Accelerating High-dimensional Nearest Neighbor Queries

Christian A. Lang
clang@cs.ucsb.edu

Ambuj K. Singh
ambuj@cs.ucsb.edu

Technical Report TRCS02-04
Department of Computer Science
University of California
Santa Barbara, CA 93106

Abstract

The performance of nearest neighbor (NN) queries degrades noticeably with increasing dimensionality of the data. This stems not only from reduced selectivity of high-dimensional data but also from an increased number of seek operations during NN-query execution.

If the NN-radii would be known in advance, the disk accesses could be reordered such that seek operations are minimized. We therefore propose a new way of estimating the NN-radius based on the fractal dimensionality and sampling. We show that the estimation error is considerably lower than for previous approaches.

Since it is possible that the radius is underestimated, we propose a way of finding a close upper bound on the radius based on a given radius estimate. This technique is applicable to any page-based index structure. We show that this upper bound is close to the correct radius.

In the second part of the paper, we present two applications of these findings. We show how the radius estimations can be used to transform k-NN queries into at most two range queries, and how it can be used to reduce the number of page reads during all-NN queries. In both cases, we observe significant speedups over traditional techniques for synthetic and real-world data.

1 Introduction

Nearest neighbor (NN) queries are an important query type for high-dimensional data, such as multimedia data, strings, and time sequences. One example for such a query is "Get the 10 closest images to a given query image" where similarity is typically defined on color distributions or texture features. Another example is DNA sequencing. Here, for each gene segment of one genome, all closest matching gene segments of another genome have to be found. Instead of asking for one best match, this query type asks for all best matches.

On large databases, such queries can be very time-consuming, especially when the number of feature dimensions is high. The main reason for the typically high response times lies in the fact that index structures supporting these queries are disk-based due to the large amount of data. Answering a single query therefore incurs a large amount of disk I/O consisting of page read and seek operations.

Much research has therefore focused on accelerating the execution of such queries by reducing the number of disk pages to read, or by reducing the number of disk seek operations during page reads. Techniques that try to reduce the number of seeks cannot guarantee a minimum number of pages to be read, and vice versa. One extreme example is the linear scan for answering NN queries: by scanning the whole data file, the number of seek operations is minimized but at the same time, all pages have to be accessed. An example for the other extreme is Hjaltason and Samet's k-NN query algorithm [14]. Their algorithm accesses only a minimal number of disk pages but induces a high amount of seek operations.

An algorithm that obtains the best of both worlds would be reading only the pages that contain the NNs with a read schedule that minimizes seeks. In other words, the NN query is transformed into a range query. The problem with this approach is that we do not know the required pages in advance. In order to determine these pages, it would be helpful to know the range of the query containing the NNs. Much work in the last years went into predicting such query ranges. However, current prediction techniques are not sufficient for this task since they predict an *average* query range for the entire dataspace. For a single query, the range can be much larger or smaller than this average. When using these estimates to transform NN queries into range queries, too few or too many pages would be accessed.

A major contribution of this paper is therefore an improved estimate of the query range based on the fractal dimensionality of data and sampling. We distinguish between two types of estimates: query-independent ones and query-dependent ones. For the former, we introduce two improvements on the fractal dimensionality-based estimation of the average query range. The latter type assumes that the query is known and a range estimate is desired. We show how the new query-independent estimate can be used to obtain an (even tighter) query-dependent query range estimate through sampling. The basic idea is to get a first estimate on the sample and then to use the query-independent estimates to compensate for the effects due to sampling.

As a second contribution, we introduce a way of obtaining a close upper bound on the NN range for page-based index structures. This bound is obtained by inspecting all points stored in pages that are intersected by the query-dependent range estimate. The NNs found in these pages are then used to obtain an upper bound on the query range estimate. Our experiments show that this estimate is within 1% of the actual radius.

As a third contribution, we introduce two new NN query algorithms based on these estimates. The first algorithm is for single k-NN queries, and the second algorithm is for all-NN queries. Both new algorithms reduce the amount of disk seeks and page reads at the same time by using our new query-dependent range estimates to determine which pages will have to be fetched from disk in the future. This set of pages is then read into memory using a read schedule that minimizes disk seeks. In case the query-dependent range estimate underestimates the real radius (and therefore would miss some NNs), we use our estimation technique to obtain an upper bound on the query range. For both query types, we observed a significant reduction in overall query time.

The paper is organized as follows. In Section 2, we show how the fractal dimensionality-based estimation of the average query range can be improved. This is then used together with sampling to obtain a query-dependent way of calculating a query range estimate in Section 3. Section 4 analyzes the quality of the obtained estimates. Section 5 introduces a technique to compute a tight upper bound on the query-dependent query range. Sections 6 and 7 present two applications of these estimates. Related work is discussed in Section 8 and we conclude in Section 9.

2 Query-independent k-NN Radius Estimation

In this section, we show how the notion of fractal dimensionality of data can be used to obtain an estimate for the average NN-query range. We describe how current techniques estimate the fractal dimensionality and how its calculation can be improved. We then describe how the fractal dimensionality-based NN-radius estimate can be further improved by taking boundary effects into account.

Before going into details, let us first define some of the terms used in the following. Given a query point q, the *k NNs of q* are defined as a set $NN_k(q)$ of at least k points such that

$$\forall o \in NN_k(q), \forall o' \in DB - NN_k(q) : d(o,q) < d(o',q)$$

where DB denotes the full dataset and $d(o,q)$ denotes the distance between o and q.

The *k-NN radius* is then defined as

$$\max_{o \in NN_k(q)} d(o,q).$$

2.1 Previous Work on NN-radius Estimation

Real datasets oftentimes exhibit self-similarity. This led to the use of the theory of fractals for characterizing datasets. More precisely, Pagel et al. [20] showed how two different fractal dimensionality measures can be used to describe certain properties of datasets. One of these measures is the so-called *correlation fractal dimensionality D_2*. It is defined as follows:[1]

Definition 1 (Correlation Fractal Dim.) *For a point set that has the self-similarity property in the range of scales $r \in (r_1, r_2)$, its Correlation fractal dimensionality D_2 for this range is measured as*

$$D_2 \equiv \frac{\partial \log \sum_i p_i^2}{\partial \log(r)} = \text{constant} \quad r \in (r_1, r_2)$$

[1] A summary of all symbols used in this paper can be found in Table 1.

N	number of data points
σ	data sampling rate
E	embedding dimensionality
D_2	Correlation fractal dimensionality of the full dataset
D'_2	Correlation fractal dimensionality of the data sample
r_{sample}	sampling-based k-NN radius estimate
$r_{expected}$	query-dependent k-NN radius estimate
r_{upper}	upper bound on k-NN radius
$d_{nn}(k)$	radius estimate based on D_2 (full dataset)
$d_{sample-nn}(k)$	radius estimate based on D_2 (sample data)
$d_{nn}^{cutoff}(k)$	radius estimate with boundary effects (full dataset)
$d_{sample-nn}^{cutoff}(k)$	radius estimate with boundary effects (sample data)
k	number of NNs

Table 1: Notation used in the paper

Input: normalized dataset
Output: fractal dimensionality

(1) Let $r := 1$.
(2) Partition the data space in $\left(\frac{1}{r}\right)^d$ boxes of side length r and count the number of boxes $N(r)$ containing points.
(3) Let $r := \frac{r}{2}$. If r is not small enough[4], go to (2).
(4) Find the linear part of the $r/N(r)$-graph in log/log-scale and compute its slope, return it.

Figure 1: The box counting algorithm

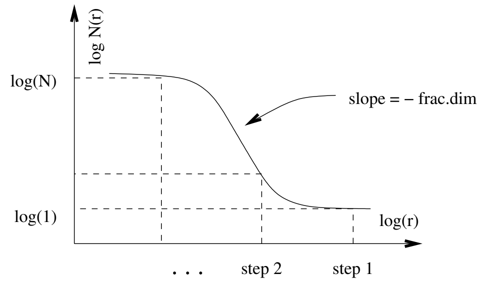


Figure 2: Typical $r/N(r)$-plot

Pagel et al. [20] show how this measure can be used to estimate the k-NN query radius for arbitrary datasets. They show that for the L_∞-norm[2], the average k-NN radius is given by

$$d_{nn}(k) = \frac{1}{2} \cdot (\frac{k}{N-1})^{\frac{1}{D_2}} \tag{1}$$

where N is the number of data points.

The next section shows how D_2 can be computed for a given dataset and how this computation can be improved.

2.2 Improved Fractal Dimensionality Estimation

The correlation dimension D_2 of point datasets is typically computed using a box-counting algorithm [3], as shown in Figure 1.[3] With each iteration of the algorithm, another point on the $r/N(r)$-graph is computed. A typical graph is shown in Figure 2. It roughly consists of two constant tails, one at log(1) (on the right) and one at log(N) (on the left), and a linear piece inbetween. Towards the right end, r becomes large enough to cover the whole dataset at some point, leading to the log(1)-tail. Towards the left end, r becomes so small that every data point falls into its own box, causing the number of occupied boxes to reach N. For fractal datasets, a linear section exists inbetween the tails. Its slope determines the fractal dimensionality.

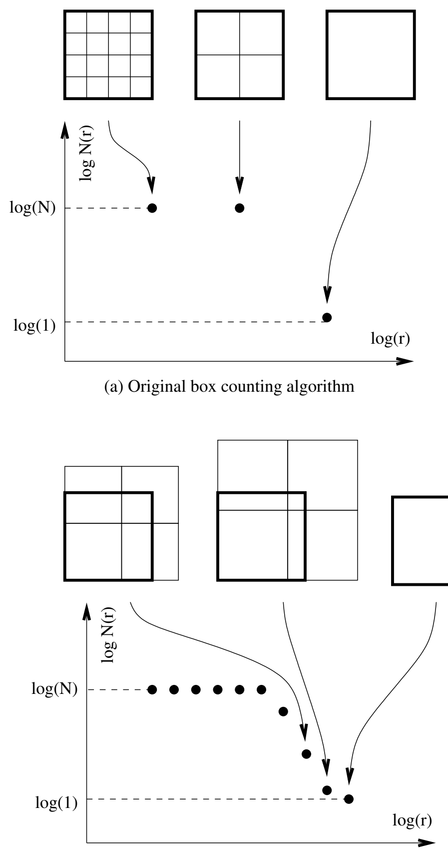
We show in this section that the box counting algorithm suffers from the sparsity of high-dimensional datasets and that the radius predictions therefore deteriorate with increasing data dimensionality. This leads to an improved way of measuring D_2. The next section shows how understanding boundary effects can further improve the quality of the radius estimation.

