Efficient Evaluation of All-Nearest-Neighbor Queries

Yun Chen Jignesh M. Patel
University of Michigan
{yunc, jignesh}@eecs.umich.edu

Abstract

The All Nearest Neighbor (ANN) operation is a commonly used primitive for analyzing large multi-dimensional datasets. Since computing ANN is very expensive, in previous works R-tree based methods have been proposed to speed up this computation. These traditional index-based methods use a pruning metric called MAXMAXDIST, which allows the algorithms to prune out nodes in the index that need not be traversed during the ANN computation. In this paper we introduce a new pruning metric called the NXNDIST, and show that this metric is far more effective than the traditional MAXMAXDIST metric.*

In this paper, we also challenge the common practice of using R-tree index for speeding up the ANN computation. We propose an enhanced bucket quadtree index structure, called the MBRQT, and using extensive experimental evaluation show that the MBRQT index can significantly speed up the ANN computation.*

In addition, we also present the MBA algorithm based on a depth-first index traversal and bi-directional node expansion strategy. Furthermore, our method can be easily extended to efficiently answer the more general All-k-Nearest-Neighbor (AkNN) queries.

1 Introduction

The All Nearest Neighbor (ANN) operation takes as input two sets of multi-dimensional data points and computes for each point in the first set the nearest neighbor in the second set. The ANN operation has a number of applications in analyzing large multi-dimensional datasets. For example, clustering is commonly used to analyze large multi-dimensional datasets, and algorithms such as the popular single-linkage clustering method [15, 17] uses ANN as its first step. A related problem, called AkNN, which reports the kNN for each data point, is directly used in the Jarvis-Patrick Clustering algorithm [16]. AkNN is also used in a number of other clustering algorithms including the *k*-means and the *k*-medoid clustering algorithms [4].

The list of applications of ANN and AkNN is quite extensive and also includes co-location pattern mining [31], graph-based computational learning [18], pattern recognition and classification [22], N-body simulations in astrophysical studies [10], and particle physics [23].

ANN is a computationally expensive operation ($O(n^2)$ in the worst case). In many applications that use ANN, especially large scientific applications, the datasets are growing rapidly and often the ANN computation is one of the main computational bottlenecks. Recognizing this problem, there has been a lot of interest in the database community in developing efficient external ANN algorithms [4, 5, 9, 13, 32]. All of these methods build R*-tree indices [3] on one or both datasets, and evaluate the ANN by traversing the index. During the index traversal, these methods keep track of nodes in the index that need to be considered, and employ a priority queue (PQ) to determine the order of the index traversal. The efficiency of these algorithms heavily depends on how many PQ entries are created and processed. The most common and effective pruning method that has been developed so far employs a pruning metric called MAXMAXDIST, which is roughly the maximum distance between any points in two minimum bounding rectangles (MBR). In this paper we introduce a new distance metric, called the MINMAXMINDIST (abbreviated as NXNDIST), and show that this new metric has a much more powerful pruning effect. Using extensive experiments we show that *this new distance metric often improves the performance of ANN operation by over 10X.*

In this paper we also explore the properties of NXNDIST and develop a fast algorithm for computing this metric. This fast algorithm is critical since for ANN queries this distance computation is evaluated frequently.

Previous index-based ANN methods [4, 5, 9, 13, 32] have exclusively focused on the "ubiquitous" R*-tree index structure. In this paper, we show that for ANN queries there is a much better choice for an index structure, the MBRQT index. MBRQT is essentially a disk-based bucket PR quadtree [25], with the addition of the MBR information for internal nodes. Experiments show that *ANN evaluation using MBRQT is around 3X faster than using R*-tree.*

In addition, we also present the *MBRQT Based ANN (MBA)* algorithm that employs the depth-first traversal technique and bi-directional node expansion method for efficient ANN processing.

The extension from quadtree to *MBRQT* is simple and straightforward, so the MBA method can be used in cases where the database system chooses to support quadtrees (for example, Oracle has support for traditional quad-trees [19]), or in cases where ANN is run on datasets that do not have a prebuilt index (such as when running ANN as part of a complex query in which a selection predicate may have been applied on the base datasets).

Besides comparing our methods with previous index-based ANN methods, we also extensively compare with the GORDER [29] method that doesn't use an index to speed up the ANN computation. These comparisons show that our method significantly outperforms previous methods.

The remainder of this paper is organized as follows: Section 2 covers related work. Section 3 outlines our new ANN approach. Section 4 contains a comprehensive experimental evaluation of our new approach, and compares it with previous methods. Finally, Section 5 contains the conclusions.

2 Related Work

Closely related to ANN processing are Distance Join algorithms [13]. A Distance Join operation works on two sets of spatial data, and computes all object pairs, one from each set, such that the distance between the two objects is less than a non-negative value d. Distance semi-join [13] produces one result per entry of the outer relation, for which incremental algorithms are also developed. Shin et al. [26] introduce a more efficient algorithm later for a related problem of k-distance join, which uses a bi-directional expansion of entries in the PQ and a plane-sweep method.

The closest body of related work is the collection of previously proposed external memory ANN algorithms. A simple approach for computing ANN is to run a NN algorithm on the inner dataset **S** for each object in the outer dataset **R**. For this approach, optimization techniques have also been proposed to reduce CPU and I/O costs [6]. However, the assumption for such optimization is that the queries fit in main memory, which makes it inefficient when the size of **R** is larger than the main memory size.

Depending on whether **R** and/or **S** are indexed, existing techniques fall into two categories: traversal of R*-tree indices using a Distance Join algorithm [9, 13], and hash-based algorithms using spatial partitions [12]. The work in [32] spans both categories. Böhm and Krebs [5] also provide a solution to the more general problem of *Nearest Neighbor Join*: namely find for each object in **R**, its k nearest neighbors in **S**, which degenerates to ANN when $k = 1$. However, a specialized index structure termed *multipage index* is proposed for the solution provided, and thus the solution in [5] does not apply to general-purpose index structures such as R*-trees or quadtrees.

