NNH: Improving Performance of Nearest-Neighbor Searches Using Histograms (Full Version. UCI Technical Report, Dec. 2003)

Liang Jin[1], Nick Koudas[2], and Chen Li[1]

[1] School of Information and Computer Science, University of California, Irvine, Irvine, CA 92612, USA
{liangj,chenli}@ics.uci.edu**
[2] AT&T Labs Research, 180 Park Avenue, Florham Park, NJ 07932, USA
koudas@research.att.com

Abstract. Efficient search for nearest neighbors (NN) is a fundamental problem arising in a large variety of applications of vast practical interest. In this paper we propose a novel technique, called NNH ("Nearest Neighbor Histograms"), which uses specific histogram structures to improve the performance of NN search algorithms. A primary feature of our proposal is that such histogram structures can co-exist in conjunction with a plethora of NN search algorithms without the need to substantially modify them. The main idea behind our proposal is to choose a small number of pivot objects in the space, and pre-calculate the distances to their nearest neighbors. We provide a complete specification of such histogram structures and show how to make use of the information they provide towards more effective searching. In particular, we show how to construct them, how to decide the number of pivots, how to choose pivot objects, how to incrementally maintain them under dynamic updates, and how to utilize them in conjunction with a variety of NN search algorithms to improve the performance of NN searches. Our intensive experiments show that nearest neighbor histograms can be efficiently constructed and maintained, and when used in conjunction with a variety of algorithms for NN search, they can improve the performance dramatically.

1 Introduction

Nearest-neighbor (NN) searches arise in a large variety of applications such as image and video databases [1], CAD, information retrieval (IR) [2], data compression [3], and string matching/searching [4]. The basic version of the k-NN problem is to find the k nearest neighbors of a query object in a database, according to a distance measurement. In these applications, objects are often

** These two authors were supported by NSF CAREER award No. IIS-0238586 and a UCI CORCLR grant.

characterized by features and represented as points in a multi-dimensional space. For instance, we often represent an image as a multi-dimensional vector using features such as histograms of colors and textures. A typical query in an image database is to find images most similar to a given query image utilizing such features. As another example, in information retrieval, one often wishes to locate documents that are most similar to a given query document, considering a set of features extracted from the documents [2].

Variations of the basic k-NN problem include high-dimensional joins between point sets. For instance, an *all-pair k-nearest neighbor join* between two point sets seeks to identify the k closest pairs among all pairs from two sets [5, 6]. An *all-pair k-nearest neighbor semi-join* between two point sets reports, for each object in one data set, its k nearest neighbors in the second set [5].

Many algorithms have been proposed to support nearest-neighbor queries. Most of them use a high-dimensional indexing structure, such as an R-tree [7] or one of its variations. For instance, in the case of an R-tree, these algorithms use a branch-and-bound approach to traverse the tree top down, and use distance bounds between objects to prune branches (minimum-bounding rectangles, MBR's) that do not need to be considered [8, 9]. A priority queue of interior nodes is maintained based on their distances to the query object. In the various forms of high-dimensional joins between point sets, a queue is maintained to keep track of pairs of objects or nodes in the two data sets. Usually this queue is ordered based on various criteria, to facilitate early stopping and/or pruning.

One of the main challenges in these algorithms is to perform effective pruning of the search space, and subsequently achieve good search performance. The performance of such an algorithm heavily depends on the number of disk accesses (often determined by the number of branches visited in the traversal) and its run-time memory requirements (priority-queue size), which indicates memory (for priority-queue storage) and processor requirements for maintaining and manipulating the queue. Performance can deteriorate if too many branches are visited and/or too many entries are maintained in the priority queue, especially in a high-dimensional space due to the well-known "curse of dimensionality" problem [1].

In this paper we develop a novel technique to improve the performance of these algorithms by keeping histogram structures (called "NNH") of the nearest-neighbor distances of a preselected collection of objects (called "pivots"). Such structures record the nearest-neighbor distances for each pivot. These distances can be utilized to estimate the distance at which the k-nearest neighbors for each query object can be identified. They can subsequently be used to improve the performance of a variety of nearest-neighbor search and related algorithms via more effective pruning. The histogram structures proposed can co-exist in conjunction with a plethora of NN search algorithms without the need to substantially modify these algorithms.

There are several challenges associated with the construction and use of such structures. (1) The construction time should be small, their storage requirements should be minimal, and the estimates derived from them should be precise. (2)

They should be easy to use towards improving the performance of a variety of nearest-neighbor algorithms. (3) Such structures should support efficient incremental maintenance under dynamic database updates. In this paper we provide a complete specification of such histogram structures, showing how to efficiently and accurately construct them, how to choose pivots effectively, how to incrementally maintain them under dynamic updates, and how to utilize them in conjunction with a variety of NN search algorithms to improve the performance of NN searches.

The rest of the paper is organized as follows. Section 2 outlines the formal definition of a nearest-neighbor histogram (NNH) structure, and describes its efficient and accurate construction. In Section 3 we show how to use such histograms to improve the performance for a variety of NN algorithms. Section 4 discusses how to choose pivots in such a structure. In Section 5 we discuss how to incrementally maintain an NN histogram structure in the presence of dynamic database updates. In Section 6 we report our extensive experimental results, evaluating the construction time, maintenance algorithms, and the efficiency of our proposed histograms when used in conjunction with a variety of algorithms for NN search to improve their performance.

1.1 Related Work

Summary structures in the form of histograms have been utilized extensively in databases in a variety of important problems, such as selectivity estimation [10–12] and approximate query answering [13, 14]. In these problems, the main objective is to approximate the distribution of frequency values using specific functions and a limited amount of space.

Nearest Neighbor (NN) queries are very popular in many diverse sources of information, including spatial, image, audio, and video databases. Many algorithms exist for efficiently identifying the nearest neighbors of low and high-dimensional data points for main memory data collections, in the field of computational geometry [15]. In databases, many different families of high-dimensional indexing structures are available [16], and various techniques are known for performing NN searches tailored to the specifics of each family of indexing structures. Such techniques include NN searches for the *entity grouping* family of indexing structures (e.g., R-trees [7]) and NN searches for the *space partitioning* family (e.g., Quad-trees [17]). The underlying principle of all such techniques is a top down traversal of the hierarchical indexing structure, using an initial bound for the distance to the NN and then iteratively refining such estimate (pruning parts of the search space, if possible), until the NN is identified. Such techniques generalize in a straightforward way to cases where the closest k neighbors (k-NN) are requested.

In addition to the importance of NN queries as stand-alone query types, a variety of other query types make use of NN searches. *Spatial or multidimensional* joins [18] are a representative example of such query types. In the context of spatial joins, different algorithms have been proposed for spatial NN semi-joins

and all-pair NN joins [5]. Such queries seek to "correlate" points from different data sets using the NN information.

NN search algorithms can benefit from the histogram structures proposed in this paper enabling them to perform more effective pruning. For example, utilizing a good estimate to the distance of the k-th nearest neighbor of a query point, one can form essentially a range query to identify nearest neighbors, using the query object and the estimated distance and treating the search algorithm as a "black box" without modifying its code.

Various studies [19–22] use notions of pivots or anchors or foci for efficient indexing and query processing. Our approach is related in principle, but our objectives and methodologies are different. We will elaborate on these issues in Section 4.3 and demonstrate the utility of our proposal in Section 6 experimentally.

2 NNH: Nearest-Neighbor Histograms

In this section we provide an overview of nearest-neighbor searches, present the formal definition of nearest-neighbor histograms, and discuss their construction.

