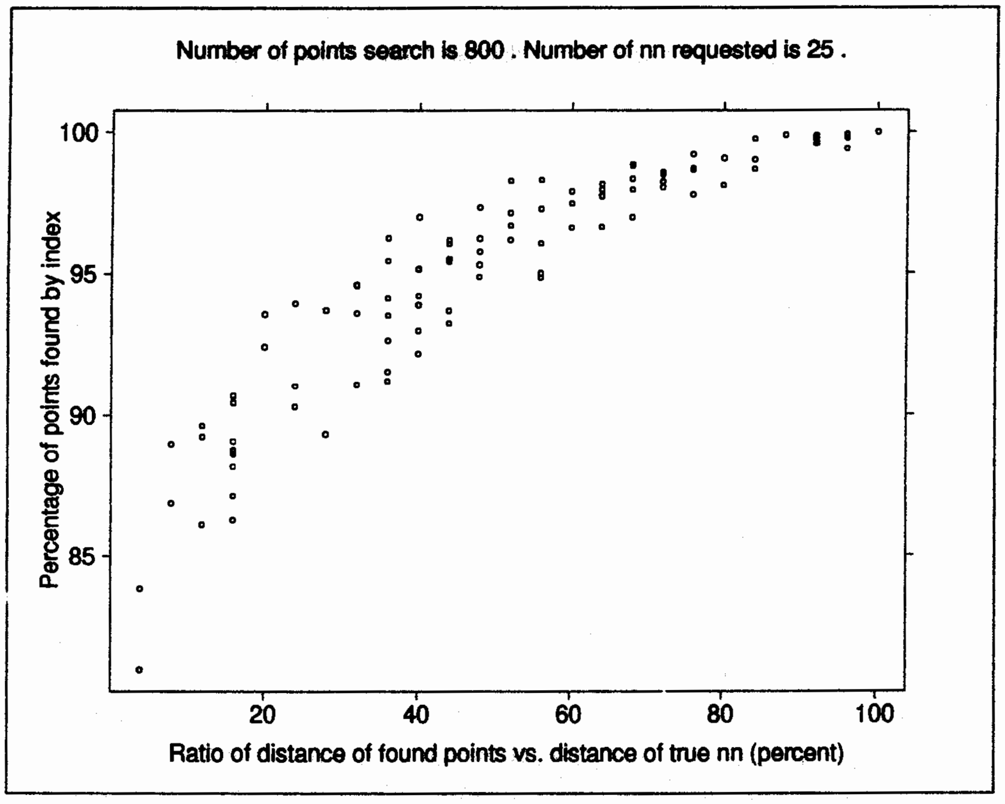

Figure 25: Distance and found ratio for 100 random queries. *RS* scheme.

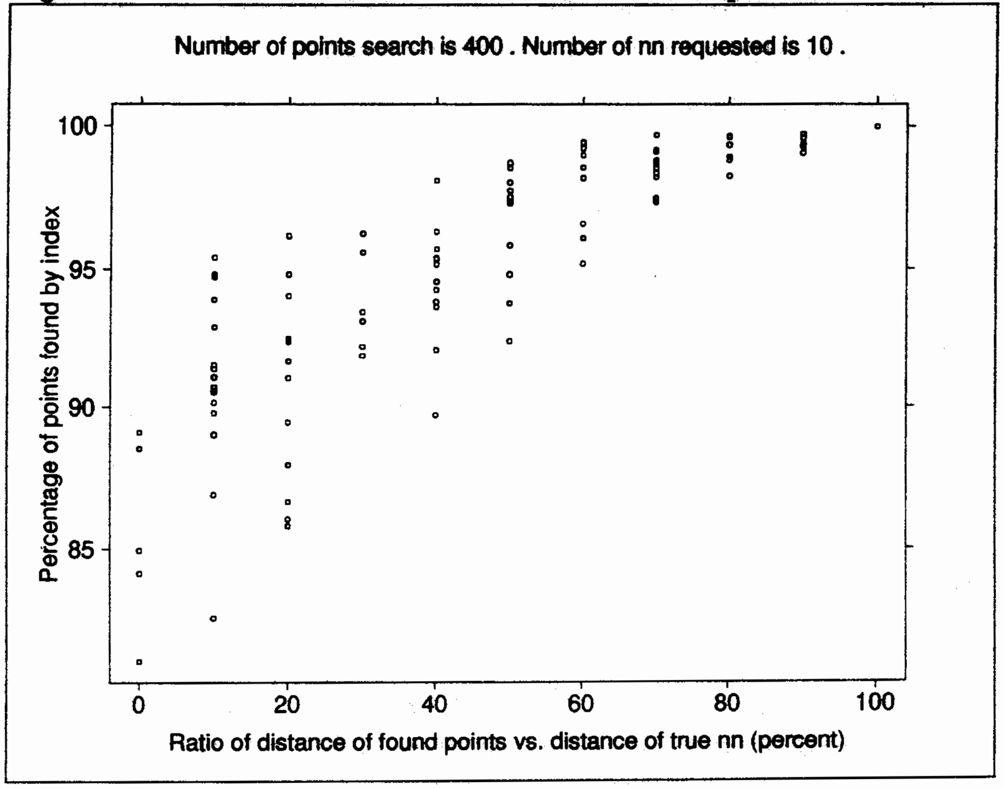

Figure 26: Distance and found ratio for 100 random queries. *RS* scheme.

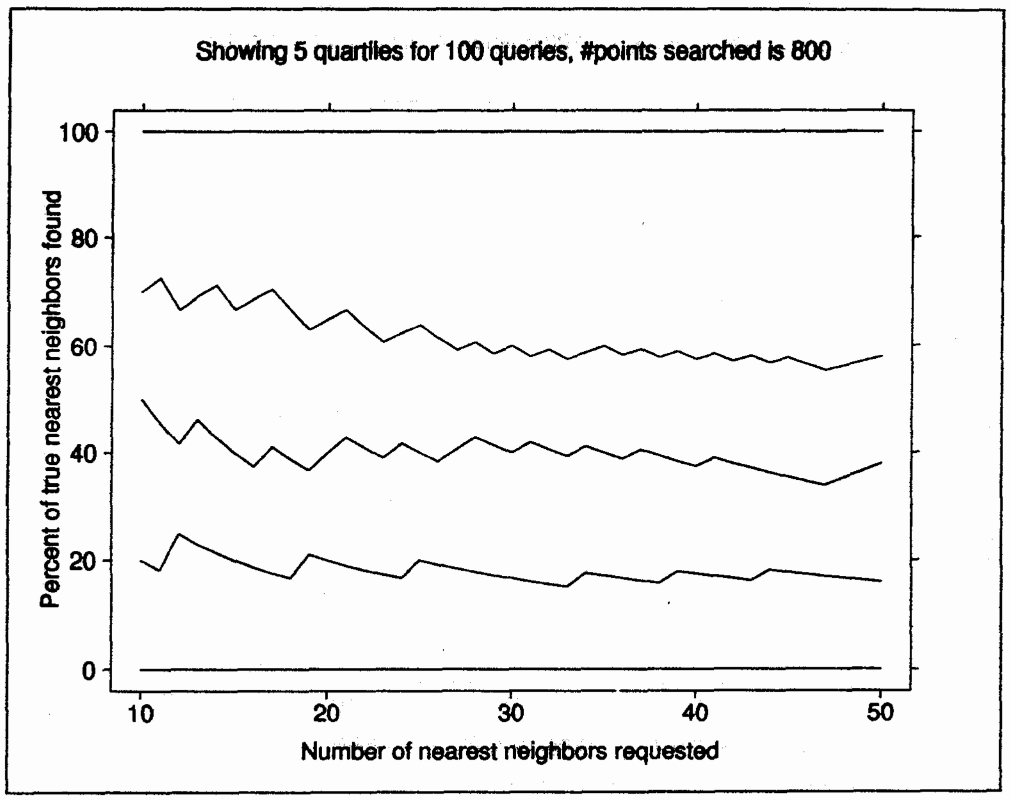

Figure 27: Five quartiles of percentage of points found over 100 queries. *RR* scheme.

