An index structure for improving closest pairs and related join queries in spatial databases

Congjun Yang
Division of Computer Science
Department of Mathematical Sciences
The University of Memphis
Memphis, TN 38152, USA.
yangc@msci.memphis.edu

King-Ip Lin
Division of Computer Science
Department of Mathematical Sciences
The University of Memphis
Memphis, TN 38152, USA.
linki@msci.memphis.edu

Abstract

Spatial databases have grown in importance in various fields. Together with them come various types of queries that need to be answered effectively. While queries involving single data set have been studied extensively, join queries on multi-dimensional data like the k-closest pairs and the nearest neighbor joins have only recently received attention.

In this paper, we propose a new index structure, the b-Rdnn tree, to solve different join queries. The structure is similar to the Rdnn-tree for the reverse nearest neighbor queries. Based on this new index structure, we give the algorithms for various join queries in spatial databases. It is especially effective for the k-closest pair queries, where earlier algorithms using R-tree can be very inefficient in many real life circumstances. To this end we present experimental results on k-closest pair queries to support the fact that our index structure is a better alternative.*

1 Introduction

Applications from navigation to mobile networks to multimedia require handling of spatial and multidimensional data. It is paramount for database systems to be able to efficiently answer queries about these data, especially in a dynamic environment.

Many queries on these kind of data are on a single table. These include point location, range, nearest neighbor, and reverse nearest neighbor queries [7]. However, another type of queries involves relating data from multiple tables – a 'join' query in database terms. These queries look for pairs of tuples in different tables that satisfy various conditions. A naive join algorithm that examines all possible pairs is prohibitively expensive. Thus, we need to devise more effective index and algorithm for these queries.

We illustrate various join queries by the following example. Suppose we have a table of residential houses in a certain state, with location information stored as x,y-coordinates. Also, we have a separate table of factories or other sources of pollution, again with location information. Various groups of people, from homeowners to environment workers, need to determine the level of pollution the factories are creating to the residents and determine what actions are needed. We can ask various queries:

1. *(Distance-based) Spatial joins*: for each house, list all factories within 100 miles. If the effect of pollution is negligible beyond 100 miles, potential homeowners can use this to determine the suitability of a house.

2. *k Closest nearest neighbor pairs*: list the top k houses, together with the corresponding closest factory to each house, in the order of closeness to the factory. This can be another metric to measure the effect of pollution, especially if we assume the effect of pollution is non-additive – i.e. the effect of pollution is primarily due to the factory closest to the house. If k equals the total number of houses, this query is also called *all pair nearest neighbors* .

3. *k Closest pairs*: find the top k factory-house pairs ordered by the closeness to one another. This gives us a measure of the seriousness of the effect of individual factory on individual household, and can give environmental workers a priority on which pair to tackle first.

All the above queries require some kind of join. Hence, it is imperative that such queries be performed effectively. As mentioned, the naive algorithm requires examining every pair of objects, which is impractical for large tables. Thus, one would like to use various techniques – such as indexes – to speed up the queries considerably.

Distance-based spatial join has been studied extensively. Many existing join algorithms are based on the R-trees [2, 3], the Seed-trees [9]; or the Breadth-First approach [6]. Other spatial join techniques exist, such as spatial merge-join [11], spatial hash-join [10], size-separation spatial join [8], and scalable sweeping-based spatial join [1]. However, closest pair related join problems have only recently been in the spotlight. For instance, Hjaltason and Samet [5] as well as Corral et al [4] propose various algorithms to solve the k-Closest pair problem. They assume that each data set is indexed using an R^*-tree (or similar index structure). Their methods traverse the indexes to find the closest pairs. This works well in cases where the two data sets do not 'overlap' – i.e. the two data sets reside in completely disjoint regions of a multidimensional space. However, in many real life applications, this assumption is invalid. For instance, in the example above, it is more likely that the houses and the factories are in the same geographic proximity. In such cases, their methods perform poorly. We believe that there is room for significant improvement.

In this paper, we propose a new index structure, the bichromatic Rdnn-Tree (bRdnn-Tree), to solve the closest pair join problem. The structure uses information about nearest neighbors to help prune the search path more effectively. Moreover, the index structure is also very efficient on various types of join queries, such as spatial join, closest nearest neighbor pairs, and all pairs nearest neighbor.

The rest of the paper is organized as follows: Section 2 defines the various problems mentioned above, discusses previous solutions to the closest pair problems, and outlines the potential for improvement. The bRdnn-Tree is presented in Section 3. Section 4 provides experimental results. Section 5 discusses future directions.

2 Problem Definitions and Existing Algorithms

In what follows, we assume that S, T are sets of points in d dimensional space. $D(p, q)$ is the distance between two points p and q. R_S denotes the R-tree containing data set S. For an R-tree, we use N_S to denote a node of the tree containing data set S.

2.1 Problem Definitions

Here we give the formal definitions of the problems we handle in this paper. Our focus is the nearest neighbor related problems. Given two data sets S and T and a query point in one data set S, one can search the nearest neighbor of a point in $p \in S$ in the other data set T. Let us call it the bichromatic nearest neighbor search. Formally, we have the following:

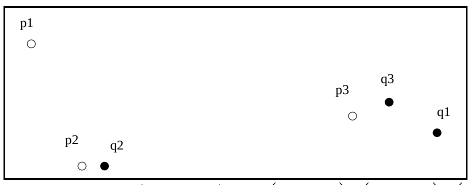
DEFINITION: (Bichromatic Nearest Neighbor Query) Given two data sets S and T of points in some d dimensional space and a query point $q \in S$, the bichromatic nearest neighbor query is to find a point $p \in T$ such that $D(q, x) \geq D(q, p) \quad \forall x \in T$

In the above definition, we call (q, p) a nearest neighbor pair (NN pair) with respect to q. The problem of finding k-closest such pairs with respect to k points in one data set is hence called the k Closest NN pairs.