Consider a data set $D = \{p_1, \ldots, p_n\}$ with n objects in a Euclidean space $\Re^d$ under some l_p form. Formally, there is a distance function $\Delta : \Re^d \times \Re^d \rightarrow \Re$, such that given two objects p_i and p_j, their distance is defined as

$$\Delta(p_i, p_j) = \Big(\sum_{1 \leq l \leq d} |x_l^i - x_l^j|^p \Big)^{1/p} \tag{1}$$

where $(x_1^i, \ldots, x_d^i)$ and $(x_1^j, \ldots, x_d^j)$ are the d coordinates of objects p_i and p_j, respectively.

Definition 1. *Nearest Neighbor Search Given a query object, a j-Nearest Neighbor (NN) search returns the j points in the database that are closest to the query object.*

Given an object p in the space, its *NN distance vector* of size t is a vector $v(p) = \langle r_1, \ldots, r_t \rangle$, in which each r_i is the distance of p's i-th nearest neighbor in the database D.

A *nearest-neighbor histogram* (NNH) of the data set, denoted H, is a collection of objects (called "pivots") with their NN vectors. In principle, these pivot points may or may not correspond to points in the database. In the rest of the paper, we assume that the pivots are *not* part of the data set for the purpose of easy dynamic maintenance, as discussed in Section 5. Initially all the vectors have the same length, denoted T, which is a design parameter and forms an upper bound on the number of neighbors NN queries specify as their desired result. We choose to fix the value of T in order to control the storage requirement for the NNH structure [23]. For any pivot p, let $H(p, j)$ denote the j-NN distance of p recorded in H. Let

$$H(p) = \langle H(p, 1), \ldots, H(p, T) \rangle \tag{2}$$

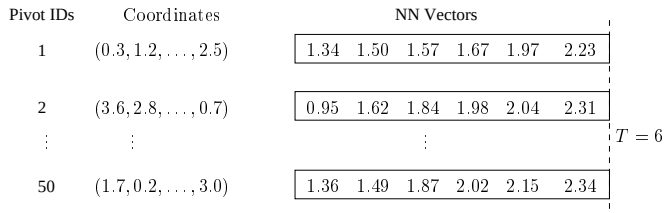


Fig. 1. An NN histogram structure H.

denote the NN vector for a pivot p in the histogram H.

Figure 1 shows such a structure. It has $m = 50$ pivots, each of which has an NN vector of size $T = 6$. For instance, $H(p_1, 3) = 1.57$, $H(p_2, 6) = 2.31$, and $H(p_{50}, 5) = 2.15$.

In Section 4 we discuss how to choose pivots to construct a histogram. Once we have chosen pivots, for each p of them, we run a T-NN search to find all its T nearest neighbors. Then, for each $j \in \{1, \ldots, T\}$, we calculate the distance from the j-th nearest neighbor to the object p, and use these T distances to construct an NN vector for p, by sorting these distances to form the vector.

3 Improving Query Performance

In this section we discuss how to utilize the information captured by nearest neighbor histograms to improve query performance. A number of important queries could benefit from such histograms, including k-nearest neighbor search (k-NN) and various forms of high-dimensional joins between point sets, such as *all-pair k-nearest neighbor joins* (i.e., finding k closest pairs among all the pairs between two data sets), and k-nearest neighbor semi-joins (i.e., finding k NN's in the second data set for each object in the first set). The improvements to such algorithms are twofold:(a) the processor time can be reduced substantially, due to advanced pruning. This reduction is important since for data sets of high dimensionality, distance computations are expensive and processor time becomes a significant fraction of overall query-processing time [24]; and (b) the memory requirement can be reduced significantly due to the much smaller queue size.

Most NN algorithms assume that there is a high-dimensional indexing structure on the data set, such as an R-tree or its variant [7]. They use a branch-and-bound approach, in which they traverse the tree from the root to the leaf nodes, and do necessary pruning in the traversal. To simplify our presentation, we choose to highlight how it is possible to utilize our histogram structures to improve the performance of common queries involving R-trees [7]. Similar concepts carry out easily to other structures (e.g. SS-tree [25], SR-tree [26], etc.) and algorithms as well.

3.1 Utilizing Histograms in k-NN Queries

Nearest neighbor searches involving R-trees request the nearest or k-NN objects to a query point q. A typical k-NN search involving R-trees follows a branch-and-bound strategy traversing the index top down. It maintains *a priority queue* of "active" minimum bounding rectangles (MBR's) [8, 9]. At each step, it maintains a bound to the k-NN distance, δ, from the query point. This bound is initialized to infinity when the search starts. Using the geometry of an MBR mbr and the coordinates of query object q, an upper-bound distance of the query point to the nearest point in the mbr, namely $MINMAXDIST(q, mbr)$, can be derived [27]. In a similar fashion, a lower bound of the distance from q to any point in mbr, namely $MINDIST(q, mbr)$, is derived. Using these estimates, we can prune MBR's from the queue as follows:[3] (1)For an MBR mbr, if its $MINDIST(q, mbr)$ is greater than $MINMAXDIST(q, mbr')$ of another MBR mbr', then mbr can be pruned. (2) If $MINDIST(q, mbr)$ for an MBR mbr is greater than δ, then this mbr can be pruned.

In the presence of NN histograms, we can utilize the distances to estimate an upper bound of the k-NN of the query object q. Recall that the NN histogram includes the NN distances for selected pivots only, and it does not convey immediate information about the NN distances of objects not encoded in the histogram structure. We can estimate the distance of q to its k-NN, denoted δ_q, using the triangle inequality between q and any pivot p_i in the histogram, as shown in Figure 2:

$$\delta_q \leq \Delta(q, p_i) + H(p_i, k), 1 \leq i \leq m.^4 \tag{3}$$

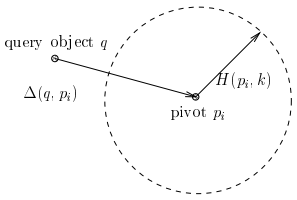


Fig. 2. Estimating the k-NN distance of query object q using an NNH.

Thus, we can obtain an upper bound estimate δ_q^{est} of δ_q as

$$\delta_q^{est} = \min_{1 \leq i \leq m} \big(\Delta(q, p_i) + H(p_i, k)\big) \tag{4}$$

The complexity of this estimation step is $O(m)$. Since the histogram structure is small enough to fit in the memory, and the number of pivots is small, the above

[3] For simplicity, we focus on the pruning steps for finding the nearest neighbor. They can be easily generalized to pruning steps for finding the k-NN objects.

[4] Note that we use $H(p_i, k)$ instead of $H(p_i, k-1)$ because in this paper we assume that the pivots are *not* part of the database.

estimation can be conducted efficiently. After computing the initial estimate δ_q^{est} for the query object q, we can use this distance to help the search process prune MBR's. That is, the search progresses by evaluating $MINMAXDIST$ and $MINDIST$ between q and an R-tree MBR mbr as before. Besides the standard pruning steps discussed above, the algorithm also checks if

$$MINDIST(q, mbr) > \delta_q^{est} \tag{5}$$

is true. If so, then this mbr can be pruned, since we know the k-NN's of q cannot be in this MBR. Thus we do not need to insert this MBR into the queue, reducing the memory requirement (queue size) and the later operations of this MBR in the queue. We will explore these advantages experimentally in Section 6.