The more recent work on ANN by Zhang et al. [32] suggests two approaches to the ANN problem when the inner dataset **S** is indexed: *Multiple nearest neighbor search (MNN)*, and *Batched nearest neighbor search (BNN)*. MNN is essentially an index-nested-loops join operation, where the locality of objects is maximized to minimize I/O. However, the CPU cost is still high because of the large number of distance calculations for each NN search. To reduce the CPU cost, BNN splits the points in **R** into n disjoint groups, and traverses index **S** only n times, greatly reducing the number of distance calculations.

For the case where neither dataset has an index, Zhang et al. [32] also propose a hash-based method (HNN) using spatial hashing introduced in [24]. However, it was pointed out that in many cases building an index and running BNN is faster than HNN, and HNN is also susceptible to poor performance on skewed data distributions [32].

The recent GORDER [29] method employs a Principal Components Analysis (PCA) technique to transform the union space of the two input datasets to a single principal component space, and then sort the transformed points using a superimposed *Grid Order*. The transformed datasets, often more uniformly distributed, are written back to disk in sorted order. A Block Nested Loops join algorithm is then executed for solving the KNN join query.

The BNN and the GORDER approaches are currently regarded as highly efficient ANN methods. To the best of our knowledge, no previous work has compared these two methods directly. In this paper we make this comparison, and compare these two methods with our new techniques.

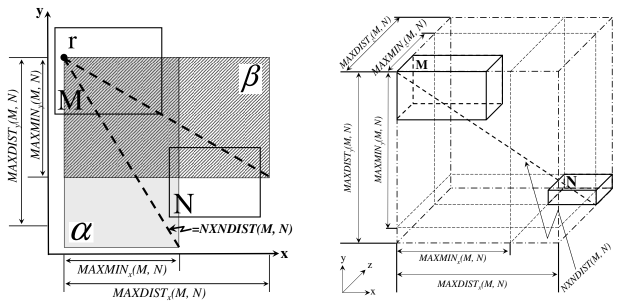
Interestingly, previous research on ANN and related join methods has not considered the use of disk-resident quadtree indices. As we show in this paper, the regular decomposition and non-overlapping properties of the quadtree make it a much more efficient indexing structure for ANN queries.

3 ANN Evaluation

In this section, we first introduce a new asymmetric distance metric, MINMAXMINDIST (abbreviated as NXNDIST), which has a higher pruning power for ANN computation compared to the traditional MAXMAXDIST metric. We also present an efficient algorithm for computing NXNDIST that has linear cost with respect to dimensionality. We then propose a new index structure called the Minimum Bounding Rectangle enhanced Quad-Tree (MBRQT). MBRQT has significant advantages over an R*-tree for ANN computation as it maximizes data locality and avoids the overlapping MBR issue inherent in an R*-tree

Table 1. Frequently Used Notations

Notation	Description
D	Dimensionality of data space
R	Query object dataset
S	Target object dataset
I_R	Index on dataset **R**
I_S	Index on dataset **S**
M	An MBR in index I_R
N	An MBR in index I_S
r	Point object in the dataset **R**
s	Point object in the dataset **S**



(a) 2-D NXNDIST (b) 3-D NXNDIST

Figure 1. NXNDIST Examples

index. Next we introduce the *MBA* algorithm together with the pruning heuristics that take advantage of the inherent properties of the NXNDIST metric for more effective pruning. Finally, we generalize our method to solve AkNN problems.

To facilitate our discussion, we will use the notations introduced in Table 1.

3.1 A New Pruning Distance Metric

As is common with current ANN algorithms, a certain distance metric is required as the upper bound for pruning entries from I_S that do not need to be explored. Traditionally the MAXMAXDIST metric has been used as such an upper bound [8, 9]. The MAXMAXDIST between two MBRs is defined as the maximum possible distance between any two points each falling within its own MBR [8,9]. We observe that the MAXMAXDIST metric is an overly conservative upper bound for ANN searches. We show that, for ANN queries a much tighter upper bound can be derived. This new upper bound guarantees the enclosure of the nearest neighbor within N for every point within M. We call this new metric the NXNDIST, and formally define it in the next section.

3.1.1 Definition and Properties of NXNDIST

For completeness and ease of comparison, first we provide brief descriptions of two related distance metrics on MBRs that have been previously defined [8]. These metrics are MINMINDIST and MINMAXDIST.

The MINMINDIST between two MBRs is the minimum possible distance between any point in the first MBR and any point in the second MBR. This metric has been extensively used in previously proposed ANN methods as the lower bound metric for pruning. We also employ this metric as a lower bound measure (NXNDIST, which we define in this section, is our upper bound metric).

The MINMAXDIST [8] between two MBRs is the upper bound of the distance between at least one pair of points, one from each of the two MBRs. We note that MINMAXDIST was proposed to address a different class of Distance Join operations (e.g. [8, 9]), and is not suitable as a pruning upper bound for ANN.

In the following discussion, we define the NXNDIST metric in arbitrary dimensions and explore its properties.

We represent a D-dimensional MBR with two vectors: a lower bound vector to record the lower bound in each of the D dimensions, and an upper bound vector to record the upper bound in each of the D dimensions. For example, MBR M is represented as $M(< l_1^M, l_2^M, ..., l_D^M >, < u_1^M, u_2^M, ..., u_D^M >)$. On the other hand, a p is represented as the vector $< p_1, p_2, ..., p_D >$.

We use $DIST(p, q)$ to denote the Euclidean distance between two points p and q, and denote the distance between p and q in dimension d as $DIST_d(p, q)$. We use $MAXDIST_d(M, N)$ to represent the maximum distance between any point within M and any point within N in dimension d.

***Definition* 3.1.** *Given two D-dimensional MBRs M and N, and an arbitrary point p in M,*
$MAXMIN_d(M, N) = \max_{\forall p \in M} (\min (|p_d - l_d^N|, |p_d - u_d^N|))$

The intuition of $MAXMIN_d(M, N)$ is "the maximum of the minimum distances in dimension d from any point within range $[l_d^M, u_d^M]$ to at least one end point l_d^N or u_d^N".