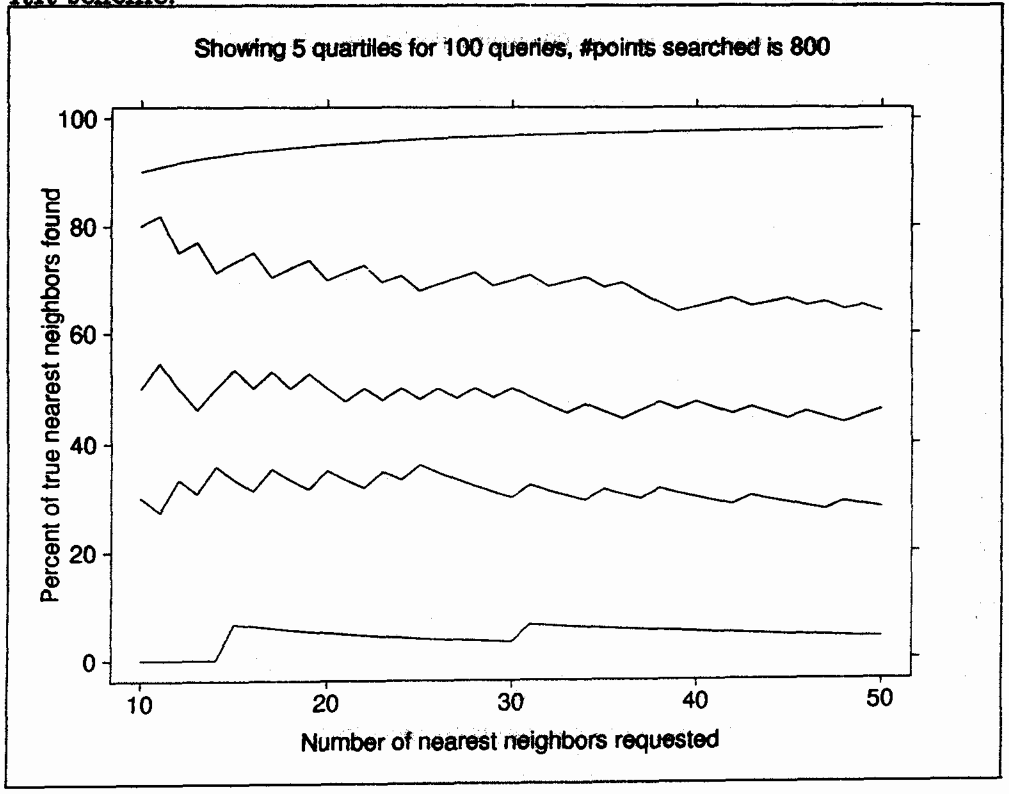

Figure 28: Same as Figure 27 for *RS* scheme.

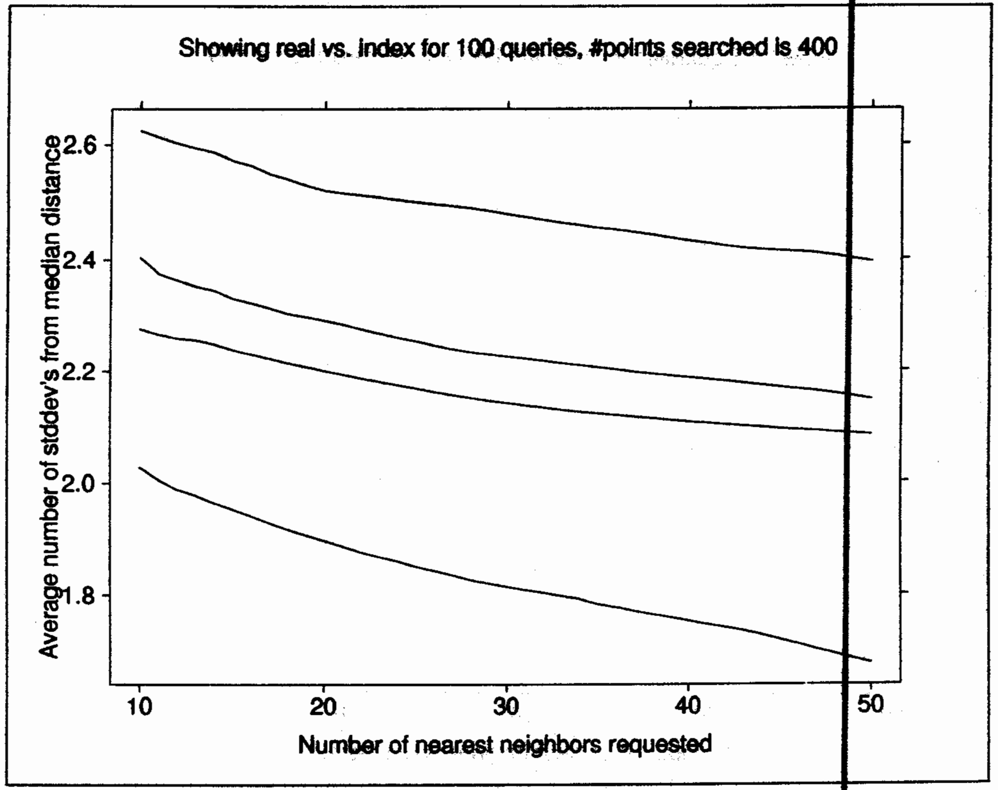

Figure 29: 1'st and 3'rd quartiles for true nearest neighbors and index structures. *RR* scheme. $c = 400$.

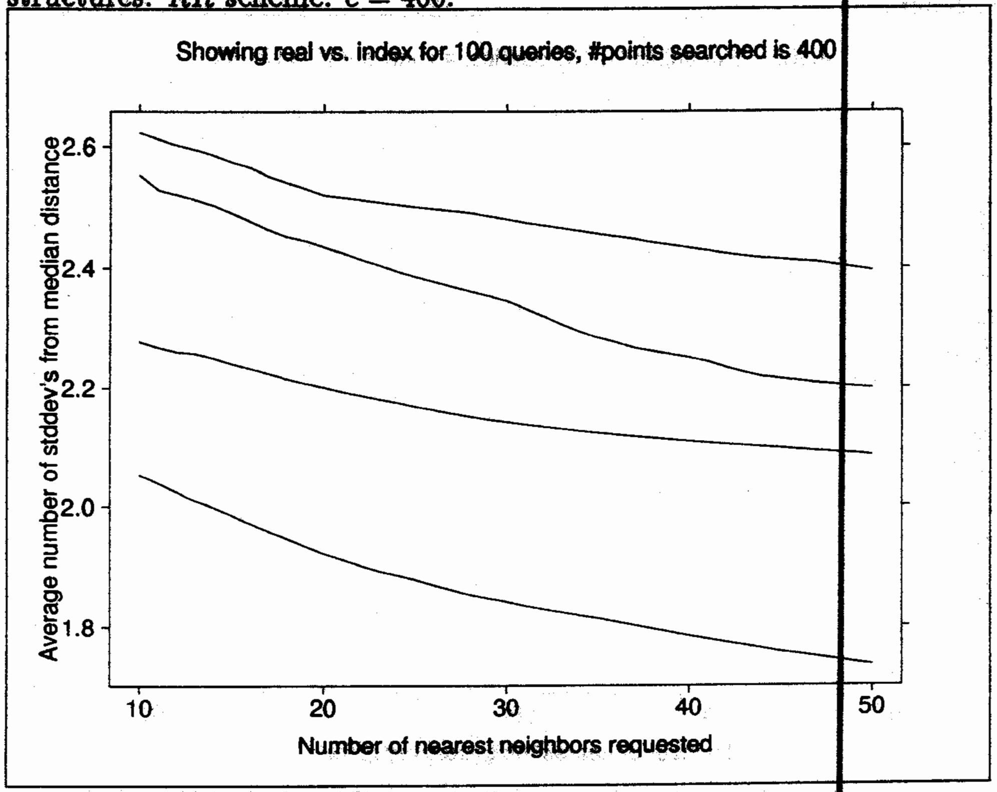

Figure 30: Same as Figure 29 for *RS* scheme.

9 Connect-4 Data Set

The Connect-4 data set is a table of all positions in the connect-4 game. The board has 42 places in it, each place can contain *blank, black* or *white*, wich are encoded by $1, 0, 2$ respectively. One extra attribute has three options regarding who is in a position to win (three options that include a draw). There is a constraint that the number of black places and the numebr of white places differ by at most 1.

- The number of points is 67557.
- The number of dimensions (arity) is 43.
- The values of data points in each dimension are integers ranging from 0 to 2.
- The dimensionality is not reduced.

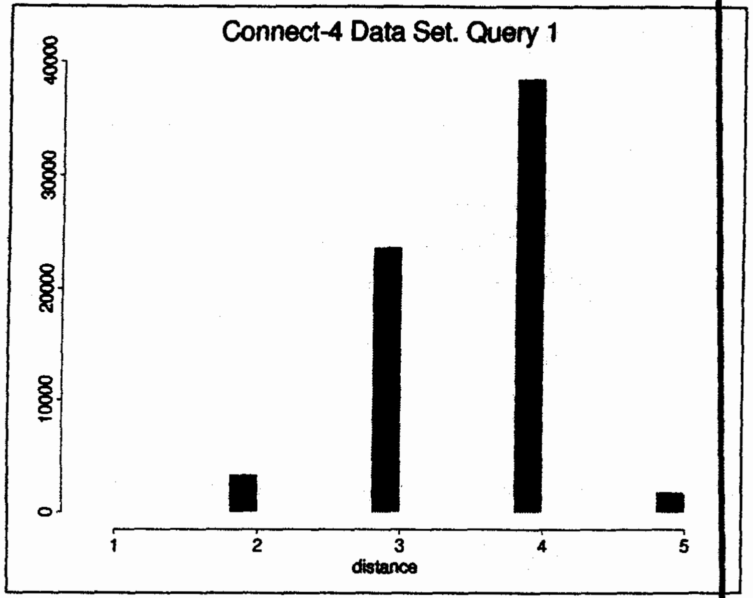

Figure 31: Distribution of distances of data points from query point number 1 (Connect-4 data set).