Notice that the algorithm in [8, 9] is shown to be IO optimal [28]. Utilizing our NNH structure may not reduce the number of IOs during the R-tree traversal. However, our structure can help reduce the size of the priority queue, and the number of queue operations. This reduction can help reduce the running time of the algorithm, as shown in our experiments (Section 6). In addition, if the queue becomes too large to fit into memory, this reduction could even help us reduce the IO cost since part of the queue needs to be paged on disk.

3.2 Utilizing Histograms in k-NN Joins

A related methodology could also be applied in the case of all-pairs k-NN join queries using R-trees. The bulk of algorithms for this purpose progress by inserting pairs of MBR's between index nodes from corresponding trees in a priority queue and recursively (top-down) refining the search. In this case, using k-NN histograms, one could perform, in addition to the type of pruning highlighted above, even more powerful pruning.

More specifically, let us see how to utilize such a histogram to do pruning in a k-NN semi-join search [5]. This problem tries to find for each object o_1 in a data set D_1, all o_1's k-nearest neighbors in a data set D_2. (This pruning technique can be easily generalized to other join algorithms.) If the two data sets are the same, the join becomes a self semi-join, i.e., finding k-NN's for all objects in the data set. Assume the two data sets are indexed in two R-trees, R_1 and R_2, respectively. A preliminary algorithm described in [5] keeps a priority queue of MBR pairs between index nodes from the two trees. In addition, for each object o_1 in D_1, we can keep track of objects o_2 in D_2 whose pair $\langle o_1, o_2 \rangle$ has been reported. If $k = 1$, we can just keep track of D_1 objects whose nearest neighbor has been reported. For $k > 1$, we can output its nearest neighbors while traversing the trees, thus we only need to keep a counter for this object o_1. We stop searching for neighbors of this object when this counter reaches k.

Suppose we have constructed a histogram H_2 for data set D_2. We can utilize H_2 to prune portions of the search space in bulk by pruning pairs of MBR's from the priority queue as follows. For each object o_1 in D_1, besides the information kept in the original search algorithm, we also keep an estimated radius δ_{o_1} of

the k-NN in D_2 for this object. We can get an initial estimate for δ_{o_1} as before (Section 3.1). Similarly, we will not insert a pair (o_1, mbr_2) into the queue if

$$MINDIST(o_1, mbr_2) \geq \delta_{o_1}. \tag{6}$$

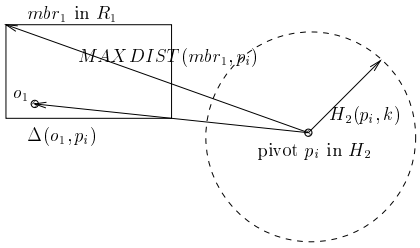


Fig. 3. Pruning MBR-MBR pairs in the queue.

The above technique can be used to prune object-MBR pairs. Now we show that this histogram is even capable to prune MBR-MBR pairs. As shown in Figure 3, for each MBR mbr_1 in tree R_1 for dataset D_1, consider a pivot p_i in the NN histogram H_2 of dataset D_2. For each possible object o_1 in mbr_1, using the triangle inequality between o_1 and p_i, we know that the k-NN distance to o_1:

$$\delta_{o_1} \leq \Delta(o_1, p_i) + H_2(p_i, k) \leq MAXDIST(mbr_1, p_i) + H_2(p_i, k). \tag{7}$$

$MAXDIST(mbr_1, p_i)$ is an upper bound of the distance between any object in mbr_1 and object p_i. Therefore,

$$\delta^{est}_{mbr_1} = \min_{p_i \in H_2} (MAXDIST(mbr_1, p_i) + H_2(p_i, k)) \tag{8}$$

is an upper bound for the k-NN distance for any object o_1 in mbr_1. This distance estimate can be used to prune any (mbr_1, mbr_2) from the queue, if

$$\delta^{est}_{mbr_1} \leq MINDIST(mbr_1, mbr_2) \tag{9}$$

where $MINDIST(mbr_1, mbr_2)$ is the lower bound of the distance between any pair of objects from mbr_1 and mbr_2 respectively. $MINDIST(mbr_1, mbr_2)$ can be easily calculated using the coordinates of the corners of the two MBR's.

In order to use this technique to do pruning, in addition to keeping an estimate for the distance of the k-NN for each object in data set D_1, we also keep an estimate of distance δ_{mbr_1} for each MBR mbr_1 in R_1. This number tends to be smaller than the number of objects in D_1. These MBR-distance estimates can be used to prune many MBR-MBR pairs, and reduce the queue size as well as the number of disk IO's.

Pruning in all-pair k-nearest neighbor joins: The pruning techniques described above can be adapted to perform more effective pruning when finding

the k-nearest pairs among all pairs of objects from two data sets [5, 6]. Notice that so far we have assumed that only D_2 has a histogram H_2. If the first data set D_1 also has a histogram H_1, similar pruning steps using H_1 can be done to do more effective pruning, since the two sets are symmetric in this join problem.

By effectively limiting the size of the queue in terms of number of pairs, not only the memory requirements are smaller, but also the running time is much shorter. As we will see in Section 6, these pruning techniques have profound impact on the performance of these join algorithms.

4 Constructing NNH Using Good Pivots

Pivot points are vital to NNH for obtaining distance estimates in answering NN queries. We now turn to the problems associated with the choice of pivot points, and the number of pivots.

Assuming we decide to choose m pivot points. The storage requirement for NNH becomes $O(mT)$, since for each pivot point p we will associate a vector of p's distances to its T-NN's. Let the m pivot points be $p_1, \ldots, p_m$. Given a query point q, we can obtain an estimate to the distance of q's k-NN, δ_q^{est}, by returning

$$\min_{1 \le i \le m} \big(\Delta(q, p_i) + H(p_i, k) \big) \tag{10}$$

This estimate is an upper bound of the true distance of q to its k-NN point and is obtained utilizing the triangle inequality, among q, a pivot point p_i, and p_i's k-NN point. Assuming that the true distance of q to its k-NN is $\Delta_k(q)$. This estimation incurs an error of

$$\min_{1 \le i \le m} \big(\Delta(q, p_i) + H(p_i, k) \big) - \Delta_k(q). \tag{11}$$

Thus, the expected error for the estimation of any k-NN query q (for $1 \le k \le T$) becomes:

$$\frac{1}{T} \sum_{k=1}^{T} \Big(\min_{1 \le i \le m} \big(\Delta(q, p_i) + H(p_i, k) \big) - \Delta_k(q) \Big) \tag{12}$$

The larger the number of pivot points m, the larger the space requirements of the corresponding NNH structure. However, at the same time, the larger m the higher the chances of obtaining a more accurate estimate δ_q^{est}. To see the reason, observe that for a query point q, the estimate obtained with $m+1$ pivot points is the same to or better than that obtained with m pivot points. It is possible that either the additional pivot point p_{m+1} is closer to q or the distance to its k-NN is tighter, or both. As a result, the distance estimate for q might be better. It is evident that by increasing the number of pivot points, we increase the storage overhead of NNH, but potentially improve its estimation accuracy.

