High-dimensional Proximity Joins

Kyuseok Shim* Ramakrishnan Srikant Rakesh Agrawal

IBM Almaden Research Center
650 Harry Road, San Jose, CA 95120

Abstract

Many emerging data mining applications require a proximity (similarity) join between points in a high-dimensional domain. We present a new algorithm that utilizes a new data structure, called the ϵ-kd tree, for fast spatial proximity joins on high-dimensional points. This data structure reduces the number of neighboring leaf nodes that are considered for the join test, as well as the traversal cost of finding appropriate branches in the internal nodes. The storage cost for internal nodes is independent of the number of dimensions. Hence the proposed data structure scales to high-dimensional data. We analyze the cost of the join for the ϵ-kd tree and the R-tree family, and show that the ϵ-kd tree will perform better for high-dimensional joins. Empirical evaluation, using synthetic and real-life datasets, shows that proximity join using the ϵ-kd tree is typically 2 to 40 times faster than the R^+ tree, with the performance gap increasing with the number of dimensions.

We also discuss how some of the ideas of the ϵ-kd tree can be applied to the R-tree family. These biased R-trees perform better than the corresponding traditional R-trees for high-dimensional proximity joins, but do not match the performance of the ϵ-kd tree.

1 Introduction

Many emerging data mining applications require efficient processing of proximity (similarity) joins on high-dimensional points. Examples include applications in time-series databases[AFS93, ALSS95], multimedia databases [Jag94, NBE$^+$93, NC91], medical databases [ACF$^+$93, TBS90], and scientific databases[Vas93]. Some typical queries in these applications include: (1) discover all stocks with similar price movements; (2) find all pairs of similar images; (3) retrieve music scores similar to a target music score. These queries are often a prelude to clustering the objects. For example, given all pairs of similar images, the images can be clustered into groups such that the images in each group are similar.

To motivate the need for multidimensional indices in such applications, consider the problem of finding all pairs of similar time-sequences. The technique in [ALSS95] solves this problem by breaking each time-sequences into a set of contiguous subsequences, and finding all subsequences similar to each other. If two sequences have "enough" similar subsequences, they are considered similar. To find similar subsequences, each subsequence is mapped to a point in a multi-dimensional

*Currently at Bell Laboratories, Murray Hill, NJ.

space. Typically, the dimensionality of this space is quite high. The problem of finding similar subsequences is now reduced to the problem of finding points that are close to the given point in the multi-dimensional space. A pair of points are considered "close" if they are within ϵ distance of each other with some distance metric (such as L_2 or L_∞ norms) that involves all dimensions, where ϵ is specified by the user. A multi-dimensional index structure (the R^+ tree) was used for finding all pairs of close points.

This approach holds for other domains, such as image data. In this case, the image is broken into a grid of sub-images, key attributes of each sub-image mapped to a point in a multi-dimensional space, and all pair of similar sub-images are found. If "enough" sub-images of two images match, a more complex matching algorithm is applied to the images.

A closely related problem is to find all objects similar to a given objects. This translates to finding all points close to a query point.

Even if there is no direct mapping from an object to a point in a multi-dimensional space, this paradigm can still be used if a distance function between objects is available. An algorithm is presented in [FL95] for generating a mapping from an object to a multi-dimensional point, given a set of objects and a distance function.

Current spatial access methods (see [Sam89, Gut84] for an overview) have mainly concentrated on storing map information, which is a 2-dimensional or 3-dimensional space. While they work well with low dimensional data points, the time and space for these indices grow rapidly with dimensionality. Moreover, while CPU cost is high for proximity joins, existing indices have been designed with the reduction of I/O cost as their primary goal. We discuss these points further later in the paper, after reviewing current multidimensional indices.

To overcome the shortcomings of current indices for high-dimensional proximity joins, we propose a structure called the ϵ-kd tree. This is a main-memory data structure optimized for performing proximity joins. The ϵ-kd tree also has a very small build time. This lets the ϵ-kd tree use the proximity distance limit ϵ as a parameter in building the tree. Empirical evaluation shows that the build plus join time for the ϵ-kd tree is typically 2 to 40 times less than the join time for the R^+ tree [SRF87],[1] with the performance gap increasing with the number of dimensions. A pure main-memory data structure would not be very useful, since the data in many applications will not fit in memory. We present a join algorithm that can handle large amounts of data while still using the ϵ-kd tree.

Problem Definition We will consider two versions of the spatial proximity join problem:

- **Self-join:** Given a set of N high-dimensional points and a distance metric, find all pairs of points that are within ϵ distance of each other.

[1]Our experiments indicated that the R^+ tree was better than the R tree [Gut84] or the R^* tree [BKSS90] tree for high-dimensional proximity joins.

- **Non-self-join:** Given two sets S_1 and S_2 of high-dimensional points and a distance metric, find pairs of points, one each from S_1 and S_2, that are within ϵ distance of each other.

The distance metric for two n dimensional points $\vec{X}$ and $\vec{Y}$ that we consider is

$$L_p = \left(\sum_{1}^{n} |X_i - Y_i|^p \right)^{1/p}, \quad 1 \leq p \leq \infty.$$

L_2 is the familiar Euclidean distance, L_1 the Manhattan distance, and L_∞ corresponds to the maximum distance in any dimension.

Properties The problem of high-dimensional proximity joins with some distance metric and ϵ parameter has the following properties:

- The feature vector chosen for similarity comparison is high-dimensional.
- Every dimension of the feature vector is mapped into a numeric value whose bounds are known.
- The distance function is computed considering every dimension of the feature vector.
- The proximity distance limit ϵ is not large, since indices are not effective when the selectivity of the proximity join is large (i.e. when every point matches with most other points).

Paper Organization. In Section 2, we give an overview of existing spatial indices, and describe their shortcomings when used for high-dimensional proximity joins. Section 3 describes the ϵ-kd tree and the algorithm for proximity joins. In Section 4, we analyze the performance for the ϵ-kd tree and the R^+ tree for proximity joins. We give a performance evaluation in Section 5. Section 6 discusses biased R^+ trees. We conclude in Section 7.

2 Current Multidimensional Index Structures

We first discuss the R-tree family of indices, which are the most popular multi-dimensional indices, and describe how to use them for proximity joins. We also give a brief overview of other indices. We then discuss problems of the current index structures for high-dimensional proximity joins.

2.1 The R-tree family

R-tree [Gut84] is a balanced tree in which each node represents a rectangular region. Each internal node in a R-tree stores a *minimum bounding rectangle (MBR)* for each of its children. The MBR covers the space of the points in the child node. The MBRs of siblings can overlap. The decision whether to traverse a subtree in an internal node depends on whether its MBR overlaps with the space covered by query. When a node becomes full, it is split. Total area of the two MBRs resulting from the split is minimized while splitting a node. Figure 1 shows an example of R-tree. This tree

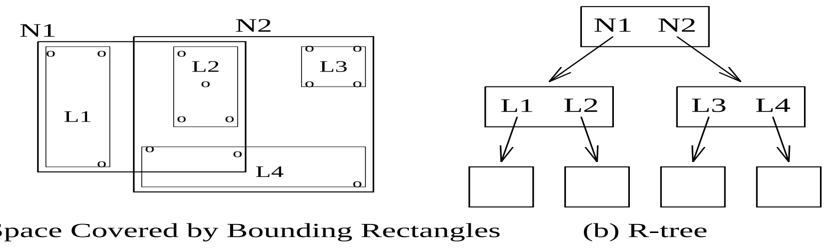


Figure 1: Example of an R-tree

consists of 4 leaf nodes and 3 internal nodes. The MBRs are N1,N2,L1,L2,L3 and L4. The root node has two children whose MBRs are N1 and N2.