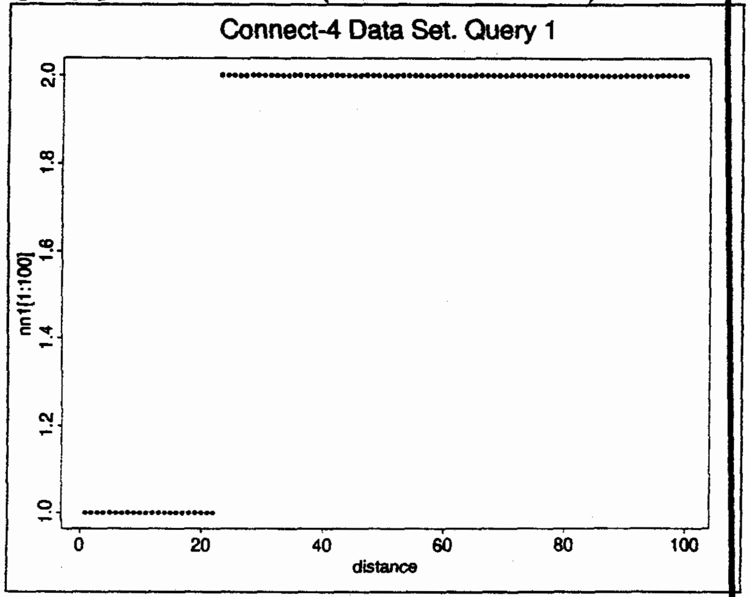

Figure 32: Distances of first 100 nearest neighbors from query point number 1 (Connect-4 data set).

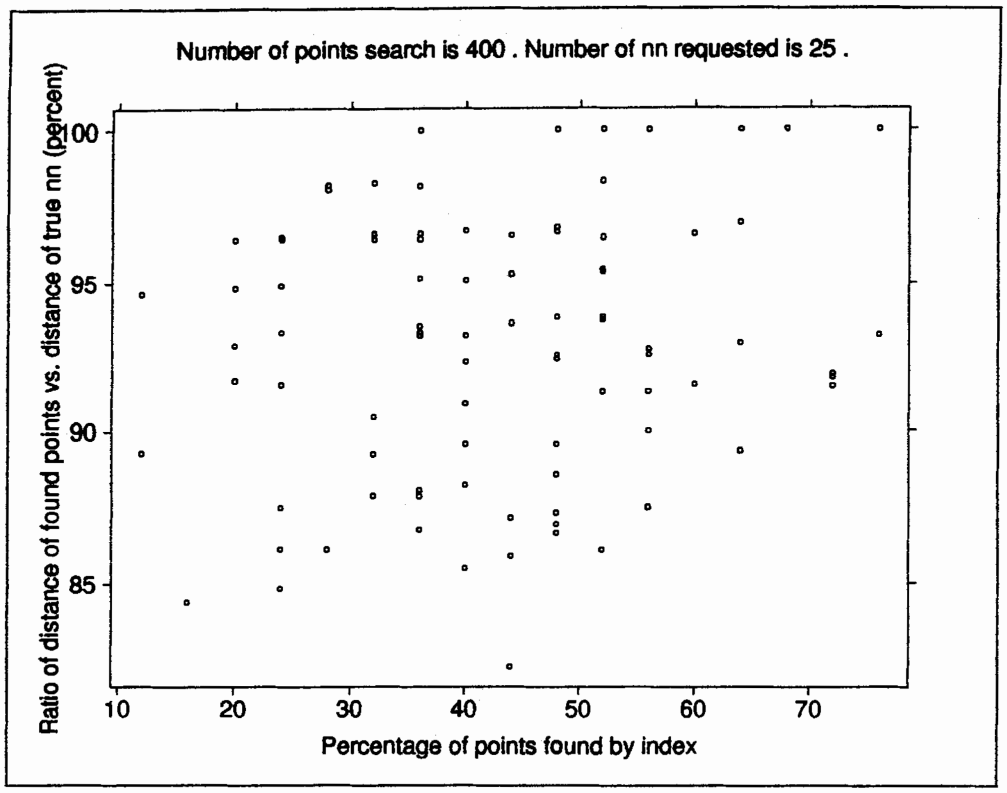

Figure 33: Distance and found ratio for 100 random queries. *RR* scheme.

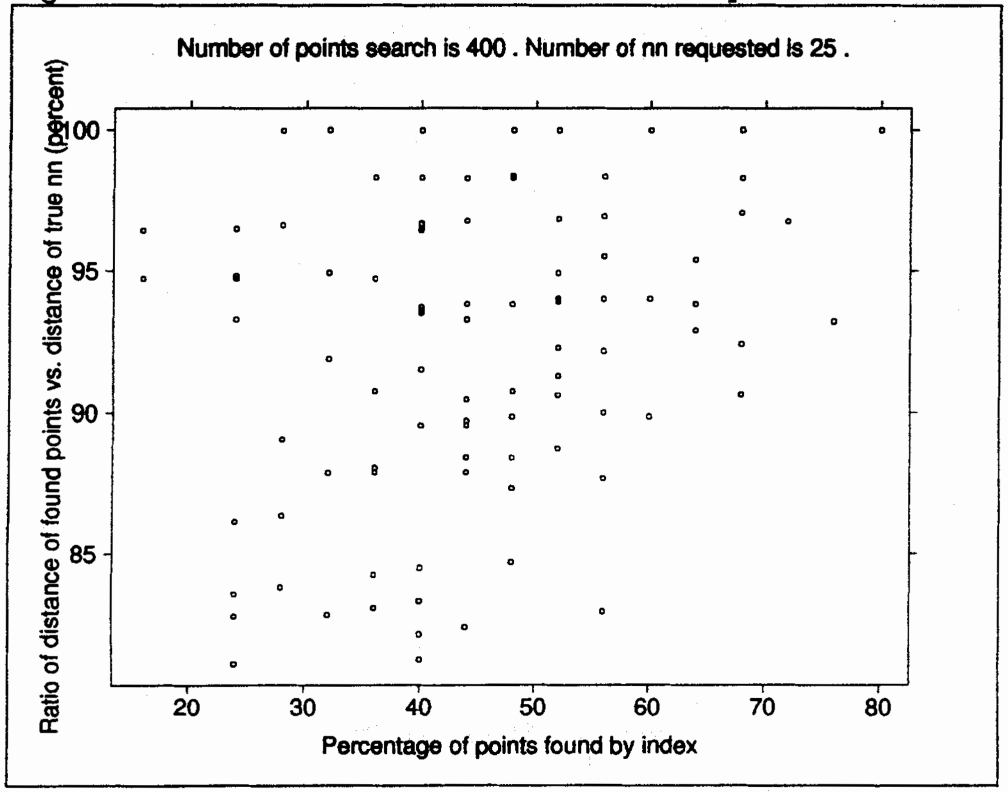

Figure 34: Distance and found ratio for 100 random queries. *RS* scheme.