4.1 Choosing Pivots

Given the number of pivot points m devoted to the NNH, we want to choose the best m pivots to minimize the expected distance estimation error for any query q. Query points are not known in advance, and any query point q on the data set D could be equally likely. Moreover, the parameter k in a k-NN distance estimate is also not known in advance, but provided at run time. At best we are aware of an upper bound for it. For this reason, we decide to minimize the term

$$\Delta(q, p_i) + H(p_i, k), 1 \leq i \leq m \tag{13}$$

providing the estimate to the k-NN query, by choosing as pivot points the set S of m points that minimize the quantity:

$$\sum_{q \in D, p_i \in S} \Delta(q, p_i) \tag{14}$$

That is, we minimize distances to pivot points assuming queries are points of D. This goal is the well-known objective of the clustering problem and consequently we obtain the m cluster centroids of D, as pivots. It is, in expectation, the best choice of m pivots, provided that query points q belong to D, assuming no knowledge of the number of NN points k-NN queries specify. This choice has the added utility of enabling a plethora of known clustering algorithms (e.g., [29, 30]) to be applied in our problem of selecting the pivots.

4.2 Choosing the Number of Pivots

We now present our methodology for deciding the number of pivots m to use when constructing an NNH structure. We start by describing a way to quantify the benefits of an NNH with a given number of pivots. Consider an NN search for a query object q. We use both the standard algorithm [8] and the improved one described in Section 3.1. The *queue-size-reduction ratio* of the NNH with m pivots for this search is:

$$\text{reduction ratio} = 1 - \frac{\text{max queue size of improved algorithm}}{\text{max queue size of standard algorithm}} \tag{15}$$

This ratio indicates how much memory the NNH can save by reducing the queue size. Notice that we could also use the kNN semi-join operation to define the corresponding queue-size-reduction ratio. Our experiments showed no major difference between the results of these two definitions, so we adopt the former operation for defining queue-size-reduction.

Given a number m, we use the approach in Section 4.1 to find m good pivots. We randomly select a certain number (say, 10) of objects in the data set to perform NN searches, and calculate the average of their queue-size-reduction ratios. The average value, denoted B_m, is a good measurement of the benefits of the NNH with m pivots. In particular, using standard tail inequality theory

[31] one can show that such an estimate converges (is an unbiased estimate) to the true benefit for random queries as the size of the sample grows. We omit the details due to lack of space.

Our algorithm for deciding the m value for a data set is shown in Figure 4. Its main idea is as follows: we first initialize m to a small number (e.g., 5) and measure the average queue-size-reduction ratio B_m. Then we increment the number of pivots (using the parameter SP) and judge how much benefit (in terms of the reduction ratio) the increased number of pivots can achieve. When there is no big difference (using the parameter ε) between consecutive rounds, the algorithm terminates and reports the current number of pivots.

Algorithm
Input: • D: data set.
• SP: step for incrementing the number of pivots.
• ε: Condition to stop searching.
Output: a good number of pivots.
Variables: • B_{new}: reduction ratio in current iteration.
• B_{old}: reduction ratio in last iteration.
• m: number of pivots.
Method:

```
B_new ← 0.0; m ← 0;
DO {
    B_old ← B_new;
    m ← m + SP;
    Choose m pivots to construct an NNH;
    Perform kNN queries for sampled objects using the NNH;
    B_new ← average reduction ratio of the searches;
}
WHILE (B_new - B_old > ε);
RETURN m;
```

Fig. 4. Algorithm for deciding pivot number

One potential problem of this iterative algorithm is that it could "get stuck" at a local "optimal" number of pivots (since the local gain on the reduction ratio becomes small), although it is still possible to achieve a better reduction ratio using a larger pivot number. We can modify the algorithm slightly to solve this problem. Before the algorithm terminates, it "looks forward" to check a larger pivot number, and see if we can gain a better reduction ratio. Techniques for doing so are standard in hill-climbing type of algorithms [32] and they can be readily incorporated if required.

4.3 Comparisons with Other Approaches

There have been studies on selecting pivots to improve similarity search in metric spaces [19–21, 33]. The studies in [20, 21] claim that it is desirable to select pivots

that are far away from each other and from the rest of the elements in the database. These studies converge on the claim that the number of pivots should be related to the "intrinsic dimensionality" of a dataset [34].

Previous studies (e.g., [19–21, 33]) focus on search in general metric spaces and use pivots and the triangle inequality to avoid unnecessary distance calculations. They deploy intersection of the distance "rings" to all the pivots to form a candidate set in a search. As a result it is desirable for the intersection to be as small as possible, and consequently choose the pivots to be far away from each other, preferably outliers [20].

Existing techniques are not suitable for choosing pivots in an NNH. The reason is that, an NNH relies on the local information kept by each pivot to estimate the distances of the nearest neighbors to a query object, so that effective pruning can be performed. Thus, the way we choose the number of pivots depends on how clustered the objects are, so does the way we choose the pivots. In particular, the number of pivots depends on how many clusters data objects form. In addition, it is desirable to choose a pivot that is very close to many objects, so that it can capture the local distance information about these objects. Thus, the selected pivots might not be too far from each other, depending upon the distribution of the objects. Clearly outliers are not good pivots for NNH, since they cannot capture the local information of the neighboring objects. In Section 6.4, we will experimentally show the performance advantages of our approach.

5 Incremental Maintenance

In this section we discuss how to maintain an NN histogram in the presence of dynamic updates (insertions/deletions) in the underlying database. Under dynamic updates, a number of important issues arise. When a new object is inserted, it could affect the structure of the nearest neighbors of many pivots, possibly prompting changes to their NN vectors. Similarly, when an object is deleted, it could also change the NN structure of pivots. We associate a separate value E_i for each pivot p_i, which identifies the number of positions (starting from the beginning) in the vector $H(p_i)$ that can be utilized to provide distance estimates. All the distances after the E_i-th position cannot be used for NN distance-estimation purposes, since as a result of some update operations they are not valid anymore. Initially, $E_i = T$ for all the pivots.

Figure 5 presents an example NN histogram in which each vector V_i has a valid length E_i. All the vectors have the same initial length $T = 6$, as shown in Figure 1. After modifications, the valid length E_1 of vector $H(p_1)$ is 6, meaning that all the 6 distances are valid. For pivot p_2, $E_2 = 4$ means we can only use its first four distances as valid radii to derive NN distance estimates, and the last two distances should be ignored. Similarly, E_{50} is 5 for the NN vector of pivot p_{50}.

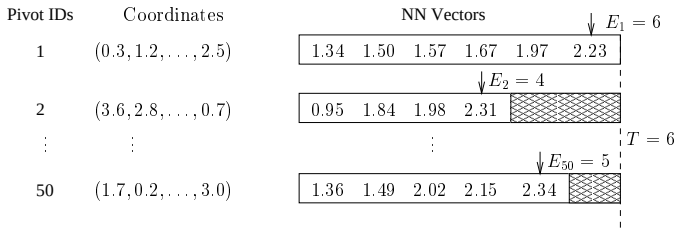


Fig. 5. Valid length for a NN vector.

5.1 Insertion

Upon insertion of an object o_{new} into the data set, we perform the following task. For each pivot p_i whose E_i nearest neighbors may include o_{new}, we need to update their NN vectors. For updates, we scan all the pivots in the histogram, and for each pivot p_i, we compute the distance between p_i and the new object, i.e., $\Delta(p_i, o_{new})$. Consider the distance vector of p_i:

$$H(p_i) = \langle r_1, \ldots, r_{E_i} \rangle$$

in the NN histogram. We locate the position of $\Delta(p_i, o_{new})$ in this vector. Assume

$$r_{j-1} \leq \Delta(p_i, o_{new}) < r_j$$

where $j \leq E_i$. There are two cases.