***Definition* 3.2.** $NXNDIST(M, N) =$

$$\sqrt{S - \max_{d=1}^{D} \begin{pmatrix} MAXDIST_d^2(M, N) \\ -MAXMIN_d^2(M, N) \end{pmatrix}}, \textit{ where}$$

$S = \sum_{d=1}^{D} MAXDIST_d^2(M, N)$.

Figure 1(a) shows an example of $NXNDIST(M, N)$ in 2-D space. Two MBRs M and N are shown, as well as an arbitrary point object $r \in M$. If an interval is constructed originating from r, with extent along the y axis equivalent to $MAXDIST_y(M, N)$ in either direction, then it is guaranteed to enclose N along the y axis. Sweeping this interval along the x axis with extent $MAXMIN_x(M, N)$, a rectangular search region is formed, which is the shaded region

α in the figure. As is shown in the figure, this rectangular search region is guaranteed to enclose at least one edge of N. Similarly, a second search region β, which is shown as the hatched rectangle, can also be formed by sweeping along the y axis. Of the two search regions α and β, the shorter diagonal length is equivalent to $NXNDIST(M, N)$.

To generalize to D dimensions, the sweeping interval is replaced by a (D-1) dimensional hyperplane, and there are a total of D different ways in which the sweeping can be performed. $NXNDIST(M, N)$ is then the minimum diagonal length among the D search regions. Figure 1(b) depicts a 3-D example of NXNDIST.

Figure 2(a) gives an illustration of two MBRs and various distance metrics between them.

It is worth mentioning that a similar metric called $minExistDNN$ was proposed in [30] for computing *Top-t Most Influential Spatial Sites*, which works the same way as NXNDIST in two dimensional cases. However, we note that the algorithm for computing the $minExistDNN$ is not scalable to dimensionality greater than 2, and thus is not applicable to multi-dimensional datasets.

Next, we prove the correctness of the NXNDIST metric as the upper bound for ANN search and reveal some of its useful properties.

***Lemma* 3.1.** *Given two MBRs, M and N, and a point object $r \in M$. Let $NN(r, N)$ denote r's nearest neighbor within N, then $DIST(r, NN(r, N)) \leq NXNDIST(M, N)$.*

Proof. From Definition 3.2, let i be the dimension in which

$$\begin{aligned} & MAXDIST_i^2(M,N) - MAXMIN_i^2(M,N) \\ = \; & \max_{d=1}^{D} \left(MAXDIST_d^2(M,N) - MAXMIN_d^2(M,N)\right) \end{aligned}$$

$NXNDIST(M, N)$ can then be expressed as:

$$\sqrt{\begin{array}{l} \sum_{d=1,\ldots,D}^{d \neq i} MAXDIST_d^2(M,N) \\ +MAXMIN_i^2(M,N) \end{array}} \qquad (1)$$

Let p be a point in M. From Definition 3.1, let q_i^N be the end point value of N in the i^{th} dimension such that:

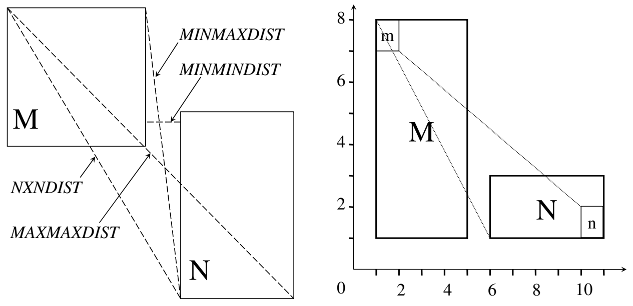
$$\begin{aligned} & \max_{\forall p \in M} |p_i - q_i^N| \\ = \; & \max_{\forall p \in M} \left(\min\left(|p_i - l_i^N|, |p_i - u_i^N|\right)\right) \end{aligned}$$

For N to be a MBR, there must exist in N a point object s such that $s_i = q_i^N$. The definition of nearest neighbor ensures the following:

$$DIST(r, NN(r, N)) \leq DIST(r, s) \qquad (2)$$

We observe the following from Definition 3.1:

$$\begin{aligned} DIST_i(r,s) \; &\leq \; MAXMIN_i(M,N) & (3) \\ \forall_{d=1}^{D} DIST_d(r,s) \; &\leq \; MAXDIST_d(M,N) & (4) \end{aligned}$$



(a) Metrics on MBRs (b) NXNDIST Properties

Figure 2. NXNDIST Properties

It follows from expression 1 and inequalities 3, 4 that

$$DIST(r, s) \leq NXNDIST(M, N) \qquad (5)$$

From inequalities 2 and 5 we obtain:

$$DIST(r, NN(r, N)) \leq NXNDIST(M, N) \; . \qquad \square$$

Lemma 3.1 establishes the foundation for the pruning heuristics presented in Sections 3.3.3 and 3.4.

***Lemma* 3.2.** *Let m be a child MBR of M, i.e., $m \subseteq M$ then $NXNDIST(m, N) \leq NXNDIST(M, N)$.*

Proof. Consider the following informal proof by contradiction: Suppose $NXNDIST(m, N) > NXNDIST(M, N)$. Then it follows that there exists some point $r \in m$ for which the following inequality holds:

$$DIST(r, NN(r, N)) > NXNDIST(M, N) \qquad (6)$$

Since $r \in M$, from Lemma 3.1, the following inequality holds:

$$DIST(r, NN(r, N)) \leq NXNDIST(M, N) \qquad (7)$$

This produces a contradiction to inequality (6). $\square$

Lemma 3.2 ensures the correctness of the traversal algorithms and pruning heuristics presented in Section 3.3.