The box counting algorithm partitions the data space into $b(i) := 2^{d\cdot(i-1)}$ boxes in the ith iteration. For uniform data, there are $\begin{pmatrix} b(i)+N-1 \\ N \end{pmatrix}$ possible distributions of N data points in $b(i)$ boxes. It follows easily that the probability that a box contains no points is $\frac{b(i)-1}{b(i)-1+N}$. For $i = 2$, $N = 10^6$ and $d = 50$, this probability is nearly 1. This means that nearly all boxes are empty. It follows that with high probability the number of boxes with points is N after the first iteration. In these cases, no linear slope

[2] We will also focus on the L_∞-norm in this paper. Similar reasoning holds for other norms as well.

[3] Note that the presented algorithm is a slightly simplified version of the algorithm for computing D_2. The basic idea is the same, however.

[4] By "small enough", we mean small enough for the linear part of the graph to be revealed.



(a) Original box counting algorithm

(b) Modified box counting algorithm

Figure 3: Relation between box coverage and log/log-plot

can be found in Step (4) of the algorithm. This is shown if Figure 3 (a).

Even though this problem is less severe for clustered data, in most cases the number of points on the linear slope is still too small for accurate calculations. For that reason, we modify the box counting algorithm as follows: instead of dividing r by 2 in Step (3), we use a value $1 < r_{div} < 2$. The boxes in direction of higher coordinates may now partly fall outside the data space (cf. Figure 3 (b)) but since they still form a complete coverage of the points, the computed dimension is correct. The modified algorithm is given in Figure 4. Now, more points can be computed between the maximal ($\log N$) and minimal ($\log 1$) values of the graph.

2.3 Taking Boundary Effects into Account

The k-NN radius estimate can be further improved by taking boundary effects into account. The formula for $d_{nn}(k)$ is derived by assuming that queries that exceed the limits of the data space are wrapped around. In other words, in a data space with extents normalized to 1, a query with radius $\epsilon = 1/2$ is assumed to cover the whole data space. In reality however, query shapes are cut off at the data space boundary. In order to cover the same number of NNs, the NN radius has to increase correspondingly. This effect becomes especially pronounced in high dimensions.

Input: normalized dataset
Output: fractal dimensionality

(1) Let $r := 1$.
(2) Partition the data space in $\lceil \frac{1}{r} \rceil^d$ boxes of side length r and count the number of boxes $N(r)$ containing points.
(3) Let $r := \frac{r}{r_{div}}$. If r is not small enough, go to (2).
(4) Find the linear part of the $r/N(r)$-graph in log/log-scale and compute its slope, return it.

Figure 4: The modified box counting algorithm

The expected volume of a query hypercube in E dimensions with radius r is

$$(2r)^E$$

if no boundary effects are taken into account.

With boundary effects, this volume depends on the location of the query point. The expected volume $\widetilde{vol}(r)$ is then

$$\int_p \prod_{i=1}^{E} (\min\{p_i + r, 1\} - \max\{p_i - r, 0\}) dp$$

or

$$\int_p \prod_{i=1}^{E} \begin{cases} 2r & \text{if } p_i + r < 1 \text{ and } p_i - r > 0 \\ p_i + r & \text{if } p_i + r < 1 \text{ and } p_i - r \leq 0 \\ 1 - p_i + r & \text{if } p_i + r \geq 1 \text{ and } p_i - r > 0 \\ 1 & \text{otherwise} \end{cases} dp$$

which is equivalent to

$$\prod_{i=1}^{E} \int_{p_i} \begin{cases} 2r & \text{if } p_i < 1 - r \text{ and } p_i > r \\ p_i + r & \text{if } p_i < 1 - r \text{ and } p_i \leq r \\ 1 - p_i + r & \text{if } p_i \geq 1 - r \text{ and } p_i > r \\ 1 & \text{otherwise} \end{cases} dp_i.$$

The integral can be calculated as

$$\int_r^{1-r} 2r \; dp_i + \int_0^{\min\{1-r,r\}} (p_i + r) \; dp_i +$$

$$\int_{\max\{1-r,r\}}^{1} (1 - p_i + r) \; dp_i + \int_{1-r}^{r} 1 \; dp_i$$

r_{div}	Rel. error (10 NN)	Rel. error (20 NN)	Rel. error (50 NN)
2 (Orig. alg.)	49.29%	58.52%	84.35%
1.15 (5x more steps)	−19.91%	−11.43%	8.66%

Table 3: NN-radius estimation errors

which is

$$\frac{1}{2} + r + \frac{1}{2}(\min\{1-r,r\}^2 + \max\{1-r,r\}^2) + r\min\{1-r,r\} - (1+r)\max\{1-r,r\} + \begin{cases} 2r - 4r^2 & \text{if } r \leq \frac{1}{2} \\ 2r-1 & \text{otherwise} \end{cases}$$

Denote this expression by $pvol(r)$. The expected volume is then

$$\widetilde{vol}(r) = pvol(r)^E.$$

Then the expected k-NN radius is the r for which

$$k = (N-1) \cdot \widetilde{vol}(r)^{\frac{D_2}{E}}. \tag{2}$$

We will denote this r by $d_{nn}^{cutoff}(k)$ in the following.

In order to evaluate the quality of the new query-independent radius estimates, we computed the box counting dimension for a large number of high-dimensional real-world datasets. We then used Formula 1 to estimate the k-NN radius. In all cases, the values returned by the modified algorithm gave better radius estimations than the original algorithm. Table 3 shows how the relative error in NN-radius estimation changes for different values of r_{div} and k for the LANDSAT-dataset[5]. The error was calculated by comparing the radius estimate $d_{nn}^{cutoff}(k)$ with an average NN-radius obtained by running 500 random k-NN queries. As can be seen, by using finer step sizes in the box counting algorithm, the estimation error can be reduced significantly. As apparent, the obtained value for D_2 is more accurate when more intermediate steps are computed.

3 Query-dependent k-NN Radius Estimation

The box counting algorithm allows the estimation of the average k-NN radius. This calculation can be done statically for a given dataset. It is possible to achieve better results if we know the actual query location and have a data sample available. We will refer to this case as *query-dependent radius estimation.*

Assume we have a dataset of size N and we obtain an in-memory sample of size $N \cdot \sigma$; σ is called the *sampling rate*[6]. Furthermore, let q be a query point and k the number of its NNs we are looking for. We can obtain a first (rough) estimate of the query radius by computing the k NNs of q on the sample. Since the sample is stored entirely in memory, this can be done very efficiently.

Let us denote this estimate by r_{sample}. Since there might be a point that is a k-NN of q but is not in the sample, r_{sample} will usually be larger than the real k-NN-radius. If we knew how the radius changes with the sampling rate, we could compensate for this change. We can use Formula 1 for this purpose. Similar to $d_{nn}(k)$, we can calculate the k-NN radius for the sample dataset:

$$d_{sample-nn}(k) = \frac{1}{2} \cdot \left(\frac{k}{N \cdot \sigma - 1}\right)^{\frac{1}{D_2'}} \tag{3}$$

where D_2' denotes the Correlation fractal dimensionality of the sample dataset. Note that $d_{sample-nn}(k)$ can also be precomputed for a given dataset sample.

Once $d_{nn}(k)$ and $d_{sample-nn}(k)$ are known, we also know the expected rate of change in the NN-radius when moving from the sample to the full dataset, namely $\frac{d_{nn}(k)}{d_{sample-nn}(k)}$. Let us denote the query-dependent estimate of the k-NN radius by $r_{expected}$. It is given as

$$r_{expected} = \frac{d_{nn}(k)}{d_{sample-nn}(k)} \cdot r_{sample}. \tag{4}$$

When improving the static radius estimates $d_{nn}(k)$ and $d_{sample-nn}(k)$ by taking boundary effects into account, as discussed in Section 2.3, we can expect to obtain an improved query-dependent radius estimate. It is denoted by $r_{expected}^{cutoff}$ and defined as

$$r_{expected}^{cutoff} = \frac{d_{nn}^{cutoff}(k)}{d_{sample-nn}^{cutoff}(k)} \cdot r_{sample} \tag{5}$$

where $d_{sample-nn}^{cutoff}$ is defined similarly to d_{nn}^{cutoff}.

The quality of these query-dependent estimates is discussed in the next section.

4 Quality of the Query-dependent Radius Estimates

The quality of the query-dependent radius estimates can be expressed in several ways. We first examine

[5] Table 2 summarizes all datasets used in this paper.

[6] We assume the sample is obtained by randomly selecting $N \cdot \sigma$ points of the dataset.

UNIFORM	100,000 8-dimensional points distributed uniformly.
TEXTURE	26,697 48-dimensional points representing texture feature vectors obtained from the Corel Image Collection (transformed using KLT).
LANDSAT	275,465 60-dimensional points representing texture feature vectors of Landsat images (transformed using KLT).

Table 2: Datasets used in the experiments

the deviation of the radius estimate from the correct NN-radius for uniform datasets (Section 4.1). We then show that similar numbers hold for real datasets (Section 4.2). Finally, since $r_{expected}$ and $r_{expected}^{cutoff}$ are based on the fractal dimensionality D_2 of the dataset, we analyze how an error in D_2 affects the radius estimate (Section 4.3).

4.1 Expected Error for Uniform Data Distribution

In this section, we analyze the expected error in $r_{expected}$ analytically. Note that since $r_{expected}^{cutoff}$ was derived under uniformity assumptions, its expected error is naturally 0 and it is therefore not further analyzed in this section.