- Such a position cannot be found. Then this new object cannot be among the E_i nearest neighbors of p_i and we do not need to update this NN vector.
- Such a position is found. As shown in Figure 6, o_{new} cannot be among the (j-1)-NN objects of p_i. Therefore, as shown in Figure 7, we can insert this distance $\Delta(p_i, o_{new})$ to the j-th slot in the NN vector, and shift the $(E_i - j)$ distances after the slot to the right. Correspondingly, we increment the number of valid distances for this vector E_i by 1. If E_i becomes larger than T, we set $E_i = T$.

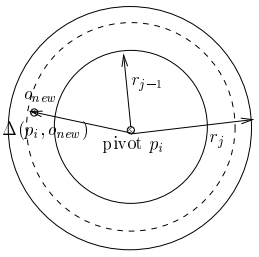


Fig. 6. Inserting an object o_{new}.

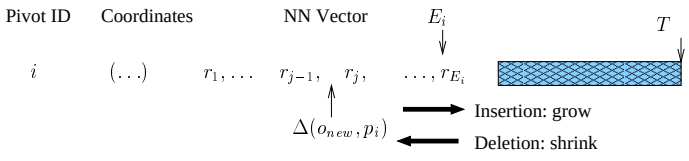


Fig. 7. Updating an NN vector.

5.2 Deletion

When deleting an object o_{del}, we need to update the vectors for the pivots whose nearest neighbors may have included o_{del}. For each pivot p_i, similarly to the insertion case, we consider the distance vector of p_i in H. We locate the position of $\Delta(p_i, o_{del})$ in this vector:

$$r_j = \Delta(p_i, o_{del})$$

where $j \leq E_i$. Two cases of interest arise:

- Such a position cannot be found. Then the deleted object can not be among the E_i nearest neighbors of p_i. Therefore, we do not need to modify the NN vector $H(p_i)$ for this pivot p_i.
- Such a position is found. We remove r_j from the NN vector, and shift the distances $r_{j+1}, \ldots, r_{E_i}$ to the left, as shown in Figure 7. We decrement E_i by 1.

As an example, consider the NN vectors shown in Figure 1. After deleting its original $H(p_2, 2)$ and $H(p_2, 5)$ for pivot p_2, and deleting the original $H(p_{50}, 3)$ for pivot p_{50}, the new histogram is shown in Figure 5.

Notice that in this paper we assume that the pivots are *not* part of the database, deleting a data point from the database will not affect the formation of pivots. It only could change the values in the NN vectors.

Lemma 1. *After the above procedures for insertions and deletions, the new distance vectors form an NN histogram of the (modified) data set.*

The worst case complexity of each procedure above for insertions and deletions is $O(m * T * log(T))$, where m is the number of pivots in the histogram, and T is the maximal length of each distance vector. The reason is that we can do a binary search to find the inserting position of $\Delta(p_i, o_{new})$ or $\Delta(p_i, o_{del})$ in the NN vector of p_i, which takes $O(log(T))$ time. If the position is found, we need $O(T)$ time to shift the rest of the vector (if we use a simple array implementation of the vector in memory). This operation is executed for all the m pivots in the histogram. Since the number of pivots is small and the histogram can fit in memory, the maintenance cost is low (see Section 6.3).

After many insertions and deletions, if the valid length of an NN vector becomes too small, we can recompute a new NN distance for this pivot by doing a T-NN search. In addition, we can periodically run the algorithms in Section 4 to choose a new collection of pivots and construct the corresponding NN histogram.

6 Experiments

In this section we present the results of an experimental evaluation assessing the utility of the proposed NNH techniques when used in conjunction with algorithms involving NN searches. We used two datasets in the evaluation:

1. A Corel image database (Corel). It consists of 60,000 color histogram features. Each feature is a 32-dimensional float vector.
2. A time-series data set from the AT&T Labs. It consists of 5 million numerical values generated for a fixed time span. We constructed datasets with different dimensionalities by using different time-window sizes. For each window of size d, we construct a d-dimensional data set by sequentially scanning the data set, assigning all points in the underlying time series falling inside the window to a multidimensional point, and we advance the window creating new points, such that successive windows do not overlap.

We conducted sets of experiments evaluating the utility of the proposed histograms for different problems of interest. The experiments evaluated the effect of such an NNH structure in terms of the reduction of queue size (memory requirement) and running time, for both a single k-NN query and a k-NN semi-join query. In addition, we want to show the effect of different numbers of pivots, and different ways to choose the number of the pivots and the pivot objects. In the experiments we also reported the cost (size and construction time) of such an NNH structure.

Both datasets exhibit similar trend in the results. We mainly focus on reporting the results of Corel image database. We will use the time-series data to show the effects of dimensionality in our NNH approach and evaluate different approaches to choosing pivots in our approach.

All experiments were performed on a SUN Ultra 4 workstation with four 300MHz CPU's and 3GB memory, under the SUN OS 4.7 operating system. The software was compiled using GNU C++, using its maximum optimization ("-O4"). For the priority queue required in the algorithm described in [8, 9], we used a heap implementation from the Standard Template Library [35].

6.1 Improving k-NN Search

We present experiments showing the effectiveness of NNH for NN searches using R-trees on the Corel database. In general, any indexing technique that follows a branch-and-bound strategy to answer NN queries can be used. As an example, we implemented the k-NN algorithm described in [8, 9], using an R-tree with a page size of 8,192 bytes. We refer to this algorithm as the "standard version" of the NN search. We assume that a query object may not belong to the data set. We begin by using the NN histogram to get the initial estimate of the k-NN distance for the query object. We perform pruning as described in Section 3.1: We use the estimated k-NN distance to discard some MBR's from the priority queue based on the MINDIST of each MBR to the query object. We refer to this version of the NN algorithm as the "improved version."

Reducing Memory Requirements and Running Time Several important factors affect the performance of a k-NN search including memory usage, running time, and number of IO's. Memory usage impacts running time significantly,

since the larger the priority queue gets, the more distance comparisons the algorithm has to perform, penalizing its running time. At large dimensionality, distance comparisons are a significant fraction of the algorithm's running time.

We use the size of the priority queue to measure memory usage, since additional memory requirements are relatively small. While performing an R-tree traversal, we kept two numbers: the maximal number of entries in the queue, and the average number of entries in the queue. Every time we inserted an entry to the queue, we recorded its queue size. After the search ended, we computed the average of these sizes as the average queue size.

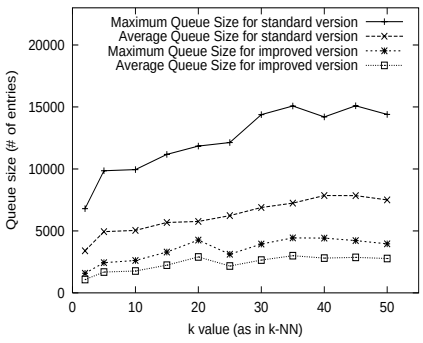


Fig. 8. Queue size of k-NN search.

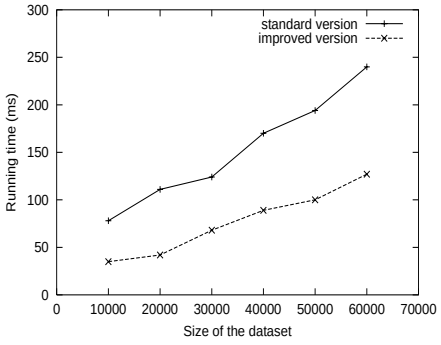


Fig. 9. Running Time for 10-NN queries.