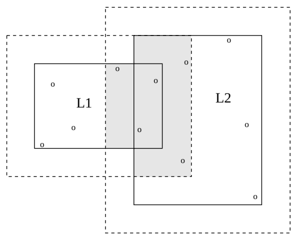
R^* **tree** [BKSS90] added two major enhancements to R-tree. First, rather than just considering the area, the node splitting heuristic in R^* tree also minimizes the perimeter and overlap of the bounding regions. Second, R^* tree introduced the notion of *forced reinsert* to make the shape of the tree less dependent on the order of the insertion. When a node becomes full, it is not split immediately, but a portion of the node is reinserted from the top level. With these two enhancements, the R^* tree generally outperforms R-tree.

R^+ **tree** [SRF87] imposes the constraint that no two bounding regions of a non-leaf node overlap. Thus, except for the boundary surfaces, there will be only one path to every leaf region, which can reduce search and join costs.

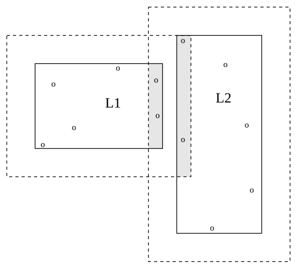
X-tree [BKK96] avoids splits that could result in high degree of overlap of bounding regions for R^*-tree. Their experiments show that the overlap of bounding regions increases significantly for high dimensional data resulting in performance deterioration in the R^*-tree. Instead of allowing splits that produce high degree of overlaps, the nodes in X-tree are extended to more than the usual block size, resulting in so called super-nodes. Experiments show that X-tree improves the performance of point query and nearest-neighbor query compared to R^*-tree and TV-tree (described below). No comparison with R^+-tree is given in [BKK96] for point data. However, since the R^+-tree does not have any overlap, and the gains for the X-tree are obtained by avoiding overlap, one would not expect the X-tree to be better than the R^+-tree for point data.

Proximity Join The join algorithm using R-tree considers each leaf node, extends its MBR with ϵ-distance, and finds all leaf nodes whose MBR intersects with this extended MBR. The algorithm then performs a nested-loop join or sort-merge join for the points in those leaf nodes, the join condition being that the distance between the points is at most ϵ. (For the sort-merge join, the points are first sorted on one of the dimensions.)

To reduce redundant comparisons between points when joining two leaf nodes, we could first *screen* points. The boundary of each leaf node is extended by ϵ, and only points that lie within the intersection of the two extended regions need be joined. Figure 2 shows an example, where the rectangles with solid lines represent the MBRs of two leaf nodes and the dotted lines illustrate the



(a) Overlapping MBRs (R tree and $R^{\star}$ tree)



(b) Non-overlapping MBRs (R^{+} tree)

Figure 2: Screening points for join test

extended boundaries. The shaded area contains screened points.

2.2 Other Index Structures

kdB tree [Rob81] is similar to the R^{+} tree. The main difference is that the bounding rectangles cover the entire space, unlike the MBRs of the R^{+} tree.

hB-tree [LS09] is similar to the kdB tree except that bounding rectangles of the children of an internal node are organized as a K-D tree [Ben75] rather than as a list of MBRs. (The K-D-tree is a binary tree for multi-dimensional points. In each level of the K-D-tree, only one dimension, chosen cyclically, is used to decide the subtree for traversal.) Further, the bounding regions may have rectangular holes in them. This reduces the cost of splitting a node compared to the kdB tree.

TV-tree [LJF94] uses a variable number of dimensions for indexing. TV-tree has a design parameter α ("active dimension") which is typically a small integer (1 or 2). For any node, only α dimensions are used to represent bounding regions and to split nodes. For the nodes close to the root, the first α dimensions are used to define bounding rectangles. As the tree grows, some nodes may consist of points that all have the same value on their first, say, k dimensions. Since the first k dimensions can no longer distinguish the points in those nodes, the next α dimensions (after the k dimensions) are used to store bounding regions and for splitting. This reduces the storage and traversal cost for internal nodes.

Grid-file [NHS84] partitions the k-dimensional space as a grid; multiple grid buckets may be placed in a single disk page. A directory structure keeps track of the mapping from grid buckets to disk pages. A grid bucket must fit within a leaf page. If a bucket overflows, the grid is split on one of the dimensions.

2.3 Problems with Current Indices

The index structures described above suffer from the following problems for performing proximity joins with high-dimensional points:

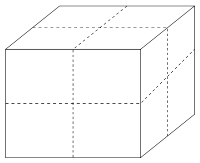


Figure 3: Number of neighboring leaf nodes.

Number of Neighboring Leaf Nodes. The splitting algorithms in the R-tree variants utilize every dimension equally for splitting in order to minimize the volume of hyper-rectangles. This causes the number of neighboring leaf nodes within ϵ-distance of a given leaf node to increase dramatically with the number of dimensions. To see why this happens, assume that a R-tree has partitioned the space so that there is no "dead region" between bounding rectangles. Then, with a uniform distribution in a 3-dimensional space, we may get 8 leaf nodes as shown in Figure 3. Notice that every leaf node is within ϵ-distance of every other leaf node. In an n dimensional space, there may be $O(2^n)$ leaf nodes within ϵ-distance of every leaf node. The problem is somewhat mitigated because of the use of MBRs. However, the number of neighbors within ϵ-distance still increases dramatically with the number of dimensions.

This problem also holds for other multi-dimensional structures, except perhaps the TV-tree. However, the TV-tree suffers from a different problem – it will only use the first k dimensions for splitting, and does not consider any of the others (unless many points have the same value in the first k dimensions). With enough data points, this leads to the same problem as for the R-tree, though for the opposite reason. Since the TV-tree uses only the first k dimensions for splitting, each leaf node will have many neighboring leaf nodes within ϵ-distance.

Note that the problem affects both the CPU and I/O cost. The CPU cost is affected because of the traversal time as well as time to screen all the neighboring pages. I/O cost is affected because we have to access all the neighboring pages.

Storage Utilization. The kdB tree and R-tree family, including the X-tree, represent the bounding regions of each node by rectangles. The bounding rectangles are represented by "min" and "max" points of the hyper-rectangle. Thus, the space needed to store the representation of bounding rectangles increases linearly with the number of dimensions. This is not a problem for the hB-tree (which does not store MBRs), the TV-tree (which only uses a few dimensions at a time), or the grid file.

Traversal Cost. When traversing a R-tree or kdB tree, we have to examine the bounding regions of children in the node to determine whether to traverse the subtree. This step requires checking the ranges of every dimension in the representation of bounding rectangles. Thus, the CPU cost of examining bounding rectangles increases proportionally to the number of dimensions of data points. This problem is mitigated for the hB-tree or the TV-tree. This is not a problem for the grid-file.

Build Time. The set of objects participating in a spatial join may often be pruned by selection predicates [LR94] (e.g. find all similar international funds). In those cases, it may be faster to perform the non-spatial selection predicate first (select international funds) and then perform spatial join on the result. Thus it is sometimes necessary to build a spatial index on-the-fly. Current indices are designed to be built once; the cost of building them can be more than the cost of the join [PD96].

Skewed Data. Handling skewed data is a problem for the grid-file. In a k-dimensional space, a single data page overflow may result in a k–1 dimensional slice being added to the grid-file directory. If the grid-file had n buckets before the split, and the splitting dimension had m partitions, n/m new cells are added to the grid after the split. Thus, the size of the directory structure can grow rapidly for skewed high-dimensional points.

Summary. Each index has good and bad features for proximity join of high-dimensional points. It would be difficult to design a general-purpose multi-dimensional index which does not have any of the shortcomings listed above. However, by designing a special-purpose data structure, we can attack these problems. We now describe a new data structure, ϵ-kd tree, which is a special-purpose data structure for this purpose. This data structure can be considered analogous to hash tables built for equality joins.

3 The ϵ-kd tree

We introduce the ϵ-kd tree in Section 3.1 and then discuss its design rationale in Section 3.2.

3.1 ϵ-kd tree definition

We first define the ϵ-kd tree[2]. We then describe how to perform proximity joins using the ϵ-kd tree, first for the case where the data fits in memory, and then for the case where it does not.

