Contrast Plots and P-Sphere Trees: Space vs. Time in Nearest Neighbor Searches

Jonathan Goldstein
Microsoft Research
jongold@microsoft.com

Raghu Ramakrishnan
Department of Computer Sciences University of Wisconsin-Madison
raghu@cs.wisc.edu

Abstract

In recent years, many researchers have focused on finding efficient solutions to the nearest neighbor (i.e. NN) problem. While there have been many efforts to find faster than linear scan processing strategies for these tasks, there has been no success at solving the problem for high dimensionality.

While previous work shows that the problem can't be solved efficiently in general, we present a technique for either in-memory or secondary storage that is guaranteed to perform well in the "good" situations previously described. Furthermore, it is the first NN strategy that allows a database administrator to easily trade space for execution time. In addition, the space/time performance for a particular dataset can be easily predicted by examining one characteristic the data. Finally, there are variants of the strategy that perform far better than the tested alternative techniques in all tested scenarios.

1 Introduction

In recent years, many researchers have focused on finding efficient solutions to the *nearest neighbor (NN)* problem, defined as follows: *Given a collection of data points and a query point in a d-dimensional metric space, find the data point that is closest to the query point.* Particular interest has focused on solving this problem in high dimensional spaces, which arise from techniques that approximate (e.g., see [32]) complex data—such as images (e.g., [18, 37, 38, 28, 38, 30, 34, 21, 4]), sequences (e.g., [3, 2]), video (e.g., [18]), and shapes (e.g., [18, 39, 34, 29])—with long "feature" vectors. Similarity queries are performed by taking a given complex object, approximating it with a high dimensional vector to obtain the query point, and determining the data point closest to it in the underlying feature space.

While there have been many efforts to find faster than linear scan processing strategies for these tasks, there has been no success at solving the problem for arbitrary high dimensional workloads. The reasons for this lack of success have, in the past several years, become more clearly understood.

Indeed, in [13], we prove that there is no faster than linear scan processing strategy to solve the problem for a wide variety of high dimensional workloads. We also establish workload based performance bounds for the problem itself. They show that a performance limiting feature of the problem is the tendency, in high dimensionality, of data and query points to all become equidistant. This effect can be summarized visually by examining the distribution of the distances between data points and a typical query point. We will call such distributions **contrast distributions** and graphs of such distance distributions **contrast plots**.

This paper introduces several NN processing techniques. These techniques are all predicated on a very important requirement: that a random sample of the query distribution is available at index build time. Exploiting this requirement, we create a class of NN processing techniques with new properties and capabilities. For instance, the techniques presented in this paper have performance that is easy to characterize in terms of contrast distributions. In addition, having sample query points allows us to integrate redundancy into the index itself to easily trade space for time; a tradeoff that is necessary to overcome problems associated with dimensionality for range queries ([23]). This redundancy, when used with a non-deterministic (but highly accurate) variant of the techniques presented here, allows unparalleled search performance on the "hard cases" identified in [13]. Our techniques can be used with any distance metric and speed up NN processing over both in memory and disk based data.

The nature of the easily characterizable behavior of our techniques has an interesting repercussion. With the additional assumption that query distribution follows data distribution, we can prove that contrast distribution spread is the primary performance limiting feature of the nearest neighbor problem. We do this by relating the behavior of the strictly worst performing variant of our techniques to the theoretical performance bounds established in [13] for the problem.

In [13], we showed that as the spread of the contrast distribution for a particular workload narrows, a sub linear NN strategy for processing that workload becomes harder

[0] **Proceedings of the 26th VLDB Conference, Cairo, Egypt, 2000.**

and harder to find, until a threshold is reached at which there is no sub linear NN strategy. The strategy described in this paper can be viewed as a constructive proof of the implication proven in [13] going the opposite direction. When combined, the two results create an "if and only if" relationship between contrast distribution spread and attainable performance. There are two important aspects of this result, the first is that it establishes contrast as not only *a* limiting factor, but *the* primary limiting factor of NN processing techniques in situations where data and query distributions are equivalent. The second is that it shows that the techniques presented in this paper perform according to the inherent difficulty of the problem (i.e. there is no unnecessary "bad" behavior).

This paper is divided into 7 sections. Section 2 describes the two simplest and worst performing variants of the algorithm. The better performing of these variants allows one to trade some user controlled level of accuracy for improved performance. Section 3 contains the highlights of our theoretical analysis of the worst performing of the algorithm variants. Section 4 contains a description of the best performing variant of our NN processing strategy. Section 5 contains a performance analysis of all variants of the algorithm presented in the previous sections while Section 6 discusses related work. This paper finishes with the conclusions in Section 7.

2 P-Sphere Trees

The query processing strategy presented in this paper involves building and searching a structure called a P-Sphere tree (probabilistic sphere tree). This structure is built using the entire dataset and assumes that a set of sample query points Q is available at index build time.

Note that in situations where query distribution follows the data distribution, we can use a random sample of the data points themselves as our sample query points. This is frequently a valid assumption for high dimensional similarity/matching problems. For instance, if we are trying to match fingerprints or retina prints for building security, queries will typically be a random sample of the people that work in the building, whose fingerprints were used to generate the database in the first place.

The structure will be built in such a manner that approximately some user specified percentage of the time, a search of the structure will yield a provably (at query run time) correct answer. More precisely, the user can specify the 95% confidence interval for the percentage of the time that a search of the structure yields a provably correct answer. The narrower the interval, the longer index construction takes. From now on, we will describe the accuracy of such trees as $u_{95\%}$ accurate since the user can control a 95% confidence interval around some desired level of accuracy u.

When a search of the structure doesn't yield a provably correct answer, the next best strategy (possibly a linear scan) will be performed. Thus, while we may sometimes need to perform a search using some alternate, worse performing technique, assuming there is acceptable contrast, on average we will get much better performance.

2.1 The Anatomy of a P-Sphere Tree

All P-Sphere trees have the same basic two level structure (see Figure 1). The top level is a single large node that contains a series of <*sphere descriptor, leaf page pointer*> pairs. Each leaf of the index contains **all** data points that lie within the sphere described in the corresponding sphere descriptor from the top level. All leaves cover the same amount of data (not the same amount of hypervolume). For instance, each sphere in Figure 1 covers LS data points.

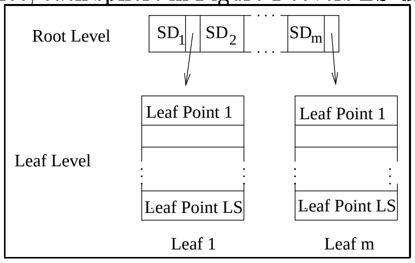

Figure 1: Anatomy of a P-Sphere Tree

Note that since each leaf contains all points that lie within a spherical subsection of the data space, spatial overlap amongst the leaves may lead to redundancy. Also, there is no guarantee that the entire data space will be covered by the sphere descriptors in the top level. As a result, there is no guarantee that all data points of the original data set will be in the generated P-Sphere tree.

2.2 Searching P-Sphere Trees

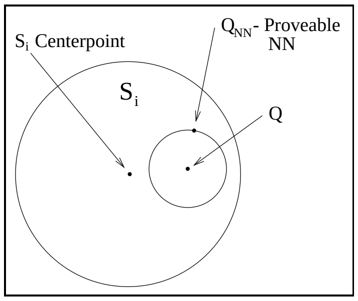

Figure 2: Provable NN

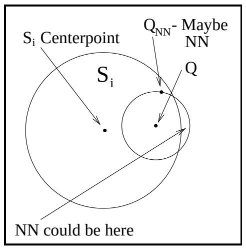

Figure 3: Unprovable NN

The search algorithm for P-sphere trees is very simple and involves finding the closest point to the query point in the sphere whose center is closest to the query point. More precisely:

1. Search the root node for the sphere/pointer pair $< S_i, LP_i >$ whose centerpoint is closest to the query point Q.
2. Return the point Q_{NN} in leaf L (pointed to by LP_i) that is closest to Q.