Figure 8 shows the maximal queue size and the average queue size for the standard and improved versions of the NN algorithm. We first ran a k-means algorithm to generate 100 pivots, and constructed the NN histogram with $T = 50$. Then we performed 10-NN queries for 100 randomly generated query objects, and calculated the average number for both the maximal queue size and the average queue size. We observe that we can reduce both the queue sizes dramatically by utilizing the NNH to prune the search. For instance, when searching for the 10-NN's for a query object, the average queue size of the standard version was $5,036$, while it was only $1,768$ for the improved version, which was only about 35% of that of the standard version. Similarly, the maximal queue size required by the standard version was $9,950$, while it was only $2,611$ for the improved version. One interesting observation is that the queue sizes do not change too much as the value of k (specified in NN search) changes.

The maximum queue size determines how much memory the algorithm requires. If a data set is very large, the queue size used by the standard version of the k-NN algorithm might not fit in memory, which can make performance deteriorate substantially. In this case, utilizing NNH to reduce the queue size can dramatically improve the performance of the search. We will see shortly that this improvement will be even more beneficial in the k-NN-join case.

Figure 9 shows the running times for different dataset sizes for both versions, when we ran 10-NN queries. We let the number of image features vary in the

Corel database. It is shown that the improved version can always speed up the query as the size increases. For instance, when the dataset is $60,000$, the standard version requires an average of 238ms to retrieve the 10 NN's for a query object, while the improved version requires only an average of 127ms. This improvement increases as the size of the dataset increases. The reason is that, increasing the size of the dataset increases the size of the priority queue maintained in the algorithm. As a result, the standard version has to spend additional processor time computing distances between points and MBR's of the index. In contrast the effective pruning achieved by the improved version reduces this overhead.

Choosing the Number of Pivots As discussed in Section 4, by choosing the pivots using cluster centroids, the larger the number of pivots, the better the distance estimates, and as a result the better performance improvement.

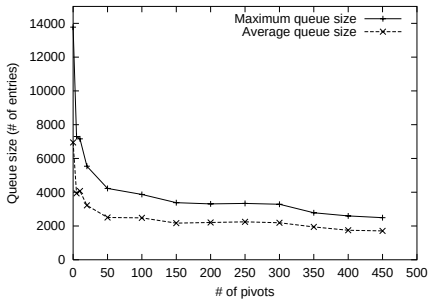


Fig. 10. Performance for different # of pivots

Figure 10 shows the performance improvement for different numbers of pivots when we ran 10-NN queries for the $60,000$ Corel data set. The first data point in the figure represents the standard version: its maximum queue size is $13,772$, its average queue size is $6,944$. It is shown that as the number of pivots increases, both the maximal queue size and the average queue size decrease. For instance, by keeping only 100 pivots, we can reduce the memory requirement by more than 66%. This improvement increases as we have more pivots. The extra gain for this data set is not significant, exhibiting a diminishing-returns phenomenon. As a result, it is evident that for this data set, an NNH of very small size can offer very large performance advantages.

Our algorithm presented in Section 4.2 will report 100 as the "optimal" number of pivots with $\epsilon = 0.1\%$ and $SP = 5$. This result is consistent with the intuition obtained by the observation of Figure 10. We also deployed the suitable queue-size-reduction ratio definition for k-NN semi-join, and obtained similar results.

6.2 Improving k-NN Joins

We implemented the k-NN semi-join algorithm in [5]. We performed a semi-join using two different subsets D_1 and D_2 (with the same size) of objects in the Corel data set, to find k-nearest neighbors for each object in D_1, allowing k to vary from 1 to 50. We constructed an NNH H_2 for D_2, with 100 pivots and $T = 50$. We used these NN vectors to get the k-NN distance estimates for all the objects in D_1. Then, we performed a traversal of the R-trees with the help of the priority queue. We used the approach described in Section 3.2 to prune the priority queue. During the traversal, we tried to perform a depth-first search on the R-tree of the first data set, so that pruning for the objects in the trees enabled by the histograms can be used as early as possible.

We compared the standard semi-join algorithm and the improved one with pruning obtained by NNH. The standard version required much more memory than the improved version. For instance, when we used $|D_1| = |D_2| = 10,000$ objects to do the semi-join, the algorithm required more than 2GB (virtual) memory, which caused the operating system to keep swapping data between memory and disk, yielding unacceptable performance for the standard version of the algorithm.

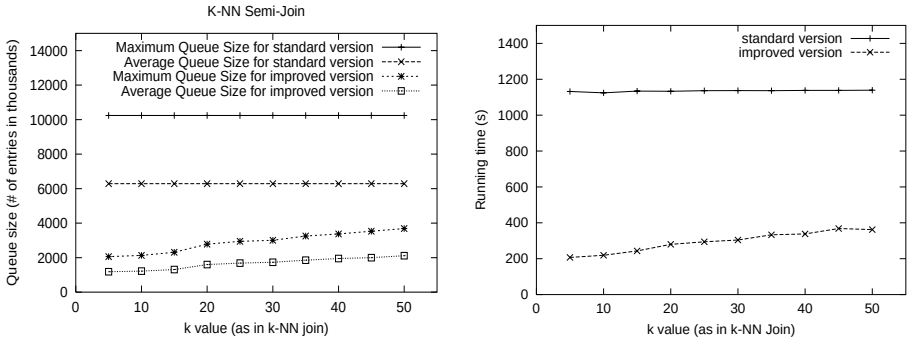
For this reason, we restrict the size of datasets to be under $8,000$ to make the standard version feasible on our test-bed. Each run required up to 1.2GB memory. We could not increase the size further since the performance of the standard algorithm deteriorated dramatically due to page thrashing. In contrast, by utilizing the histogram to perform more effective pruning, the improved algorithm required only 230MB of memory, which is around 20% of that of the standard version.

Reducing Memory Requirements and Running Time Similar to the case of single-object k-NN searches, we collected the maximal and average queue sizes for both the standard version and the improved one. Figure 11(a) presents the results. It is shown that additional pruning using the NNH histograms makes the semi-join algorithm much more efficient. For instance, when we performed a 10-NN semi-join search, the maximal and average queue size of the standard version were 10.2M (million) and 6.3M respectively, while it was only 2.1M and 1.2M for the improved version.

We also measured the running time for both versions. The results are shown in Figure 11(b). Clearly by using the improved version, we can reduce the running time dramatically. For instance, when performing a 10-NN semi-join, utilizing the NNH structure we can reduce the time from $1,132$ seconds to 219 seconds.

Figure 12(a) and (b) show the running times and queue sizes for different $|D_1| = |D_2|$ data sizes. The figures show that as the data size increases, the semi-join with pruning achieves significant performance advantages.

To evaluate the effect of different dimensionalities, we used the time-series dataset. We constructed each record with different window sizes from the dataset to achieve different dimensionalities. We extracted two data sets D_1 and D_2, and each contained 2000 records. Figure 13(a) shows that the NNH structure can



(a) Queue size. (b) Running time

Fig. 11. Semi-join performance improvement for different k values.