In order to derive a formula for the expected error of $r_{expected}$, let us assume a uniform data distribution and a normalized dataspace. In this setting, the expected query radius $d_{nn}^{cutoff}(k)$ is (according to Equation 2) the r for which

$$k = (N-1) \cdot \widetilde{vol}(r)^{\frac{D_2}{E}}.$$

Since $D_2 = E$ for uniform data, this is equivalent to

$$k = (N-1) \cdot \widetilde{vol}(r).$$

The same holds for the sample, resulting in the following equation for the expected sample radius r_{sample}:

$$k = (N \cdot \sigma - 1) \cdot \widetilde{vol}(r_{sample}).$$

Thus, we know the expected correct query radius for the full dataset and the sample. However, what is the expected value of $r_{expected}$? Since we assume uniformity, $D_2 = D_2'$ and therefore

$$\frac{d_{nn}(k)}{d_{sample-nn}(k)} \approx \left(\frac{N \cdot \sigma}{N}\right)^{\frac{1}{D_2}}.$$

Therefore,

$$r_{expected} \approx d_{sample-nn}^{cutoff}(k) \cdot \sigma^{\frac{1}{D_2}}. \qquad (6)$$

The relative error of $r_{expected}$ is then given as

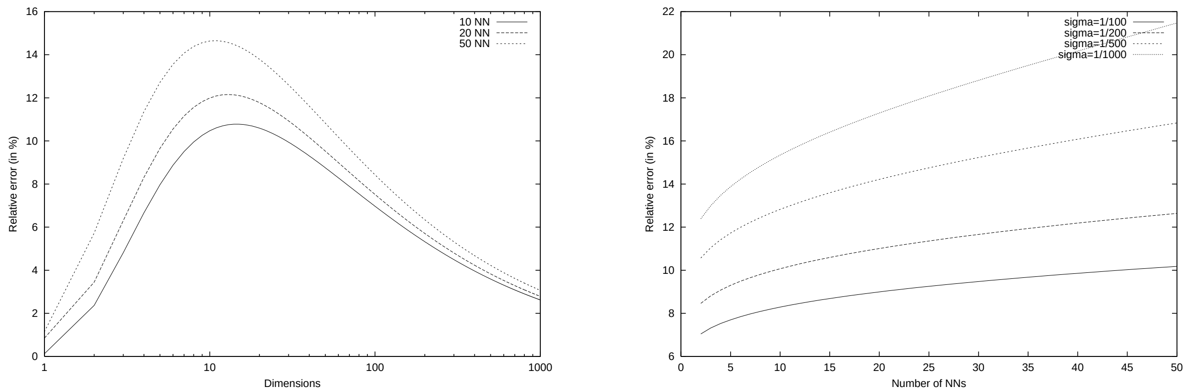
$$\frac{r_{expected} - d_{nn}^{cutoff}(k)}{d_{nn}^{cutoff}(k)}$$

which is plotted in Figure 5. Figure 5(a) plots the relative error as a function of the underlying data dimensionality. The sampling rate σ is assumed to be $\frac{1}{100}$. For lower dimensionalities, the error is small because the estimate $r_{expected}$ is close to the correct radius value. For higher dimensionalities, all points tend to be equidistant (see [8]). Therefore, the error in the k-NN radius estimate has to drop as well. This leaves us with a maximum at around 10 dimensions. Depending on k, this error ranges between 10% and 14%. Figure 5(b) plots the same relative error for varying k (number of NNs). With increasing k, the error of the radius estimate increases. However, it stays below 22% even for a sampling rate of 1/1000. This shows that (at least for uniform data) our query-dependent radius estimate is very close to the correct NN-radius. We can also see that the relative error is always positive for uniform data, meaning that $r_{expected}$ overestimates the correct radius. Real data, however, can cause an underestimation as we will see in the next section.

4.2 Measured Error for Real Data

This section shows how the relative error of $r_{expected}$ varies for real datasets (cf. Table 2 for a description of all datasets used). Figure 6 shows the results. Our first dataset, LANDSAT, contains more than a quarter million 60-dimensional points and is highly clustered. Figure 6(a) shows that — similar to the uniform case — with increasing k, the relative error of $r_{expected}$ increases slightly. The same holds for the sampling rate. The smaller the sample, the higher the relative error. However, two differences can be noted compared to the uniform case of the last section. First, for a sample rate of 1/100, the relative error drops below zero for $k > 3$. This means that $r_{expected}$ underestimates the real radius.

The second difference that can be seen from the graphs is that the error is higher for very small k and then drops quickly until it reaches a minimum. This can be explained as follows. Since the data is clustered, the distances between data points vary a lot (compared to uniform data). Since our queries are density-biased, the distance of data points from the query point varies a lot as well. If k is large, the effect of sampling is alleviated by the large number



(a) Relative error of $r_{expected}$ (analytical, $N = 100,000$, $\sigma = \frac{1}{100}$)

(b) Relative error of $r_{expected}$ (analytical, $N = 100,000$, $E = 60$)

Figure 5: Relative error of $r_{expected}$

of points in the query range. If k is small, sampling can cause $r_{expected}$ to be much larger than the real radius because the point distances have such a high variance. Even the compensation via the fractal dimensionality cannot counteract this effect because it describes the dataset globally, whereas this effect is local.

Figure 6(b) plots the relative error of $r_{expected}$ and $r_{expected}^{cutoff}$ for the TEXTURE dataset. The U-shape of the curve is more pronounced and the errors are overall higher. This is because TEXTURE contains less than $30,000$ points. A sample rate of $\sigma = 1/100$ leaves us with less than 300 points. The 50 NNs are $1/6$ of all points! But even with such a small sample, the errors stay below 65%, and are typically below 40%. This graph shows also the effect of taking the dataspace cutoff into account. As expected, the error is reduced for all values of k. For smaller k-values the compensation is less pronounced since again local effects prevail.

As a comparison, in the TEXTURE example, the statically computed radius $d_{nn}(k)$ bottomed out at about 400% relative error. This clearly shows the advantage of query-dependent radius estimation.

4.3 Stability of $r_{expected}$ with Respect to D_2

Unless the fractal dimensionality of a dataset is known in advance (e.g. for synthetic datasets), it has to be computed in some way (e.g. using the box-counting method described in Section 2.2). This computation can, however, induce some error in the fractal dimensionality. This error can again cause an error in the NN-radius estimation. In this section, we examine the impact of an error in D_2 on the quality of $r_{expected}$.

Let $\triangle D_2$ denote the absolute error in D_2. Then

$$\sigma^{\frac{1}{D_2+\Delta D_2}} = \sigma^{\frac{1}{D_2}} \cdot \sigma^{-\frac{1}{\frac{D_2^2}{\Delta D_2}+D_2}} = \sigma^{\frac{1}{D_2}} \cdot \left(1+\sigma^{-\frac{1}{\frac{D_2^2}{\Delta D_2}+D_2}} - 1\right).$$

Now let $\widetilde{r_{expected}}$ denote the query-dependent radius estimate with error. From Equation 4, we have

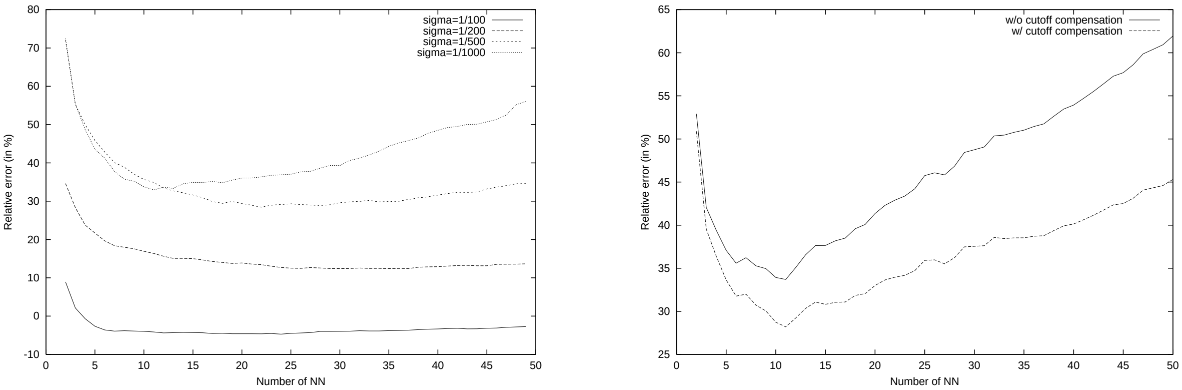
$$\widetilde{r_{expected}} = r_{sample} \cdot \sigma^{\frac{1}{D_2}} \cdot \left(1+\sigma^{-\frac{1}{\frac{D_2^2}{\Delta D_2}+D_2}} - 1\right).$$

In other words, the relative error in the radius estimation is

$$\sigma^{-\frac{1}{\frac{D_2^2}{\Delta D_2}+D_2}} - 1.$$

Note that this formula is independent of k.

For typical fractal dimensionalities between 4 and 8 and for typical sampling rates between $1/100$ and $1/500$, the relative error in the radius estimation corresponds nearly linearly to the relative error in D_2: an underestimation of D_2 by 10% causes an underestimation of $r_{expected}$ by approximately 10%, and vice versa for overestimations. This can be explained as follows. If D_2 is underestimated, the dataset appears lower dimensional than it is. Therefore, erroneously, the effect of sampling is underestimated and hence, too little compensation is applied



(a) Relative error of $r_{expected}$ (LANDSAT dataset)

(b) Relative error of $r_{expected}$ and $r^{cutoff}_{expected}$ (TEXTURE dataset, $\sigma = 1/100$)

Figure 6: Relative error of $r_{expected}$ for real data

to r_{sample}. On the other hand, if D_2 is overestimated, the effect of sampling is overestimated and too much compensation is applied to r_{sample}.

In our experiments on synthetic datasets (where D_2 is known), the relative error in D_2 computed with the modified box-counting algorithm was less than 2%. Even though it can not be known whether this holds true also for real clustered data, it is a strong indicator that the error caused by the fractal dimensionality calculation is negligible.