ϵ-kd tree We assume, without loss of generality, that the co-ordinates of the points in each dimension lie between 0 and +1. Pointers to the data points are stored in leaf nodes. We start with a single leaf node. Whenever the number of points in a leaf node exceeds a threshold, the leaf node is split, and converted to an interior node. If the leaf node was at level i, the ith dimension is used for splitting the node. The node is split into $\lfloor 1/\epsilon \rfloor$ parts, such that the width of each new leaf node in the ith dimension is either ϵ or slightly greater than ϵ. In the rest of this section, we assume without loss of generality that ϵ is an exact divisor of 1. An example of ϵ-kd tree for two dimensional space, with $\epsilon = 0.25$, is shown in Figure 4.

[2]It is really a trie, but we call it a tree since it is conceptually similar to kdB tree.

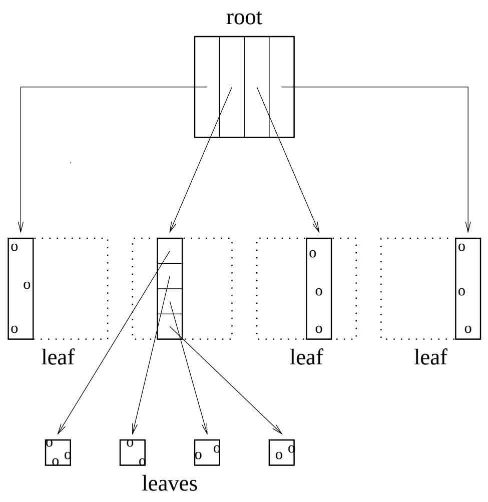


Figure 4: ϵ-kd tree

Proximity Join using the ϵ-kd tree Let x be an internal node in the ϵ-kd tree. We use x[i] to denote the ith child of x. Let f be the fanout of the tree. Note that $f = 1/\epsilon$. Figure 5 describes the join algorithm. The algorithm initially calls self-join(root), for the self-join version, or join(root1, root2), for the non-self-join version. The procedures leaf-join(x, y) and leaf-self-join(x) perform a sort-merge join on leaf nodes.

For high-dimensional data, the ϵ-kd tree will rarely use all the dimensions for splitting. (For instance, with 10 dimensions and a ϵ of 0.1, there would have to be more than 10^{10} points before all dimensions are used.) Thus we can usually use one of the free unsplit dimension as a common "sort dimension". The points in every leaf node are kept sorted on this dimension, rather then being sorted repeatedly during the join. When joining two leaf nodes, the algorithm does a sort-merge using this dimension.

Note that we can use the same join code for all the L_p distance metrics, with only the final test between a pair of points being metric-dependent.

Memory Management The value of ϵ is often given at run-time. Since the value of ϵ is a parameter for building the data structure, it may not be possible to build a disk-based version of the ϵ-kd tree in advance. Instead, we sort the multi-dimensional points with the first splitting dimension and keep them as an external file.

We first describe the join algorithm, assuming that main-memory can hold all points within a 2ϵ distance on the first dimension, and then generalize it. The join algorithm first reads points whose values in the sorted dimension lie between 0 and 2ϵ, builds the ϵ-kd tree for those points in main memory, and performs the proximity join in memory. The algorithm then deallocates the space used for the points whose values in the sorted dimension are between 0 and ϵ, reads points whose values are between 2ϵ and 3ϵ, build the ϵ-kd tree for these points, and performs the join

```
procedure join(x, y)
begin
   if leaf-node(x) and leaf-node(y) then
      leaf-join(x, y);
   else if leaf-node(x) then begin
      for i = 1 to $f$ do
         join(x, y[i]);
   end
   else if leaf-node(y) then begin
      for i = 1 to $f$ do
         join(x[i], y);
   end
   else begin
      for i = 1 to $f - 1$ do begin
         join(x[i], y[i]);
         join(x[i], y[i+1]);
         join(x[i+1], y[i]);
      end
      join(x[f], y[f]);
   end
end
```

```
procedure self-join(x)
begin
   if leaf-node(x) then
      leaf-self-join(x);
   else begin
      for i = 1 to $f-1$ do begin
         self-join(x[i], x[i]);
         join(x[i], x[i+1]);
      end
      self-join(x[f], x[f]);
   end
end
```

Figure 5: Join algorithm

(a) Step 1 (b) Step 2 (c) Step 3 (d) Step 4

Figure 6: Memory Management: Example 1

procedure again. This procedure is continued until all the points have been processed. Note that we only read each point off the disk once. This procedure is illustrated in Figure 6. The shaded portion illustrates the data present in main memory in each step.

This procedure works because the build time for the ϵ-kd tree is extremely small. It can be generalized to the case where a 2ϵ chunk of the data does not fit in memory. The basic idea is to partition the data into ϵ^2 chunks using an additional dimension. Then, the join procedure (i.e read points into memory, build ϵ-kd , perform join and so on) is instead repeated for each $4\epsilon^2$ chunk of the data using the additional dimension. This process is shown in Figure 7. The first square shows the data partitioned into a $\epsilon \times \epsilon$ grid, with the four ϵ^2 chunks in the top left corner (11, 12, 21 and 22) shaded. These 4 chunks are read into memory, and the join is performed between these chunks, and for each chunk. Chunks 11 and 21 are then dropped, chunks 13 and 23 read into memory, and the joins between 12, 22, 13 and 23 performed. This process continues till all joins are complete.

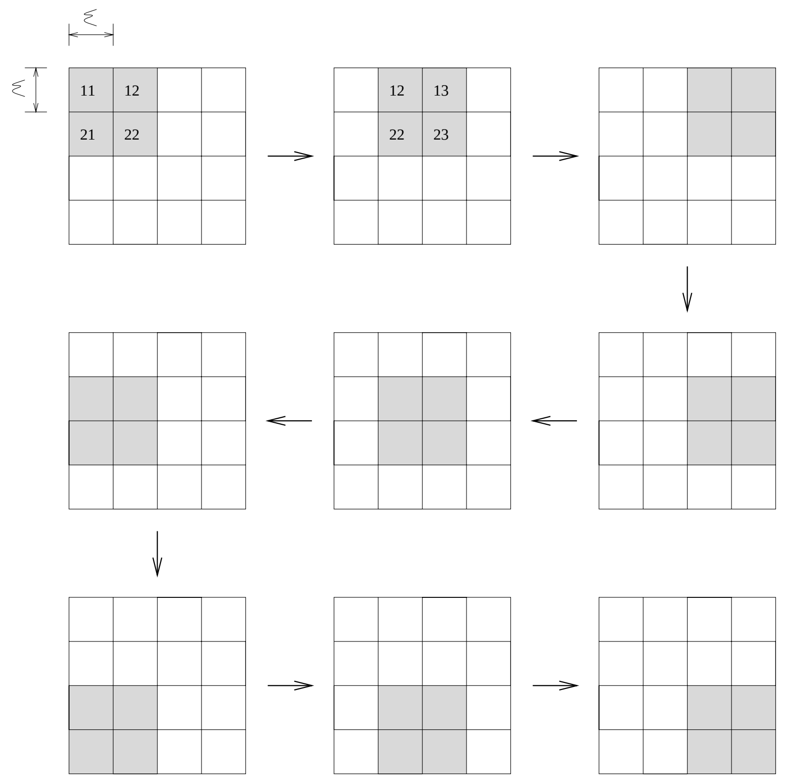


Figure 7: Memory Management: Example 2

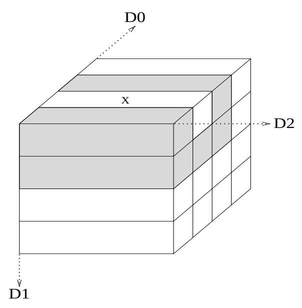
3.2 Design Rationale

Two distinguishing features of ϵ-kd tree are:

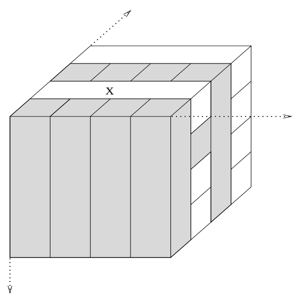
- *Biased Splitting* : We do not treat all dimensions equally for choosing split points. Instead, we preferentially split one dimension multiple times.
- *ϵ Sized Splitting* : When we split a node, we split the node in ϵ sized chunks.