***Lemma* 3.3.** *Let m be a child MBR of M, and let n be a child MBR of N, then $MINMINDIST(m, n)$ is not always smaller than $NXNDIST(M, N)$.*

Proof. Suppose that the following inequality always holds:

$$MINMINDIST(m, n) < NXNDIST(M, N) \qquad (8)$$

We construct a counter example in Figure 2(b) to contradict this claim. As shown in the figure, $m \subset M$ and $n \subset N$. Simple calculations show that $NXNDIST(M, N) = \sqrt{74}$, and $MINMINDIST(m, n) = \sqrt{89}$. This produces a contradiction to inequality 8. $\square$

Lemma 3.3 presents an important property of the NXNDIST that makes it a more efficient upper bound for pruning than the MAXMAXDIST metric.

We also note that NXNDIST is not commutable, i.e., $NXNDIST(M, N) \neq NXNDIST(M, N)$. We omit the proof here in the interest of space.

Algorithm 1: $NXNDIST(M, N)$

```
1  $MAXDIST[D] \Leftarrow [0], MAXMIN[D] \Leftarrow [0]$;
2  $S \Leftarrow 0, minS \Leftarrow 0$;
3  for $d = 1$ to $D$ do
4      $MAXDIST[d] \Leftarrow$
       $\max(|l_d^M - u_d^N|, |l_d^M - l_d^N|, |u_d^M - u_d^N|, |u_d^M - l_d^N|)$ ;
5      $S += MAXDIST[d]^2$;
6  $minS \Leftarrow S$;
7  for $d = 1$ to $D$ do
8      $MAXMIN[d] \Leftarrow MAXMIN(l_d^M, u_d^M, l_d^N, u_d^N)$;
9      $minS \Leftarrow$
       $\min(minS, S - MAXDIST[d]^2 + MAXMIN[d]^2)$;
10 return $\sqrt{minS}$;
```

3.1.2 Computing NXNDIST

Since NXNDIST is computed frequently during the evaluation of ANN, it is crucial to have an efficient algorithm for computing it. From Definition 3.2 we have developed an $O(D)$ algorithm for computing NXNDIST, which is shown in Algorithm 1.

Algorithm 1 proceeds in two iterations: the first iteration accumulates $S = \sum_{d=1}^{D} MAXDIST^2[d]$; the second iteration computes the $MAXMIN[d]$ value in each dimension d and obtains $NXNDIST(M, N)$. A 3-D example of Algorithm 1 is shown in Figure 1(b).

The MAXMIN procedure in Algorhtm 1 calculates the MAXMIN value in each dimension using Definition 3.1. It suffices to mention that the MAXMIN procedure takes constant computation time.

3.2 MBRQT

In a number of previous ANN works [8, 9, 13, 26, 32], the "ubiquitous" R*-tree index has been used. However it is natural to ask if other indexing structures have an advantage over the R*-tree for ANN processing. Notice that the R*-tree family of indices basically partition the underlying space based on the actual data distributions. Consequently, the partition boundaries for two R*-trees on two different datasets will be different. As a result when running ANN, the effectiveness of the pruning metrics such as NXNDIST will be reduced, as the pruning heuristic relies on this metric being smaller than some MINMINDIST. In contrast, an indexing method that imposes a regular partitioning of the underlying space is likely to be much more amenable to the pruning heuristic. A natural candidate for a regular decomposition method is the quadtree [25]. We do note that quadtrees are not a balanced data structure, but they can be mapped to disk resident structures quite effectively [11, 14], and some commercial DBMSs already support quadtrees [19]. The question that we raise, and answer, in this paper is how effective is a quadtree index compared to an R*-tree index for ANN processing.

Note that with a traditional quadtree, spatially neighboring nodes all border each other and the pairwise MINMINDIST value is zero. This may inevitably cause excessive computational and memory overhead due to large queue or stack size resulting from a low pruning rate. To mitigate this problem, we associate an explicit MBR with each internal node, which produces a tighter approximation of the entries below that node (at the cost of increasing storage). Essentially, we propose to enhance a regular PR bucket quadtree with MBRs. This enhanced indexing structure is called the MBR-quadtree, or simply MBRQT. As our experimental results show this index structure is significantly more effective than R*-trees for ANN processing.

3.3 ANN Algorithms

Before presenting the ANN algorithms, we briefly describe two data structures that are used in these algorithms.

3.3.1 Data Structures

The first data structure is the Local Priority Queue (LPQ). During the ANN procedure, each entry within I_R becomes the $owner$ of exactly one LPQ, in which a priority queue stores entries from I_S. Each entry e within the priority queue keeps a MIND and a MAXD field, accessible as e.MIND and e.MAXD. These fields indicate the lower and upper bound of the distance from the $owner$'s MBR to e's MBR. The priority queues inside the LPQs are ordered by the MIND field of the entries. In addition, each LPQ also keeps a MAXD field which records the minimum (for ANN) or maximum (for AkNN) of all e.MAXD values in the priority queue, as the upper bound for pruning un-wanted entries.

There are two advantages in using LPQ: (i) By requiring the $owner$ of the LPQs to be unique, we avoid duplicate node expansions from I_R (thus improving beyond the bitmap approach of [9, 13], since the bitmap approach only builds a bitmap for the point data objects within **R**, but not the intermediate node entries); (ii) LPQ gives us the advantages of the Three-Stage pruning heuristics, which we discuss in detail in Section 3.3.3.

The second data structure is simply a FIFO Queue, which serves as a container for the LPQs.

3.3.2 The MBA Algorithm

Based on how the index is traversed (depth-first or breadth-first) and intermediate nodes from I_R and I_S are expanded (bi-directional or uni-directional [26]), a choice of four ANN algorithms is available. Among these algorithms we choose the one with depth-first traversal and bi-directional

Algorithm 2: $MBA(I_R, I_S)$

```
1 $Q_{root} \Leftarrow New\ QUEUE()$;
2 $LPQ_{root} \Leftarrow New\ LPQ(I_R.root, \infty)$ ;
3 $Distances(LPQ_{root}.owner, I_S.root)$;
4 $LPQ_{root}.ENQUEUE(I_S.root)$;
5 $ExpandAndPrune(LPQ_{root}, Q_{root})$;
6 while $LPQ_{new} \Leftarrow Q_{root}.DEQUEUE()$ do
7     ANN-DFBI($LPQ_{new}$);
```

Algorithm 3: $ANN - DFBI(LPQ_{in})$

```
1 $Q_{out} \Leftarrow New\ QUEUE()$ ;
2 $ExpandAndPrune(LPQ_{in}, Q_{out})$;
3 while $LPQ_{child} \Leftarrow Q_{out}.DEQUEUE()$ do
4     ANN-DFBI($LPQ_{child}$);
```

node expansion (ANN-DFBI), which proves to outperform the others in extensive experiments. We omit the experimental details here in the interest of space.