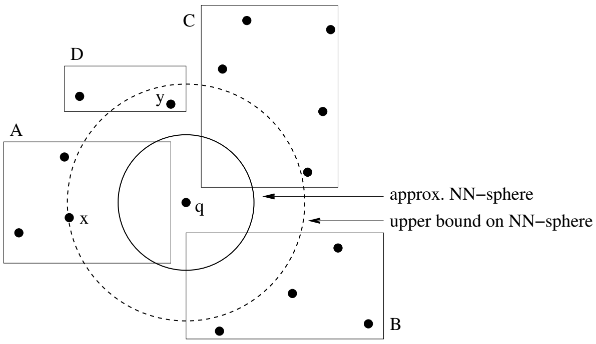


Figure 7: Upper Bound Computation

5 Obtaining an Upper Bound on NN-radius

In the previous sections, we saw how a query-dependent NN-radius estimate can be obtained via fractal dimensionalities and sampling. However, since this estimate may also underestimate the correct radius, its application can lead to false misses. If this is not acceptable, as is the case in our upcoming applications (cf. Sections 6 and 7), an upper bound on the radius has to be obtained. This section shows how a close upper bound on the NN-radius can be obtained if an approximate NN-radius is known and if the underlying index structure is page-based.

5.1 Upper Bound Computation

The basic idea behind the upper bound computation is shown in Figure 7. Assume, we are given the radius estimate as shown (labelled with "approx. NN-sphere"). Furthermore assume that the index structure distributed the dataset (indicated as dots) into four pages (labelled with A, B, C, and D). By reading all points from the pages that are intersected by the radius estimate (in this case, A, B, and C), we can determine the closest point in these pages (labelled with x). The distance to this point is an upper bound on the 1-NN radius (shown as "upper bound on NN-sphere"). Note that it is not necessarily the smallest upper bound (as point y shows).

Let us describe this estimation more formally now for arbitrary values of k. Let P be the set of index pages intersected by the approximate k-NN-sphere. Let C_{eff} be the effective capacity of each disk page. This is usually less than the page capacity due to the reduced page utilization in index structures. Assume that $C_{eff} \cdot |P| \geq k$. Then we can simply pick q's k closest points from the pages in P and calculate the smallest radius that covers them.

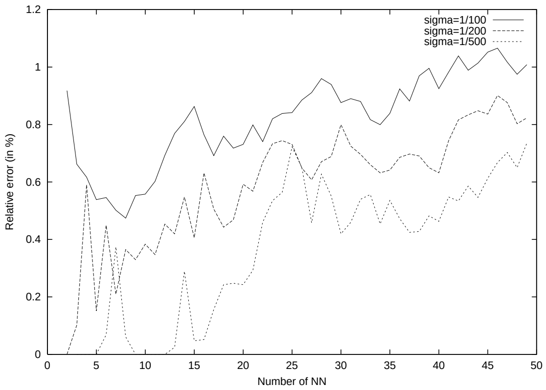


Figure 8: Relative error of r_{upper} (LANDSAT dataset)

We will denote this second radius estimate by r_{upper}. Obviously, r_{upper} is larger than the real radius since the corresponding query sphere centered at q covers k points read from the pages, but not necessarily the k globally closest ones.

What happens if $C_{eff} \cdot |P| < k$? This means that we do not have k points in the pages touched by the approximate k-NN-sphere. Obviously we can always pick the smallest MAXDIST from all pages as r_{upper}. This upper bound is not very tight, however. In our applications, we therefore make use of a data sample stored in memory. If no page is intersected by the approximate k-NN sphere, we just pick the smallest distance between q and the sample points as r_{upper}.

5.2 Quality of the Upper Bound

In order to examine the tightness of r_{upper}, we performed some experiments on real data. Figure 8 shows the results for the LANDSAT dataset and a R-tree index structure. The graph was obtained as follows. First, 500 random query points were picked. For each query point q, we computed r_{upper} as described above by choosing $r^{cutoff}_{expected}$ to be the initial radius estimate. We then computed the relative error (the amount of overestimation) of r_{upper} and plotted the averages for varying k-values. As can be seen, with increasing k, the error increases slightly but it stays overall below 1.2%. This shows that the points read from the pages touched by the initial radius estimate produce a very accurate upper bound.

It has to be noted, however, that this holds only for index structures that cluster points in pages, such as the R*-tree, the X-tree, or the SS/SR-tree. Nonclustering index structures cause a high overestimation of r_{upper}, as we will see for the VA-file later.

Input: Query point q, number of NNs k
Output: k NNs of q

(1) Let $queue := \{root\}$.
(2) Pick page or point p from $queue$ which has smallest MINDIST from q.
(3) If p is a page, read p into memory and place its children in $queue$.
(4) Otherwise, report p as a NN.
(5) If k NNs were reported, stop. Otherwise, goto (2).

Figure 9: The access optimal k-NN query algorithm

6 Application 1: Accelerated k-NN Query Algorithm

In this section, we present the first application of our query-dependent radius estimators. It aims at accelerating k-NN query processing by reducing the number of random disk accesses.

6.1 The Access Optimal Algorithm

Before going into the details, let us first revisit the optimal k-NN query algorithm for page-based tree index structures by Hjaltason and Samet [14]. Their algorithm works as follows (cf. Figure 9). First, a priority queue is initialized with the root page of the tree (Step 1). This queue is sorted by the MINDIST between q and the page. For a query q, an element p is removed from the queue whose MINDIST is closest to q (Step 2). If this element was a point, it is reported as a NN (Step 4). Otherwise, the element was a page. In this case, the page has to be fetched from disk and the elements stored in it are sorted into the waiting queue by their MINDIST. If the page is an inner page, the elements are typically pointers to child pages. If it is a leaf page, the elements are data points.

We assume in the following that all levels of an index tree is stored in memory. Or, in other words, only the leaf level pages have to be fetched from disk. This means that Step (3) causes only disk I/O if p was a leaf page. From a disk access point of view, the whole algorithm then reduces to the following: sort all leaf pages by their MINDIST from q and fetch them from disk in that order while keeping a list of the current k closest NNs found in the read pages. The algorithm terminates as soon as the most distant NN in the list is closer to q than the next leaf page to be read. Hjaltason and Samet prove that this algorithm is optimal in that it accesses a minimal number of pages, namely only the pages intersected by the k-NN query sphere. We will refer to this optimality as *access optimal.*

It is clear that especially for high dimensional data, the pages close to q can be scattered widely over the disk. This causes a large amount of seek operations during the query processing. If all pages to be read during a NN query would be known in advance, these expensive seeks could be avoided. We refer to an algorithm that minimizes the overall response time as *response time optimal* algorithm. Our new accelerated algorithm reduces the response time significantly by utilizing our new radius estimators. As we will see, its performance is very close to a (hypothetical) response time optimal k-NN query algorithm.

6.2 The New Algorithm

This basic idea of our new k-NN query algorithm is as follows. For every incoming NN-query, we compute $r_{expected}^{cutoff}$ (cf. Section 3) as a first estimate of its NN-query radius. Using this estimate, we can perform a range query on the full dataset. Since all pages are known in advance for range queries, the amount of seek operations can be minimized by employing a specific page read scheduler. If the first estimate was too large, at least k points were in the range, and we can return the k closest ones as the result set. If the first estimate was too small, less than k points were in the range and we need to increase the search radius and perform another range query. Since we have already a first radius estimate, we can use r_{upper} (cf. Section 5) as this second estimate. Since r_{upper} is an upper bound on the real radius, we read at least k points during the second range query. Therefore, at most two range queries are necessary to answer the k-NN query, resulting in a worst case cost of two linear scans over a subset of the data pages (if an optimal page read scheduler is used). Figure 10 gives the algorithm in more detail.

Note that a simple modification yields an approximate k-NN query algorithm with less I/O cost. Instead of performing a second range query, we can stop the algorithm after step (3) and simply report the k closest points encountered during the first range query. This approximate k-NN query algorithm is given in Figure 11. As we saw in Section 4.2, for typical applications, $r_{expected}^{cutoff}$ is already close to the real radius. Therefore, the quality of the approximate NNs can be expected to be high.

Before being able to run these algorithms for a given dataset, we need to precompute a sample of the dataset, the Correlation fractal dimensionality D_2 and D'_2 of the full dataset and the sample dataset, respectively. This can be done in $O(N \log N)$ time with the box counting algorithm discussed in Section 2.2. Note that this cost has to be paid only once and is therefore amortized over time.

Input: Query point q, number of NNs k
Output: k NNs of q

(1) Perform a k-NN query on sample in memory; this yields a k-NN-radius r_{sample}.
(2) Let $r_{expected}^{cutoff} := r_{sample} \cdot \frac{d_{nn}^{cutoff}(k)}{d_{sample-nn}^{cutoff}(k)}$.
(3) Perform a range query with radius $r_{expected}^{cutoff}$ on full dataset; this results in a set R of data points and a set P of index leaf pages accessed during the query.
(4) If $|R| \geq k$, return the k closest points from R, stop.
(5) Otherwise (more NNs need to be found), compute the k closest points stored in P and calculate the smallest radius r_{upper} enclosing them.
(6) Perform a modified range query around q with radius r_{upper} on full dataset; this results in a set R' of data points and a set P' of index leaf pages accessed during the query.
(7) Return the k closest points from R', stop.

Figure 10: The accelerated k-NN query algorithm

Input: Query point q, number of NNs k
Output: k approximate NNs of q

(1) Perform a k-NN query on sample in memory; this yields a k-NN-radius r_{sample}.
(2) Let $r_{expected}^{cutoff} := r_{sample} \cdot \frac{d_{nn}^{cutoff}(k)}{d_{sample-nn}^{cutoff}(k)}$.
(3) Perform a range query with radius $r_{expected}^{cutoff}$ on full dataset; this results in a set R of data points and a set P of index leaf pages accessed during the query.
(4) Return the k closest points stored in P, stop.

Figure 11: The approximate k-NN query algorithm

In the next section, we discuss the best strategy for accessing the disk pages during query execution.

6.2.1 Disk Page Read Strategy

Only steps (3) and (6) of our accelerated k-NN query algorithm induce disk I/O due to the two range queries. The range query in step (3) is a regular range query provided by the indexing system. The range query in step (6) is modified as follows. Since the k closest points from the pages P are already known, this range query needs to read only the pages from $P' - P$. Note that it is possible to perform step (6) with a regular range query without affecting the correctness of the algorithm but the modified version reduces the amount of unnecessary page reads and thereby leads to lower I/O cost.