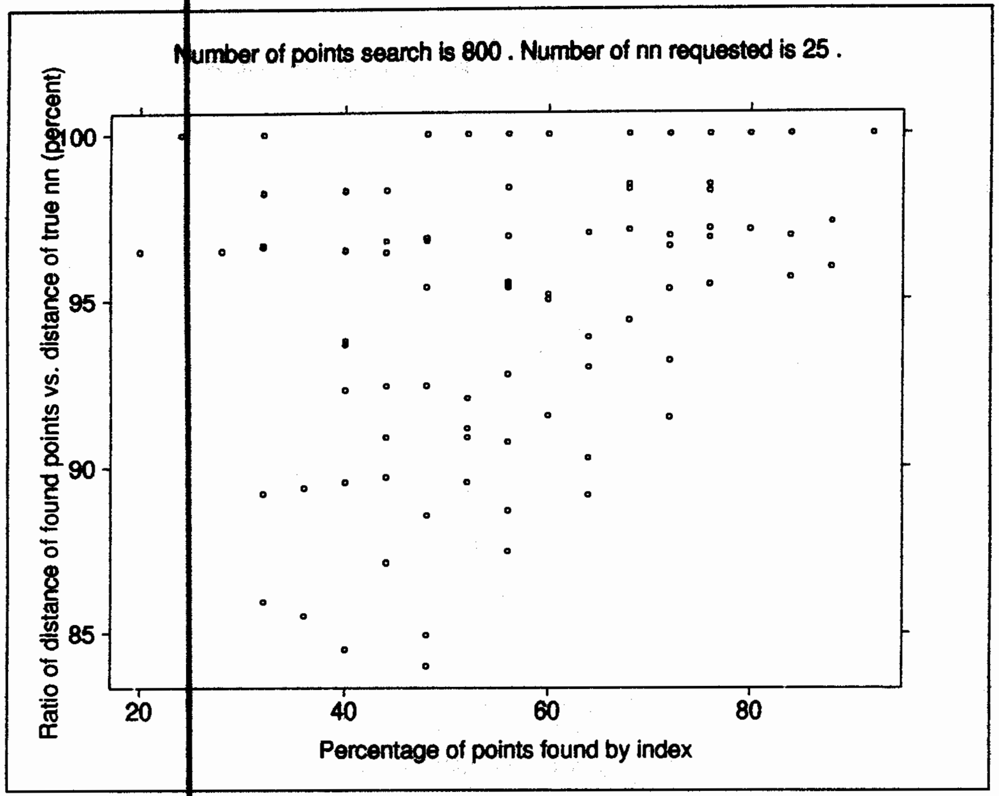

Figure 35: Distance and found ratio for 100 random queries. *RR* scheme.

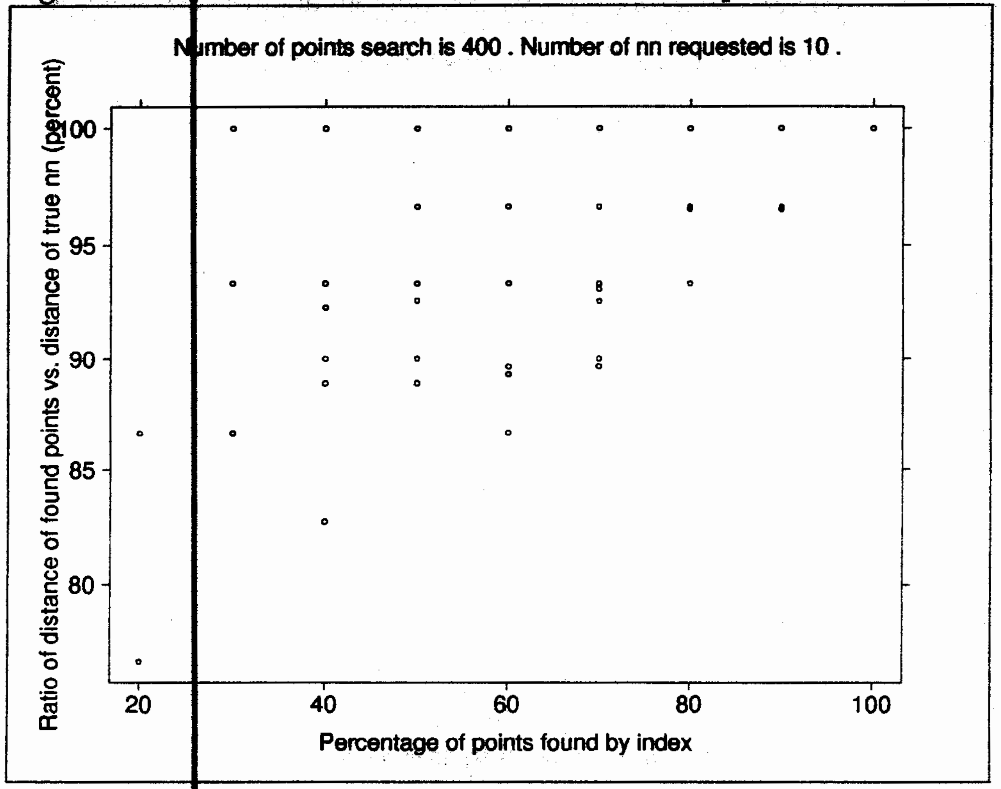

Figure 36: Distance and found ratio for 100 random queries. *RS* scheme.

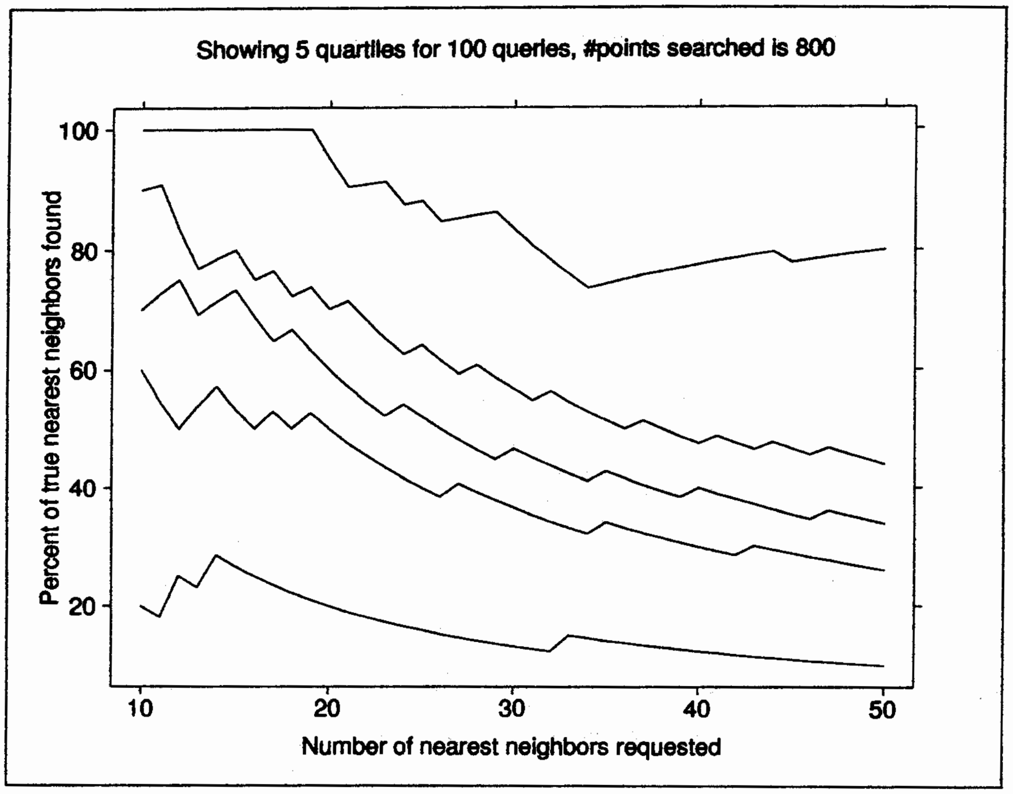

Figure 37: Five quartiles of percentage of points found over 100 queries. *RR* scheme.

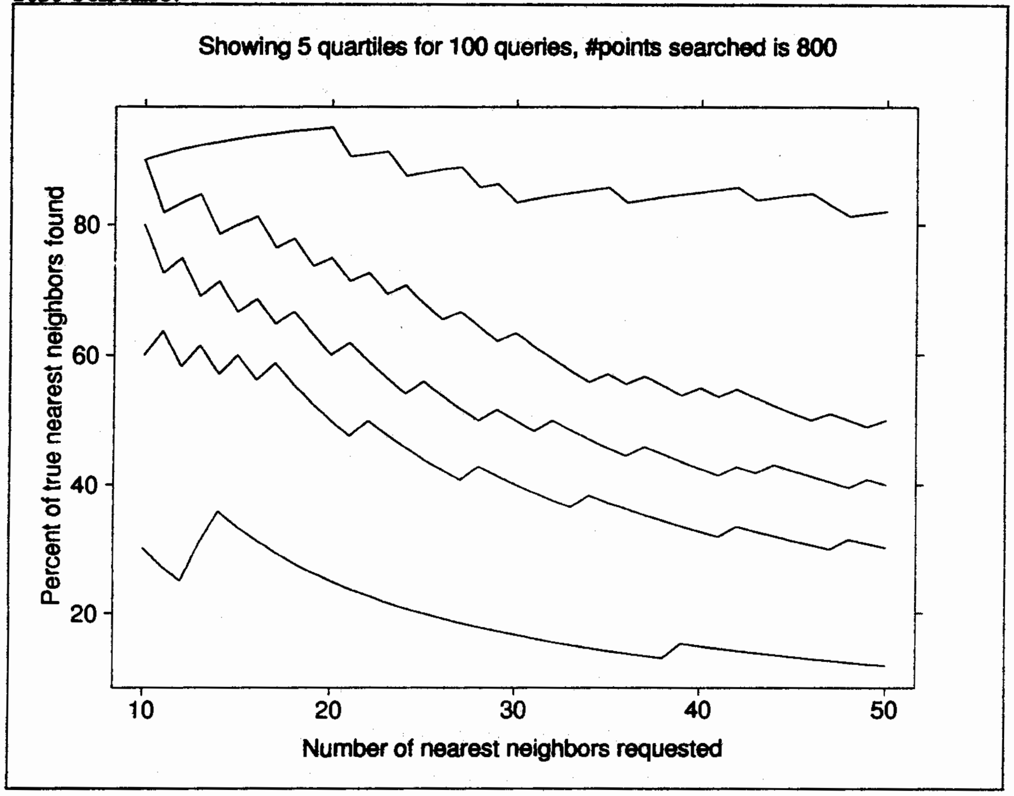

Figure 38: Same as Figure 37 for *RS* scheme.