Algorithm 2 shows the top level MBA algorithm, which simply expands the root nodes from both I_R and I_S and iteratively calls the ANN-DFBI routine.

The ANN-DFBI algorithm is shown in Algorithm 3. In this algorithm, index I_R is explored recursively in a depth-first fashion. As a result, the FIFO Queue at each level will only contain LPQs obtained by expanding both the *owner* entry of the higher level LPQ and the entries residing inside the priority queue contained within that LPQ, reducing memory consumption. In addition, bi-directional node expansion implies synchronous traversal of both indexes, data locality is also maximized, which improves I/O efficiency.

Note that the MBA is a general purpose algorithm and is also applicable to the R*-tree index structure, which we implement in the experiments and call it the RBA (R*-tree Based ANN) algorithm.

3.3.3 Pruning Heuristics

The basic heuristic for pruning is as follows: Let PM (MAXMAXDIST or NXNDIST) represent the chosen pruning metric between two MBRs M and N, if $MINMINDIST(M, N) > PM(M, N')$, for some N', then the path corresponding to (M, N) can be safely pruned.

The LPQ owned by each unique entry on I_R acts as the main filter, and enforces three stages of pruning: Expand Stage, Filter Stage, and Gather Stage, realized in the $ExpandAndPrune$ procedure presented in Algorithm 4.

The Expand Stage refers to the stage when internal nodes on I_R are expanded, and new lower level LPQs are created for and owned by child entries. This stage corresponds to lines $11 - 18$ in Algorithm 4. In this stage, the MAXD field from the input LPQ is passed on to the new LPQs as the

Algorithm 4: ExpandAndPrune(LPQ_{in}, Q_{out})

```
1 if $LPQ_{in}.owner$ is OBJECT then
2     while $n \Leftarrow LPQ_{in}.DEQUEUE()$ do
3         if $n$ is an OBJECT then
4             Return result $< LPQ_{in}.owner, n >$;
5         else
6             forall $e \in n$ do
7                 $Distances(LPQ_{in}.owner, e)$;
8                 if $e.MIND \leq LPQ_{in}.MAXD$ then
9                     $LPQ_{in}.ENQUEUE(e)$ ;
10 else
11     forall $c \in LPQ_{in}.owner$ do
12         $LPQ_c \Leftarrow new\ LPQ(c, LPQ_{in}.MAXD)$;
13     while $n \Leftarrow LPQ_{in}.DEQUEUE()$ do
14         forall $e \in n$ do
15             forall $LPQ_c$ do
16                 $Distances(LPQ_c.owner, e)$;
17                 if $e.MIND \leq LPQ_c.MAXD$ then
18                     $LPQ_c.ENQUEUE(e)$ ;
19     $Q_{out}.ENQUEUE$(all non-empty $LPQ_c$) ;
```

initial pruning upper bound. As entries from I_S are popped from the input LPQ, their MIND field is compared against the MAXD field of the new LPQs. If it's smaller, these entries are expanded; their child entries are probed against all the new LPQs, their MIND and MAXD values are computed against the owners of the new LPQs (this happens inside the $Distances$ function in Algorithm 4). These new expanded child entries are either discarded or queued by the new LPQs, and if queued, updating the LPQs' MAXD fields. In this stage NXNDIST has additional pruning advantages over MAXMAXDIST due to Lemma 3.3, namely early pruning becomes possible even when the MAXD field of the new LPQs has not yet been updated, which is not possible with MAXMAXDIST.

It is likely that during the Expand Stage, the MAXD of a new incoming entry may become smaller than the MIND of some entries that are already on the queue. This may lead to more nodes than necessary being expanded/explored in the next iteration and thus cause performance degradation. To mitigate this effect, we activate the Filter Stage which happens in the $ENQUEUE()$ function in Algorithm 4.

During the Filter Stage, as a new entry is being pushed into the priority queue inside a LPQ, its MAXD is compared against the MIND field of all the entries that it passes. Entries with a MIND greater than the MAXD of the new entry are immediately discarded. Ties on the MIND field are broken by comparing the MAXD fields of these two entries. In doing so, we are essentially optimizing the locality of pruning heuristics. Since NXNDIST is a much tighter metric, the Filter Stage has much stronger pruning power with NXNDIST than with MAXMAXDIST.

The Gather Stage corresponds to lines $2 - 9$ in Algorithm 4. This stage occurs when the $owner$ of the input LPQ is a data object, then as entries are popped out of the input LPQ, the first data object that occurs is the result for the $owner$ data object.

Note that the Three-Stage-Pruning strategy proposed here is a general-case optimization technique for ANN processing and can be easily adapted on any indices where the upper bound is non-increasing during the search.

3.4 Extension to AkNN

The extension of our methods to AkNN processing can be realized through slight modifications of Algorithm 4, using NXNDIST and the parameter k as the combined pruning criteria. In the interest of space we omit the details here, but give an intuition of the extension.

The intuition behind the extension of our method to compute AkNN is as follows: An entry e from I_S can only be pruned away when there are at least k entries in the LPQ and the MINMINDIST from the $owner$ MBR to that of e is greater than the MAXD field of the LPQ.

4 Experimental Evaluation

In this section, we present the results of our experimental evaluation. We compare our ANN methods with previous ANN algorthms. Of all the previously proposed ANN methods, the recent batch NN (BNN) [32] and GORDER [29] methods are considered to be the most efficient. Consequently, in our empirical evaluations, we only compare our methods with these two algorithms.

We note that BNN and GORDER haven't actually been compared to each other in previous work. A part of the contribution that we make via our experimental evaluation is to also evaluate the relative performance of these two methods.