In order to determine with certainty that the point generated from the algorithm above is the correct one, we determine if S_i contains the sphere C with centerpoint Q and radius $Distance(Q, Q_{NN})$. If S_i contains C, then we know that there is no closer point in the dataset or it would have been in L (See Figures 2 and 3). More precisely, the final search algorithm is shown in Figure 4

2.3 Creating a P-Sphere tree

This section describes the manner in which a P-sphere tree is generated from a dataset such that some specified $u_{95\%}$

```
1. Search the root node for the sphere/pointer pair $< S_i, LP_i >$ whose centerpoint is closest to the query point $Q$.
2. Find the point $Q_{NN}$ in leaf $L$ (pointed to by $LP_i$) which is closest to $Q$.
3. If $S_i$ contains the sphere with centerpoint $Q$ and radius $Dist(Q, Q_{NN})$, return $Q_{NN}$.
4. Otherwise return the result using the next best strategy.
```

Figure 4: Deterministic P-Sphere Tree Search Algorithm

accuracy during search is met. Note that the width of the CI is controlled by one of the algorithm inputs, $| Q |$, and that the relationship between $| Q |$ and CI is presented in Section 2.3.1.

There are three parameters to adjust in the tree: the fanout of the root, the centerpoints in the sphere descriptors, and the leaf size. We will fix the fanout of the root to be some constant greater than one. While we assume in this section that the fanout is constant, it is, in fact, a parameter that must be determined at index build time. Chapters 3 and 5 contain a further discussion on the choice of fanout. The centerpoints will be generated by taking a random sample of the data set. This is useful for theoretical analysis as it implies that the sphere centerpoint distribution follows the data distribution. Note that the only parameter to determine is the leaf size. This parameter is the most difficult to determine. The problem of determining leaf size can be summarized as follows:

Given a data set, sample query points, user specified $u_{95\%}$ accuracy, and fanout and centerpoints of the final P-Sphere Tree, determine the leaf size of the P-Sphere tree such that the user specified $u_{95\%}$ accuracy is met.

The strategy we employ to determine leaf size involves "reverse engineering" our tree to work correctly for query distributions like the one we set aside from the original data set. This is done by observing that for every query, there is an associated leaf size that is just large enough, given the centerpoints of the P-sphere tree, to return the provably correct answer when searching the resulting tree. In other words, there is an associated distribution of leaf sizes for a given workload such that the area under the curve between 0 and a given leaf size LS is the expected accuracy of the P-Tree with leaf size LS. We use our queries to sample this leaf size distribution. Based on this sample and our accuracy goal, we decide on an appropriate leaf size.

More precisely, the algorithm in Figure 5 empirically samples the implicit leaf size distribution using the query points as the basis of those samples. Note that the number of queries determines the number of samples taken of the distribution of leaf sizes and is the basis for the 95% confidence interval for accuracy established in Section 2.3.1.

The first loop simply calculates the nearest neighbor of every query in a manner that insures we make only 1 pass over the data. This is useful in reducing I/O if your query points and their nearest neighbors fit in memory, which is typically the case.

The second loop determines, for each query, the leaf that would have been searched had that query been run on the P-Sphere tree being constructed. In addition, the algorithm determines the minimum radius of the found leaf needed to ensure that the leaf, when searched, contains the provably nearest neighbor for that query.

The last loop simply counts, for each query, the number

```
1. Initialize all $Q_{NN}$
2. For each data point $D$
    (a) For each query point $Q$
        i. If $Q_{NN}$ is further from $Q$ then $D$
            A. $Q_{NN} = D$
3. For each query $Q$
    (a) $Q_{Center}$ = the center point of the leaf closest to $Q$
    (b) $Q_{Radius} = Dist(Q_{Center}, Q) + Dist(Q, Q_{NN})$
4. For each data point $D$
    (a) For each query $Q$
        i. If $Q$ falls inside the sphere described by $Q_{Center}$ and $Q_{Radius}$
            A. Increment the $Q_{Leafsize}$
```

Figure 5: Determining Leaf Size Distribution

of data points that lie within the sphere whose centerpoint is Q_{Center} and radius Q_{Radius}. Once again, the loop is performed such that one pass over the data is all the I/O typically needed to perform the computation.

Upon termination of the algorithm, there is associated with each query a leaf size ($q_{Leafsize}$) which when divided by the number of data points, is a sample of the leaf size distribution discussed above. To meet the accuracy goal, we determine the correct leaf size of our P-Sphere tree by sorting the calculated leaf sizes and picking the one whose percentile matches the user defined level of correctness. For instance, if there were 100 queries, and the user wanted 90% accuracy, we would sort the 100 leaf sizes and pick the 90*th* leaf size in the list.

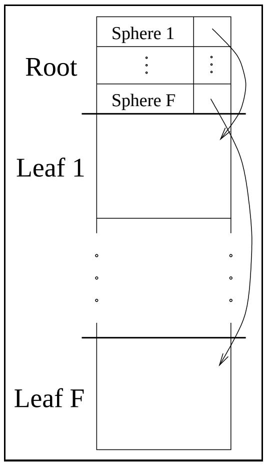

Figure 6: Layout of a P-Sphere Tree in a File

We now have all the parameters needed to build our P-Sphere Tree. The first level of the tree is trivial to build since it simply consists of <*centerpoint, pointer*> pairs. The pointers to appropriate size chunks of disk space can be set up ahead of time since the sizes of the chunks are known in advance (See Figure 6).

Setting up the second level is harder. Since each leaf must contain the LS closest data points to that leaf's centerpoint, a scan of the data must be made in order to calculate that list. In addition, we want to avoid as many disk I/Os as possible since the entire tree is not likely to fit in memory. As a result, we must calculate these leaves a group at a time, such that for each group, one pass is made over the data set. During these passes, a priority queue for each leaf with LS entries is used to keep track of the LS closest points encountered during the pass. The number of leaves calculated during each pass is chosen to minimize construction time. More precisely, Figure 7 shows how the tree is constructed from its parameters.

The full algorithm (making use of the previously listed algorithms) for computing the P-Sphere tree is given in

```
1. Write out the first level
2. For each group of leaves
    (a) Populate the priority queues associated with the current group's leaves with the first $LeafSize$ data points
    (b) For each data point $D$ beyond the first $LeafSize$ data points
        i. For each leaf $L$ within the current group
            A. If $D$ is closer to the centerpoint of $L$ than the furthest point in the associated priority queue $Q$
                • Remove the furthest point from $Q$
                • Insert $D$ into $Q$
    (c) Write out the priority queues to the appropriate places on disk
```

Figure 7: Building the Actual P-Sphere Tree

Figure 8.

```
1. Set aside some random number of data points to be query points, removing them from the data point list.
2. Randomly sample the data points to determine leaf centerpoints
3. Determine, for each query, $Q_{Leafsize}$ (see Figure 5)
4. Make a sorted list $LeafSizeList$ out of all $Q_{Leafsize}$
5. $LeafSize = LeafSizeList[UserAccuracy * ListSize]$
6. Write out the tree (see Figure 7)
```

Figure 8: Full Algorithm for P-Sphere Tree Construction

2.3.1 Determining the Number of Queries

One important parameter to the P-Sphere tree algorithm is the number of query points $\mid Q \mid$ used to build the tree. This number determines the confidence interval for the accuracy of the resulting P-Sphere tree. This confidence interval is established by mapping the problem into a classic problem in probability theory.

This mapping is the result of thinking of every query as a coin flip, where the coin comes up heads if the P-Sphere tree search algorithm results in a provably correct answer, and comes up tails if it doesn't produce a provably correct answer. We can then think of the confidence interval for the P-Sphere tree accuracy as the confidence interval for the bias of a coin which was flipped as many times as we have queries, and resulted in the user specified accuracy percent heads.