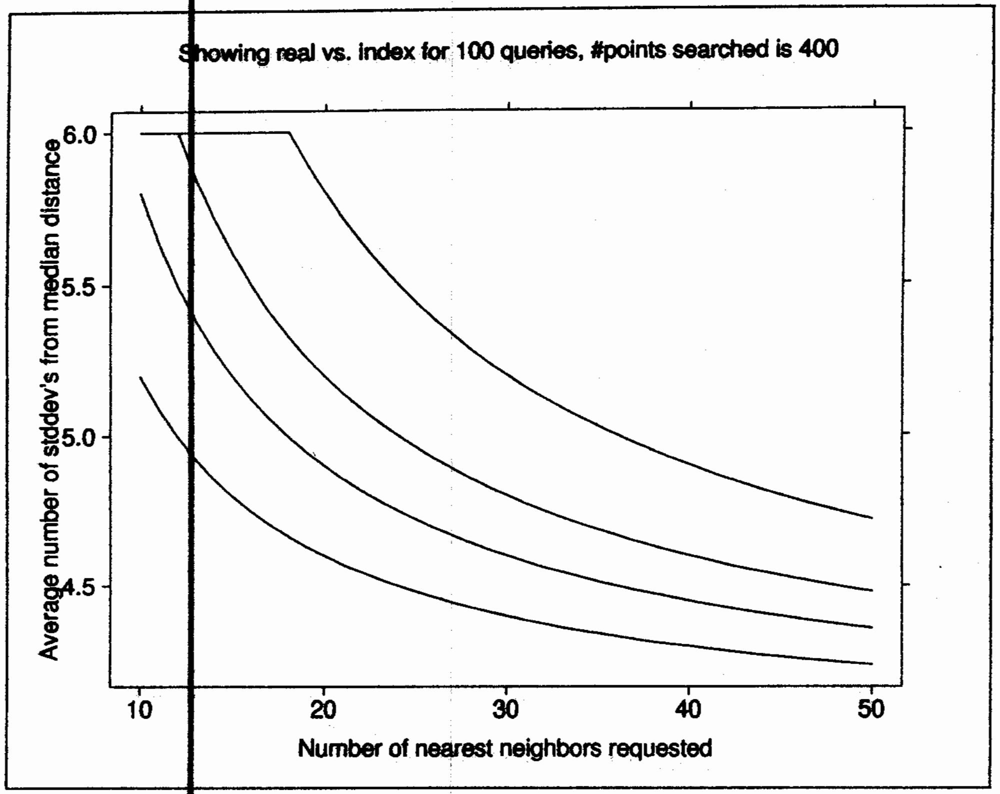

Figure 39: 1'st and 3'rd quartiles for true nearest neighbors and index structures. *RR* scheme. $c = 400$.

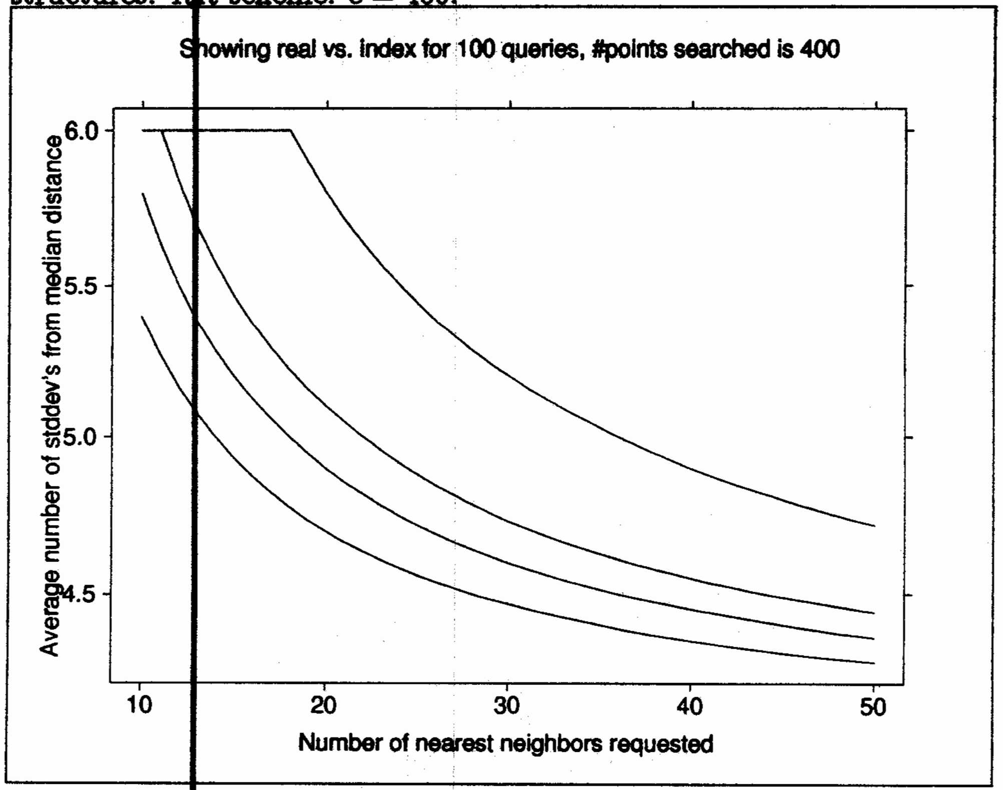

Figure 40: Same as Figure 39 for *RS* scheme.

10 Adult Data Set

The Adult data set has data about people. Numeric attributes were kept unchanged. Low cardinality attributes were split into several attribtues such that the distance between two tuples on those attribtues is 100 if the values are different, and 0 if they are the same. The original data set had 15 attribtues.

- The number of points is 30162.
- The number of dimensions (arity) is 90.
- The values of data points in each dimension are integers ranging from −560 to 1596.
- The dimensionality is reduced to 74 while keeping 98% of the variance.

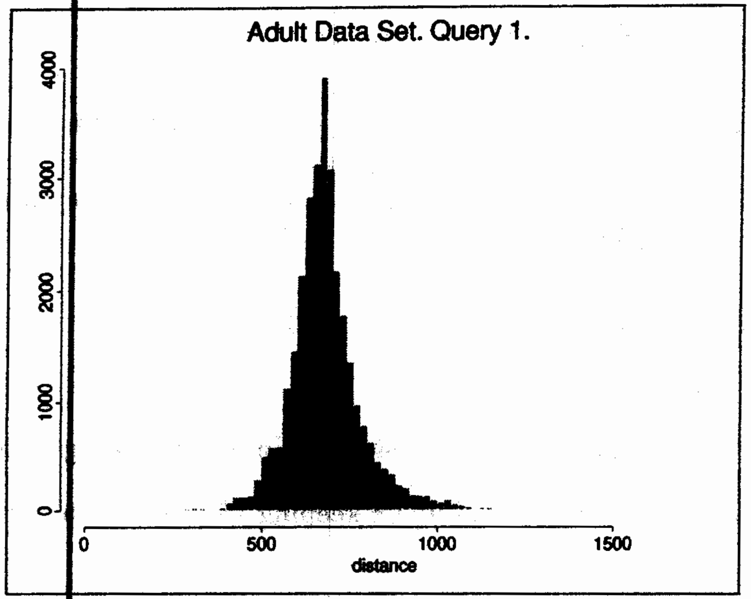

Figure 41: Distribution of distances of data points from query point number 1 (Adult data set).