We discuss below how these features help ϵ-kd tree solve the problems with current indices outlined in Section 2.

Number of Neighboring Leaf Nodes. Recall that with current indices, the number of neighboring leaf pages may increase exponentially with the number of dimensions. The ϵ-kd solves this problem because of the *biased splitting.* When the length of the bounding rectangle of each leaf nodes in the split dimension is at least ϵ, at most two neighboring leaf nodes need to be considered for the join test. However, as the length of the bounding rectangle in the split dimension becomes less than ϵ, the number of neighbor leaf nodes for join test increases. Hence when a leaf node becomes full, we split the node into several children, each of size ϵ in the split dimension. We split the node into several children at once, rather than gradually, in order to reduce the build time and the number of children in each neighbor.



(a) Choosing split dimension globally



(b) Choosing split dimension locally

Figure 8: Global and Local Ordering of Split Dimensions

We have two alternatives for choosing the next split dimension: global ordering and local ordering. Global ordering uses the same split dimension for all the nodes in the same level, while local ordering chooses the split dimension based on the distribution of points in each node. Examples of these two cases are shown in Figure 8, for a 3-dimensional space. For both orderings, the dimension $D0$ is used for splitting in the root node (i.e. level 0). For global ordering, only $D1$ is used for splitting in level 1. However, for local ordering, both $D1$ and $D2$ are chosen alternatively for neighboring nodes in level 1. Consider the leaf node labeled X. With global ordering, it has 5 neighbor leaf nodes (shaded in the figure). The number of neighbors increases to 9 for local ordering. Notice that the space covered by the neighbors for global order is a proper subset of that covered by the neighbors for local ordering. The difference in the space covered by the two orderings increases as ϵ decreases. Hence we chose global ordering for split dimensions, rather than local ordering.

When the number of points are so huge that the ϵ-kd tree is forced to split every dimension, then the number of neighbors will be comparable to other indices. However, till that limit, the number of neighbors depends on the number of points (and their distribution) and ϵ, and is independent of the number of dimensions.

The order in which dimensions are chosen for splitting can significantly affect the space utilization and join cost if correlations exist between some of the dimensions. This problem can be solved by statistically analyzing a sample of the data, and choosing for the next split the dimension that has the least correlation with the dimensions already used for splitting.

Space Requirements. For each internal node, we simply need an array of pointers to its children. We do not need to store minimum bounding rectangles because they can be computed. Hence the space required depends only on the number of points (and their distribution), and is independent of the number of dimensions.

Traversal Cost. Since we split nodes in ϵ sized chunks, traversal cost is extremely small. The join procedure never has to check bounding rectangles of nodes to decide whether or not they may

Total number of points	T
Range of points	0 to 1

	R^+ tree	ϵ-kd tree
Average number of points per leaf node	N_r	N_e
Number of dimensions used for splitting	D_r	D_e

Table 1: Notation

contain points within ϵ distance.

Build time. The build time is small because we do not have complex splitting algorithms, or splits that propagate upwards.

Skewed data. Since splitting a node does not affect other nodes, the ϵ-kd tree will handle skewed data reasonably.

4 Analysis

In this section, we analyze the number of join tests for the ϵ-kd tree, and the number of join and screen tests for the R^+ tree. The goal is to understand the behavior of the indices as the number of dimensions and number of points vary. For simplicity, we assume that the MBRs of the R^+ tree cover the whole space. (That is, we really use a K-D-B tree as a proxy for the R+ in this analysis. For datasets with a lot of points, the gap between the MBRs will be small; hence this is a reasonable approximation.) In this analysis, we only consider uniform distribution of points. We also assume that the R^+ tree will always split a rectangle on a dimension which has not been used for splitting (if such a dimension is available),[3] and that the bounding rectangle is split at its midpoint. Further, we assume that the set of free dimensions is the same for all leaf pages. This issue is similar to the issue of global vs. local splitting for the ϵ-kd tree; hence this assumption is favorable to the R^+ tree.

We first consider the cases where both the R^+ tree and the ϵ-kd tree have unsplit dimensions left, and then extrapolate to the case where either the R^+ tree, or both indices, have no unsplit dimensions. Table 1 summarizes the notation we will use in this section.

4.1 R^+ tree: analysis

Recall that in the R+ tree, the join test is performed for points in each leaf node, as well as between pairs of leaf nodes that have an overlap when their MBRs are extended by ϵ. A naive approach to performing the join test would be to use the nested-loop join. In other words, for each point in

[3]The traditional splitting heuristic which tries to minimize both the volume and perimeter of the hyper-rectangles will result in the R^+ tree usually splitting the MBR on a dimension which has not been used for splitting.

(a) 1-dimensional boundary (b) 0-dimensional boundary

Figure 9: Effect of Screening

left leaf node, we examine all points in right leaf node. However, as shown earlier in Figure 2, the algorithm can first screen the points in each leaf nodes to see if they fit within the extended MBR of the other leaf node.

Let D_r be the number of split dimensions used by the R^+ tree. Then the R^+ tree will have 2^{D_r} pages, with $T/2^{D_r}$ points in each. Let L_m be the maximum number of points in a leaf page. Since $T/2^{D_r} < L_m$, $D_r = \lceil \log_2(T/L_m) \rceil$.

Note every leaf page falls within the extended MBR of every other leaf node, since they all meet at the mid-point of the space. Thus for each leaf page, the points in every other leaf page have to be screened. Since there are T/N_r pages, and T points have to be screened for each page, the total number of screen tests is given by

$$\#\ \textbf{Screen Tests} \approx T^2/N_r \tag{1}$$

Next, we look at the number of join tests. If two leaf nodes have a $(D_r - k)$-dimensional common boundary (among the dimensions used for splitting), the expected number of points in each node left after screening is $N_r \times (2\epsilon)^k$, where N_r is the average number of points in a leaf node. (For each dimension which is not a boundary, 2ϵ of the remaining points are left after screening, since the size of the page in each dimension is 1/2.) This is illustrated in Figure 9. For the two leaf nodes with a 1-dimensional (0-dimensional) common boundary, there are 2ϵ ($4\epsilon^2$) of the points left after screening on each node. The number of join tests for two leaf pages with a $(D_r - k)$-dimensional common boundary would be $(N_r \times (2\epsilon)^k)^2$, using a nested-loop join. If we first sort the points on one of the dimensions, and use a sort-merge join, the cost drops to

$$(N_r \times (2\epsilon)^k)^2 \times 2\epsilon = {N_r}^2 \times (2\epsilon)^{2k+1} \tag{2}$$

We do not count the time for the sort, since we can sort all the points on one of the unsplit dimensions at the time the tree is built.

Hence, if we know the number of neighbor leaf nodes that have a common $(D_r - k)$-dimensional boundary, we can estimate the number of join tests. We can represent the position of each leaf page in D_r-dimensional space as a D_r-dimensional boolean array, where "True" in the k-th dimension would correspond to the node being above the mid-point of the space in the k-th dimension. If two leaf pages had the same value for k dimensions, they would have a k-dimensional common boundary. Since there are $\begin{pmatrix} D_r \\ k \end{pmatrix}$ ways of choosing k dimensions that are the same, each leaf page has $\begin{pmatrix} D_r \\ k \end{pmatrix}$ neighbors that have a k-dimensional common boundary.