DEFINITION: (k Closest NN Pairs Query) Given two data sets S and T of points in d dimensional space, the k-closest NN pair query (with respect to S) is to find k points $s_1, s_2, \cdots, s_k \in S$ and the nearest neighbor $t_i \in T$ for each s_i such that $\forall s \in S \setminus \{s_1, \cdots, s_k\}$ and its nearest neighbor t in T we have $D(s, t) \geq D(s_i, t_i) \quad \forall i \in \{1, 2, \cdots, k\}$

In other words, the k-closest NN pair problem with respect to a data set is to find k points in the data set that have smaller nearest neighbor distances than any other points in the same data set. If k is the same as the size of the data set, the problem can also be viewed as the bichromatic version of the all pair nearest neighbor problem. Another common join query is called the k-closest pair problem, which we formally give the definition as follows:

DEFINITION: (k-Closest Pair Join) Given two data sets S and T of points in some d dimensional space, the *k-Closest Pairs (k-CPs)* of S and T is a collection of k ordered pairs $KCP = \{(s_1, t_1), (s_2, t_2), \cdots, (s_k, t_k)\}$ where $s_i \in S$ and $t_i \in T \; \forall i \in \{1, 2, \cdots, k\}$, such that for any $(s, t) \in S \times T - KCP$ we have $D(s, t) \geq D(s_i, t_i) \quad \forall i \in \{1, 2, \cdots, k\}$



3 Closest NN Pair (w.r.t. P): $(p_2, q_2), (p_3, q_3), (p_1, q_2)$
3 Closest Pairs : $(p_2, q_2), (p_3, q_3), (p_3, q_1)$

Figure 1. Example: k Closest Pair vs. k Closest NN Pair

Figure 1 shows that the k-closest NN pairs is not necessarily the k closest pairs and vice versa. The main difference is that the closest NN pairs is always with respect to one of the data set invloved. For that set, each object can only appear in the result once. For closest pairs queries, there is no such restriction. The difference in applications has been outlined in section 1. However, the two types of queries are closely related to each other, as shown by the following theorem (it will be used in section 3):

THEOREM 2.1 *Given two data sets P and Q of points from some d dimensional space, assume $\{(p_1, q_1), (p_2, q_2), \cdots, (p_k, q_k)\}$ is the k-closest NN pairs with respect to P (or Q) $\forall i \in \{1, 2, \cdots, k\}$, and (p, q) is one of the k Closest Pairs. Then we have*

$$p \in \{p_1, p_2, \cdots, p_k\} \qquad (\textit{or } q \in \{q_1, \cdots, q_k\})$$

Proof: Without loss of generality, we assume that $D(p_1, q_1) \leq D(p_2, q_2) \cdots \leq D(p_k, q_k)$ since (p, q) is one of the k-closest pairs in $P \times Q$, it is clear that $D(p, q) \leq D(p_k, q_k)$. Suppose $p \notin \{p_1, p_2, \cdots, p_k\}$, then (p, q) is not an NN pair with respect to p. In other words, q is not the nearest neighbor of p in Q. Assume q' is the nearest neighbor of p in Q, then we have $D(p, q') \leq D(p, q) \leq D(p_k, q_k)$ and hence (p, q') is one of the k-closest NN pairs with respect to p. Therefore, $p \in \{p_1, p_2, \cdots, p_k\}$, a contradiction. This proves that $p \in \{p_1, p_2, \cdots, p_k\}$. Similarly, we can show that $q \in \{q_1, q_2, \cdots, q_k\}$. *QED*

2.2 Existing Algorithms

The significant work in this field has been described in two separate papers. Hjaltason and Samet [5] introduced several incremental distance join algorithms, while Corral et al. [4] introduced various algorithms based on the R-tree family. Due to space limitation we focus on the methods by Corral. The interested reader is directed to [4] for a comparison between the two algorithms.

In [4], it is assumed that for each data set there is an R-tree (or one of its variants) constructed, and the indexes are used to avoid the naive nested-loop join. The basic algorithm traverses the two trees together using a branch-and-bound approach. The search starts at the root of the two trees, and it keeps track of the current best solution (which is ∞ initially). At any stage, if a pair of internal nodes are retrieved, the algorithm examines the bounding rectangles of all pairs of branches (one from each tree) and decides which pairs need to be traversed and which pairs can be pruned; when a pair of leaves is reached, all data in the nodes are examined to update the current solution. This continues until all possible pairs are either traversed or pruned. Various measures are used to decide which pairs to prune: MINMINDIST (lower bound of minimum distance between of the bounding rectangles), MINMAXDIST (upper bound of minimum distance), and MAXMAXDIST (upper bound of maximum distance). These provide bounds for the distances between any pair of objects that are descendents of the nodes. Also, various order of traversal like the Depth-First and the Best-Firest are explored. Furthermore, the number of available buffers affects the performance of algorithms: small buffer size favors using a priority queue while depth-first traversal benefits from a larger number of buffers.