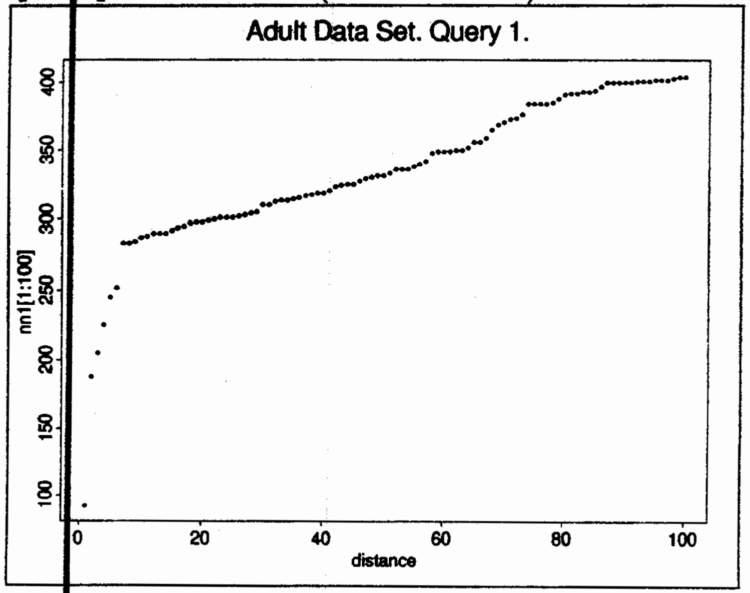

Figure 42: Distances of first 100 nearest neighbors from query point number 1 (Adult data set).

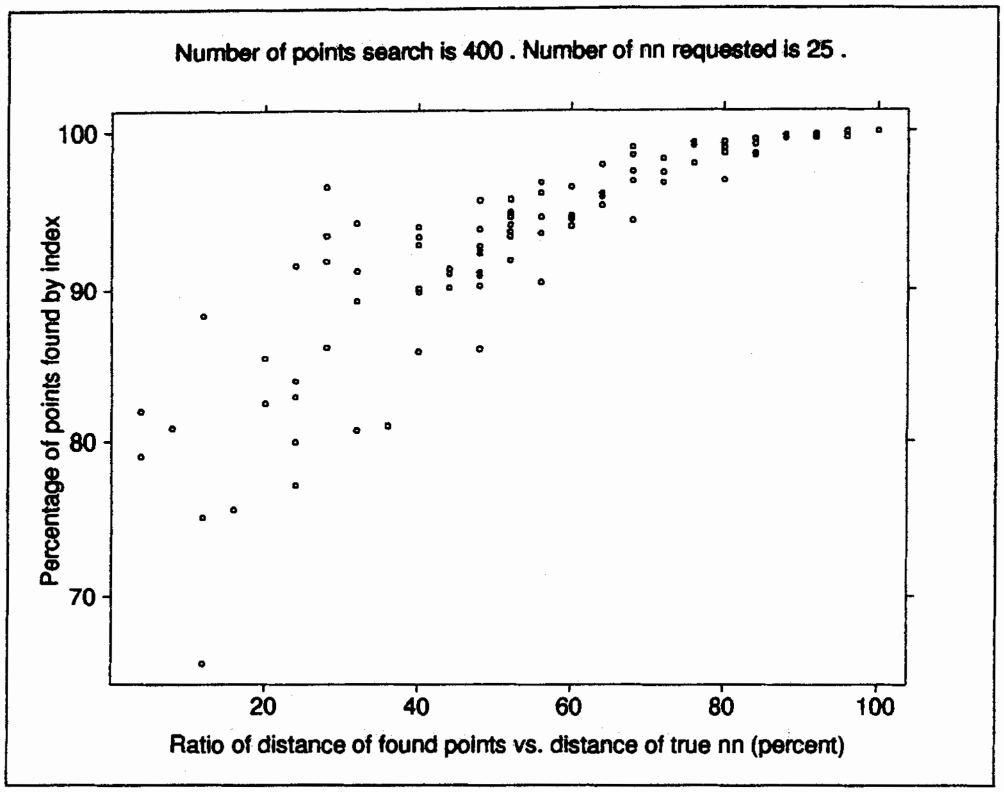

Figure 43: Distance and found ratio for 100 random queries. *RR* scheme.

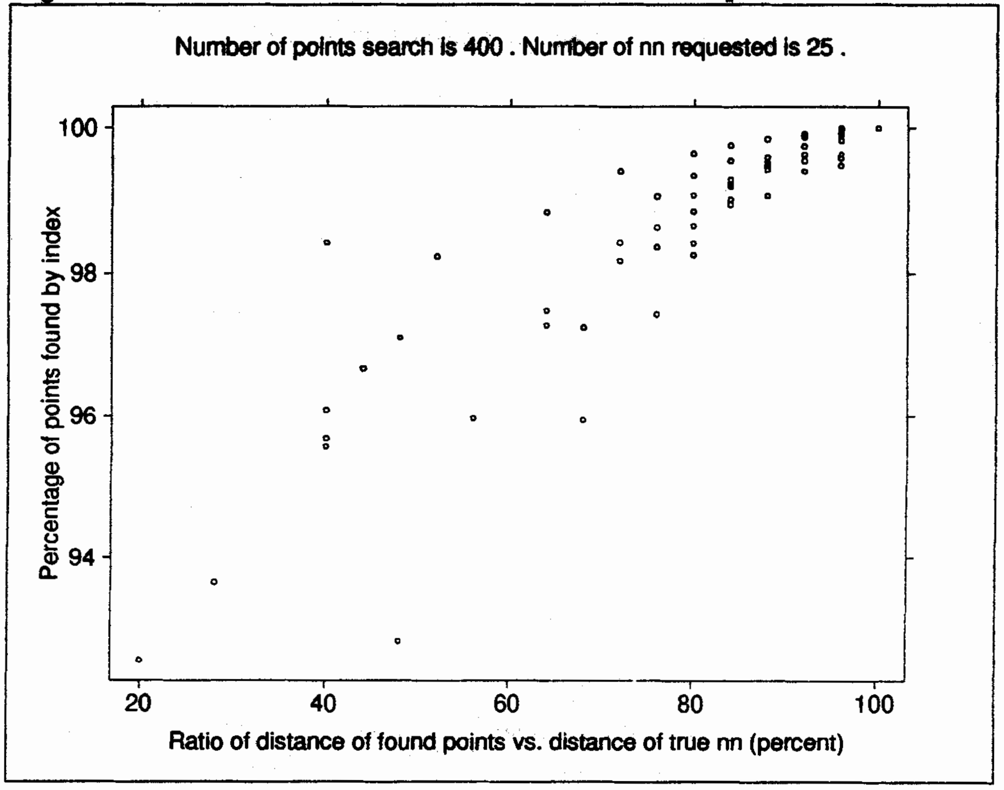

Figure 44: Distance and found ratio for 100 random queries. *RS* scheme.

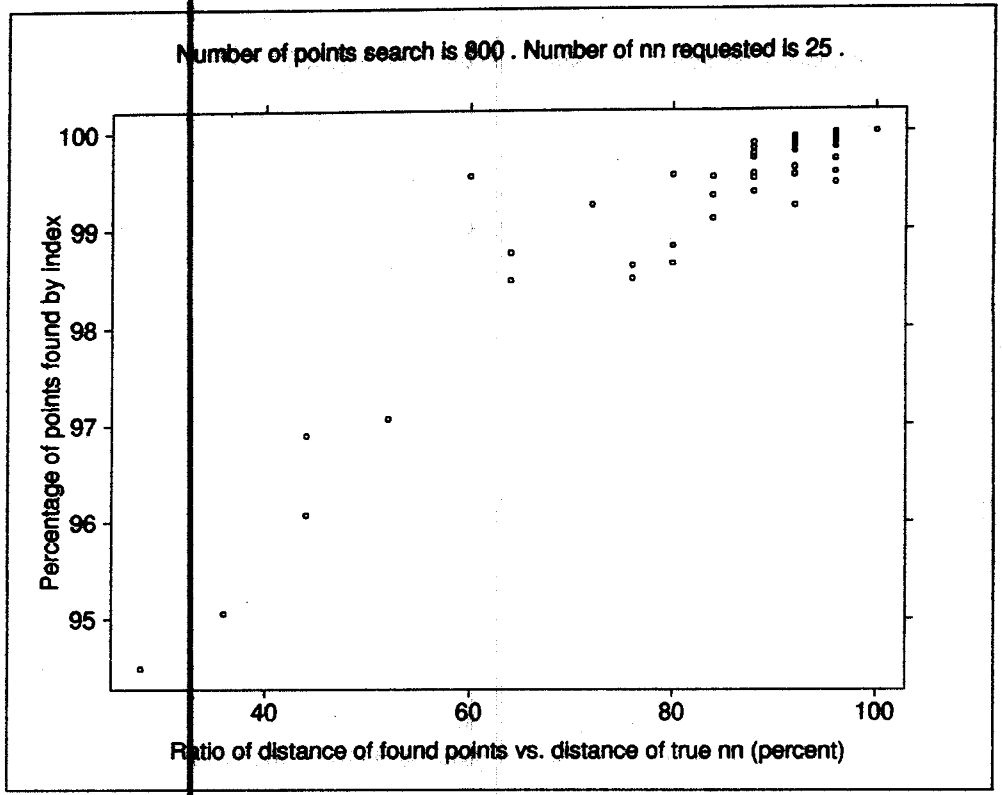

Figure 45: Distance and found ratio for 100 random queries. *RS* scheme.