Another way to state the problem is that we want a confidence interval on the p-value of a Bernoulli process that was independently sampled as many times as we have queries. More precisely, if $\widehat{p}$ is the user specified accuracy goal (and also, as a result, the estimator for the likelihood of success for the coin flip), the 95% confidence interval for the actual accuracy is approximately

$$\widehat{p} \pm 2\sqrt{\frac{\widehat{p}(1-\widehat{p})}{\mid Q \mid}} \qquad [14] \qquad (1)$$

Note that this analysis will apply to all variants of P-Sphere trees.

3 Theoretical analysis of Deterministic P-Sphere Trees

This section contains highlights from the theoretical analysis of deterministic P-Sphere trees found in [22]. For ease of reading, all references in this section to P-Sphere trees are, more specifically, deterministic P-Sphere trees. In this analysis, the overall behavior of P-Sphere trees is described by examining the effect of datasets on the parameters of P-Sphere trees that have a known level of accuracy (not just a CI). Among the qualities examined are run time and space requirements as well as various upper and lower bounds. Note that these results apply equally to both CPU and disk costs.

3.1 Contrast Plots

The contrast plot of a query with respect to a dataset is a histogram of distance between points in the dataset and that query point. A plot of the integral of the function in a contrast plot is the cumulative contrast plot. Normalized contrast plots are contrast plots where distances are normalized so that the distance to the nearest neighbor is 1. For instance, the normalized contrast plot displayed in Figure 9 indicates that half the points in the dataset are less than twice as far as the nearest neighbor.

Note that contrast plots are constructed about a *particular* query point and are therefore unique to each query. As a result, every workload has an associated distribution of contrast plots. This distribution is implicitly sampled in P-Sphere tree construction by using multiple queries to determine leaf size. It is useful to note, however, that [13] shows that as the workload becomes harder and harder to index, the distribution of contrast plots converge to the same constant distribution.

3.2 Analysis of the P-Sphere Tree with Respect to Contrast Plots

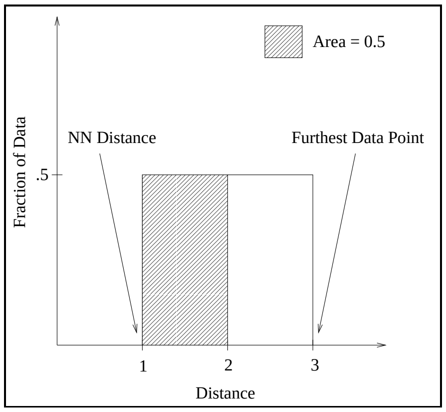

Figure 9: Sample Contrast Plot

Interestingly, if we make the simplifying assumption that all queries have a contrast plot identical to some "typical" contrast plot, the contrast plot is enough to precisely determine the median space and time behavior of P-Sphere trees applied to that workload. While such an analysis is not strictly accurate (because of the simplifying assumption), it lends much insight into the conditions under which this algorithm does and does not perform well. In addition, the upper and lower bounds described in this paper are completely accurate (the simplifying assumption is not needed).

3.2.1 General Space and Time Analysis

Since the fanout of the P-Sphere Tree is fixed at index construction time, the only parameter that determines space and time is the leaf size. Determining the leaf size is the purpose of Lemma 1. In this lemma, we assume that we are given the top level of a P-sphere tree, and a level of accuracy that we meet exactly (not a CI), and wish to determine leaf size.

In this lemma, we construct a cumulative distribution $F(x)$, which is the percent of data which is at most x distance away from a particular query point. Note that this function corresponds to a cumulative contrast plot. As mentioned earlier in this chapter, we will make the simplifying assumption that all random query points Q produce identical contrast plots, and therefore, $F(x)$ is independent of Q. While this is not generally true, the resulting analysis gives us insight into the relationship between contrast plots and leaf sizes. The fact that different queries have different contrast plots is a second order averaging effect that happens "on top of" the one described in this analysis. Furthermore, experimental results in [13] found that, in practice, contrast plots do not vary widely amongst query points.

Lemma 1 *Given:*

- *A dataset D with n datapoints.*
- *A P-Sphere tree T over D with fanout m (we assume in this proof that each of the m centerpoints were sampled with replacement) which returns the provably correct answer (u*100)% of the time and whose leaf size is S percent of the entire dataset.*
- *A random query point Q.*
- *A set of cumulative distributions $F_Q(x)$, such that for a particularly query point Q, $F_Q(x)$ equals the percent of data which is at most x distance away from Q. Because of an assumption we make later, $F_Q(x)$ is identical for any assignment of Q. We will therefore refer only to $F(x)$, which is the distribution for any assignment of Q.*
- *A random variable B_C, or nearest bucket centerpoint, which is, by definition of the P-Sphere tree algorithms, the closest of m points sampled from the original dataset.*
- *NN, the nearest neighbor of Q, which is $DMIN$ distance away from Q.*

and assuming that Q and $F(x)$ are independent, the leaf size S of T as a fraction of the total dataset is

$$F(F^{-1}(1-[1-u]^{(1/m)})+DMIN) \tag{2}$$

With the above lemma, we can now produce the main result of this theoretical analysis, which is the following theorem about the behavior of P-Sphere Trees:

Theorem 1 *Given all assumptions and symbol definitions of Lemma 1, the space/time (tree search time) requirements (as a fraction of linear scan space/time) for the P-Sphere tree (disregarding space/time overhead incurred by storing/reading pointers in the root) are:*

$$\mathit{Space} = \frac{m}{n} + S*m,\ \mathit{Time}^1 = \frac{m}{n} + S, \tag{3}$$

$$S = F(F^{-1}(1-[1-u]^{(1/m)})+DMIN) \tag{4}$$

Furthermore, the best possible space/time (tree search) behavior of P-Sphere trees over all datasets (assuming no duplicates) is:

$$\mathit{Space} = \frac{m}{n} + S*m,\ \mathit{Time} = \frac{m}{n} + S,\ S = 1-[1-u]^{(1/m)} \tag{5}$$

Also, the worst possible space/time behavior of P-Sphere trees over all datasets is:

$$\mathit{Space} = \frac{m}{n} + m,\ \mathit{Time} = \frac{m}{n} + 1,\ S = 1 \tag{6}$$

3.2.2 Leaf Size and Contrast Plots

It is clear from the above theorem that leaf size is the determining factor of overall P-Sphere Tree behavior. It is interesting to note that the formula for leaf size,

$$S = F(F^{-1}(1-[1-u]^{(1/m)})+DMIN) \tag{7}$$

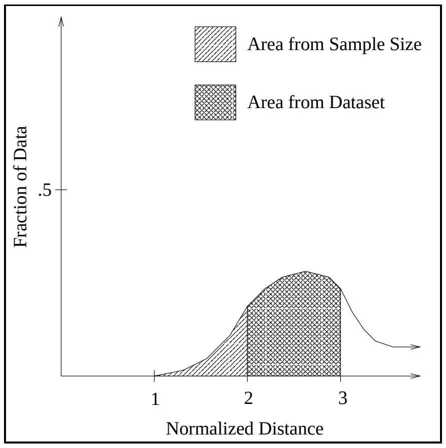

Figure 10: The affect of DMIN on Leaf Size

can be intuitively understood in terms of normalized contrast plots. One way of interpreting the above expression is that the leaf size is $1-[1-u]^{(1/m)}+$ the area under the curve of the normalized contrast plot from $1-[1-u]^{(1/m)}$ into the plot and spanning the subsequent interval of size 1(See Figure 10).

For instance, an example contrast plot that would result in the minimum possible leaf size, $1-[1-u]^{(1/m)}$ is shown in Figure 11.