In general, both range queries can make use of the heuristic suggested by Seeger et al. [25] to minimize the I/O cost for reading a set of disk pages. The authors assume that there is a certain set of pages on disk that are to be read into memory. Between these pages are so-called *gap pages* that do not contain any useful information. In order to minimize the I/O cost, it can be advantageous to read gap pages instead of seeking over them since disk seeks are expensive. Seeger et al. [25] present a heuristic to determine the maximum number g_{max} of gap pages that should be read. If the number of gap pages is more than g_{max}, a seek operation is performed.

In more detail, g_{max} is calculated as

$$g_{max} = \frac{t_{seek}}{t_{xfer}} - \frac{N}{C_{eff} \cdot |P|} - \frac{1}{\log(1 - \frac{C_{eff} \cdot |P|}{N})}$$

where P corresponds to the set of pages to be read in the first range query of our accelerated k-NN query algorithm. The average disk seek and transfer times are denoted by t_{seek} and t_{xfer}, respectively. Note that, even though this formula was derived for unlimited buffer size, we use it also for the limited buffer case in our experiments. We found that the improvement is marginal when using the correct (but more complex) formula for the limited buffer case.

6.3 Experimental Results

In order to evaluate our accelerated k-NN-query algorithm, we performed experiments for a large number of datasets. Here we present the results for a synthetic (UNIFORM) and two real datasets (LANDSAT and TEXTURE). For each dataset we ran 100 k-NN queries where k varied between 10 and 50 and measured the amount of seeks and page transfers. All queries are density-biased, i.e. query points are picked with higher probability from a region with higher density. The sampling rate is always 1/100. For the underlying harddisk we assume a 20 MB/s transfer rate and an average seek time of 10 ms. This, together with the measured seek and transfer numbers, is then used to compute the average query response time.

Furthermore, we consider three different index structures in our experiments: the R*-tree, the X-tree, and the VA-file (Section 6.3.3 discusses how other indexing schemes fit into our algorithm). Here is how we utilize the range query algorithms of these index structures:

R*-tree Range queries in the R*-tree [2] are performed by descending those subtrees whose bounding box overlaps the query range until leaf pages are reached. For each reached leaf page, the entries that fall into the range are determined and returned as result set. Since the whole leaf page has to be read anyway, we can additionally inspect the other points in the page to support the computation of r_{upper}.

X-tree Range queries in the X-tree [7] work the same way as in the R*-tree. The only difference is the way the bounding boxes are constructed. In our experiments, we used the bulkloaded X-tree as presented in [6]. This bulkloading algorithm ensures that no overlaps occur between the bounding boxes and that the page utilization is maximized.

VA-file For the VA-file [27], we used the following range query algorithm: in a first step, the approximate vectors that intersect the query range are determined. This is done without disk I/O since the approximate vectors are stored in memory. In a second step, the corresponding exact vectors are read from disk. For this step, the disk page scheduler is used to minimize seek time. Since the VA-file organizes the exact vectors in pages, we can inspect the other vectors in a read page to support the computation of r_{upper}.

Note, however, that the VA-file does not cluster points in pages. It rather stores the points in whatever order they are given. This causes a deterioration of the second radius estimation quality as we will see in the experimental section.

In case of the R*-tree and the X-tree, we used the optimal NN-query algorithm proposed by Hjaltason and Samet [14]. For the VA-file, we used the near-optimal NN-query algorithm proposed by Weber and Blott [27]. For all index structures used, the index page capacity is 8 KBytes and that the upper part of the index tree is kept in memory. Therefore, only leaf page accesses cause disk I/O.

6.3.1 Results for Synthetic Data

In this and the next subsection, we compare the performance of the access optimal NN-query algorithm with our accelerated version, a hypothetical algorithm with perfect radius estimator, and the linear scan. The hypothetical algorithm, we will denote it by "ORACLE", always picks the correct query radius for the first range query and reads therefore always the minimal number of pages possible for our algorithm. In other words, ORACLE provides a lower bound of our algorithm's response time. Note further that the linear scan reads all pages in a linear fashion and therefore requires no seek operations. It is therefore a good point of reference for judging the impact of filtering out pages by our radius estimates.

The first set of experiments was performed on the 8-dimensional UNIFORM dataset. Figure 12 shows the results for the R*-tree and X-tree. For each index structure and k-value, we show four I/O costs: for the access optimal query algorithm (denoted by "Orig."), for our accelerated query algorithm (denoted by "Accel."), for ORACLE (denoted by "Oracle"), and for the linear scan (denoted by "Scan"). Each cost value is an average over 100 queries.

Figure 12(a) shows the I/O costs for the R*-tree. The cost of the original query algorithm increases with increasing k and is well above the linear scan. The accelerated query algorithm performs slightly worse than the linear scan. This is due to the high amount of overlaps in the bounding boxes of an R*-tree. The same holds for ORACLE: due to the high overlap, a large amount of pages have to be read, resulting in a higher read cost than for the linear scan[7]. Despite these overlaps, our technique speeds up the NN-query by a factor of $2-4$ (depending on k) and resembles ORACLE closely. Note that our technique can never perform worse than two linear scans over a subset of the data (when employing an optimal page read scheduler), since we perform at most two range queries. Obviously, the original query algorithm cannot give this guarantee.

Figure 12(b) shows the results for the X-tree. Since we use an elaborate bulkloading algorithm to build the tree, the page utilization is high and no page overlaps occur. This improves the query performance drastically. For 10-NN queries, the index is even faster than the linear scan. For larger k, the performance deteriorates again and becomes worse than scanning. When using our acceleration technique, the performance is improved even further, as can be seen in the second bars. Compared to the original X-tree, we achieve speed-ups between 3 and 5. Additionally, the accelerated query algorithm is at least 3 times faster than the linear scan.

6.3.2 Results for Real Data

In this section, we present experimental results for two real datasets. The first set of experiments was performed on the LANDSAT dataset. The corresponding plots can be found in Figure 13.

In case of the R*-tree (Figure 13(a)), we observe much higher query costs than for the UNIFORM dataset. The reason is the increased amount of overlap that occurs for this 60-dimensional dataset. When applying our technique, the I/O cost is reduced by a factor of about 9. The cost again ranges between 1 and 2 linear scans and only slightly above ORACLE.

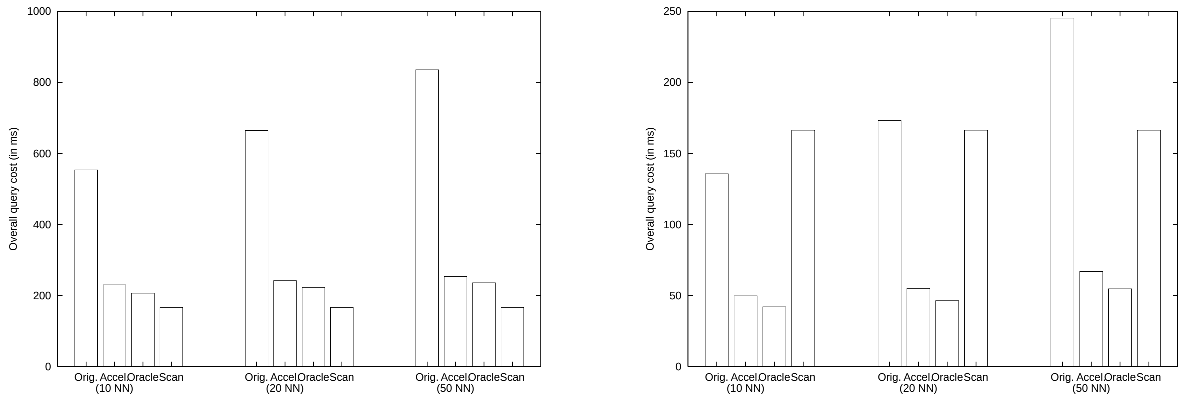
Two points should be noted from the X-tree plots (Figure 13(b)). First, the X-tree stays well below the linear scan for all queries. This is even more remarkable considering the fact that the X-tree algorithm performs only random page accesses, whereas the linear scan uses only sequential ones. This shows that for highly clustered data, a well tuned index structure is still very useful for NN-queries.

Even more remarkable is the second observation that our accelerated query algorithm outperforms this well-tuned index structure by a factor of $3-5$. The reason is simple. Even though the accelerated query algorithm needs to read more pages, it knows them in advance and can read as many of them sequentially as necessary. In a way, it gets the best of both worlds, the X-tree and the linear scan: it reads nearly as few pages as the optimal NN-query algorithm of the X-tree, and it performs nearly as few seeks as the linear scan.

Figure 13(c) shows the performance results for the VA-file. Note that the VA-file performance deteriorates similar to the R*-tree and X-tree with increasing k. Moreover, its performance even approaches the linear scan. The reason for this behavior lies in the second phase of the VA-file query algorithm: since the second phase reads the exact vectors according to their approximative distance from the query point, most disk accesses will be random. The linear scan, on the other hand, causes no random seeks.

Even though our algorithm improves the query performance of the VA-file, the speed-up is less than for the other index structures. This can be explained by the way the index structures cluster the data. The R*-tree and X-tree perform a clustering of points in pages, whereas the VA-file does not. It stores the points in whatever order they were given. Distant points can therefore end up in the same VA-file page. Since our algorithm determines the second radius estimate from the points found in the pages of the first range query, this can cause a high overestimation. A large second radius estimate leads to high I/O costs for the second range query. This can also be inferred from the ORACLE performance: since ORACLE knows the exact query radius, its cost is much lower than for our algorithm. We can therefore conclude that our second radius estimator works best for index structures that perform clustering of

[7] An optimal page read scheduler would have caused the I/O cost of ORACLE to be slightly below the linear scan.



(a) R*-tree (b) X-tree

Figure 12: Overall query cost for UNIFORM dataset

points in pages, while non-clustering index structures cause a higher I/O cost in the second phase.