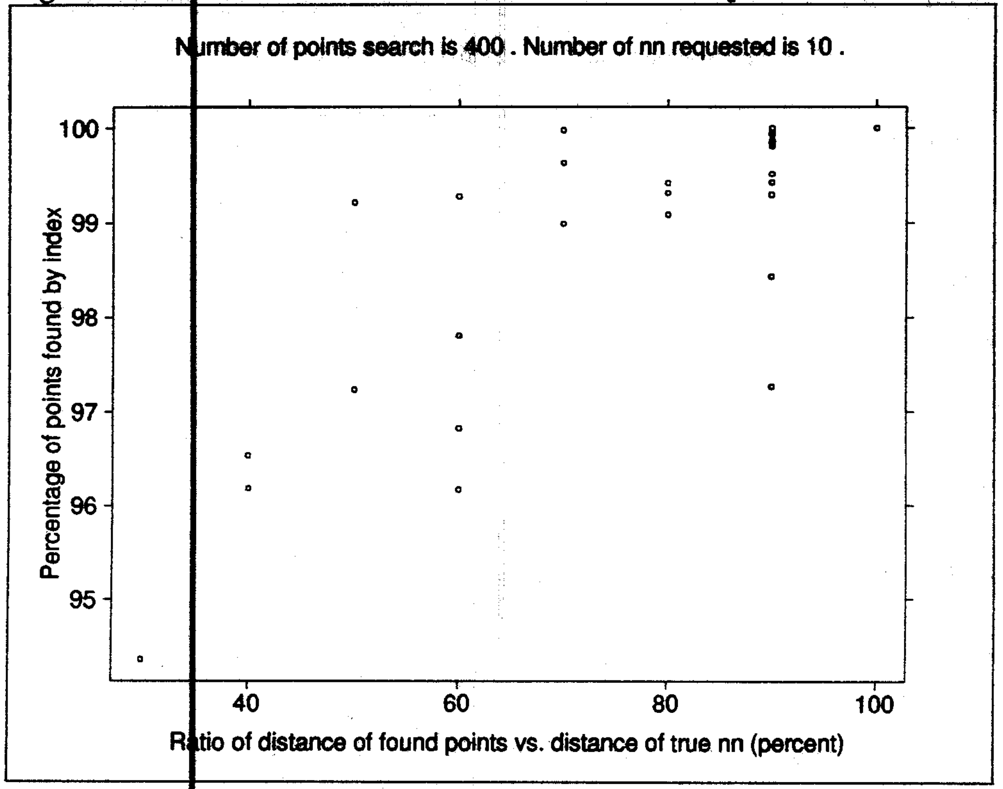

Figure 46: Distance and found ratio for 100 random queries. *RS* scheme.

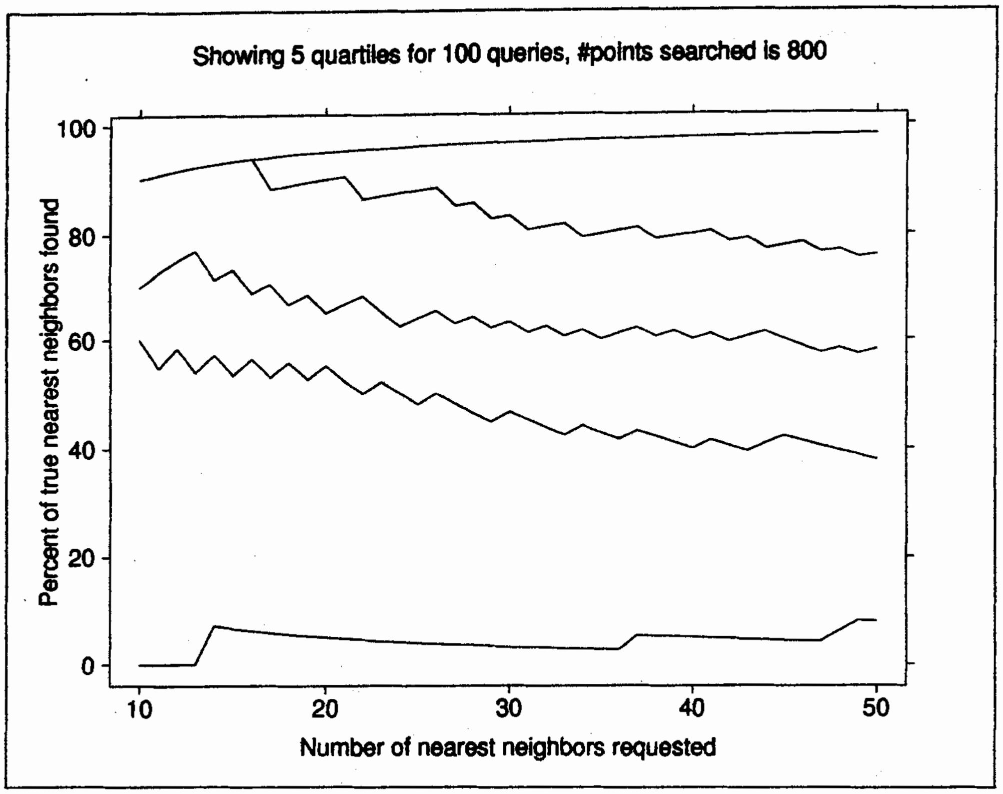

Figure 47: Five quartiles of percentage of points found over 100 queries. *RR* scheme.

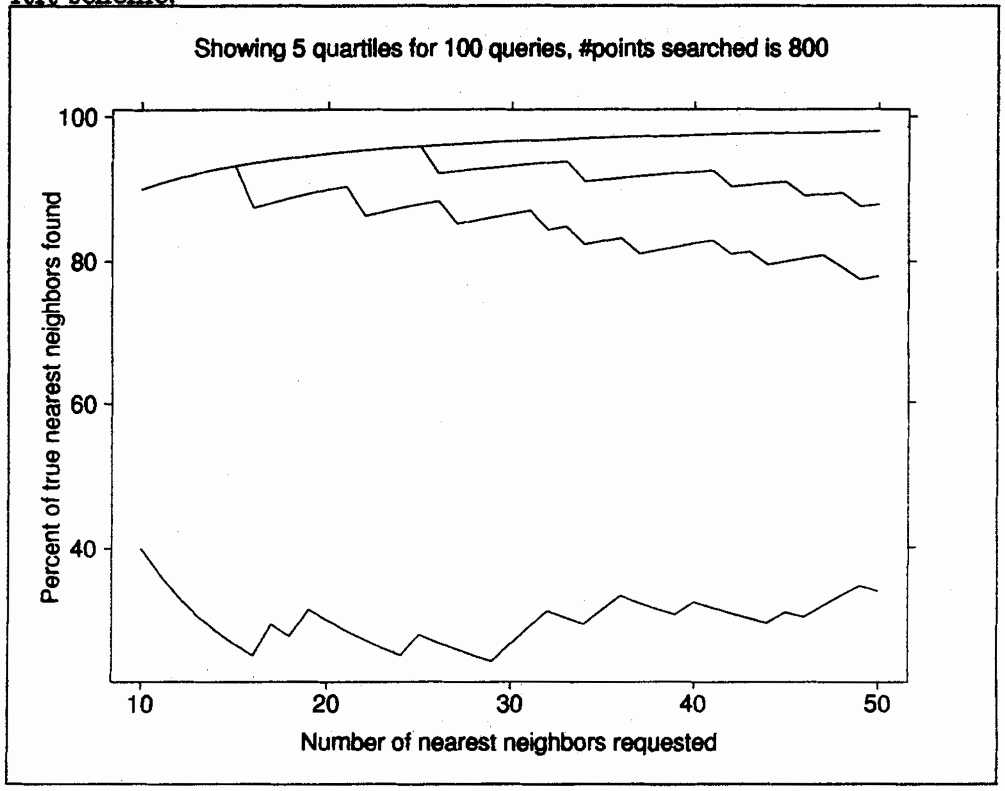

Figure 48: Same as Figure 47 for *RS* scheme.