We can now compute the total number of join tests, for all (T/N_r) leaf pages, by plugging in (2).

$$\# \text{ Join Tests} = (T/N_r) \times \sum_{i=1}^{D_r} \begin{pmatrix} D_r \\ i \end{pmatrix} \times {N_r}^2 \times (2\epsilon)^{2i+1} = T \times N_r \times \sum_{i=1}^{D_r} \begin{pmatrix} D_r \\ i \end{pmatrix} \times (2\epsilon)^{2i+1}$$

Since ϵ is typically quite small, ϵ^2 would be considerably smaller than D_r. Hence we can ignore terms with ϵ^5 or higher powers of ϵ, resulting in

$$T \times N_r \times D_r \times 8\epsilon^3 \tag{3}$$

The above formula does not include the cost of a self-join on the points in each leaf page. Since this requires $N_r^2 \times 2\epsilon$ comparisons per leaf page, and there are (T/N_r) pages, the total number of join tests for these self-joins is

$$T \times N_r \times 2\epsilon \tag{4}$$

Combining (3) and (4), we get

$$\textbf{\# Join Tests} \approx T \times N_r \times 2\epsilon \times (1 + D_r \times 4\epsilon^2) \tag{5}$$

Since typically $N_r\epsilon < (T/N_r)$ (the number of points per pages times ϵ is less than the number of leaf pages), and $D_r \times 4\epsilon^2 < 1$, we get

$$T \times N_r \times 2\epsilon < T \times T/N_r$$

That is, the number of join tests (Equation 5) is less than the number of screen tests (Equation 1).

Someone might argue that not performing screen test may improve the cost of proximity join. Assume that we do not perform screen and perform sort-merge join only. Then, the number of join tests between a pair of leaf nodes becomes $N_r \times N_r \times 2\epsilon$. The cost of screening points for a pair of leaf pages is $2N_r$. Since $N_r \times \epsilon > 1$, the join cost without screening is more expensive than the cost of screening.

4.2 ϵ-kd tree: analysis

Let the ϵ-kd tree have a depth of D_e. Each leaf has $N_e \approx T\epsilon^{D_e}$ points on average. Recall that there is no screen cost for the ϵ-kd . If we use a sort-merge join, the join cost between a pair of leaf nodes is

$$N_e \times N_e \times 2\epsilon$$

We do not count the time for the sort since that can be done at the time the tree is built (that is, once per leaf node, rather than once per pair of leaf nodes within ϵ distance).

A leaf node in the ϵ-kd has at most $3^{D_e} - 1$ neighbors within ϵ distance. Now, $3^{D_e} - 1 \approx (T/N_e) \times (3\epsilon)^{D_e}$, since (T/N_e) is the number of leaf nodes, and $(3\epsilon)^{D_e}$ the fraction of leaf nodes within ϵ distance.

Thus the total number of join tests is

$$(T/N_e) \times [(T/N_e) \times (3\epsilon)^{D_e}] \times {N_e}^2 * 2\epsilon = T^2 \times 2\epsilon \times (3\epsilon)^{D_e}$$

We can simplify the formula by multiplying by a fudge factor of 1.5 (at the cost of penalizing the ϵ-kd tree a little). Thus we get

$$\textbf{\# Join Tests} \approx T^2 \times (3\epsilon)^{D_e+1} \tag{6}$$

4.3 Comparison of costs

Since there are no screen costs for the ϵ-kd tree, Equation 6 shows the dominant factor behind the performance of the ϵ-kd tree. We can now compare this with the dominant factor for the R^+ tree, the number of screen tests given in Equation 1. From the two equations, the ϵ-kd tree will be faster than the R^+ tree when

$$(3\epsilon)^{D_e+1} < (1/N_r) \tag{7}$$

Plugging in some typical values, $N_r = 50, \epsilon = 0.05$ into Equation 7, the number of tests for the ϵ-kd tree is considerably smaller even for $D_e = 2$. Note that the performance of the R^+ tree cannot be improved by increasing the size of leaf pages (thus decreasing $1/N_r$), since the join cost would become dominant as the size increased.

Equation 7 ignores traversal cost, which is much lower for the ϵ-kd tree than the R^+ tree since the ϵ-kd does not have to check for intersection of MBRs. Further, Equation 7 ignores the number of join tests for the R^+ tree completely and just considers screen costs. On the other hand, since the R^+ tree does not partition the space, but uses MBRS, the number of screens for the R^+ tree is likely to be somewhat less than suggested by the above formulae.

In Section 5.5, we give empirical results for the number of join tests for the ϵ-kd tree, and the number of join and screen tests for the R^+ tree, while varying several parameters. These results correlate well with the above analysis.

4.4 Other Cases

We now consider the case where both indices have no unsplit dimensions, followed by the case where only the R^+ tree has unsplit dimensions. (Since the R^+ tree does unbiased splitting, it will always run out of dimensions before the ϵ-kd tree.)

If the ϵ-kd tree has no unsplit dimensions left, we expect the number of join tests to be similar for the ϵ-kd tree and the R^+ tree, since both of them will partition the space in a similar manner. For high-dimensional data, we expect this case to be very rare. For example, with a ϵ of 0.1, and 10 dimensional data, there have to be more than 10^{11} points before this occurs.

If only the R^+ tree has no unsplit dimensions left, the performance gap will narrow compared to the case where both trees have unsplit dimensions. As the R^+ tree fills the space, the screen cost becomes less dominant compared to the join cost. However, the sum of the costs is still higher than for the ϵ-kd tree, since the R^+ tree still operates in a higher dimensional space than the ϵ-kd tree. Another way to look at this case is to consider it being in between the case where both trees having unsplit dimensions and case where both trees having no unsplit dimensions. Since there is a gradual transition, the performance gap narrows till the number of join plus screen tests become comparable when both trees have no unsplit dimensions left.

5 Performance Evaluation

We empirically compared the performance of the ϵ-kd tree with both $R+$ tree and a sort-merge algorithm. The experiments were performed on an IBM RS/6000 250 workstation with a CPU clock rate of 66 MHz, 128 MB of main memory, and running AIX 3.2.5. Data was stored on a local disk, with measured throughput of about 1.5 MB/sec.

We first describe the algorithms compared in Section 5.1, and the datasets used in experiments in Section 5.2. Next, we show the performance of the algorithms on synthetic and real-life datasets in Sections 5.3 and 5.4 respectively. Finally, we explain the observed performance by looking at the number of join tests and screen counts in Section 5.5.

5.1 Algorithms

ϵ-kd tree. We implemented the ϵ-kd tree algorithm described in Section 3.1. A leaf node was converted to an internal node (i.e. split) if its memory usage exceeded 4096 bytes. However, if there were no dimensions left for splitting, the leaf node was allowed to exceed this limit. The execution times for the ϵ-kd tree include the I/O cost of reading an external sorted file containing the data points, as well as the cost of building the data structure. Since the external file can be generated once and reused for different value of ϵ, the execution times do not include the time to sort the external file.

R^+ tree. Our experiments indicated that the R^+ tree was faster than the R^* tree for proximity joins on a set of high-dimensional points. (Recall that the difference between R^+ tree and R^* tree is that R^+ tree does not allow overlap between minimum bounding rectangles. Hence it reduces the number of overlapping leaf nodes to be considered for the spatial proximity join, resulting in faster execution time.) We therefore used R^+ tree for our experiments. We used a page size of 4096

	ϵ-kd	R^+ tree	Sort-Merge
Join Cost	Yes	Yes	Yes
Build Cost	Yes	No	–
Sort Cost (first dim.)	No	–	No

Table 2: Costs included in the execution times.

Parameter	Default Value	Range of Values
Number of Points	100,000	10,000 to 1 million
Number of Dimensions	10	4 to 28
ϵ (join distance)	0.1	0.01 to 0.2
Range of Points	-1 to +1	-same-
Distance Metric	L_2-norm	L_1, L_2, L_∞ norms

Table 3: Synthetic Data Parameters

bytes. In our experiments, we ensured that the R^+ tree always fit in memory and a built R^+ tree was available in memory before the join execution began. Thus, the execution time for R^+ tree does not include any build time — *it only includes CPU time for main-memory join.* (Although this gives the R^+ tree an unfair advantage, we err on the conservative side.)