4.1 Implementation Details

We have implemented a persistent MBRQT and an R*-tree on top of the SHORE storage manager [7]. We compiled the storage manager with 8KB page size, and set the buffer pool size to 64 pages ($512KB$). The purpose of having a relatively small buffer pool size is to keep the experiments manageable, which also essentially follows the experimental design philosophy used in previous research [20, 21, 27, 32].

We have also experimented with various buffer pool sizes, and the conclusions presented in this section also hold for these larger buffer pool sizes. In the interest of space, these additional experiments are suppressed in this presentation. One exception to this behavior, is the performance of GORDER, which is very sensitive to the buffer pool size for high-dimensional data. To quantify this effect, we present one experiment with varying buffer pool sizes (in Section 4.4).

Table 2. Experimental Datasets

Dataset	Cardinality	Description
500K2D	500K	2D point data
500K4D	500K	4D point data
500K6D	500K	6D point data
TAC	700K	2D Twin Astrographic Catalog Data
FC	580K	10D Forest Cover Type data

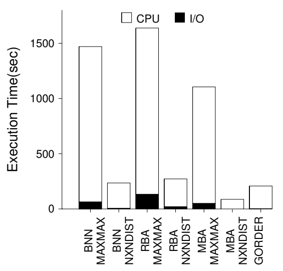
For the set of experiments that compare the MBRQT approach against previous methods, we take advantage of the original source code generously provided by the authors of [32] and [29]. For consistency, we modified the BNN implementation, switched the default page size from 4KB to 8KB, and retained the LRU cache size of 512KB. The parameters used for the GORDER methods are chosen using the suggested optimal values in the experimental section of [29], and K is set to 1 for all of the experiments comparing the ANN performance of these methods.

All experiments were run on a 1.2GHz Intel Pentium M processor, with 1GB of RAM, running Red Hat Linux Fedora Core 2. For each measurement that we report, we run the experiment five times and report the average of the middle three numbers.

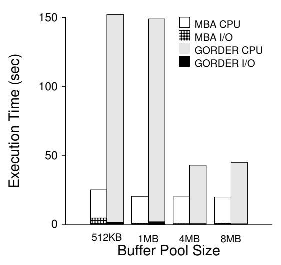
4.2 Experimental Datasets and Workload

We perform experiments on both real and synthetic datasets. Two real datasets are used: The Twin Astrographic Catalog dataset (TAC) from the U.S. Naval Observatory site [2], and the Forest Cover Type (FC) from the UCI KDD data repository [1]. The TAC data is a 2D point dataset containing high quality positions of around 700K stars. The Forest Cover dataset contains information about various 30 x 30 meter cells for the Rocky Mountain Region (US Forest Service Region 2). Each tuple in this dataset has 54 attributes, of which 10 attributes are real numbers. The ANN operation is run on these 10 attributes (following similar use of this dataset in previous ANN works, such as [29]).

We also modified the popular GSTD data generator [28] to produce multi-dimensional synthetic datasets. Although we experimented with various combinations of datasets with a wide range of sizes, in the interest of space, we only present selected results from a few representative workloads. The synthetic datasets that we use in this section are $500K$ point data. To test the effect of data dimensional-



(a) TAC Data(2D)



(b) FC Data(10D)

Figure 3. Comparison of Methods: Real Data

ity on the ANN methods, three datasets of cardinality 500K are generated, with dimensionality of 2, 4, and 6, respectively. Table 2 summarizes the datasets that we use in our experiments.

4.3 Effectiveness of the NXNDIST Metric

In this experiment, we evaluate the effectiveness of the NXNDIST metric and compare it with the traditional, looser pruning metric – MAXMAXDIST. For this experiment, we use the TAC dataset. Since BNN [32] is currently the most efficient R*-tree based ANN method, we compare both our MBA and RBA methods with BNN.

In Figure 3(a), results for BNN, MBA, and RBA approaches are shown, with both the MAXMAXDIST and the new NXNDIST pruning metric. (Similar results are also observed with the synthetic datasets, which we omit here in the interest of space.) Note that the original BNN algorithm of [32] corresponds to the bars labeled as "BNN MAXMAXDIST", and the BNN algorithm with NXNDIST corresponds to the bars labeled "BNN NXNDIST".

From Figure 3(a), we notice that for all three methods, BNN, MBA, and RBA, the use of NXNDIST metric dramatically improves the query performance. *Observe the order-of-magnitude improvement for the MBA method, and a 6X performance gain for both the BNN and RBA methods, by simply switching to the NXNDIST metric*.

The reasons for the drastic improvement of NXNDIST over MAXMAXDIST are as follows: (a) NXNDIST by itself is a much tighter upper bound than MAXMAXDIST, so the chances of the NXNDIST of a new entry being less than the MIND field of an existing entry in the queue become much higher. (b) As the search descends down the indices, the reduction in the length of NXNDIST is faster than that of MAXMAXDIST (see Lemma 3.3), resulting in better pruning as more un-wanted intermediate nodes are discarded – this drastically reduces the number of the next level nodes to examine. Also, the reduced effect of NXNDIST on BNN and RBA can be attributed to the MBR overlapping problem inherent with R*-trees (see Section 3.2).

4.4 Comparison of BNN, MBA, and GORDER

In Figure 3 we show the results comparing BNN, MBA, and GORDER using the two real datasets.

BNN v/s MBA: For this comparison, consider Figure 3(a). Comparing BNN and MBA in this figure, we observe that with the same pruning metric, *MBA is superior to the R*-tree BNN algorithm, both in CPU time and the I/O cost*. The superior performance of MBA over BNN is a result of the underlying MBRQT index, which has the advantages of the regular non-overlapping decomposition strategy employed by the quadtree (see Section 3.2 for details).