However, an even more important factor of the performance of the join algorithm, not mentioned in previous papers, is the amount of "overlap" of the data in the two sets. This can be defined as the overlapping area of the bounding rectangles of the two data sets S and T. For example, with their proposed technique, we run experiments joining two 2D data sets with 80,000 data points each. We build an R-tree (with 2K page size) for each data set and apply the join algorithms. Table 1 shows the result:

Overlap %	0%		100%	
k	1	12,500	1	12,500
Pairs compared, DFS	1	21.6	5074.4	5504
Pairs compared, PQ	1	21.6	5073	5312.8
Page faults, DFS	2	14.8	2570.4	2592
Page faults, PQ	2	14.8	3164.2	3320.8

DFS : depth-first search; PQ : Priority queue (160 buffers)
Average number of leaf nodes per tree : 1080

Table 1. Comparison of join performance with different overlap

In table 1, the "pairs compared" corresponds to the total number of pairs of leaf nodes compared. This measures the performance when there is no buffer available. The "page faults" measures the actual cost with LRU buffering. The table shows that the performance of the algorithms vary significantly with respect to the difference in overlap, regardless of whether buffering is available. When the overlap is 100%, practically both trees have to be completely traversed 2-3 times (depending on k and the traversal method). The poor performance is due to the inability of the join algorithm to prune nodes. For example, when the two dataset has 100% overlap, for each node N_1 (with bounding rectangle R_1) in the R^*-tree corresponding to one data set, it is highly likely that there is at least another node N'_1 (with bounding rectangle S_1) in the other R^*-tree such that R_1 and S_1 intersects. In fact, one can expect R_1 to intersect with bounding rectangles of quite a few nodes in the other tree. However, the join algorithms have to examine all pairs of overlapping nodes. This leads to the poor performance of the algorithm. The same effect occurs in the incremental algorithms by Hjaltason and Samet, as their algorithms tranverse the tree based on increasing distance, thus all the overlapping pairs must be traversed.

The above discussion suggests that a new indexing scheme for join queries over multiple data sets is needed.

3 The Index Structure and Algorithms

3.1 Proposed Index Structure: bichromatic Rdnn-Tree (bRdnn-Tree)

As Theorem 2.1 shows, the k-closest pair and the k-closest NN pair problems are closely related. Thus, if we can solve the k-closest NN pair problem effectively, we can use it to find the k-closest pairs. One way to solve the k-closest NN pair problem is to try to pre-compute and store the nearest neighbor information in an index. We propose a new index structure, the bichromatic Rdnn-Tree (bRdnn-Tree), that dynamically maintains the nearest neighbor pair information. The structure is similar to the Rdnn-tree for the Reverse Nearest Neighbor problem [12]. Recall that the reverse nearest neighbor (RNN) of a point p in a data set S is a collection of points in S that have p as their nearest neighbor. In this case, as we are given two data sets, we construct two trees, one for each data set. In what follows, we denote them as the Red tree and the Blue tree. For a point p in R, $NN_S(p)$ denotes its nearest neighbors in data set S, and $RNN_S(p)$ its reverse nearest neighbors in S.

In each tree, a leaf node contains entries of the form (pt, dnn), where pt refers to a d-dimensional point in the data set and dnn is the distance between the NN pair with respect to pt. In other words, the dnn of a leaf entry in the Blue tree is the distance from a blue point pt to its nearest neighbor in the Red tree. Formally, for a blue point b we have $dnn = dnn_R(b) = \min_{r \in R} D(b, r)$. The dnn can be defined similarly for any red points.

A non-leaf node contains an array of branches of the form $(ptr, Rect, max\_dnn, min\_dnn)$; ptr is the address of a child node in the tree. If ptr points to a leaf node, $Rect$ is the minimum bounding rectangle of all points in the leaf node. If ptr points to a non-leaf node, $Rect$ is the minimum bounding rectangle of all rectangles that are entries in the child node; max_dnn (min_dnn) is the maximum (minimum) distance of each point to its nearest neighbor in the other data set. More specifically, for a node N of the Red tree containing data set R we have $max\_dnn = \max_{p \in N}\{dnn_B(p)\}; (min\_dnn = \min_{p \in N}\{dnn_B(p)\})$. Similarly, we can define max_dnn (min_dnn) for any node in the Blue tree.

3.2 Algorithms

Insertion and Deletion When a point p' is to be inserted into the Red tree, we first perform an NN and a RNN search on the Blue tree to find the nearest neighbor ($NN_B(p')$) and the reverse nearest neighbor of p' ($RNN_B(p')$) respectively. With $NN_B(p')$, we can compute $dnn(p')$ to create the entry for p'. It is easy to see that $RNN_b(p')$ are the points that are affected as they are the points in the Blue tree that have the new red point p' as their nearest neighbor. Typically, their dnn field and hence the max_dnn and min_dnn in their parent nodes need to be updated. Therefore, we have a pre-insertion phase to search the blue tree for $NN_B(p')$ and $RNN_B(p')$ for a red point p' before we insert p' into the Red tree, and vice versa. We update the entries for $RNN_B(p')$ and their parent nodes while searching for $NN_B(p')$ and hence finish two tasks in one pass. Formally, we have the following Pre-insert algorithm.

```
Algorithm 1 Pre-insert (blueNode $n$, redPoint $p'$)
  Input: The root $n$ of the blue tree and a red point $p'$
  Output: the adjusted blue tree and $NN_B(p')$
  1) Initialize the candidate nearest neighbor $c$
  2) If $n$ is a leaf node, then for each entry $(pt, dnn)$
        do: If $D(p', pt) < D(p', c)$, then let $c = pt$;
        If $D(p', pt) < dnn$, output $pt$ and return
        $D(p', pt)$
  3) If $n$ is a non-leaf node, then for each branch
        $Bch = (ptr, Rect, max\_dnn, min\_dnn)$ do:
        If $D(p', Rect) < max\_dnn$ or $D(p', Rect) <$
        $D(p', c)$ call Pre-Insert($ptr$, $p'$); If $ptr$ was
        adjusted, adjust $max\_dnn$ and $min\_dnn$ for
        $Bch$.
```