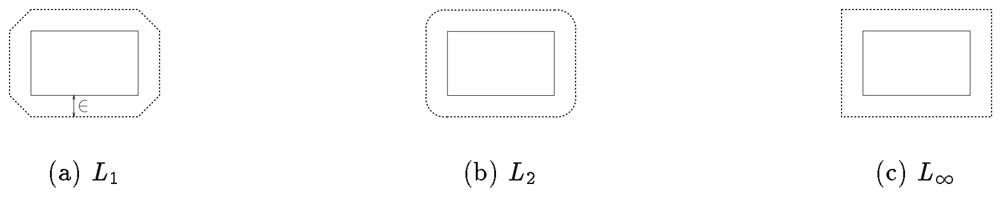
2-level Sort-Merge. Consider a simple sort-merge algorithm, which reads the data from a file sorted on one of the dimensions and performs the join test on all pairs of points whose values in the sort dimension are closer than ϵ. We implemented a more sophisticated version of this algorithm, which reads a 2ϵ chunk of the sorted data into memory, further sorts in memory this data on a second dimension, and then performs the join test on pairs of points whose values in the second sort dimension are close than ϵ. The algorithm then drops the first ϵ chunk from memory and reads the next ϵ chunk, and so on. The execution times reported for this algorithm also do not include the external sort time.

Table 2 summarizes the costs included in the execution times for each algorithm.

5.2 Data Sets and Performance Metrics

Synthetic Datasets. We generated two types of synthetic datasets: uniform and gaussian. The values in each dimension were randomly generated in the range -1.0 to 1.0 with either uniform or gaussian distribution. For the Gaussian distribution, the mean and the standard deviation were 0 and 0.25 respectively. Table 3 shows the parameters for the datasets, along with their default values and the range of values for which we conducted experiments.

Distance Functions We used L_1, L_2 and L_∞ as distance functions in our experiments. The extended bounding rectangles obtained by extending MBRs by ϵ differ slightly in R^+ tree depending on distance functions. Figure 10 shows the extended bounding regions for the L_1, L_2 and L_∞ norms.



(a) L_1 (b) L_2 (c) L_∞

Figure 10: Bounding Regions extended by ϵ

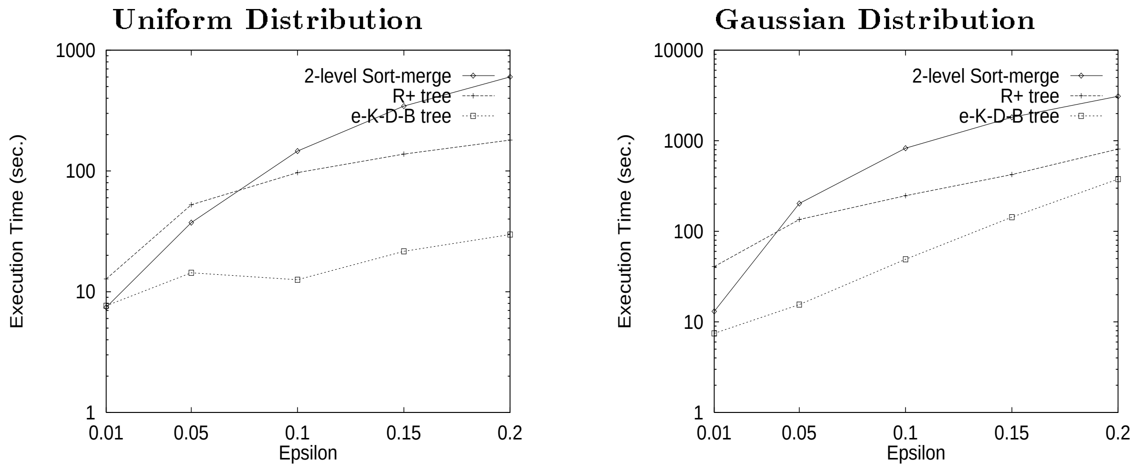


Figure 11: Performance on Synthetic Data: ϵ Value

The rectangles with solid line represents the MBR of a leaf node and the dashed lines the extended bounding regions. This difference in the regions covered by the extended regions may result in a slightly different number of intersecting leaf nodes for a given a leaf node. However, in the R-tree family of spatial indices, the selection query is usually represented by rectangles to reduce the cost of traversing the index. Thus, the extended bounding rectangles to be used to traverse the index for both L_1 and L_2 become the same as that for L_∞.

5.3 Results on Synthetic Data

ϵ value. Figure 11 shows the results of varying ϵ from 0.01 to 0.2, for both uniform and gaussian data distributions. L_2 is used as distance metric. We did not explore the behavior of the algorithms for ϵ greater than 0.2 since the join result becomes too large to be meaningful. Note that the execution times are shown on a log scale. The ϵ-kd tree algorithm is typically around 2 to 20 times faster than the other algorithms. For low values of ϵ (0.01), the 2-level sort-merge algorithm is quite effective. In fact, the sort-merge algorithm and the ϵ-kd algorithm do almost the same actions, since the ϵ-kd will only have around 2 levels (excluding the root). For the gaussian distribution, the performance gap between the ϵ-kd tree and the R^+ tree narrows for high values of ϵ because the join result is very large.

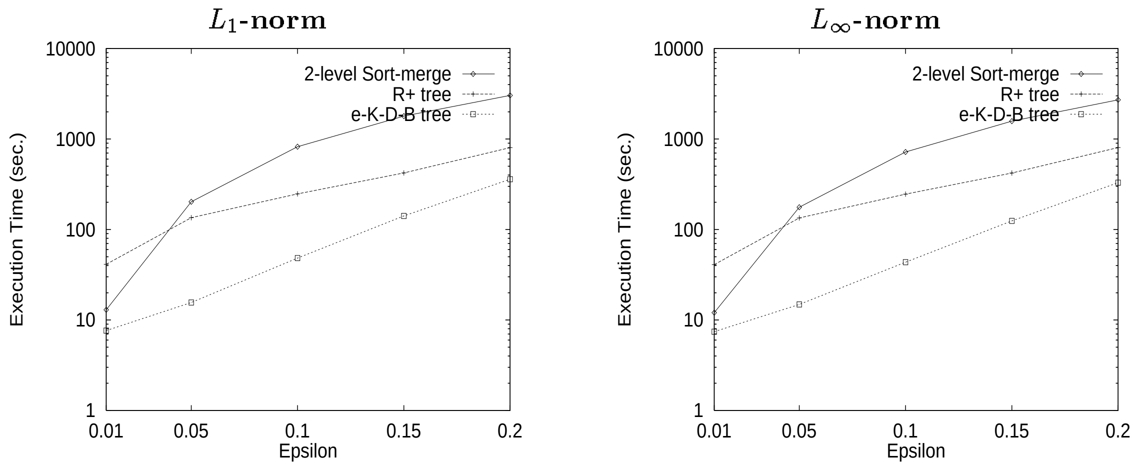


Figure 12: Performance: Distance Metrics (Gaussian Distribution)

Distance Metric. Figure 12 shows the results of varying ϵ for the L_1 and L_∞ norms for the gaussian distribution. The results for the same datasets for the L_2 norm were shown in Figure 11. The relative performance of the algorithms is almost identical for the three distance metrics. Although not shown, we obtained similar results for the uniform distribution, and in our other experiments as well. Hence we only show the results for the L_2-norm in the remaining experiments.

Number of Dimensions. Figure 13 shows the results of increasing the number of dimensions from 4 to 28. Again, the execution times are shown using a log scale. The ϵ-kd algorithm is around 5 to 19 times faster than the sort-merge algorithm. For 8 dimensions or higher, it is around 3 to 47 times faster than the R^+ tree, the performance gap increasing with the number of dimensions. For 4 dimensions, it is only slightly faster, since there are enough points for the ϵ-kd tree to be filled in all dimensions.

For the R^+ tree, increasing the number of dimensions increases the overhead of traversing the index, as well as the number of neighboring leaf nodes and the cost of screening them. Hence the time increases dramatically when going from 4 to 28 dimensions.[4] Even the sort-merge algorithm performs better than the R^+ tree at higher dimensions. In contrast, the execution time for the ϵ-kd remains roughly constant as the number of dimensions increases.