GORDER v/s BNN: From Figure 3(a) we observe that in general the *GORDER algorithm is superior to the BNN method*. There are two main reasons: (a) Both methods employ techniques to group the datasets to maximize locality. However, BNN does this only for **R**, while in GORDER the locality optimization is achieved by partitioning both input datasets and by using a transform to produce nearly uniform datasets. (b) In BNN, an R*-tree index is built for **S**. The inherent problem of overlapping MBRs in an R*-tree results in both higher I/O and CPU costs during the index traversal. In GORDER, however, the two datasets are disjointly partitioned, which leads to better CPU and I/O characteristics.

We also compared GORDER and BNN for the synthetic datasets, and found that GORDER was faster than BNN in all cases (these results have been suppressed in the interest of space). For the remainder of this section we only present results comparing our MBA method with GORDER.

GORDER v/s MBA: The results in Figure 3(a) show that *MBA outperforms GORDER by at least 2X* on the two-dimensional TAC dataset. The reasons for these performance gains are three-fold: (a) GORDER requires repeated retrievals of the dataset **S**, while MBA traverses the indices I_R and I_S simultaneously. This synchronized traversal of the indices results in better locality of access, which results in fewer buffer misses; (b) The pruning metric employed in GORDER is essentially MAXMAXDIST, which is less effective than NXNDIST (as discussed in Section 4.3); (c) With MBRQT, the pruning happens at multiple levels of the index structure, where early internal node level pruning saves a significant amount of computation. GORDER, on the other hand, is essentially a block nested-loops join, with the pruning happening only on the block and object levels, and thus incurs significantly more computation.

The *performance advantages of MBA over GORDER continue for higher dimensional datasets*. Figure 3(b) shows the execution time for these two algorithms on the 10-dimensional FC dataset. We also use this experiment to illustrate the effect of buffer pool size on the GORDER method when using high-dimensional datasets[1]. To quan-

[1]We note that the performance of GORDER is sensitive to the buffer pool size only for high-dimensional datasets. For low-dimensional datasets

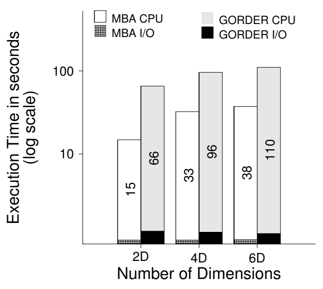


Figure 4. Effect of D

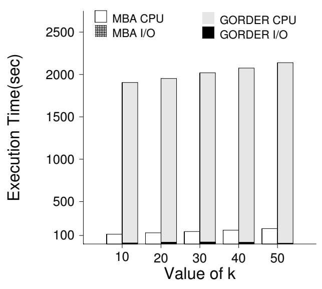


Figure 5. AkNN on TAC Data

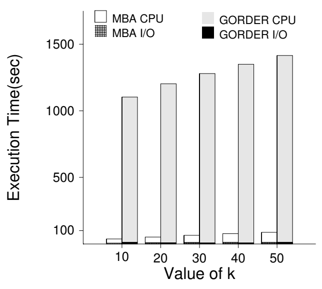


Figure 6. AkNN on FC Data

tify this effect, for this experiment, we vary the buffer pool size from 512KB to 8MB.

The first observation to make in Figure 3(b) is the performance of GORDER improves rapidly as the buffer pool size increases from 1MB to 4MB, and stabilizes after the 4MB point. The reason for this behavior of GORDER is as follows: GORDER executes a block nested loops join and is joining a single block of the outer relation R with a number of blocks of the inner relation S. Before executing an in-memory join of the data in "matching" R and S blocks, GORDER uses a distance-based pruning criteria to safely discard pairs of blocks that are guaranteed to not produce any matches. This pruning is more effective when there are larger number of S blocks to examine, which happens naturally at larger buffer pool sizes. Since the pruning criteria is influenced by the number of neighbors of a grid cell (which grows rapidly as the dimensionality increases), the impact of the smaller buffer pool size is more pronounced at higher dimensions. On the other hand, as discussed in Section 3.3.2, the MBA algorithm only keeps a small number of candidate entries from I_S inside the LPQ for each $_R$ entry. Spatial locality is thus preserved and the performance is not significantly affected by the size of the buffer pool.

The second observation to make in Figure 3(b) is that MBA is consistently faster than GORDER for all buffer pool sizes. For larger buffer pool sizes *MBA is 2X faster*, and for smaller buffer pool sizes it is *6X faster*.

4.5 Effect of Dimensionality

For this experiment, we generated a number of synthetic datasets, with varying cardinalities and dimensionalities. In the interest of space we show in Figure 4 results for a representative workload, namely the 500K2D, 500K4D, and 500K6D datasets. (The numbers in the bars in this graph show the actual CPU costs in seconds.)

As is shown in the figure, *MBA consistently outperforms GORDER by approximately 3X for all 2D, 4D, and 6D datasets*. As the dimensionality of the data increases, the CPU time for both methods increases very gradually, and the I/O time also elegantly scales up. This observation is consistent with both the TAC and FC datasets in Figure 3.

As we have noted previously, ANN is a very computationally intensive operation, with most of the execution time spent on distance computation and comparisons. Thus, having an efficient distance computation algorithm for high-dimensional data is crucial to the performance of ANN methods. Examining the CPU time for MBA (which uses the NXNDIST metric) in Figure 4, we observe that the CPU cost is not increasing sharply as the dimensionality increases, which shows the effectiveness of the $O(D)$ NXNDIST algorithm (Algorithm 1 presented in Section 3.1.2).

4.6 Evaluating AkNN Performace

We use both real-world datasets, TAC and FC, for the experiment comparing AkNN performance of MBA against GORDER. We follow the example in [29] and vary k value from 10 to 50, with increment of 10. Figures 5 and 6 show the results of this experiment.

As can be seen in these figures, on both the TAC and FC datasets, the execution time of MBA and GORDER increases as the k value goes up. However, *MBA is over an order of magnitude faster than GORDER in all cases*. The reasons for this performance advantage for MBA over GORDER are similar to those described in Section 4.4.