After finding $NN_B(p')$, the nearest blue neighbor of a red point p', it is straightforward to insert the point p' in the Red tree since each tree is essentially an R-tree with embedded "bichromatic" nearest neighbor distance. Here we formally present the following Insertion algorithm.

```
Algorithm 2 Insert (blueNode $bn$, redNode $rn$, redPoint $p'$)
  Input: Root nodes $bn$ $rn$ and red point $p'$ to be inserted
  Output: the red tree with $p'$ inserted and blue tree
        adjusted
  1) Pre-Insert($bn, p'$)
  2) Call $R^*$-tree insertion algorithm to insert entry
        $(p', dnn_S(p'))$ into $rn$
```

Now we turn to deletion. Notice that deleting a point from the Red (Blue) tree also affects the reverse nearest neighbors in the Blue (Red) tree. In order to maintain the integrity of the trees while deleting a red (blue) point p'', an NN search needs to be done for each point in $RNN_B(p'')$ ($RNN_R(p'')$). As with the Rdnn-tree, we can do a Batch-NN search[12], finding the nearest neighbors for multiple query points in one pass. To remove the point, the standard R^*-tree deletion algorithm can be applied.

```
Algorithm 3 Delete (redPoint $p''$)
  Input: a red point $p''$ to be deleted
  Output: $p''$ deleted red tree and blue tree adjusted
  1) Call $R^*$-tree algorithm to delete $p''$ from red tree
  2) Call RNN-Search($p''$) to find $RNN_B(p'')$
  3) Call Batch-NN-Search($RNN_B(p'')$) on the red
       tree
  4) Adjust the $dnn$ for each point in $RNN_B(p'')$
       and propagate the change up to the root
```

Spatial Join Any standard distance-based spatial join algorithms can be applied to the bRdnn-tree. Moreover, the min_dnn value (the minimum distance of any point under the subtree to its nearest neighbor in the other tree) in each node provides extra pruning power. Assuming d being the distance threshold for a spatial join query, if for a node N we have $min\_dnn > d$, then N can be pruned since no pairs with distance less that d can be formed from any point contained under N.

k **Closest NN Pairs** Since we pre-compute the "bichromatic" nearest neighbor distance for each point while building the bRdnn-Tree, searching for the k-closest NN pairs in our index structure is straight forward. Each non-leaf node contains a number min_dnn, the minimum of the distances from each point in the subtree to it's nearest neighbor in the other data set. In a leaf node, each point is accompanied by such a nearest neighbor distance of its own. On an index structure with the above properties, a branch-and-bound approach for the k-closest NN pairs is natural. During the search, k candidate pairs are kept in a priority queue with the pair of largest distance on the top. Starting from the root of one tree with empty candidate pairs, select one branch with the smallest min_dnn to descend the tree. Prune any branch whose min_dnn field is larger than the distance of the pair on the queue top. Here we formally give the algorithm for the k-closest NN pairs as follows:

```
Algorithm 4 $k$ Closest NN Pairs
  1) Start from the root of the red tree with an empty
       candidate queue.
  2) Sort the branches in ascending order according
       to the $min\_dnn$ field.
  3) Recursively visit each branch in the order. Prune
       the branches that have $min\_dnn$ larger than
       the distance of the pair on the queue top.
  4) If a leaf node is reached, retriete each entry. If
       the dnn of the entry is smaller than that of
       queue top element, insert it into the queue.
  5) Repeat the above steps on the blue tree.
```

k **Closest Pairs** From Theorem 2.1 we can see that the k-closest NN pairs with respect to each data set gives us the end points for the k-closest pairs. To find the k-closest pairs, we first find the k-closest NN pairs (as shown above), then apply the following method:

Given two data sets R and S, after performing the k-closest NN pair search in the bRdnn-Tree containing S, we get k NN pairs $(p_1, q_1), (p_2, q_2), \cdots, (p_k, q_k)$ with respect to R, in ascending order according to the distance $D(p_i, q_i)$ for $i \in \{1, 2, \cdots, k\}$. Note that for $i \neq j$, p_i and p_j are not necessarily different. Hence, if we let $P = \{p'_1, \cdots, p'_{k_s}\}$ $(\subseteq R)$ be the set of distinct points among $p_1, p_2, \cdots, p_k$, we have $k_s \leq k$. Similarly, if we interchange R and S, we can get the corresponding subset Q in S such that $Q = \{q'_1, q'_2, \cdots, q'_{k_t}\}$. With these two sets of points P and Q, we can derive the k-closest pairs. By theorem 2.1, we know that if (p, q) is one of the k-closest pairs, then $p \in P$ and $q \in Q$. Hence, to find k-closest pairs in R and S, we only need to examine possible pairs from P and Q to build the k pairs incrementally in the following way. For any two points $p'_i \in P$ and $q'_j \in Q$ where $1 \leq i \leq k_s$ and $1 \leq j \leq k_t$:

- Case 1: (p'_i, q'_j) is an NN pair. If it is the k^{th} pair, stop the search. Otherwise, for all $i' < i$ and $j' < j$, consider $(p'_{i'}, q'_{j'})$. If it is better than the worst of the current candidate solution set, insert it into the candidate solution.