In this example, the data is divided into clusters. There is the cluster which contains the query point that corresponds to the first bump in the plot. The rest of the clusters are contained in the second larger bump. There are two interesting properties of these bumps:

[1]Note that this equation is tree search time only and does not include the time for using an alternative strategy when necessary.

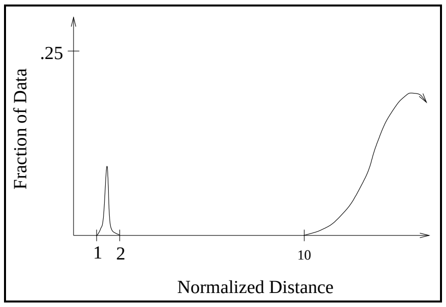

Figure 11: Contrast Plot of Clustered Data

- The first bump has area under the curve $1 - [1-u]^{(1/m)}$.
- The distance between bumps is larger than 1.

As a result of the above two properties, the addition of $DMIN$ to $F^{-1}(1-[1-u]^{(1/m)})$ has no effect on the value of F, which is unchanging in the interval between the 2 bumps.

Overall, it is easy to see that the determining factor of leaf size is the area under the curve of the normalized contrast plot of a certain interval of size 1. Thus, the leaf size is determined by the spread of the normalized contrast plot in that region.

3.3 ND P-Sphere Trees

The P-Sphere trees discussed above find the provably correct answer with $u_{95\%}$ accuracy. While this approach is useful if one always needs the correct answer, there are many cases where the exact answer is not always needed. In these cases, it is enough to know that with $u_{95\%}$ accuracy, the algorithm returns the correct answer, but correct and incorrect answers can not be distinguished by the search algorithm.

For instance, if we are solving an approximate matching problem such as identifying matching fingerprints, we can compare the actual fingerprints as part of a postprocessing step to determine correctness outside the algorithm. Furthermore, if we're performing some kind of similarity heuristic (document, image, sound), being correct most of the time is typically sufficient.

We will refer to P-Sphere trees which return the unprovably correct answer close to some specified percentage of the time as ND (non-deterministic) P-Sphere Trees. The method of construction is straightforward. The only difference between the construction of deterministic P-Sphere trees and ND P-Sphere Trees is that, for each query, we determine the leaf size needed to return the correct answer, not the provably correct answer. Thus, in terms of the algorithm in Figure 5, Q_{Radius} should be just large enough to include Q_{NN}, but no larger. Thus, $Q_{Radius} = Dist(Q_{Center}, Q_{NN})$. The rest of the algorithm remains unchanged.

The only change to the search algorithm is that we remove the verification step that determines if the answer is provably correct and the subsequent alternate strategy.

Observe that every deterministic P-Sphere tree is also an ND P-Sphere tree with differing accuracy and vice-versa. This is obvious when one considers that deterministic P-Sphere trees frequently return correct answers that can't be proven correct. In addition, ND P-Sphere trees will sometimes return the provably correct answer, but not very often. In fact, the only difference between the two types of P-Sphere trees is the size of the leaves. ND P-Sphere trees of some given accuracy have smaller leaves than a deterministic P-Sphere tree of the same accuracy. Thus the space and time performance of ND P-Sphere trees is strictly better than P-Sphere trees given the same level of accuracy over the same data.

Note that since this strategy is a strict performance improvement over deterministic P-Sphere trees, all the theoretical statements we made about deterministic P-Sphere trees are upper bounds for ND P-Sphere trees.

4 Pk-Sphere Trees

While P-Sphere trees are simple, and as a result, lend themselves to analysis, there are variants of the basic algorithm that are more complex but are strictly and significantly more efficient. The particular variant of deterministic and non-deterministic P-Sphere trees discussed in this section involves maintaining the $u_{95\%}$ accuracy discussed in the previous section using a slightly different search algorithm.

4.1 ND Pk-Sphere Trees

In the original ND search algorithm, we simply returned the closest point in the closest leaf. In the ND Pk-Sphere tree search algorithm, we instead return the closest point in the k closest leaves where k is a given constant. More precisely, the search algorithm becomes:

1. Search the root node for the k sphere/pointer pairs $< S_i, LP_i >$ whose centerpoints are closest to the query point Q.
2. Return the point Q_{NN}, the point closest to Q amongst the points in the leaves pointed to by the entries found in the previous step.

While the change in the search algorithm is simple, the resulting changes in the tree construction are more complex. In particular, the algorithm shown in Figure 5, which determines, per query, the leafsize needed to return the correct answer for that query is significantly more complicated. The new version of this algorithm will now determine, for each query, the size of the leaf, amongst the k closest, that can best accommodate the nearest neighbor of the current query. More precisely:

```
1. Initialize all $Q_{NN}$
2. For each query point $Q$
    (a) $Q_{Leafsize} = n$
    (b) For each of the $k$ closest leaves $L$
        i. $L_{Radius} = Dist(L_{Center}, Q_{NN})$
        ii. If (leafsize of $L$ assuming radius $L_{Radius}$) $< Q_{LeafSize}$
            • $Q_{LeafSize}$ = Size of $L$ assuming radius $L_{Radius}$
```

Of course, there is one aspect of this algorithm that is more complex than it appears. Calculating the size of L assuming radius L_{Radius} is a very compute intensive task.

We will therefore create, for each leaf, a histogram of the dataset with respect to distance from the leaf. These histograms are used to estimate leafsizes given radii. A more precise description of the search algorithm can be found in Figure 12.

```
1. Initialize all $Q_{NN}$
2. For each leaf $L$
   (a) Create a histogram $L_{Hist}$ of distance to $L_{Center}$ of the dataset.
3. For each query point $Q$
   (a) $Q_{Leafsize} = n$
   (b) For each of the $k$ closest leaves $L$
       i. $L_{Radius} = Dist(L_{Center}, Q_{NN})$
       ii. If (estimated (using $L_{Hist}$) leafsize of $L$ assuming radius $L_{Radius}$) $< Q_{LeafSize}$
           • $Q_{LeafSize}$ = estimated size of $L$ assuming radius $L_{Radius}$
```

Figure 12: Determining Leafsize Distribution for ND Pk-Sphere Trees

5 Performance Analysis

This section contains a series of experiments that test the behavior of P-Sphere trees and the variants described in this paper. These experiments are used to obtain an understanding of the following issues:

- The effect of fanout for all P-Sphere tree variants on space/time performance.
- The effect of k for PK-Sphere tree variants on space/time performance.
- The performance of the fastest P-Sphere tree variants on low contrast data.
- How the variants of P-Sphere trees compare to existing techniques on real data.

To address the first three issues, we used synthetic datasets. For these datasets, we intentionally chose the most difficult datasets to index. All datasets were created using identical and independently distributed dimensions, which is a very difficult case to handle ([13]). The data set size was 100,000 tuples. Note that larger datasets perform better than smaller ones ([13]), so the really difficult datasets to handle are the smallest ones. In all experiments, various parameters of the various types of trees were tested, and dimensionality of the dataset was varied. Space and search time numbers were collected for each combination of variables.

To address the last issue, we measured the performance of P-Sphere trees as compared to the performance of the SR-Tree([25]) for two real datasets. One of these datasets was also used to evaluate density based indexing in [24]. We therefore include their results in this paper. In these experiments, we were able to beat both alternative strategies' query times by approximately a factor of 20.

For experiments using synthetic data, in memory query times were measured (although the number of seeks if these had been I/O times is easily predicted as the number of leaf pages searched+1), while I/O times were measured for the real datasets. This allowed us to examine the affect of seeks (frequently the dominating component of I/O performance) independently from the affect of contrast on leaf sizes.