5 Conclusions

In this paper we have presented a new metric, called NXNDIST, and have shown that this metric is much more effective for pruning ANN computation than previously proposed methods. We have also explored the properties of this metric, and have presented an efficient $O(D)$ algorithm for computing this metric, where D is the data dimensionality. In addition, we have presented the MBA algorithm that traverses the index trees in a depth-first fashion and expands the candidate search nodes bi-directionally. With the application of NXNDIST, we have also shown how to extend our

the buffer pool effects are very small. For example, with the TAC data changing the buffer pool size from 512KB to 8MB only improved the performance of GORDER by 5%.

solution to efficiently answer the more general AkNN question.

Finally, we have shown that for ANN queries, using a quadtree index enhanced with MBR keys for the internal nodes, is a much more efficient indexing structure than the commonly used R*-tree index. Overall the methods that we have presented generally result in significant speed-up of at least 2X for ANN computation, and over an order of magnitude for AkNN computation over the previous best algorithms (BNN [32] and GORDER [29]), for both low and high-dimensional datasets.

6 Acknowledgments

This research was supported by the National Science Foundation under grant IIS-0414510, and by the Department of Homeland Security under grant W911NF-05-1-0415.

References

[1] The UCI Knowledge Discovery in Databases Archive. Downloadable from http://kdd.ics.uci.edu/.

[2] Twin Astrographic Catalog Version 2 (TAC 2.0), 1999. Downloadable from http://ad.usno.navy.mil/tac/.

[3] 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*, pages 322–331, 1990.

[4] C. Böhm and F. Krebs. Supporting KDD Applications by the k-Nearest Neighbor Join. In *DEXA*, 2003.

[5] C. Böhm and F. Krebs. The k-Nearest Neighbor Join: Turbo Charging the KDD Process. *KAIS*, 6(6), 2004.

[6] B. Braunmüller, M. Ester, H.-P. Kriegel, and J. Sander. Efficiently Supporting Multiple Similarity Queries for Mining in Metric Databases. In *ICDE*, 2000.

[7] M. Carey and et al. Shoring Up Persistent Applications. In *SIGMOD*, pages 383–394, 1994.

[8] A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos. Closest Pair Queries in Spatial Databases. In *SIGMOD*, pages 189–200, 2000.

[9] A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos. Algorithms for Processing K-closest-pair queries in spatial databases. *TKDE*, 49(1):67–104, 2004.

[10] D. J. Eisenstein and P. Hut. Hop: A new group-finding algorithm for n-body simulations. *The Astrophysical Journal*, 498:137–142, 1998.

[11] I. Gargantini. An Effective Way to Represent Quadtrees. *Commun. ACM*, 25(12):905–910, 1982.

[12] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-Memory Computational Geometry. In *Proceedings of the 34th Annual Symposium on Foundations of Computer Science*, pages 714–723, 1993.

[13] G. R. Hjaltason and H. Samet. Incremental Distance Join Algorithms for Spatial Databases. In *SIGMOD*, 1998.

[14] G. R. Hjaltason and H. Samet. Speeding up Construction of PMR Quadtree-based Spatial Indexes. *VLDB Journal*, 11(2):109–137, 2002.

[15] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: A review. *ACM Computing Surveys*, 31(3):264–323, 1999.

[16] R. Jarvis and E. Patrick. Clustering using a similarity measure based on shared near neighbors. 22:1025–1034, 1973.

[17] S. C. Johnson. Hierarchical clustering schemes. *Psychometrika*, 2:241–254, 1967.

[18] S. Koenig and Y. Smirnov. Graph learning with a nearest neighbor approach. In *Proceedings of the Conference on Computational Learning Theory*, pages 19–28, 1996.

[19] R. K. V. Kothuri, S. Ravada, and D. Abugov. Quadtree and R-tree Indexes in Oracle Spatial: A Comparison Using GIS Data. In *SIGMOD*, pages 546–557, 2002.

[20] S. Leutenegger and M. Lopez. The Effect of Buffering on the Performance of R-trees. In *IEEE TKDE*, pages 33–44, 2000.

[21] S. Šaltenis, C. Jensen, S. Leutenegger, and M. Lopez. Indexing the Positions of Continuously Moving Objects. In *SIGMOD*, pages 331–342, 2000.

[22] R. Nock, M. Sebban, and D. Bernard. A simple locally adaptive nearest neighbor rule with application to pollution forecasting. *Internal Journal of Pattern Recognition and Artificial Intelligence*, 17(8):1–14, 2003.

[23] M. Pallavicini, C. Patrignani, M. Pontil, and A. Verri. The nearest-neighbor technique for particle identification. *Nucl. Instr. and Meth.*, 405:133–138, 1998.

[24] J. M. Patel and D. J. DeWitt. Partition Based Spatial-merge Join. In *SIGMOD*, pages 259–270, 1996.

[25] H. Samet. The Quadtree and Related Hierarchical Data Structures. *ACM Computing Surveys*, 16(2):187–260, 1984.

[26] H. Shin, B. Moon, and S. Lee. Adaptive Multi-Stage Distance Join Processing. In *SIGMOD*, pages 343–354, 2000.

[27] Y. Tao, D. Papadias, and J. Sun. The TPR*-Tree: An Optimized Spatio-Temporal Access Method for Predictive Queries. In *VLDB*, pages 790–801, 2003.

[28] Y. Theodoridis, J. R. O. Silva, and M. A. Nascimento. On the Generation of Spatiotemporal Datasets. *Lecture Notes in Computer Science*, 1651:147–164, 1999.

[29] C. Xia, H. Lu, B. C. Ooi, and J. Hu. GORDER: An Efficient Method for KNN Join Processing. In *VLDB*, pages 756–767, 2004.

[30] T. Xia, D. Zhang, E. Kanoulas, and Y. Du. On computing top-t most influential spatial sites. In *VLDB*, pages 946–957, 2005.

[31] J. S. Yoo, S. Shekhar, and M. Celik. A join-less approach for co-location pattern mining: A summary of results. In *IEEE International Conference on Data Mining(ICDM)*, 2005.

[32] J. Zhang, N. Mamoulis, D. Papadias, and Y. Tao. All-Nearest-Neighbors Queries in Spatial Databases. In *SSDBM*, 2004.