- Case 2: (p'_i, q'_j) is not an NN pair. If it is better than the worst of the current candidate solution set, insert it into the candidate solution.

```
Algorithm 5 $k$ Closest Pairs
  1) Perform $k$ NN pair search on the blue tree to
       find the subset $P$ of points that form the $k$ NN
       pairs.
  2) Repeat the above on the red tree to find the cor-
       responding subset $Q$
  3) If $(p'_i, q'_j)$ is an NN pair, then do:
         If it's the $k^{th}$ pair, stop the search.
         Else, for all $i' < i$ and $j' < j$ do:
             If $(p'_{i'}, q'_{j'})$ is better than the worst of the
                 current candidate solution set, insert
                 it into the candidate solution.
  4) If $(p'_i, q'_j)$ is not an NN pair, than if it is better
       than the worst of the current candidate solu-
       tion set, insert it into the candidate solution.
```

Theorem 2.1 shows that, in any of k-closest pair (p, q), point p and q come from the k-closest NN pairs. Some of the k closest pairs are NN pairs while some are not. In the above algorithm, we search for the non-NN pairs from points in the NN pairs to find the k-closest pairs. The worst case is a nested loop of k iterations. For large k, this may not be desirable. However, for small k, our experiments show that this algorithm is much more efficient than the existing solutions, especially when the data spaces overlap.

4 Experimental results

This section presents the results of our experiments. We focus on the k-closest pair query as we can compare with previously designed algorithms. Also note that, when k is small enough so that the k closest NN pairs can be put in the buffer, the k closest pair and the k closest NN pair query have the same number of disk accesses.

We ran our tests on a machine with two 500-MHz Pentium II processors with 512 MB RAM under SCO UNIX. For comparison, we implemented the R*-tree and both algorithms mentioned in [4], the depth-first branch and bound (DFS) (called sorted-distance recursive algorithm in [4]) and the heap algorithm. For those algorithms, we also included a LRU buffer[1], and measured the number of page faults. For the bRdnn-Tree, since we never have to revisit a leaf node, we do not have to rely on the buffer (except one to store the solution).

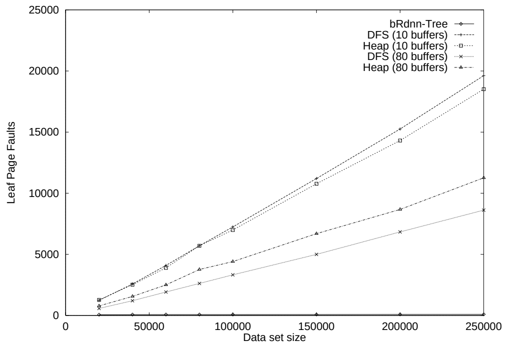
We experimented with different types of data sets. They included uniformly distributed data, clustered data, as well as real geographical 2D data from the US National Mapping Information web site (URL: `http://mappings.usgs.gov/www/gnis/`). For each type, we generated data sets of various sizes. For each size, we generated five different pairs of data and ran experiments on them. The average number of leaf node access is the main measure of search cost. We also measured the total number of nodes accessed. However, the two measures behave similarly, so in this paper we mainly show the leaf accesses.

4.1 Uniformly distributed data sets

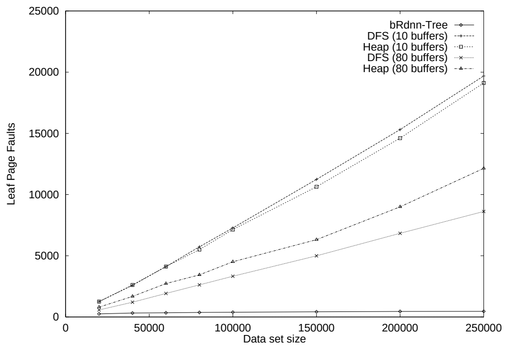
We first present the results on uniform data with 100% overlap. We measure the number of leaf page faults (i.e. number of page faults caused by reading data from disk). We have similar results when we consider the total page faults (including non-leaf pages). Figure 2 shows the results for the three different approaches: the bRdnn-Tree, the depth-first-search with R^*-tree (DFS), and the heap algorithm with R^*-tree (Heap). The DFS and Heap algorithms are given a buffer of 10 or 80 pages.

From figure 2, we can see that our index structure consistently outperforms both the depth-first and the heap algorithm, even when they are given a 10 or 80 page buffer

[1]Strictly speaking, it is level-wise LRU buffer – i.e. each level of a tree has an LRU buffer



k = 20



k = 100

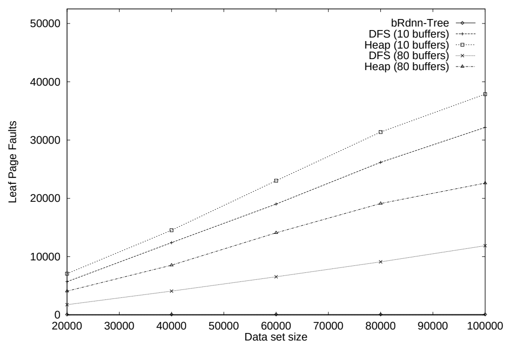
Figure 2. Comparison of performance for 2D data (fixed k)

advantage. When k is small, where our algorithm is most beneficial, the savings can be of a couple of order of magnitude. For instance, with k = 20 and data set size 80,000, the depth-first search takes more than 2,600 leaf page faults while our index requires less than 80 leaf accesses. (The numbers are similar on total node accesses).