5.1 Varying the fanout of P-Sphere Trees

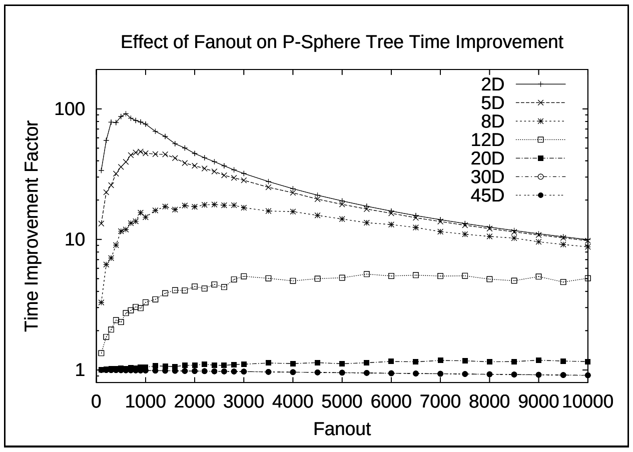

Figure 13: Effect of fanout on time for P-Sphere Trees

The experiments in this section were designed to test the effect of varying fanout for deterministic and nondeterministic P-Sphere Trees. In these experiments, an accuracy goal of 95% was used for 1000 queries. Note that performance in this section is in terms of in-memory performance. The only difference between in-memory and disk performance is the addition of a seek plus latency for each leaf being searched (assuming the root stays in memory). Note that a linear scan of the data on disk also results in one seek plus latency.

While the experiments on deterministic P-Sphere trees indicate that using deterministic P-Sphere trees for medium to high dimensionality is impractical, they are presented to highlight the effectiveness of the nondeterministic algorithms as well as the value of searching multiple leaves. The importance of P-Sphere trees lies in their theoretical value. They provide upper bounds (for all P-Sphere tree variants) that follow the inherent difficulty of the indexing problem.

Figures 13 and 14 show the effect of varying fanout on nondeterministic P-Sphere trees on time and space respec-

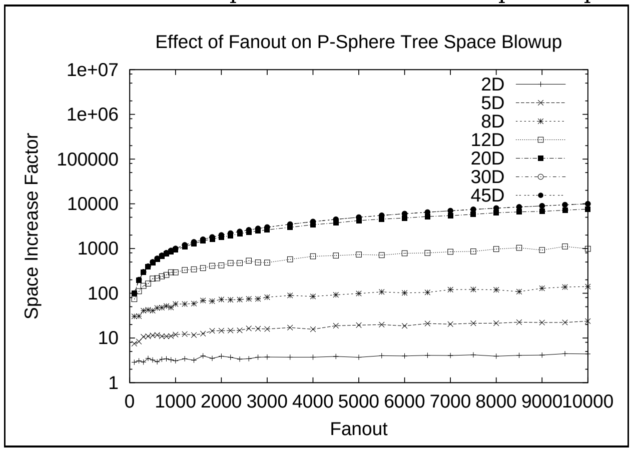

Figure 14: Effect of fanout on space for P-Sphere Trees

tively. There are several interesting features.

In Figure 13, which shows time improvement relative to linear scan, increasing leafsize helps for a while, but begins to have a negative impact after a certain point. This is especially true for lower dimensionality. At first, this seems odd since we know that increasing the fanout will reduce the leafsize, and hence the subsequent scan of the leaf. But we must remember that, especially for low dimensionality, the leaf sizes are very small, and begin to be dominated by the time to search the root, which grows linearly with leaf size. the highest dimensions are less affected since the leaves are very large in all cases, and the root only grows in these experiments to one tenth the size of the dataset.

Figure 14, which shows space blowup relative to the original dataset, is very easy to interpret. Basically, increasing fanout always increases overall size, although very slowly in low dimensionality and very quickly in high dimensionality.

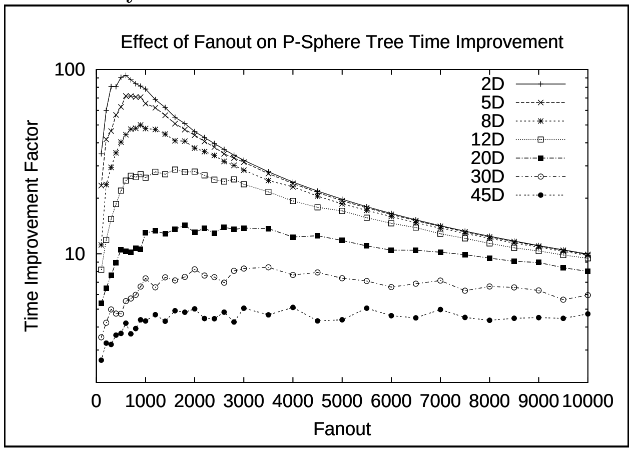

Figure 15: Effect of fanout on time for ND P-Sphere Trees

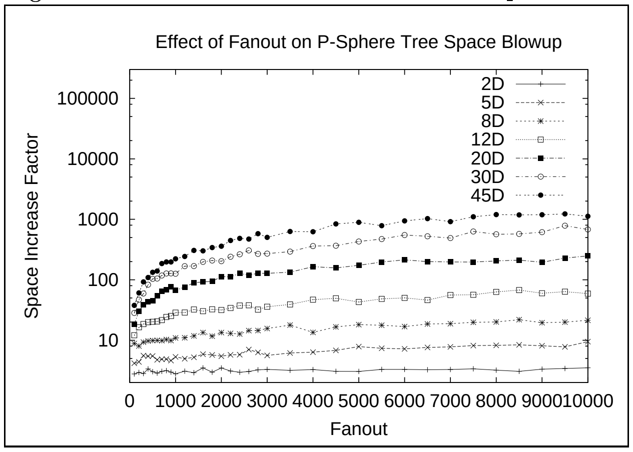

Figure 16: Effect of fanout on space for ND P-Sphere Trees

Figures 15 and 16 show corresponding graphs to the previous two for ND P-Sphere trees. Note that the overall behavior is identical, except that both space and time are considerably improved for higher dimensions. In particular, examine the time improvement of the 30 dimensional case in Figure 15. In this case, we are actually still achieving a factor of 8 speedup over linear scan! Even for the 45 dimensional case, we still achieved a factor of 3 speedup over linear scan. This is very impressive considering that other techniques fail to beat linear scan at around 10 dimensions.

For instance, [40] provides us with information on the performance of both the SS tree and the R* tree in finding the 20 nearest neighbors. Conservatively assuming that linear scans cost 15% of a random examination of the data pages, linear scan outperforms both the SS tree and the R* tree at 10 dimensions in all cases. In addition, in [25], linear scan vastly outperforms the SR tree for all synthetic datasets of dimensionality greater than 10. Lastly, in [15], performance numbers are presented for NN queries where bounds are imposed on the radius used to find the NN. While the performance in high dimensionality looks good in some cases, in trying to duplicate their results we found that the radius was such that few, if any, queries returned an answer.

While the degree to which we beat other techniques in query time is impressive, there is a down side. Unfortunately, time improvement isn't the only factor. Examine the blowup in space for the same 30 dimensional case (Figure 16). There is approximately a 100 to 200 times increase in storage requirements to achieve this 8 fold increase in performance. The 45 dimensional case is even worse, with a space blowup of nearly 800. Fortunately, the best performing variant of P-Sphere Trees, ND Pk Sphere Trees, will address this issue.