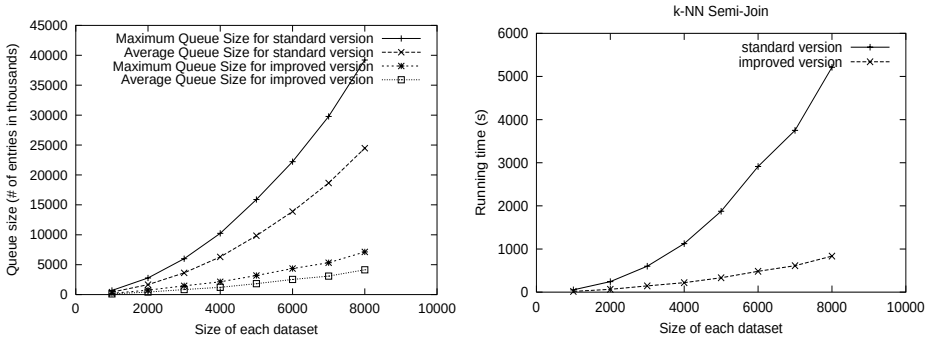
consistently help to reduce the maximal and average queue sizes. As shown in Figure 13(a), it appears that there is no strong correlation between the dimensionality and the effect of the NNH structure. The reason is that for the range of dimensions shown in the figure, the number of clusters in the underlying datasets remains stable.

Figure 13(b) shows the same trend for the running time for different dimensionalities. The running time of the improved version takes around 33% of the time of the standard version. The effectiveness of our NNH structure remains stable for different dimensionalities.

Effects of Varying Number of Pivots We evaluated the effect of different numbers of pivots in the k-NN semi-join. We performed a 10-NN semi-join on the two data sets D_1 and D_2 from the time-series dataset, each with $4,000$ objects. The dimensionality for both datasets is 30. Figure 14 reports the reduced maximum queue size and average queue size during the join process, when an NNH for dataset D_2 is used. Similarly to Section 6.1, the first data point represents the standard version at 6.7M and 4.3M, for the maximal and average queue size, respectively. This result demonstrates that increasing the number of pivots in an NNH can further reduce the memory requirement for the queue, since we have better estimates of the NN distances of the objects in D_1.

6.3 Costs and Benefits of NN Histograms

While it is desirable to have a large number of pivots in an NNH, we should also consider the costs associated with such a structure. In particular, we should pay attention to (1) the storage space; (2) the initial construction cost (off-line); and (3) the incremental-maintenance cost in the case of database updates (online).



(a) Queue size. (b) Running time

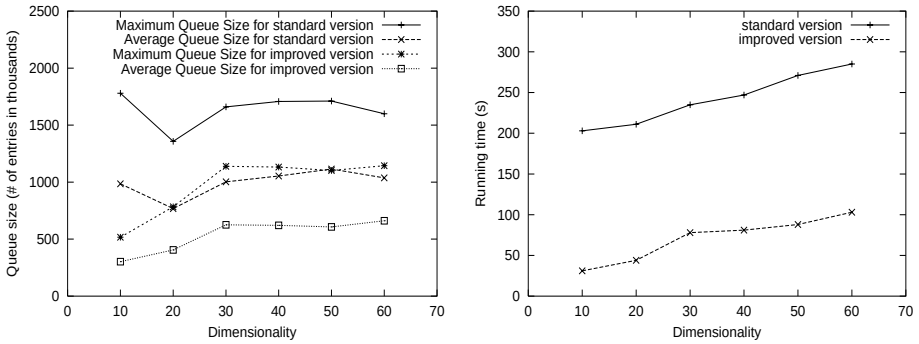
Fig. 12. Semi-join performance for different data sizes.

We measured these different costs, as well as the performance improvement for different numbers of pivots. We performed 10-NN queries and semi-joins on our 60,000-object Corel dataset with different numbers of pivots, and measured the costs. For each pivot, we maintain its 50 NN radii, thus, $T = 50$. Table 1 shows the time to construct the NNH, the required storage space for NNH, and the time for a single dynamic maintenance operation (insertion or deletion). Notice that entries for the maintenance time per each update are all zero, corresponding to times below the precision of our timing procedures. The two queue-size rows represent the ratio of the improved maximal queue sizes versus that of the standard version. For instance, when we had 100 pivots in the NNH, the maximal-queue-size-reduction ratio was 72% for a single 10-NN query. We do not include the numbers for the average-queue size since they follow the same trend as the maximal queue sizes.

Table 1. Costs and benefits of an NNH structure.

Number of pivots	10	50	100	150	200	250	300	350	400
Construction Time (sec)	2.1	10.2	19.8	28.2	34.1	40.8	48.6	53.4	60.2
Storage space (kB)	2	10	20	30	40	50	60	70	80
Time for dynamic maintenance (ms $\approx$)	0	0	0	0	0	0	0	0	0
Queue reduction ratio(k-NN) (%)	60	70	72	76	76	76	77	80	82
Queue reduction ratio (semi-join) (%)	70	72	74	77	77	78	79	79	80

From the table we can see that the size of the NNH is small, compared to the size of the data set, 28.5MB. The construction time is very short (e.g., less than 20 seconds for an NNH with 100 pivots). The incremental-maintenance time is almost negligible. The performance improvement is substantial: for instance, by



(a) Queue size. (b) Running time

Fig. 13. Semi-join performance for different dimensionalities.

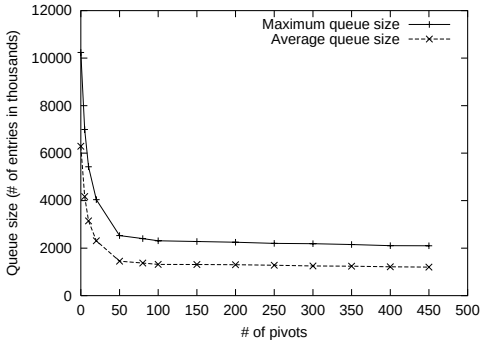


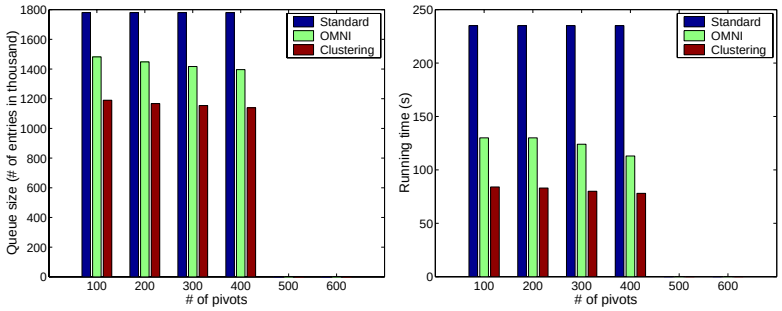
Fig. 14. Performance for different numbers of pivots in a semi-join.

keeping 100 pivots we can reduce the maximal queue size by 72% in a 10-NN search, and 74% in a semi join.

6.4 Comparison with Other Methods of Choosing Pivots

The final set of experiments we present compares our "clustering" approach of choosing pivots to ones existing in the literature [19–22]. Previous methods of utilizing the concept of pivots have been proposed in the context of search in metric spaces. These methods locate pivots that are far away from each other, and far away from other points in the dataset. Although such an approach makes sense in the context of the problem these works address, it is not suitable for the problem at hand. We chose the approach presented in [21] as a representative of these methods, denoted as "OMNI" in the experiments that follow. We compared this approach and our clustering approach of choosing pivots in NNH in terms of their effect of reducing the queue size and running time in an NN search.

Figures 15(a) and (b) show the semi-join queue size and running time for both approaches, respectively. In the experiments, both data sets consist of $2,000$ objects from the time-series dataset. In both figures, the tallest bar is the queue size and running time of the standard version without using NNH pruning, respectively. Apparently our clustering approach of choosing pivots is more effective in reducing the queue size and the running time than the OMNI approach.