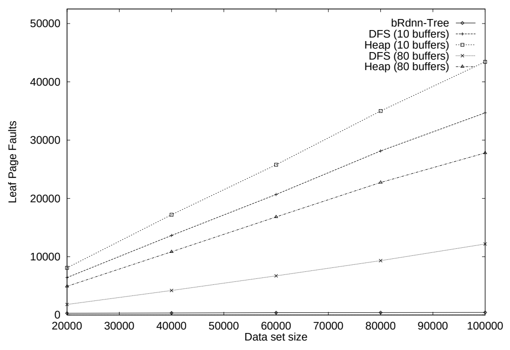
As we mentioned, the R^*-tree based algorithms fail due to the overlap between the two data sets. This implies the upper-level nodes cannot be pruned, as each node intersects with some other nodes of the other index. In fact, in all the experiments conducted on overlapping data sets, every node of the two R^*-trees has to be accessed. However, by storing the nearest neighbor distance, our method can prune each bRdnn-Tree index separately at a high level. Thus, it outperforms the R^*-tree algorithm by a large margin.

Also note that in our approach, one page is retrieved at a time. In that page, the dnn or the max_dnn of each leaf entry or branch is compared with the current candidates. In the R^*-tree algorithm, a pair of nodes from each tree is read and the distances between pairs of branches or leaf entries are computed before compared to the current candidate solution set. This is a nested loop and hence computationally expensive.

From the figure, we also notice that the performance of the R^*-tree algorithms is sensitive to the number of buffers available. However, as we mentioned earlier, our index is very robust in terms of buffer dependency.



k = 20



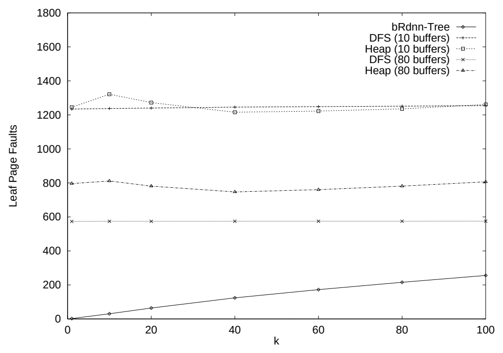
k = 100

Figure 3. Comparison of performance for 4D data (fixed k)

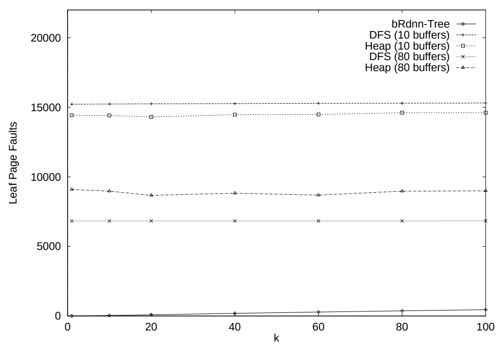
Next, we tested the applicability of the index in multimedia applications in higher dimensions by experimenting with 4D data, Figure 3 shows that the bRdnn-Tree still outperforms the existing R^*-tree algorithms consistently.

The performance with varied k is also measured. Figure 4 shows the results for 2D data. (Similar results are obtained for 4D data sets.)

Comparing the results, we see that for reasonably sized k, the bRdnn-Tree performs exceptionally well. With overlapping data sets, even for a small k, both R^*-tree based algorithms, DFS and Heap, need to search the whole tree. This is because, with overlapping regions, each node from one tree very likely overlaps with a node from the other tree. This means that this pair cannot be pruned away (because the distance between the bounding regions is 0). Thus even if $k = 1$, one has to traverse both trees completely. On the other hand, when k is small, the pruning power of the nearest neighbor distance is very strong. Thus, our algorithm prevails when k is not too large.



N = 20,000



N = 200,000

Figure 4. Comparison of performance for 2D data (varying k)

We also ran experiments on larger values of k. Typically, the performance of the R^*-tree catches up with our structure when k is around 10,000 if there is a large number of buffers. For instance, with data set size equal to 200,000, the R^*-tree only catches up when $k = 10,000$. Even then, it requires some buffer size (in this case, 80 buffer pages). For smaller buffer size (like 10-20 buffers), the bRdnn-Tree beats the R^*-tree based algorithms.

We also experimented with non-overlapping data sets. In this case, the R^*-tree based algorithms become more effective, since many branches can be pruned away at the root level (as the bounding rectangles are non-overlapping). This is true when k is large. However, when k is small, the performance of the two structures are similar.

4.2 2D clustered data sets

We also experimented with clustered data as they arise often in spatial databases. For instance, locations of houses are often highly clustered.

In our experiments, we created pairs of data sets. Each set contains 5 small clusters of radius 10 inside a 1000 x 1000 box. The center of a cluster was randomly generated.

	Leaf Page Fault		Total Page Fault	
	$k = 20$	$k = 100$	$k = 20$	$k = 100$
bRdnn-Tree	2 - 4	6 - 11	6 - 8	10 - 15
DFS (80 buffer)	3 - 1090	5 - 1091	7 - 1120	9 - 1121
Heap (80 buffer)	2 - 1090	3 - 1091	6 - 1120	7 - 1121

Table 2. Performance with 2D clustered data (N = 80,000)

At the same time, we also ensure that no two clusters in any pair of data sets intersect with each other. For each data set size, we generated 5 separate pairs of data sets and ran tests on them. Table 2 shows some representative results with a data set of 80,000 points.