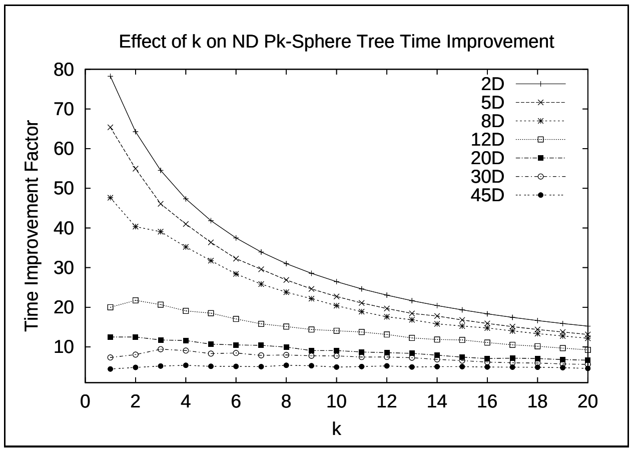

Figure 17: Effect of k on time for ND Pk-Sphere Trees with Fanout 1000

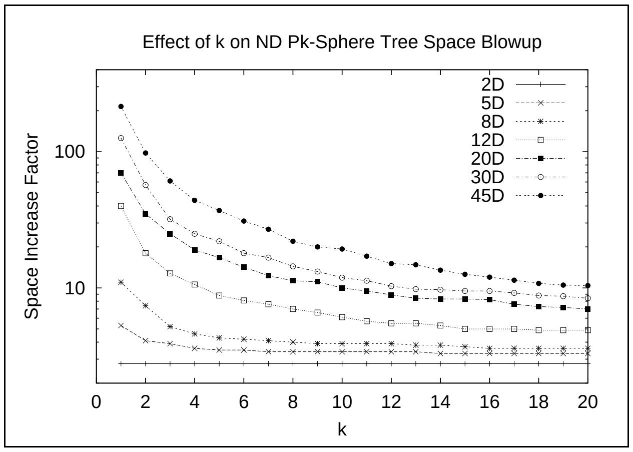

Figure 18: Effect of k on space for ND Pk-Sphere Trees with Fanout 1000

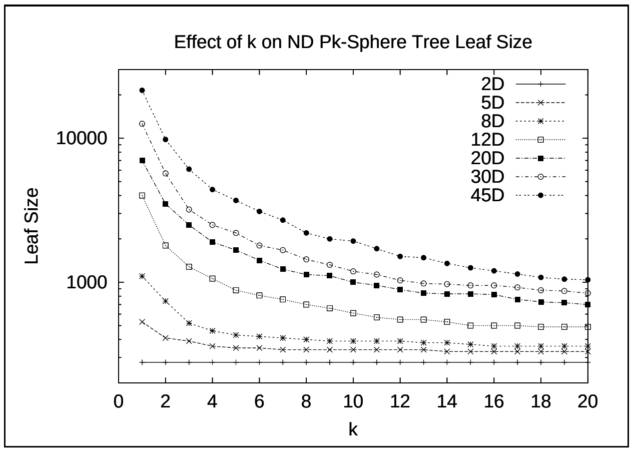

Figure 19: Effect of k on Leaf Size for ND Pk-Sphere Trees with Fanout 1000

5.2 The Performance of Pk-Sphere Trees

This section examines the performance of ND Pk-Sphere trees, the best performing NN technique described in this paper. The experiments involved varying fanout and k. Since the overall effect of fanout on these trees was identical to the P-Sphere trees described in the previous section, we will simply show the effect of varying k for two different settings of fanout.

The results for the first setting of fanout, 1000, are shown in Figures 17 and 18, which show the impact of k on time and space respectively. Note that for low dimensionality, k has little impact on space while having a detrimental impact on time. As dimensionality increases, however, the detrimental impact that k has on time becomes much diminished. In addition, for higher dimensionality, k has an extremely beneficial effect on space.

The reasons behind such behavior are more obvious when one considers Figure 19, the effect of k on leaf size. While in low dimensionality, increasing k has little effect on leaf size, in high dimensionality, increasing k dramatically reduces leaf size. As a result, in low dimensionality, increasing k has little impact on space, but requires us to search more leaves, resulting in poor search times. In higher dimensionality, however, increasing k dramatically reduces leaf size. As a result, searching additional leaves incurs little penalty since the leaves become smaller, but overall space becomes greatly reduced.

The results for a fanout of 3000, shown in Figures 20, 21, and 22 are very similar to those for a fanout of 1000, except that the larger fanout resulted in smaller leaf sizes. For low dimensionality, the savings in space and time due to smaller leaves was dominated by the increase in the number of leaves and the increase in the size of the root respectively. As a result, neither overall space nor total query time improved. For higher dimensions and high values of k, however, the increase in fanout resulted in a noticeable improvement in time with a slight penalty for space, resulting in a better overall space/time tradeoff.

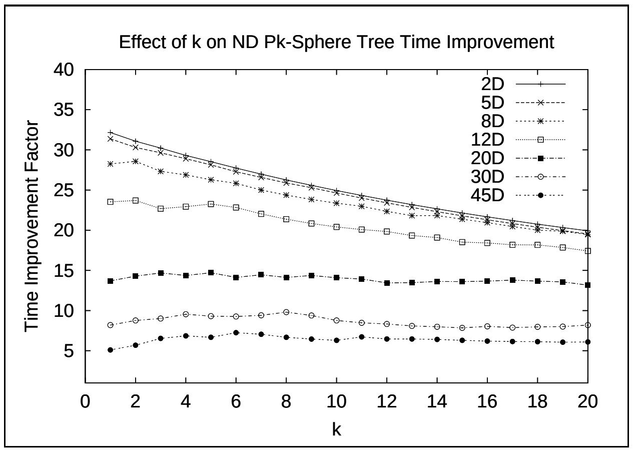

Figure 20: Effect of k on time for ND Pk-Sphere Trees with Fanout 3000

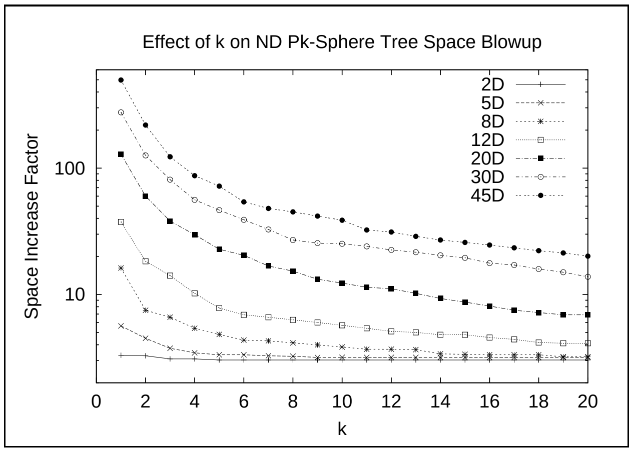

Figure 21: Effect of k on space for ND Pk-Sphere Trees with Fanout 3000

5.3 P-Sphere Tree Performance on Real Data

This section describes the performance of the three variants of P-Sphere trees on two real datasets. In each case, the space and time performance were measured against data set size and linear scan respectively. Note that time performance in this section refers to the I/O time (including seeks) for running the queries cold. In all experiments, an accuracy level of 95% was used with 1000 queries. Note that the number of queries used was enough to obtain very close to 95% accuracy ($\pm$ 1.5%). The obtained results are also compared to the SR-Tree [25] applied to the same dataset.

To determine the best setting of root fanout, all settings of fanout between 1000 and 3000 were tested at intervals of 100. The results of varying k are discussed individually for each dataset.

The first dataset was the astronomy dataset used with the probabilistic nearest neighbor processing technique described in [24]. As a result, we show their performance results for an accuracy level of 95%. This dataset is a 29 dimensional dataset with over half a million rows. The results are shown in Figure 23. Note that the speedups over linear scan are considerable. In addition, ND P-Sphere trees beat both alternative strategies by a factor of 20 to

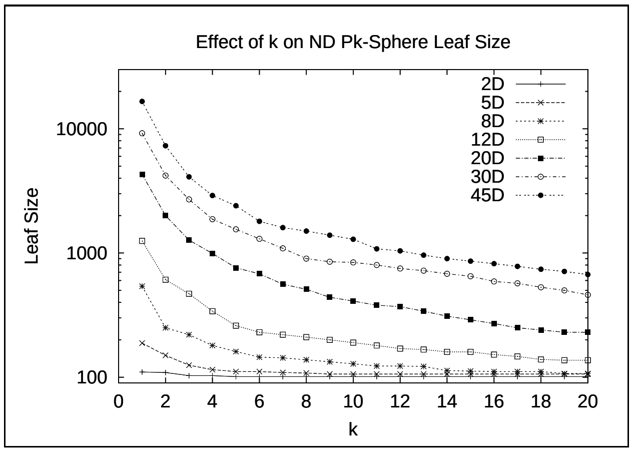

Figure 22: Effect of k on Leaf Size for ND Pk-Sphere Trees with Fanout 3000

30 for query time.