(a) Queue-size reduction (b) Running-time reduction

Fig. 15. Comparison of different ways to choose pivots

This observation can be explained by examining the different ways the approaches utilize pivots to perform pruning of the respective search space. The approach in [21] essentially records all distances between all the points and the pivots. When a query arrives, it produces the candidate set by identifying the overlap of the "valid answer ring" of every pivot point. The main design goal is to minimize the overlap and thus minimize the candidate set. In this context, it is reasonable to choose the far away points/outliers as the pivots. However, our clustering approach relies on the local pivots to give a best possible estimation of the NN radius. The closer a point gets to the pivot, the better estimation it usually achieves. So by choosing far away pivots, the estimations are inevitably bad due to the large distance between points and pivots.

Summary: our intensive experiments have established that the improved pruning enabled via the use of an NNH structure can substantially improve the performance of algorithms involving k-NN searches, while the costs associated with such a structure are very low. These improvements increase as the size of the data sets increases, leading to more effective and scalable algorithms. It is evident that standard algorithms for k-NN search and join problems are significantly challenged when the size of the data sets increases. Their run-time memory requirements increase significantly and performance degrades rapidly. The improved pruning techniques via the use of NNH can alleviate these problems, reducing the run-time memory requirements (and subsequently processor requirements) significantly.

7 Conclusions

In this paper we proposed a novel technique that uses nearest-neighbor histogram structures to improve the performance of NN search algorithms. Such histogram structures can co-exist in conjunction with a plethora of NN search algorithms without the need to substantially modify them. The main idea is to preprocess the data set, and selectively obtain a set of pivot points. Using these points, the NNH is populated and then used to estimate the NN distances for each object, and make use of this information towards more effective searching.

We provided a complete specification of such histogram structures, showing how to efficiently and accurately construct them, how to incrementally maintain them under dynamic updates, and how to utilize them in conjunction with a variety of NN search algorithms to improve the performance of NN searches. Our intensive experiments showed that such a structure can be efficiently constructed and maintained, and when used in conjunction with a variety of NN-search algorithms, could offer substantial performance advantages.

References

1. Faloutsos, C., Ranganathan, M., Manolopoulos, I.: Fast Subsequence Matching in Time Series Databases. Proceedings of ACM SIGMOD (1994) 419–429
2. Salton, G., McGill, M.J.: Introduction to modern information retrieval. McGraw-Hill (1983)
3. Gersho, A., Gray, R.: Vector Quantization and Data Compression. Kluwer (1991)
4. Ferragina, P., Grossi, R.: The String B-Tree: A New Data Structure for String Search in External Memory and Its Applications. Journal of ACM 46,2, pages 237–280, Mar. 1999 (1999)
5. Hjaltason, G.R., Samet, H.: Incremental distance join algorithms for spatial databases. In: SIGMOD. (1998)
6. Shin, H., Moon, B., Lee, S.: Adaptive multi-stage distance join processing. In: SIGMOD. (2000)
7. Guttman, A.: R-trees: A dynamic index structure for spatial searching. In: Proceedings of ACM SIGMOD. (1984) 47–57
8. Hjaltason, G.R., Samet, H.: Ranking in spatial databases. In: Symposium on Large Spatial Databases. (1995) 83–95
9. Hjaltason, G.R., Samet, H.: Distance browsing in spatial databases. ACM Transactions on Database Systems **24** (1999) 265–318
10. Jagadish, H.V., Koudas, N., Muthukrishnan, S., Poosala, V., Sevcik, K.C., Suel, T.: Optimal Histograms with Quality Guarantees. Proceedings of VLDB (1998) 275–286
11. Kooi, R.P.: The Optimization of Queries in Relational Databases. PhD Thesis, CWRU (1980)
12. Mattias, Y., Vitter, J.S., Wang, M.: Dynamic Maintenance of Wavelet-Based Histograms. Proceedings of VLDB, Cairo, Egypt (2000) 101–111
13. Acharya, S., Gibbons, P., Poosala, V., Ramaswamy, S.: The Aqua Approximate Query Answering System. Proceedings of ACM SIGMOD, Philladephia PA (1999) 574–578

14. Vitter, J., Wang, M.: Approximate computation of multidimensional aggregates on sparse data using wavelets. Proceedings of SIGMOD (1999) 193–204
15. Preparata, F.P., Shamos, M.I.: Computational Geometry. Springer-Verlag, New York-Heidelberg-Berlin (1985)
16. Gaede, V., Gunther, O.: Multidimensional Access Methods. ACM Computing Surveys (1998)
17. Samet, H.: The Design and Analysis of Spatial Data Structures. Addison Wesley (1990)
18. Brinkhoff, T., Kriegel, H.P., Seeger, B.: Efficient Processing of Spatial Joins using R-trees. Proceedings of ACM SIGMOD (1993) 237–246
19. Brin, S.: Near neighbor search in large metric spaces. In: The VLDB Journal. (1995) 574–584
20. Bustos, B., Navarro, G., Ch'avez, E.: Pivot selection techniques for proximity searching in metric spaces. In: Proc. of the XXI Conference of the Chilean Computer Science Society (SCCC'01). (2001)
21. Filho, R.F.S., Traina, A.J.M., Jr., C.T., Faloutsos, C.: Similarity search without tears: The OMNI family of all-purpose access methods. In: ICDE. (2001) 623–630
22. Vleugels, J., Veltkamp, R.C.: Efficient image retrieval through vantage objects. In: Visual Information and Information Systems. (1999) 575–584
23. Tsaparas, P., Palpanas, T., Kotidis, Y., Koudas, N., Srivastava, D.: Ranked Join Indicies. ICDE (2003)
24. Weber, R., Schek, H., Blott, S.: A Quantitative Analysis and Performance Study for Similarity Search Methods in High Dimensional Spaces. VLDB (1998)
25. White, D.A., Jain, R.: Similarity indexing with the ss-tree. In: ICDE. (1996) 516–523
26. Katayama, N., Satoh, S.: The SR-tree: an index structure for high-dimensional nearest neighbor queries. In: Proceedings of ACM SIGMOD. (1997) 369–380
27. Roussopoulos, N., Kelley, S., Vincent, F.: Nearest neighbor queries. In: SIGMOD. (1995) 71–79
28. Berchtold, S., Böhm, C., Keim, D.A., Kriegel, H.P.: A cost model for nearest neighbor search in high-dimensional data space. In: PODS. (1997) 78–86
29. Guha, S., Rastogi, R., Shim, K.: CURE: An Efficient Clustering Algorithm for Large Databases. Proceedings of ACM SIGMOD (1998) 73–84
30. Ng, R.T., Han, J.: Efficient and effective clustering methods for spatial data mining. In: VLDB, Los Altos, USA, Morgan Kaufmann Publishers (1994) 144–155
31. Motwani, R., Raghavan, P.: Randomized Algorithms. Prentice Hall (1997)
32. Bishop, C.: Neural Networks for Pattern Recognizion. Oxford University Press (1996)
33. Yianilos: Data structures and algorithms for nearest neighbor search in general metric spaces. In: SODA: ACM-SIAM. (1993)
34. Chavez, E., Navarro, G., Baeza-Yates, R.A., Marroquin, J.L.: Searching in metric spaces. ACM Computing Surveys **33** (2001) 273–321
35. Standard Template Library: *http://www.sgi.com/tech/stl/ (2003)*