Number of Points. To see the scale up of ϵ-kd tree, we varied the number of points from 10,000 to 1,000,000. The results are shown in Figure 14. For R^+ tree, we do not show results for 1,000,000 points because the tree no longer fit in main memory. None of the algorithms have linear scale-up; but the sort-merge algorithms has somewhat worse scaleup than the other two algorithms. For the gaussian distribution, the performance advantage of the ϵ-kd tree compared to the R^+ tree remains fairly constant (as a percentage). For the uniform distribution, the relative performance advantage

[4]The dip in the R^+ tree execution time when going from 4 to 8 dimension for the gaussian distribution is because of the decrease in join result size. This effect is also noticeable for the ϵ-kd tree, for both distributions.

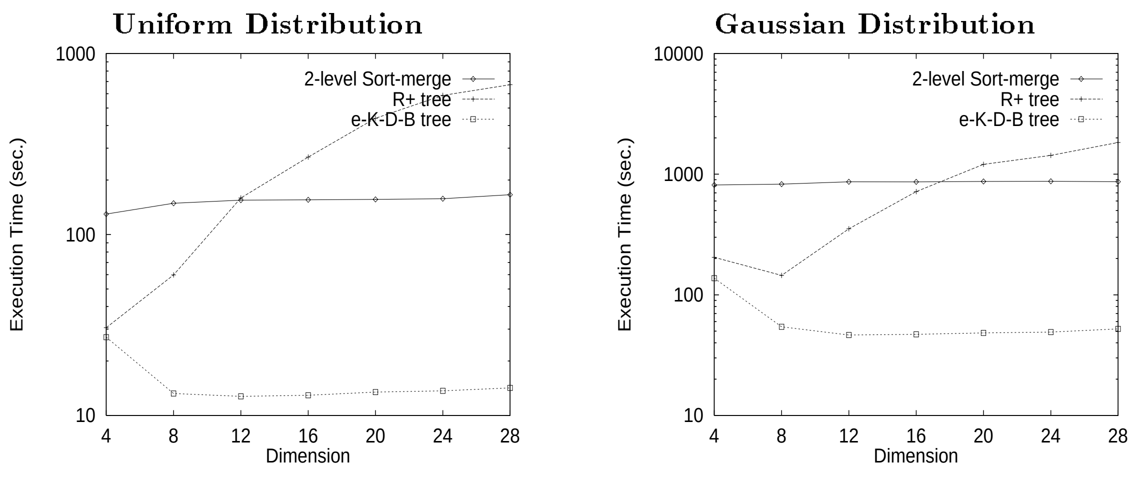


Figure 13: Performance on Synthetic Data: Number of Dimensions

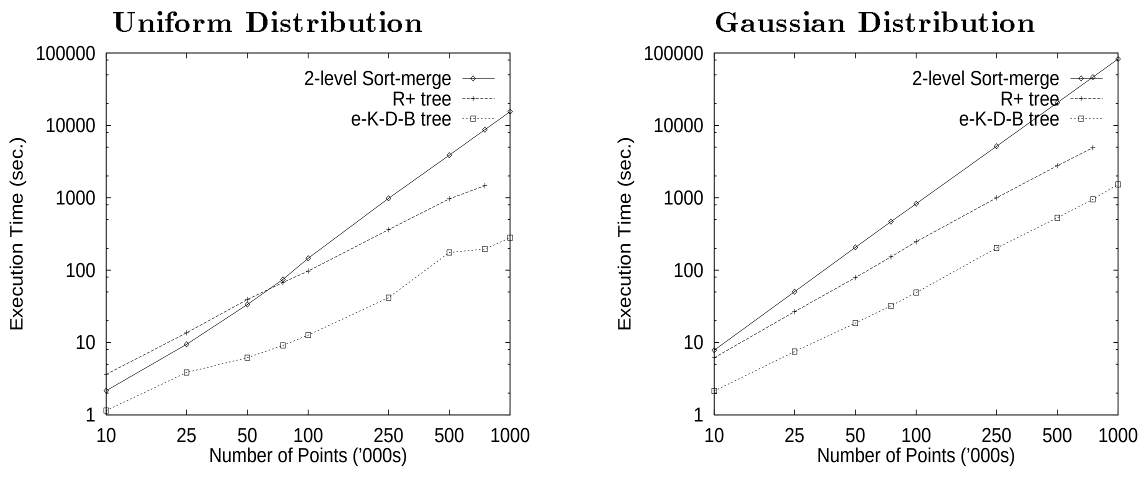
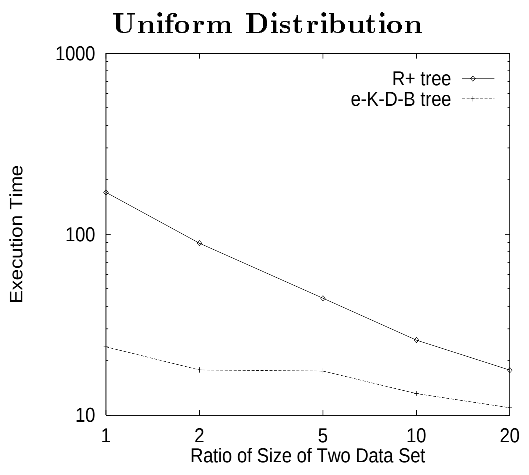


Figure 14: Performance on Synthetic Data: Number of Points

of the ϵ-kd tree varies since the average depth of the ϵ-kd tree does not increase gradually as the number of points increases. Rather, it jumps suddenly, from around 3 to around 4, etc. These transitions occur between 20,000 and 50,000 points, and between 500,000 and 750,000 points.

Non-self-joins. Figure 15 shows the execution times for a proximity join between two different datasets (generated with different random seeds). The size of one of the datasets was fixed at 100,000 points, and the size of the other dataset was varied from 100,000 points down to 5,000 points. For experiments where the second dataset had 10,000 points or fewer, each experiment was run 5 times with different random seeds for the second dataset and the results averaged. With both datasets at 100,000 points, the performance gap between the R^+ tree and the ϵ-kd tree is similar to that on a self-join with 200,000 points. As the size of the second dataset decreases, the performance gap also decreases. The reason is that the time to build the index is included for the ϵ-kd tree, but not for the R^+ tree.



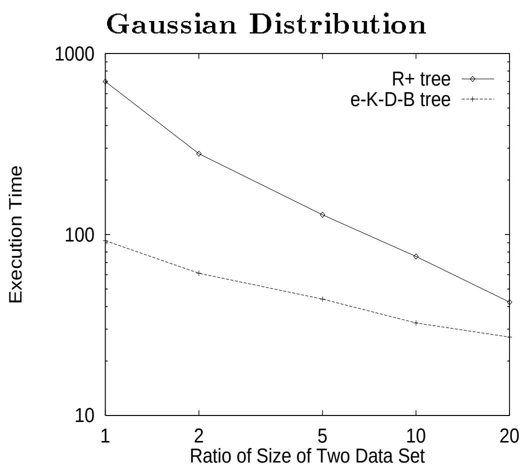
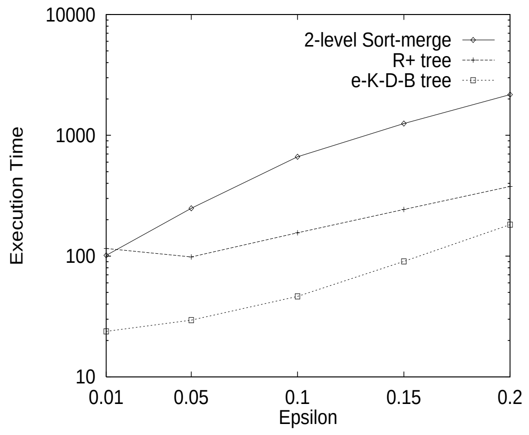


Figure 15: Non-self-joins



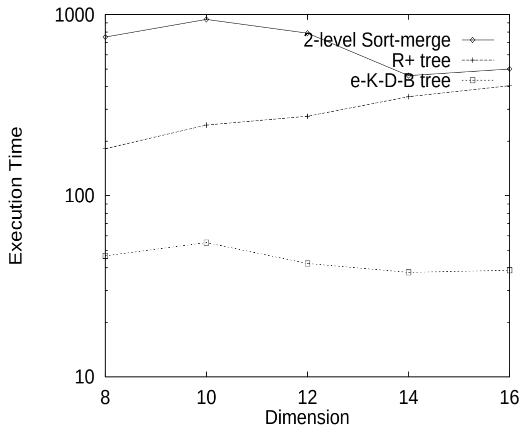


Figure 16: Performance on Mutual Fund Data

5.4 Experiment with a Real-life Data Set

We experimented with the following real-life dataset.