Technique	Blowup	Speedup over LS
P-Sphere	40	52
ND P-Sphere k=1	8	120
Density Based Indexing	1	6.6
SR-Tree	1.4	6.4

Figure 23: Performance Results for Astronomy Data

Technique	Blowup	Speedup over LS
P-Sphere	98	7.5
ND P-Sphere k=1	13	27
ND P-Sphere k=2	9	19
ND P-Sphere k=3	8	15
ND P-Sphere k=4	7	12
SR-Tree	1.5	.84

Figure 24: Performance Results for Color Histogram Data

The second dataset was a dataset derived from color histograms of images. This dataset is a publicly available dataset available from the University of California at Irvine Machine Learning Repository. This dataset has 32 dimensions and nearly 70,000 tuples. As we can see from the results in Figure 24, this is a difficult dataset to index. This is obvious from the space blowup versus query speedup of the deterministic P-Sphere tree. Despite this, we still achieved nearly a factor of 30 speedup over linear scan, and more than a factor of 30 over the SR-Tree.

Observe that the SR-Tree was outperformed by linear scan. It is interesting to note that the SR-Tree managed to prune the search space to less than 10% of the dataset, but that the cost of performing one seek per node visited so dominated the cost expression, that the pruning was irrelevant. As the gap between sequential versus random disk throughput widens (as it has dramatically done for the last 10 years), the tendency for the number or seeks to determine disk throughput will strengthen.

The color histogram dataset clearly shows the benefits of using a probabilistic search algorithm. Both space and time were improved considerably over the deterministic case. In addition, increasing k leads to an overall reduction in both space blowup and speedup. The decrease in speedup was caused by the domination of the seek associated with accessing each leaf. In this particular case, the actual amount of data accessed as k increased hardly changed. Thus, for a larger sample of this dataset, where the cost of scanning the leaf dominates the seek, increasing k would have little effect on speedup for small k. For this particular sample of this dataset, one should use the lowest setting of k for which the space blowup is acceptable.

While it is difficult to determine exactly how other techniques such as X-Trees([11]) and R*-Trees([7]) would have performed on these datasets, it is worth noting that the color histogram dataset was an example of a low contrast dataset, and was roughly equivalent in contrast to a 15 dimensional 1 million tuple iid uniform dataset. Alternative techniques are known to perform poorly on such datasets. In addition, the performance studies in [25] and [11] present evidence that R*-Trees perform significantly worse than both SR-Trees and X-Trees, and that SR-Trees and X-Trees are within a factor of 2 of one another. It is therefore likely that we would have beaten both X-Trees and R*-Trees by at least an order of magnitude.

6 Related Work

There are many processing strategies from the database community for tackling the high dimensional nearest neighbor problem. These include [40, 11, 25, 15, 10], all of which were designed with low contrast situations (high dimensionality) in mind. By making use of the assumption that sample query points are available at index construction time, the algorithms presented in this paper far outperform these strategies (for low contrast cases), all of which are beaten by linear scan around 10 dimensions ([13], [36]). Also, none of them provide asymptotic optimality guarantees for the important case where query distribution follows data distribution.

[16] is of interest in that they provide informal arguments that unnormalized contrast plots are important in evaluating the difficulty of NN processing on a given dataset. They also introduce a NN processing strategy for in-memory NN query processing. They significantly reduce the number of distance computations in medium to low contrast situations, but point out that the technique is not really suitable for disk based searching. This seems likely given that the behavior of the stucture would be very seek time dominated.

A processing strategy called the VA-File ([36]) achieves improvements over linear scan, even on low contrast datasets, by using a lossy compression technique on the entire dataset. Note that this strategy is complementary to ours and could be used on each leaf in a P-Sphere tree to further improve performance.

There has recently been interest in hashing based techniques for determining approximate nearest neighbors ([33] and [1]). Unfortunately, all available experimental results are for relatively high contrast situations, providing no insight into the cases that are truly difficult to index. In addition, their notion of approximation requires the user specified error criteria to decrease with contrast to maintain a constant level of discrimination between data points. For instance, if we used their hashing technique to determine one of the five closest data points to a random query point, the necessary level of user specified error would decrease with contrast. Thus there is a dependence between contrast and user specified error that is not made explicit in their theoretical statements about overall behavior.

The nearest neighbor processing technique presented in [31] is noteworthy in that they use a notion of approximation (returning the correct answer some percentage of the time) similar to the one presented in this paper. Unfortunately, they present no precise way of controlling or predicting the level of accuracy for a particular dataset. A precise comparison between the effectiveness of their techniques and ours is difficult since they do not present any information about the behavior of their techniques on identical and independently distributed data.

[24] also uses a notion of approximation similar to the one presented in this paper. Their approach consists of modeling the data as a mixture of Gaussians. This model guides the construction of their data subdivision (distribution based rather than space based), and ultimately guides their search algorithm. In many ways, the difference between our approaches can be summarized as sampling (our approach) versus modeling. There are, however, further important differences between their work and ours. For instance, we leverage redundancy to improve performance. In addition, dataset distribution does not affect the confidence interval for our accuracy. We also relate our results to theoretical bounds established on the overall problem [13], and not just to a specific kind of workload ([24] evaluate their approach for clustered datasets). Finally, for the same dataset with the same level of accuracy, ND P-Sphere trees were nearly 20 times faster with a space blowup of 8.

The computational geometry community has also been interested in the nearest neighbor problem and has discovered various query processing strategies [5, 6, 9, 12]. Unfortunately, none of these techniques were designed with low contrast (high dimensionality) in mind. As a result, while some of them perform quite well and are well understood in 2 or 3 dimensions, they perform very poorly in higher dimensionality. In cases where upper and lower bounds are available, there are constants that scale exponentially with dimensionality. One noteworthy technique from this community was published in [17]. Like P-Sphere trees, they use redundancy and a random sample of query points at index build time. Their structure is, however, quite different and would require many more seeks than P-Sphere trees while searching. They also show that for *fixed* dimensionality, their technique scales logarithmically with the number of data points. But like other techniques from the computational geometry community, there are constants in the bounds that scale exponentially with dimensionality. It is worth mentioning that their search time bounds become asymptotically linear as the spread of the contrast distribution becomes negligible. However, since there are no published performance results for this technique, it is impossible to compare their strategy directly to ours.

There has been much work recently on trying to capture the properties of a high dimensional dataset that makes various forms of query processing, including nearest neighbor, difficult [20, 8, 19, 27, 13]. Of these, only principal components analysis, which led to the TV-Tree ([27]), has resulted in any new query processing techniques. It is easy to find situations, however, where principal components analysis fails to recognize properly whether datasets are hard to index. As a result, the TV-Tree is not guaranteed to perform well in all situations that P-Sphere Trees are guaranteed to perform well. An interesting piece of concurrent related work, ([26]), relates fractal dimensionality, a concept very similar to contrast, to the performance of NN queries on R-Trees. They show that under certain conditions, the performance of the queries is directly related to the fractal dimensionality.

Perhaps the most relevant piece of related work is [13], which introduced the concept of contrast plots. In addition, they established asymptotic bounds on the overall problem that match the behavior of our own strategy. This is powerful evidence that the type of strategy presented in this paper is a promising new approach to the problem in general. [35] is notable in that it further develops the ideas in [13] by relating contrast to concentration of measure.