Finally, Figure 14 shows the experimental results for the TEXTURE dataset. The result looks similar to the LANDSAT dataset. Our accelerated approach performs about 15 times faster than the R*-tree and $3-4$ times faster than the X-tree. The accelerated X-tree shows a speed-up of 5 compared to the linear scan and resembles ORACLE closely.

6.3.3 Other Index Structures

In this section, we briefly elaborate on two other indexing schemes and how they fit into our accelerated k-NN query framework: the SS/SR-tree and the Pyramid Technique.

SS/SR-tree Both the SS-tree [28] and the SR-tree [18] have range queries similar to the R*-tree. The only difference is the bounding shape. Our technique will therefore yield results similar to the R*-tree.

Pyramid Technique The Pyramid technique [5] partitions the dataspace into $2d$ hyper-pyramids. These pyramids are then used to map the d-dimensional points onto 1-dimensional values that can be stored in a B+-tree. In [5], Berchtold et al. introduce a range query algorithm for this index structure. They transform the d-dimensional query range into a set of 1-dimensional range queries for the B+-tree. During the execution of such a range query, B+-tree pages need to be fetched from disk. The disk read scheduler can be used to accelerate this step. Additionally, the points that are stored in a page but not included in the range query, can be used to support the second radius estimator. Due to the design of the pyramid mapping function, some amount of locality between points is preserved. We can therefore expect meaningful second radius estimates. However, the main achievement is the support of NN queries for an index structure that offers only range queries.

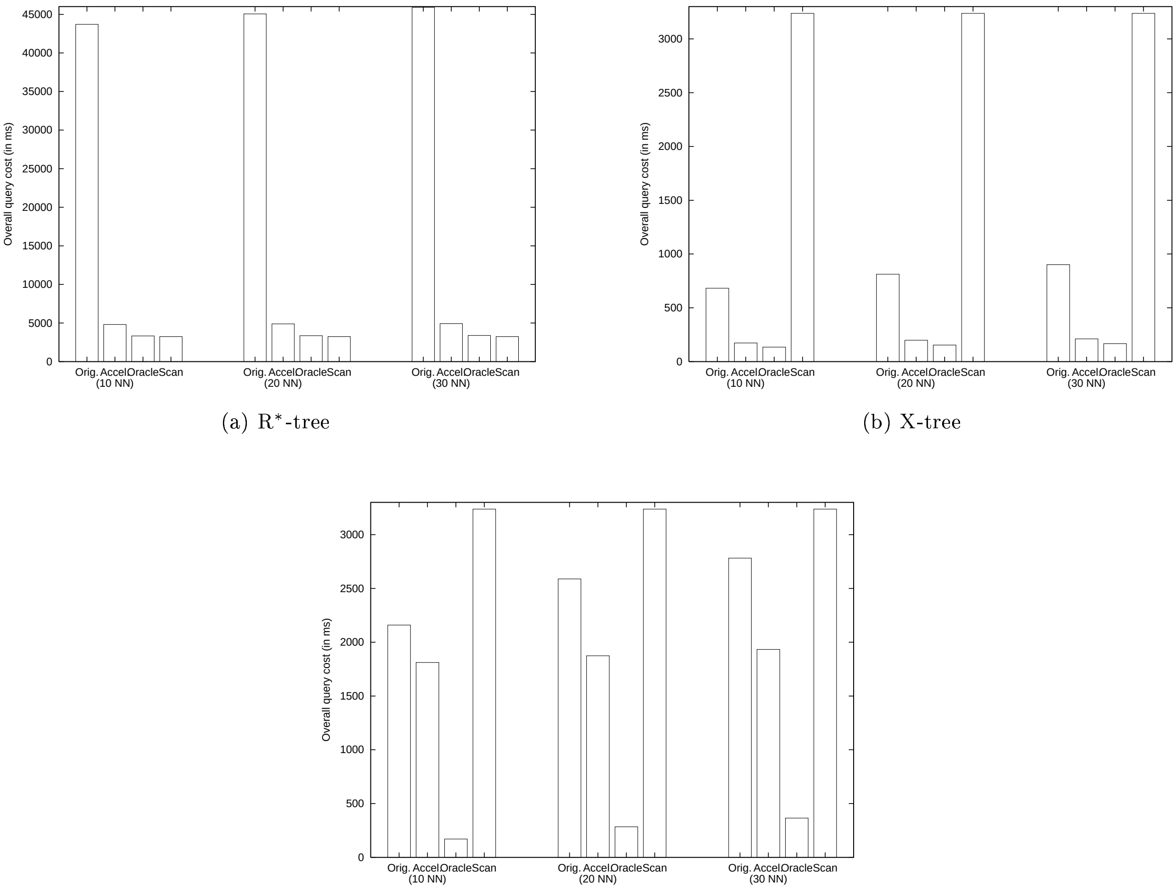
7 Application 2: Accelerated Disk-based All-NN Query Algorithm

This section discusses the second application of our improved radius estimators: an accelerated all-NN query algorithm.

7.1 The Block-nested-loops Join Algorithm

The problem of computing all NNs of one dataset relative to another dataset is not new. It can be described as follows. Assume, we are given two point datasets, D_1 and D_2. For each point in D_1 we want to determine its NN among all points in D_2. This problem can easily be generalized to all k-NN queries.

One straightforward way of answering such queries is by performing $|D_1|$ regular NN queries on D_2 (which may then be accelerated using our technique presented in the last section). However, this solution causes a high number of disk seeks for high-dimensional datasets since typical caches are not large enough to store common pages between queries. It would be beneficial to perform these operations in bulks, all points in one page at a time.



(a) R*-tree

(b) X-tree

(c) VA-file

Figure 13: Overall query cost for LANDSAT dataset

Input: Dataset D_1, dataset D_2
Output: NNs of all $q \in D_1$ taken from D_2

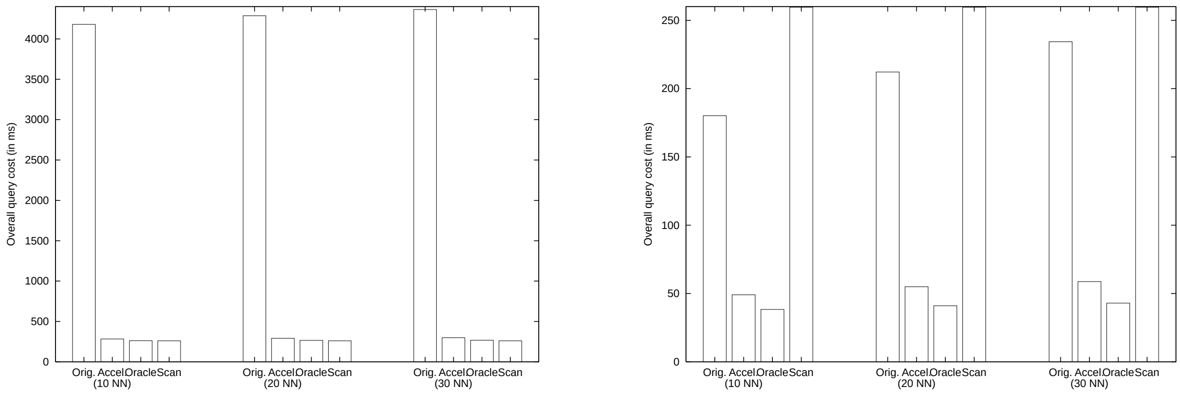
(1) Scan D_1 in chunks of size $M/2$ (half of the buffer);
for each chunk of D_1-pages:
(2) Read $M/2$ pages from D_2 (fill other half);
for each chunk of D_2-pages:
(3) Update NNs for each point in the D_1-pages.
(4) Output NNs for all points in the D_1-pages.
(5) Stop.

Figure 15: The block-nested-loops join algorithm for answering all-NN queries

One way of achieving this is by utilizing a block-nested-loops join algorithm (cf. for example [24]) since the all-NN problem can be viewed as the problem of joining D_1 and D_2 with the join condition being $\sigma(x, y) =$ "x has y as a NN" where $x \in D_1$ and $y \in D_2$. The pseudocode for the block-nested-loops join algorithm is given in Figure 15. The algorithm consists mainly of two nested loops, one reading pages from D_1 and one reading pages from D_2. These page reads are performed in a sequential fashion in order to reduce disk seeks. We assume that a buffer of size M (in number of pages) is provided. The algorithm fills the first half with D_1-pages and the second half with D_2-pages. For each point in a D_1-page we store additionally its best NN so far. This information is updated in Step (3).

7.2 The New Algorithm

In this section, we show how our new radius estimators can be used to accelerate the block-nested-loops join algorithm for answering all-NN queries. The basic idea is as follows: instead of estimating the NN-radius for a single point, we estimate it for all points in the D_1-pages stored in the buffer. Once we have calculated the NN-radius-estimates for all points of a page of D_1, we can predict which pages of D_2 have to be accessed in order to compute the join. This is where the I/O cost is reduced significantly com-



(a) R*-tree (b) X-tree

Figure 14: Overall query cost for TEXTURE dataset

Input: Dataset D_1, dataset D_2
Output: NNs of all $q \in D_1$ taken from D_2

(1) Scan D_1 in chunks of size M (whole buffer);
for each chunk of D_1-pages:
(2) Compute expected influence region $R_{expected}(P)$
for each D_1-page P.
(3) Scan D_1 in chunks of size $M/2$ (half of the buffer);
for each chunk of D_1-pages:
(4) Read $M/2$ consecutive pages from D_2
that intersect at least one influence region
of the D_1-pages (fill other half of M);
for each chunk of D_2-pages:
(5) Update NNs for each point in the D_1-pages.
(6) Update $R_{expected}(P)$ for each D_1-page P.
(7) Read all pages (at most $M/2$ at a time)
of D_2 that intersect at least one influence
region of the D_1-pages (fill other half of M);
for each chunk of D_2-pages:
(8) Update NNs for each point in the D_1-pages.
(9) Output NNs for all points in the D_1-pages.
(10) Stop.

Figure 16: The accelerated all-NN query algorithm

pared to a block-nested-loops join algorithm which accesses *every* page of D_2 for a page of D_1.