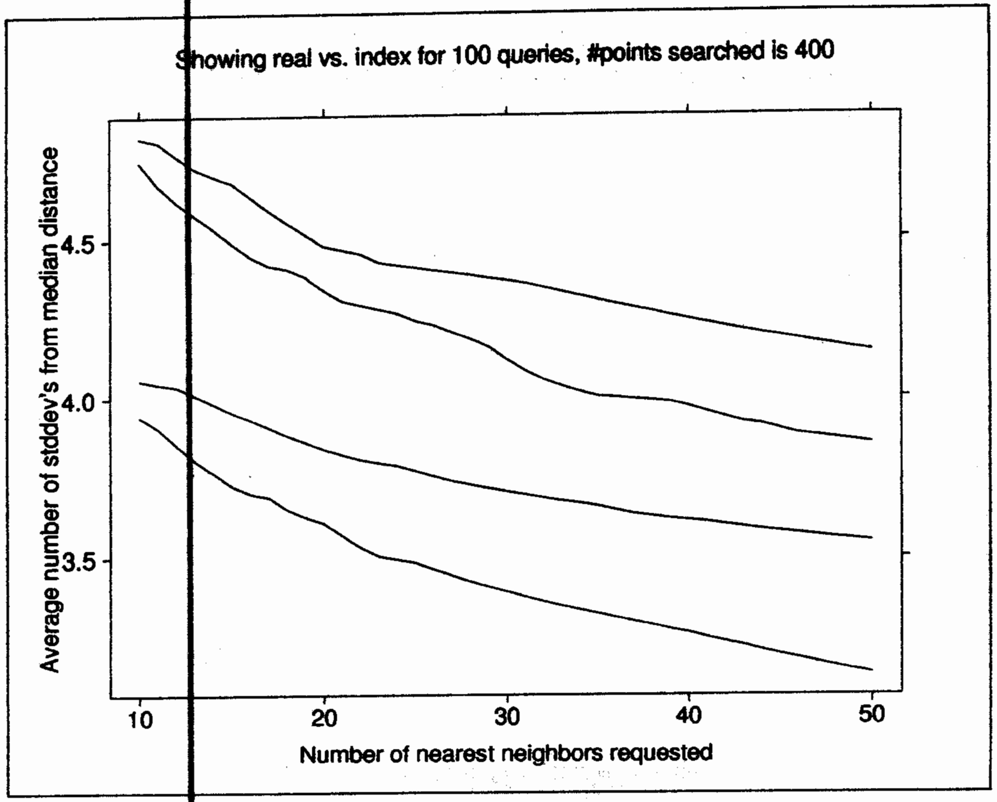

Figure 49: 1'st and 3'rd quartiles for true nearest neighbors and index structures. *RR* scheme. $c = 400$.

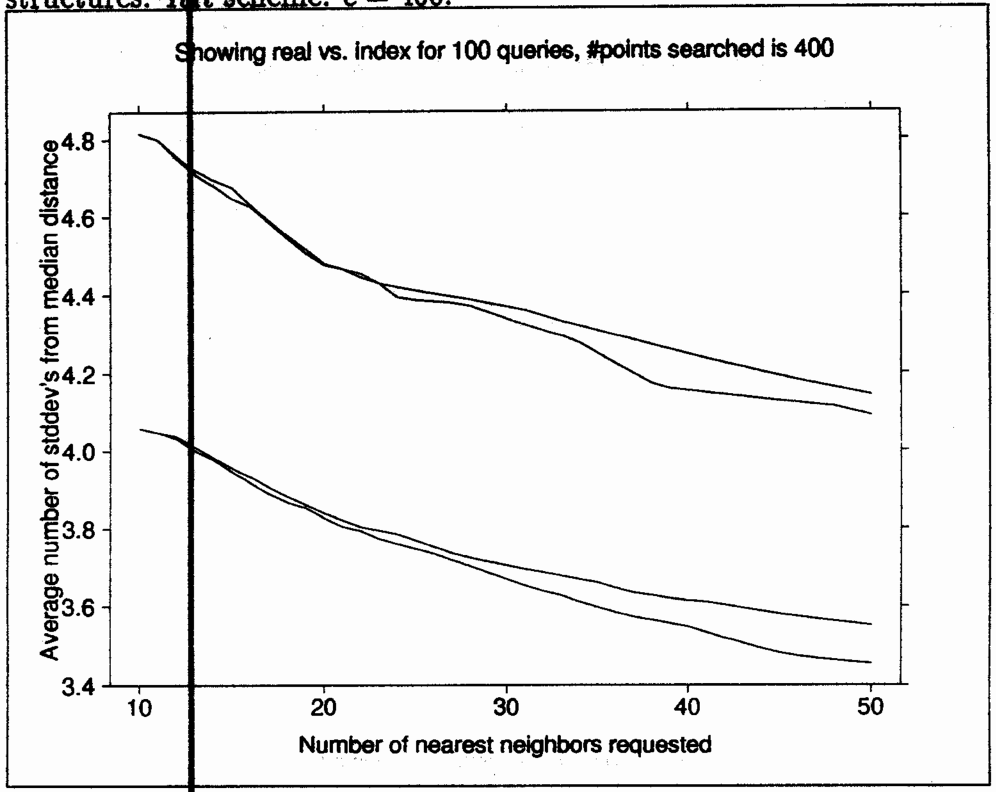

Figure 50: Same as Figure 49 for *RS* scheme.

11 Estimated performance

We have implemented an experimental version of the method described in this paper in order to test the number of points required to be checked for achieving certain accuracy. In this section we estimate the crossover point in terms of the size of the database, where our method is expected to be faster than a linear scan of the entire database. The advantage of a linear scan is that it reads a disk effectively 10 times faster than random scan.

We work under the following assumptions regarding disk performance:

1. A data page holds 4 kilo-bytes.
2. Data can be read sequentially at a speed of which is 10 times faster than random scan. We therefore define the time unit as the time it takes to read a page in a sequential scan (a typcial number would be 1 milisecond, but the analysis below is independent of this number)

These assumptions hold, for example, in a system consisting of a Pentium/Pro processor (so that CPU time is not the bottleneck), a medium speed 4GB IBM disk drive.

Denote the number points in the database by N, and denote the dimension by n (typically, n could be equal to 250). each point has 250 coordinates. If each coordinate takes 4 bytes, the total storage is 4 nN bytes. For example, if $N = 1,000,000$ and $n = 250$, the storage is about $1GB$ Let k denote the number of nearest neighbors sought.

For N (say ,one million) database points stored in dimension 250, each We consider the k nearest neighbors problem. If the problem is solved in one sequential scan of the data, assuming that CPU time is less than I/O time, this takes $4nN/4000 = .001\,nN$ time units. Suppose we implement our method so that we generate K orderings and then report $c \cdot k$ candidates for which the k nearest are to be picked, (for example, $c = 50$). If we use one relational table store all the orderings, we need $N\,K$ rows, with an average storage of 32 bytes per row, so the whole table takes $32\,N\,K$ bytes. Suppose we employ a B-tree of height at most $h = \lceil \log_{128}(NK) \rceil$ (for example, if N is a million and $K = 50$, we get about $h = 3$, assuming the root is in memory). Then a query amounts to fetching about $K(h+1)$ data pages plus fetching the full coordinates of $c \cdot k$ points from possibly $c \cdot k$ different pages. Since fetching a page takes about 10 time units, the total I/O time is $10(K(h+1) + ck)$ time units.

The crossover point for our method to be faster than complete sequential scan is when

$$N = \frac{10^4}{n}\left(K(h+1) + c \cdot k\right) .$$

So, if $n = 250$, $K = 100$, $h = 3$, $c = 50$ and $k = 10$, the multi-ordering method is better for $N > 36,000$. The factor of improvement is proportional to N as long as h does not grow. So, for example, for $N = 1,000,000$, we expect an improvement by a factor of 30.

For much larger databases the improvement is much larger since h grows very slowly. It may allow for exact searches still in time much shorter than complete sequential scan.

References

[1] A. R. Butz, "Space filling curves and mathematical programming," *Information and Control* **12** (1968) 314–330.

[2] A. R. Butz, "Convergence with Hilbert's Space filling curve," *J. Computer and System Sciences* **3** (1969) 128–146.

[3] A. R. Butz, "Alternative algorithm for Hilbert's space-filling curve," *IEEE Trans. on Computers* **C-20** (1971) 424–426.

[4] D. P. Huttenlocher and J. M. Kleinberg, "Comparing point sets under projection," in: *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms* (1994), pp. 1–7.

[5] P. Prusinkiewicz, A. Lindenmayer and F. D. Fracchia, "Synthesis of Space-filling Curves on the Square Grid," in: *Fractals in the Fundamental and Applied Sciences*, edited by H.-O. Peitgen et al., Elsevier Science Publishers, 1991, pp. 341–366.

[6] H. Sagan, *Space-filling curves*, Springer-Verlag, New York, 1994.

[7] R. J. Stevens, A. F. Lehar, and F. H. Perston, "Manipulation and presentation of multidimensional image data using the Peano scan," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, PAMI-5(5) (1983) pp. 520–526.