7 Conclusions

This paper has introduced several exciting new nearest neighbor query processing techniques, that by making use of the assumption that a random sample of the query distribution is available at index build time, have the following properties:

- They allow the data administrator to easily trade redundancy for time.
- Performance improvements apply equally to CPU and disk. The techniques are therefore suitable for both in-memory and secondary storage applications
- They can be applied to any scenario where the distance function is a metric.
- We present variants of the basic algorithm that offer excellent performance, if the user is willing to accept the fact that a small (user-specified) percentage of the time, the returned answer is not the nearest neighbor. These variants are particularly effective in low contrast situations. For instance, for 30 dimensional (identically distributed independent dimensions) uniform data, one of the techniques presented in this paper achieves about an 8 fold increase in speed (relative to linear scan) with about an equal blowup in space (relative to the dataset). It is well established that other techniques fail to beat linear scan at around 10 dimensions ([13], [36]) for the same datasets.
- P-Sphere trees consistently beat two alternative strategies' (SR-Tree [25], Density based indexing [24]) query times by a factor of 20 to 30 on the real datasets they were tested on.
- The theoretical results presented in this paper establish that dimensionality in and of itself is irrelevant to the performance of this structure, and instead relate the performance to a simple aspect of the workload, contrast distribution. This form of analysis greatly simplifies the task of describing overall behavior and leads to surprisingly simple and precise statements about how we can expect particular workloads to perform.
- For situations in which the query distribution follows the data distribution, the theoretical results presented in this paper, when combined with the results in ([13]) establish contrast as *the* performance limiting feature of sub linear strategies for NN query processing. A corollary of this is that the techniques presented in this paper are asymptotically optimal in the sense that their performance scales with the inherent difficulty of the problem.

Interesting future work includes theoretical studies that lead to tighter upper bounds for ND P-Sphere Trees, deterministic PK-Sphere Trees, and ND PK-Sphere Trees. In addition, a more thorough performance study comparing the techniques presented in this paper to a wider variety of NN processing techniques would definitely be useful. Handling updates and K-NN searches are also important extensions to this work.

References

[1] R. Motwani A. Gionis, P. Indyk. Similarity search in high dimensions via hashing. In *VLDB*, 1999.

[2] R. Agrawal, C. Faloutsos, and A. Swami. Efficient similarity search in sequence databases. In *FODO*, 1993.

[3] S. F. Altschul, W. Gish, W. Miller, E. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215, 1990.

[4] Y. H. Ang, Zhao Li, and S. H. Ong. Image retrieval based on multidimensional feature properties. In *SPIE vol. 2420*, 1995.

[5] S. Arya. *Nearest Neighbor Searching and Applications.* PhD thesis, Univ. of Maryland at College Park, 1995.

[6] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for nearest neighbor searching. In *Proc. 5th ACM SIAM Symposium on Discrete Algorithms*, 1994.

[7] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-Tree: An efficient and robust access method for points and rectangles. In *SIGMOD*, 1990.

[8] A. Belussi and C. Faloutsos. Estimating the selectivity of spatial queries using the 'correlation' fractal dimension. In *VLDB*, 1995.

[9] J. L. Bentley, B. W. Weide, and A. C. Yao. Optimal expected-time algorithms for closest point problem. *ACM Transactions on Mathematical Software*, 6(4), 1980.

[10] S. Berchtold, C. Böhm, B. Braunmüller, D. A. Keim, and H.-P. Kriegel. Fast parallel similarity search in multimedia databases. In *SIGMOD*, 1997.

[11] S. Berchtold, C. Böhm, and H.-P. Kriegel. The X-Tree: An index structure for high-dimensional data. In *VLDB*, 1996.

[12] M. Bern. Approximate closest point queries in high dimensions. *Information Processing Letters*, 45, 1993.

[13] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is nearest neighbors meaningful? In *ICDT*, 1999.

[14] G. Box, W. Hunter, and J. Hunter. *Statistics for Experimenters.* Wiley and Sons, 1978.

[15] T. Bozkaya and M. Ozsoyoglu. Distance-based indexing for high-dimensional metric spaces. In *PODS*, 1997.

[16] S. Brin. Near neighbor search in large metric spaces. In *VLDB*, 1995.

[17] K. Clarkson. Nearest neighbor queries in metric spaces. In *STOC*, 1997.

[18] C. Faloutsos et al. Efficient and effective querying ny image content. *Journal of Intelligent Information Systems*, 3(3), 1994.

[19] C. Faloutsos and V. Gaede. Analysis of n-dimensional quadtrees using the Housdorff fractal dimension. In *SIGMOD*, 1996.

[20] C. Faloutsos and I. Kamel. Beyond uniformity and independence: Analysis of R-trees using the concept of fractal dimension. In *PODS*, 1994.

[21] U. M. Fayyad and P. Smyth. Automated analysis and exploration of image databases: Results, progress and challenges. *Journal of intelligent information systems*, 4(1), 1995.

[22] J. Goldstein. *Improved Query Processing and Data Representation Techniques.* PhD thesis, University of Wisconsin - Madison, 1999.

[23] J. M. Hellerstein, E. Koutsoupias, and C. H. Papadimitriou. On the analysis of indexing schemes. In *PODS*, 1997.

[24] D. Geiger K. Bennett, U. Fayyad. Density-based indexing for approximate nearest-neighbor queries. In *Density-Based Indexing for Approximate Nearest-Neighbor Queries*, San Diego, California, 1999.

[25] N. Katayama and S. Satoh. The SR-tree: An index structure for high-dimensional nearest neighbor queries. In *PODS*, 1997.

[26] Philip Korn. Deflating the dimensionality curse using multiple fractal dimensions. In *ICDE*, San Diego, CA, February 2000.

[27] K.-I. Lin, H. V. Jagadish, and C. Faloutsos. The TV-Tree: An index structure for high-dimensional data. *VLDB Journal*, 3(4), 1994.

[28] B. S. Manjunath and W. Y. Ma. Texture features for browsing and retrieval of image data. In *IEEE Trans. on Pattern Analysis and Machine Learning*, volume 18(8), 1996.

[29] R. Mehrotra and J. E. Gary. Feature-based retrieval of similar shapes. In *ICDE*, 1992.

[30] H. Murase and S. K. Nayar. Visual learning and recognition of 3D objects from appearance. *Int. J. of Computer Vision*, 14(1), 1995.

[31] U. Shaft N. Megiddo. Efficient nearest neighbors indexing based on a collection of space filling curves. Technical Report RJ 10093(91909), IBM Almaden Research Center, November 1997.

[32] S. A. Nene and S. K. Nayar. A simple algorithm for nearest neighbor search in high dimensions. In *IEEE Trans. on Pattern Analysis and Machine Learning*, volume 18(8), 1996.

[33] R. Motwani P. Indyk. Approximate nearest neighbor - towards removing the curse of dimensionality. In *STOC*, 1998.

[34] A. Pentland, R. W. Picard, and S. Scalroff. Photobook: Tools for content based manipulation of image databases. In *SPIE Volume 2185*, 1994.

[35] Vladimir Pestov. On the geometry of similarity search: dimensionality curse and concentration of measure. *Information Processing Letters*, To Appear.

[36] S. Blott R. Weber, H.-J. Schek. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, 1998.

[37] M. J. Swain and D. H. Ballard. Color indexing. *Inter. Journal of Computer Vision*, 7(1), 1991.

[38] D. L. Swets and J. Weng. Using discriminant eigenfeatures for image retrieval. In *IEEE Trans. on Pattern Analysis and Machine Learning*, volume 18(8), 1996.

[39] G. Taubin and D. B. Cooper. Recognition and positioning of rigid objects using algebraic moment invariants. In *SPIE Vol. 1570*, 1991.

[40] D. A. White and R. Jain. Similarity indexing with the SS-Tree. In *ICDE*, 1996.