Let us discuss the new algorithm in more depth. It is given in Figure 16. In steps (1) and (2), a so-called *influence region* $R_{expected}(P)$ is computed for every page P in D_1. This can be seen as a generalization of the computation of $r_{expected}$ for the accelerated k-NN query algorithm. The influence region can be regarded as the union of the expected NN-spheres of all points in P, as shown in Figure 17. In this example, page P contains three points. The influence region is shown as the shaded area. Two of the D_2-pages (shown as dashed boxes) are intersected by P's influence region and have to be retrieved in Step (4) of the algorithm. Page A would

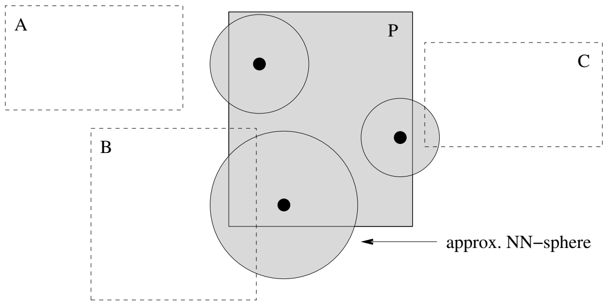


Figure 17: Influence region of page P

not be retrieved at this time.

In case D_2-pages that have to be retrieved are less than $M/2$ pages apart on disk, they can be read together in Step (4). This helps reducing disk seeks. In our implementation, we access the $M/2$ pages around the intersected D_2-page in disk-placement order. Since most index structures try to place pages nearby on the disk that are spatially close in the data space, pages stored before and after the intersected page have a high probability to contain points that are also close by. This leads to better updated estimates in Step (6) of the algorithm.

After $M/2$ D_2-pages were read in Step (4), the NN-information is updated in Step (5) similar to the block-nested-loops join algorithm. Then the points found in the $M/2$ D_2-pages is used to improve $R_{expected}(P)$ for each D_1-page in the buffer. This can be seen as the equivalent to Step (5) of our accelerated k-NN query algorithm. Similar to there, the updated $R_{expected}(P)$ represents an upper bound on the correct influence region of P.

The updated $R_{expected}(P)$ is used in Step (7) to read the pages from D_2 that are intersected by this upper bound influence region. In the remainder of the algorithm, the NN-info of all points in D_1-pages are updated and output once all D_2-pages are processed.

An important observation is that *the accelerated all-NN search algorithm can never perform more than twice as many page accesses as the block-nested-loops join algorithm.* This can be seen as follows: in the worst case, all pages of D_2 are accessed in line (4) and again in line (7), since Step (4) reads additional (possibly unnecessary) pages that may be refetched in Step (7).

Let us close this discussion with two technical remarks: first, for a concrete implementation, the reads in Step (7) would ideally be implemented using the near-optimal page read algorithm discussed in Section 6.2.1. And second, the NNs should be stored in a compressed way in Step (9). For example, for two datasets storing 10^6 points, it is sufficient to store $10^6 \cdot \log(10^6)$ bits, which is less than 3 MBytes. Since this is well within the limit of typical memory sizes, we assume that Step (7) does not cause any disk I/O.

7.3 Experimental Results

This subsection presents our experimental results for the accelerated all-NN-query algorithm. We ran two experiments: on two-dimensional synthetic uniform data, and on 60-dimensional real data. The uniform datasets consist of 10,000 points each and the real datasets consist of approximately 13,000 points each extracted randomly from the LANDSAT dataset.

In the case of joins, the overall running times are not necessarily I/O-bound. Due to the high number of distance computations required, the CPU cost becomes an important factor. Hence, in our experiments, we compare not only the I/O costs but also the amount of distance computations performed. The I/O cost is computed as before by counting the number of seeks and page transfers. The CPU cost is measured as the number of distance computations required.

Figure 18 depicts both cost measures and the combined CPU and I/O cost for both experiments. The y-axis shows the corresponding cost measure, whereas the x-axis indicates the buffer size used. This size is given as the percentage of the overall number of pages.

Let us first focus on the I/O cost. For the uniform data, the accelerated algorithm (ACC) clearly outperforms the block-nested-loops (BNL) approach for all buffer sizes (cf. Figure 18(a)). In case of the LANDSAT data, the accelerated algorithm is superior for smaller buffer sizes but degrades for larger ones. BNL benefits from an increasing buffer size since fewer seeks need to be issued. ACC, on the other hand, can not make as much use of the larger buffer since consecutive data pages do not necessarily contain nearby points in this high-dimensional setting. This leads to high upper bounds for the influence region estimation. Remember, however, that ACC can never perform more than twice as many page accesses as BNL.

When looking at the CPU cost, in both experiments, ACC outperforms BNL for all buffer sizes. For the lower-dimensional data, only 1/5 of all distance computations have to be performed. For the higher-dimensional data, only 1/4 of the computations are necessary. This shows that the pruning caused by our influence region estimation is very effective. Note, however, that the cost of a single distance computation is much higher for 60-dimensional data than for 2-dimensional data, making the algorithm CPU-bound for high-dimensional datasets. This can be seen in Figures 18(e) and 18(f) where the sum of CPU cost and I/O cost is plotted. These plots were obtained by multiplying the observed number of distance calculations with the measured time for one such calculation. In the uniform case, the overall response time drops by a factor of 4 and in case of LANDSAT by a factor of 3.

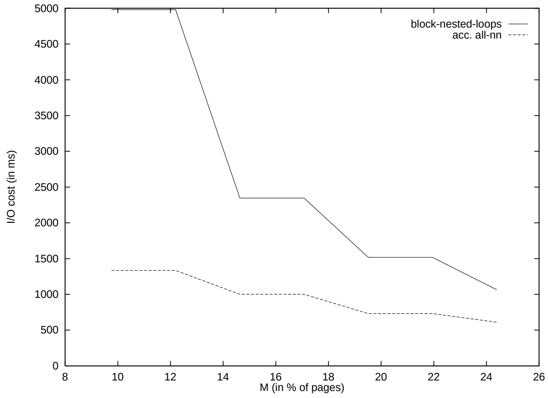
8 Previous Work

Similar to the structure of the paper, we divide the previous work section in three areas: work on the fractal dimensionality, on accelerating k-NN queries, and on accelerating all-NN queries.

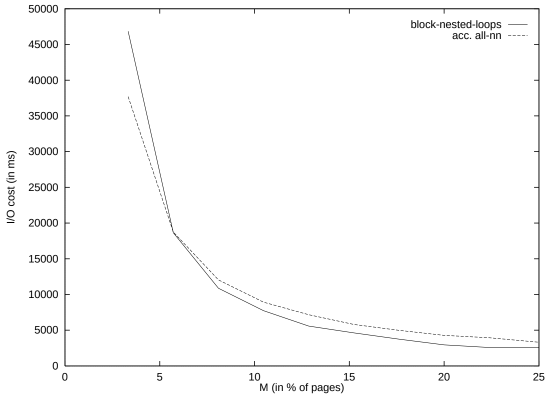
Fractal Dimensionality and NN-radius Estimation Faloutsos et al. [13] presented the first cost model for R-trees based on the fractal dimensionality which is claimed to be the "inherent dimensionality" of a dataset. This first model was restricted to range queries but later work by Papadopoulos and Manolopoulos [23] extended it for 1-NN queries in R-trees. Korn et al. [20] present a version for k-NN queries.

Common to all these approaches is the use of the box-counting algorithm with a step size r_{div} of 2. Moreover, none of these approaches take boundary effects into account.

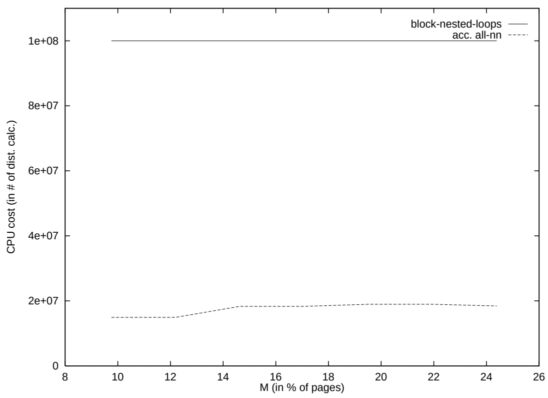
Accelerated k-NN Query Algorithms Many researchers have investigated models for predicting



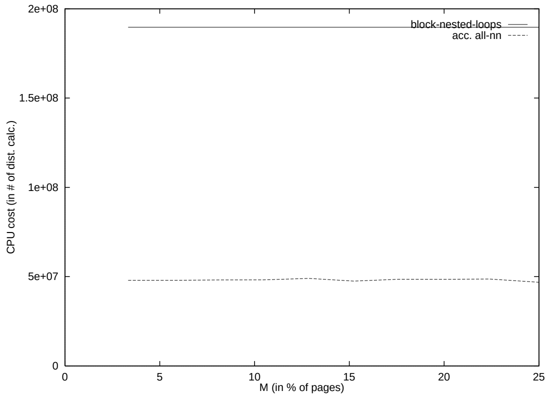
(a) Uniform data (I/O cost)



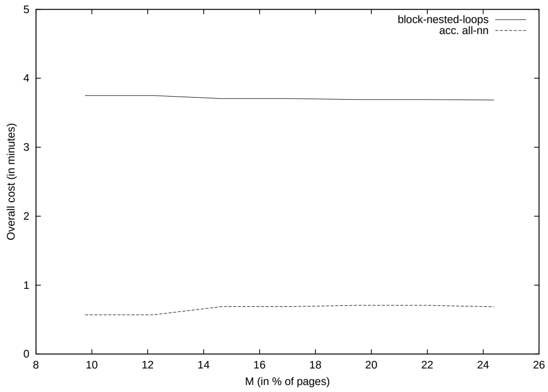
(b) LANDSAT data (I/O cost)



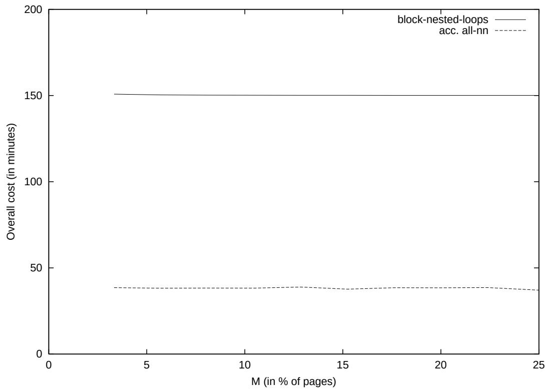
(c) Uniform data (CPU cost)



(d) LANDSAT data (CPU cost)



(e) Uniform data (overall cost)



(f) LANDSAT data (overall cost)

Figure 18: Costs of accelerated all-NN-search algorithm

the performance of index structures. Some of this work focuses on predicting the index performance for an average workload, some of it focuses on the expected cost of a single query. Since the technique presented in this paper aims at accelerating single queries, we will concentrate on the latter body of work.