While the bRdnn-Tree produces consistently excellent results, the search costs on the R^*-tree with the DFS and the Heap algorithm vary greatly. This is due to the bounding regions of the data set, which may or may not overlap in our data sets. When they do not overlap, the performances of the various methods are comparable. However, if the bounding regions overlap significantly, then even though the data is well clustered, the R^*-tree based algorithms still take much longer to execute. Thus, we can see the robustness of the bRdnn-Tree with regard to some non-uniform data distributions.

4.3 2D Real data set

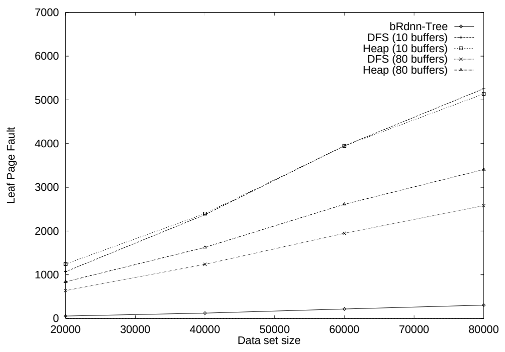
We also experimented on the US geographical data set. It contains more than 160,000 populated places in the USA, represented by latitude and longitude coordinates. We split the set into pairs of smaller sets of various sizes. Each pair of the data sets has no location in common.

We ran tests on the various data sets and the results are shown in figure 5. We can see that the bRdnn-Tree once again outperforms the R^*-tree based algorithms.

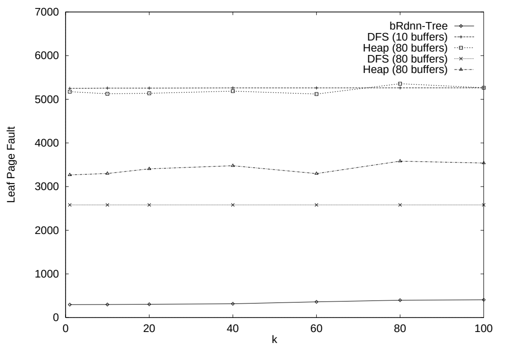
4.4 Index maintenance

While the bRdnn-tree outperforms the R^*-tree based algorithms in queries, it does require extra cost in terms of index maintenance, like insertion. This is because inserting a point into the bRdnn-tree requires finding the reverse nearest neighbors of the point to be inserted, which means one has to traverse both trees, as opposed to the case of the R^*-tree, where only one tree need to be inserted.

We measure the cost of index maintenance by inserting 2D points one at a time. We measure the average cost of insertion for the last 1,000 points. We also assume that there is a 10-page buffer for each tree. (For comparison's sake, for 200,000 data points, each bRdnn-tree has more than 4,000 nodes). The results in Table 3 are the averages of 5 different data sets.



k = 20, varying data set size



Data set size = 800,000, varying k

Figure 5. Comparison of performance on real data set

From Table 3 we see that an insertion into the bRdnn-tree takes around 2.4 times that of the R^*-tree. However, this is easily offset in the case when the two data sets overlap, as the bRdnn-tree requires an order of magnitude less time than the standard R^*-tree based joins.

4.5 Is an index worthwhile?

So far, we have been comparing the bRdnn-tree with the standard R^*-tree. However, a more fundamental question needs to be answered: is an index necessary at all?

If both data sets are static, one obvious option is to precompute the result once and store it in a separate file for

Data points	60,000	100,000	150,000	200,000	250,000
R^*-tree	1.7072	1.8233	1.8944	1.922	1.9449
bRdnn-tree	4.0556	4.2388	4.2895	4.6779	4.7525
ratio bRdnn-tree/R^*-tree	2.38	2.325	2.264	2.433	2.44

Table 3. Comparison of insertion cost for 2D data

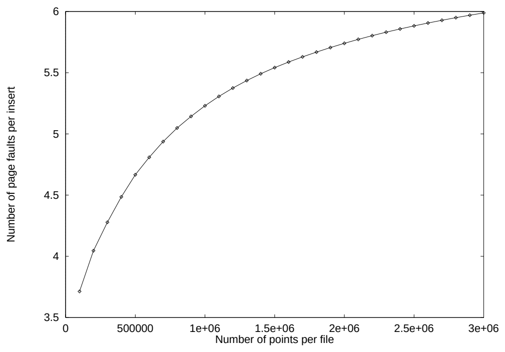
later retrival. This is especially useful if an upper bound of k can be reasonably estimated or assumed. For instance, for 2D data, a page of size 2K can store around 125 pairs of points. Thus, if we assume $k \leq 500$, we can run the nested loop algorithm once and store the top 500 pairs. Subsequently, all queries with $k \leq 500$ require only reading four pages, cheaper than any indexing methods. Moreover, for small data sets, a simple nested loop is typically less costly than building the two indexes from scratch. Thus, in such cases, it is better to apply a simple nested loop for the first query and then cache the results.

However, when the data set size grows, the cost of a simple nested loop query grows very fast – $O(n \times m)$, where m and n are the sizes of the data sets. However, the cost of building the index grows in a much slower pace. Figure 6 illustrates this fact.