The work that comes most closely to our technique was presented by Berchtold et al. [4]. They propose a new index structure, called IQ-tree, and give a probability-based method to optimize page reads during NN queries. The authors estimate the probability that a page will be read during a query and use this probability to decide which pages should be read together with the next page in order to avoid expensive seeks. The probability that a page will be accessed is defined as the percentage of the page volume covered by the NN sphere.

In contrast to our technique, this model assumes uniform data distribution within the pages. This can lead to high estimation errors in the access probability. Since the page reads are controlled by this probability estimate, many unnecessary pages are accessed. Furthermore, at least one page needs to be accessed to get an initial radius estimate.

Other authors extend index structures by statistical information, which can be used to estimate the query radius and the query cost. Ciaccia et al. [12] extend the M-tree nodes by statistics (the distance distribution) in order to predict the CPU and I/O cost of range and NN queries. Most work is based on histograms over the dataset. Theodoridis and Sellis [26] give a model for predicting the performance of range queries on R*-trees. They generate what they call a *density surface* which is basically a two-dimensional histogram representing the local densities of the dataset. Acharya et al. [1] show how the prediction error of histograms can be reduced by reducing the variance of densities within the histogram regions. Korn et al. [19] demonstrate how discontinuities at the histogram region edges can be avoided by using splines. Finally, Jin et al. [17] extend the histogram approach for spatial non-point data.

The advantage of these techniques is their high accuracy in modeling data by considering local effects. A disadvantage of the histogram approaches is that they are not applicable in high dimensions since either the number of histogram regions becomes too large or these regions contain too much empty space and become inaccurate.

In [22], sampling is used to overcome this problem. In contrast to this paper, the sample is used to predict the overall query cost of a given index structure. More specifically, the sample is used to predict the index page layout. Here, on the other hand, we use the sample to predict the query radius for one particular query. Moreover, we employ the fractal dimensionality to compensate for sampling, while in [22], uniformity is assumed to adjust the page layout prediction.

Accelerated All-NN Query Algorithms Several papers (e.g. [11, 16]) focus on efficient processing of spatial joins on R-trees. However, they do not take seek and transfer costs into account. Instead, they try to minimize the number of page misses during a join. Since these algorithms perform the join by navigating through the index tree rather than through the disk pages, they cause many random page accesses.

Other papers (e.g. [21], [9]) investigate new indexing and query processing techniques in order to reduce the number of page accesses during spatial joins. In contrast to our approach, these techniques do not make use of existing index structures and are restricted to join conditions of the type $\sigma(x, y) =$ "the distance between x and y is less than d".

Hjaltason and Samet [15] propose an incremental join algorithm that returns the closest join pair first. This work does not try to accelerate the search for the full join result as we do.

A different approach by Braunmüller et al. [10] reduces the I/O and CPU-cost of multiple NN-queries. They process one query after the other but take results of older queries into account in order to accelerate newer ones. This process is certainly faster than single NN-queries but it can be expected to perform worse when searching for all NNs since the page accesses are still random. Our approach, on the other hand, tries to maximize the number of sequential page accesses.

9 Conclusions

We showed in this paper how query-dependent query range estimates can be used to accelerate two important query algorithms. In order to obtain the estimates, we introduced an improved way of computing the fractal dimensionality of data. We then showed how the error in estimating query-independent query ranges can be reduced by taking boundary effects into account. This leads to estimation errors that are lower than for previous techniques. As another contribution, we introduced the notion of query-dependent query range estimates and gave an algorithm for their computation based on sampling. Our analysis indicates that the estimation error for the expected radius is below 14% and that errors in the fractal dimensionality estimation have only minor impact on the accuracy. For the upper bound estimate, the observed error was even lower, namely below 1%.

In the experimental section, we introduced two new algorithms that we accelerated using the estimates discussed in the first part of the paper: one for answering k-NN queries, and one for answering all-NN queries. The accelerated k-NN query algorithm performs up to 15 times faster than the R*-tree and nearly 5 times faster than a bulkloaded X-tree-like index structure. Even the VA-file, which does not cluster points in pages, benefits from our acceleration. Surprisingly, our technique outperformed the NN-query algorithms proposed as part of these index structures. The all-NN-query algorithm we presented outperforms a block-nested-loops join algorithm by a factor of $3-4$.

For the future, we plan on applying our technique to index structures for general metric spaces (e.g. M-tree) and index structures storing only a subset of the data dimensions (e.g. reduced via KLT).

References

[1] Swarup Acharya, Viswanath Poosala, and Sridhar Ramaswamy. Selectivity estimation in spatial databases. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 13–24, 1999.

[2] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 322–331, 1990.

[3] Alberto Belussi and Christos Faloutsos. Estimating the selectivity of spatial queries using the 'correlation' fractal dimension. In *Proceedings of the Int. Conf. on Very Large Data Bases*, pages 299–310, 1995.

[4] S. Berchtold, C. Böhm, H. V. Jagadish, H.-P. Kriegel, and J. Sander. Independent quantization: An index compression technique for high-dimensional data spaces. In *Proc. Int. Conf. on Data Engineering*, 2000.

[5] S. Berchtold, C. Böhm, and H.-P. Kriegel. The pyramid technique: Towards breaking the curse of dimensionality. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 142–153, 1998.

[6] Stefan Berchtold, Christian Böhm, and Hans-Peter Kriegel. Improving the query performance of high-dimensional index structures by bulk load operations. In *Proc. Int. Conf. on Extending Database Technology*, pages 216–230, 1998.

[7] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. The X-tree : An index structure for high-dimensional data. In *Proceedings of the Int. Conf. on Very Large Data Bases*, pages 28–39, 1996.

[8] Kevin S. Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When is "nearest neighbor" meaningful? In *Proc. Int. Conf. Database Theory*, pages 217–235, 1999.

[9] Christian Böhm, Bernhard Braunmüller, Florian Krebs, and Hans-Peter Kriegel. Epsilon grid order: An algorithm for the similarity join on massive high-dimensional data. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 2001.

[10] B. Braunmüller, M. Ester, H.-P. Kriegel, and J. Sander. Efficiently supporting multiple similarity queries for mining in metric databases. In *Proc. Int. Conf. on Data Engineering*, pages 256–267, 2000.

[11] T. Brinkhoff, H. Kriegel, and B. Seeger. Efficient processing of spatial joins using R-trees. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 1993.

[12] P. Ciaccia and M. Patella. Bulk loading the M-tree. In *9th Australasian Database Conference (ADC'98)*, pages 15–26, 1998.

[13] Christos Faloutsos and Ibrahim Kamel. Beyond uniformity and independence: Analysis of R-trees using the concept of fractal dimension. In *Proc. ACM Symp. on Principles of Database Systems*, pages 4–13, 1994.

[14] Gisli R. Hjaltason and Hanan Samet. Ranking in spatial databases. In *Advances in Spatial Databases — Fourth International Symposium (SSD)*, pages 83–95, 1995.

[15] Gisli R. Hjaltason and Hanan Samet. Incremental distance join algorithms for spatial databases. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 237–248, 1998.

[16] Yun-Wu Huang, Ning Jing, and Elke A. Rundensteiner. Spatial joins using r-trees: Breadth-first traversal with global optimizations. In *Proceedings of the Int. Conf. on Very Large Data Bases*, pages 396–405, 1997.

[17] J. Jin, N. An, and A. Sivasubramaniam. Analyzing range queries on spatial data. In *Proc. Int. Conf. on Data Engineering*, 2000.

[18] Norio Katayama and Shin'ichi Satoh. The SR-tree: An index structure for high-dimensional nearest neighbor queries. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 369–380, 1997.

[19] Flip Korn, Theodore Johnson, and H. V. Jagadish. Range selectivity estimation for continuous attributes. In *Proc. International Conference on Scientific and Statistical Database Management*, pages 244–253, 1999.

[20] Flip Korn, Bernd-Uwe Pagel, and Christos Faloutsos. Deflating the dimensionality curse using multiple fractal dimensions. In *Proc. Int. Conf. on Data Engineering*, 2000.

[21] Nick Koudas and Kenneth C. Sevcik. High dimensional similarity joins: Algorithms and performance evaluation. In *Proc. Int. Conf. on Data Engineering*, pages 466–475, 1998.

[22] Christian A. Lang and Ambuj K. Singh. Modeling high-dimensional index structures using sampling. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 2001.

[23] Apostolos Papadopoulos and Yannis Manolopoulos. Performance of nearest neighbor queries in R-trees. In *Proc. Int. Conf. Database Theory*, volume 1186 of *Lecture Notes in Computer Science*, pages 394–408, 1997.

[24] R. Ramakrishnan and J. Gehrke. *Database Management Systems (2nd Edition)*. McGraw-Hill, 2000.

[25] Bernhard Seeger, Per-Åke Larson, and Ron McFayden. Reading a set of disk pages. In *Proceedings of the Int. Conf. on Very Large Data Bases*, pages 592–603, 1993.

[26] Yannis Theodoridis and Timos K. Sellis. A model for the prediction of R-tree performance. In *Proc. ACM Symp. on Principles of Database Systems*, pages 161–171, 1996.

[27] R. Weber and S. Blott. An approximation based data structure for similarity search. Technical Report 24, ESPRIT project HERMES (no. 9141), October 1997.

[28] David A. White and Ramesh Jain. Similarity indexing with the SS-tree. In *Proc. Int. Conf. on Data Engineering*, pages 516–523, 1996.