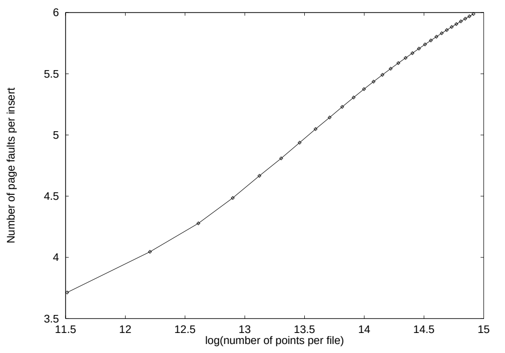
We performed tests with different data sets of up to 3,000,000 points and built the tree with 10 buffers for each bRdnn-tree. We measure the cost per insertion. Figure 6 (b) shows that the cost per insertion grows logarithmically. As mentioned previously, an insertion requires both a reverse nearest neighbor search and placing the point in the corresponding tree. The cost of the latter is proportional to the height of the tree. The former depends on the cost of finding the reverse nearest neighbor of the point to be inserted. We observe that the number of neighbors is small, and the reverse nearest neighbor search typically requires traversing only a couple of branches of the tree. Thus, when the data set size becomes large, it is worthwhile to build the index, instead of using the nested loop, even if it is just for precomputing results.

Now we turn our attention to the dynamic case. One can still decide to simply precompute and cache the list of current k closest pairs. However, when a new point is inserted, we need to compare it with all the points of the other set and update the list accordingly. Here, one needs to read one of the data set sequentially. This is costly when the data set becomes large, as illustrated by table 4.

To interpert the results of table 4, let's assume that a new query comes after 250 points (1 page of data for 2K page size) have been inserted into one table. If we precompute and cache the results without an index, then we have to compare the pairwise distances between the new points and the existing table. The cost is that of a linear scan on one table, which corresponds to the third row in table 4. In order to



(a) normal plot



(b) log-plot

Figure 6. Per insertion cost of building the bRdnn-tree

use the index, one needs to do 250 seperate insertions to the index. This corresponds to the second row of the table. As it can be seen from the table, when the database becomes large, the index becomes a far more cheaper choice. Notice that querying the index takes about a few hundred page accesses, for a million data points – still much cheaper than a sequential scan. Thus, it is crucial in the dynamic case to maintain an index.

In summary, in a relatively static environment with a relatively small data set, it is appropriate to use the naive nested-loop algorithm and cache the closest pairs. However, in a dynamic environment or with large amount of data, it is preferable to use the bRdnn-tree.

Data set size ($\times 1000$)	100	500	1,000	1,500	2,000
Average Cost per insert (bRdnn-tree)	4.2388	5.4905	6.0437	6.2576	6.4212
Cost for inserting 250 points to the bRdnn-tree	1059.7	1372.62	1510.92	1574.4	1605.3
Cost for sequential scan of a table	391	1954	3907	5860	7813

Table 4. Comparing indexing vs. caching cost for dynamic case

5 Conclusion and future work

In this paper, we present the bRdnn-Tree - a new index structure suitable for many kinds of join queries. The index structure keeps track of the nearest neighbor distance for each data point. This is shown to be a powerful pruning method for various kinds of queries, such as k-closest pairs and k-closest NN pairs. We experimented with k-closest pair queries and the results show that our index is efficient as well as robust – it can handle various distributions as well as work with limited resources.

Our immediate attention is to explore the possibility of efficiently bulk-loading the bRdnn-Tree. As we stated, for large data sets, it is worthwhile to build the index. Currently in our experiments, we built the tree by inserting the data point by point. If the data set is given beforehand, there should be many ways we can make use of it to build the index faster.

One other direction we would look at is to incorporate our structure into a query optimization and automatic index maintenance environment. As stated, our structure works extremely well when the data sets to be joined are overlapping, while the R^*-tree works well when the data is non-overlapping. Thus, an interesting question is how our index can provide hints and information to the system as to when to use or build such a structure and when to use the R^*-tree instead. This will have tremendous impact on automatic database tuning and other query optimization processes.

Acknowledgments

We would like to thank Diane Mittelmeier and Janis Dublin for proofreading the manuscript.

References

[1] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. Scalable sweeping-based spatial join. In *Proceedings of the Twenty-fourth International Conference on Very Large Databases*, pages 570–581, 1998.

[2] T. Brinkhoff, H.-P. Kriegel, R. Schneider, and B. Seeger. Multi-step processing of spatial joins. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 197–208, Minneapolis, MN, May 1994.

[3] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient processing of spatial joins using R-trees. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 237–246, Washington, D.C., May 1993.

[4] A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos. Closest pair queries in spatial databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 189–200, May 2000.

[5] G. R. Hjaltason and H. Samet. Incremental distance join algorithms for spatial databases. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, pages 237–248, 1998.

[6] Y.-W. Huang, N. Jing, and E. A. Rundensteiner. Spatial joins using r-trees: Breadth-first traversal with global optimizations. In *Proceedings 23th International Conference on Very Large Data Bases*, pages 396–405, Aug. 1997.

[7] F. Korn and S. Muthukrishnan. Influence sets based on reverse nearest neighbor queries. In *Proceedings of 2000 ACM SIGMOD International Conference on Management of Data*, pages 201–212, May 2000.

[8] N. Koudas and K. C. Sevcik. Size separation spatial join. In *Proceedings of 1997 ACM SIGMOD International Conference on Management of Data*, pages 324–335, 1997.

[9] M.-L. Lo and C. V. Ravishankar. Spatial joins using seeded trees. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 209–220, Minneapolis, MN, May 1994.

[10] M.-L. Lo and C. V. Ravishankar. Spatial hash-joins. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 247–258, Montreal, Quebec, Canada, 4–6 June 1996.

[11] J. M. Patel and D. J. DeWitt. Partition based spatial-merge join. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 259–270, 1996.

[12] C. Yang and K.-I. Lin. An index structure for efficient reverse nearest neighbor queries. In *Proceedings of the 17th IEEE International Conf. on Data Engineering*, pages 485–492, Apr. 2001.