Similar Time Sequences Consider the problem of finding similar time sequences. The algorithm proposed in [ALSS95] first finds similar "atomic" subsequences, and then stitches together the atomic subsequence matches to get similar subsequences or similar sequences. Each sequence is broken into atomic subsequences by using a sliding window of size w. The atomic subsequences are then mapped to points in a w-dimensional space. The problem of finding similar atomic subsequences now corresponds to the problem of finding pairs of w-dimensional points within ϵ distance of each other, using the L_∞ norm. (See [ALSS95] for the rationale behind this approach.)

The time sequences in our experiment were the daily closing prices of 795 U.S. mutual funds, from Jan 4, 1993 to March 3, 1995. There were around 400,000 points for the experiment (since each sequence is broken using a sliding window). The data was obtained from the MIT AI Laboratories' Experimental Stock Market Data Server (http://www.ai.mit.edu/stocks/mf.html). We varied the window size (i.e. dimension) from 8 to 16 and ϵ from 0.05 to 0.2. Figure 16 shows the resulting execution times for the three algorithms. The results are quite similar to those obtained on the synthetic dataset, with the ϵ-kd tree outperforming the other two algorithms.

5.5 Number of join and screen tests

Figure 17 shows the number of join tests and screen counts for the 3 algorithms. In general, the R^+ tree has fewer join tests than the ϵ-kd tree, but considerably more screen tests.

Notice that the relative curves for the join tests for the ϵ-kd and sort-merge, and the screen tests for the R^+ tree, are very similar to the execution times shown in Section 5.4.

The relative number of join and screen tests for the R^+ tree, and the join tests ϵ-kd tree are also as predicted by the analysis in Section 4. In particular, consider the graphs showing the numbers of tests while varying the number of dimensions. At higher dimensions, the R^+ tree has a lot more screen tests than join tests. The gap decreases at lower dimensions as the R^+ trees "fills out" the space. Further, the number of join tests for the ϵ-kd tree is independent of the number of dimensions.

As the number of points increases, the number of join tests for the ϵ-kd tree increases smoothly for the gaussian data, since the average height of the tree increases smoothly. For uniform data, the number of join tests actually decreases when going from 25,000 to 50,000 points and from 500,000 to 750,000 points. The reason is that the average depth of the ϵ-kd tree jumps from around 3 (excluding the root) to around 4 and from 4 to 5, decreasing the number of join tests.

5.6 Summary

The ϵ-kd tree was typically 2 to 47 times faster than the R^+ tree on self-joins, with the performance gap increasing with the number of dimensions. It was typically 2 to 20 times faster than the sort-merge. The 2-level sort-merge was usually slower than R^+ tree. But for high dimensions (> 15) or low values of ϵ (0.01), it was faster than the R^+ tree.

For non-self-joins, the results were similar when the datasets being joined were not of very different sizes. For datasets with different sizes (e.g. 1:10 ratio), the ϵ-kd tree was still faster than the R^+ tree. But the performance gap narrowed since we include the build time for the ϵ-kd tree, but not for the R^+ tree.

The distance metric did not significantly affect the results: the relative performance of the algorithms was almost identical for the L_1, L_2 and L_∞ norms.

6 Biased R-trees

We showed that the ϵ-kd is a fast data structure for high-dimensional proximity joins. Since the R-tree family is a very popular index structure, and many commercial systems have implemented

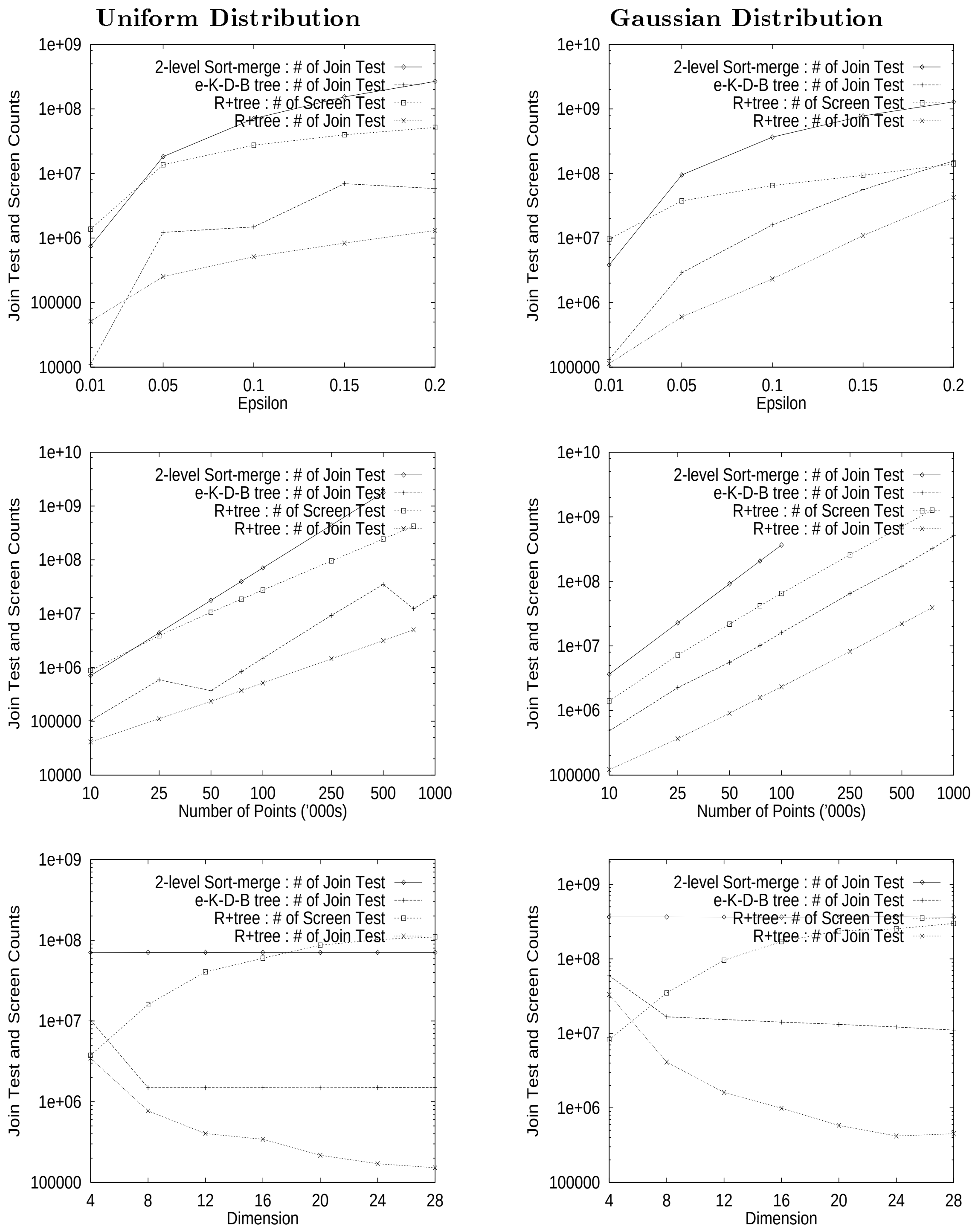


Figure 17: Join Test and Screen Counts

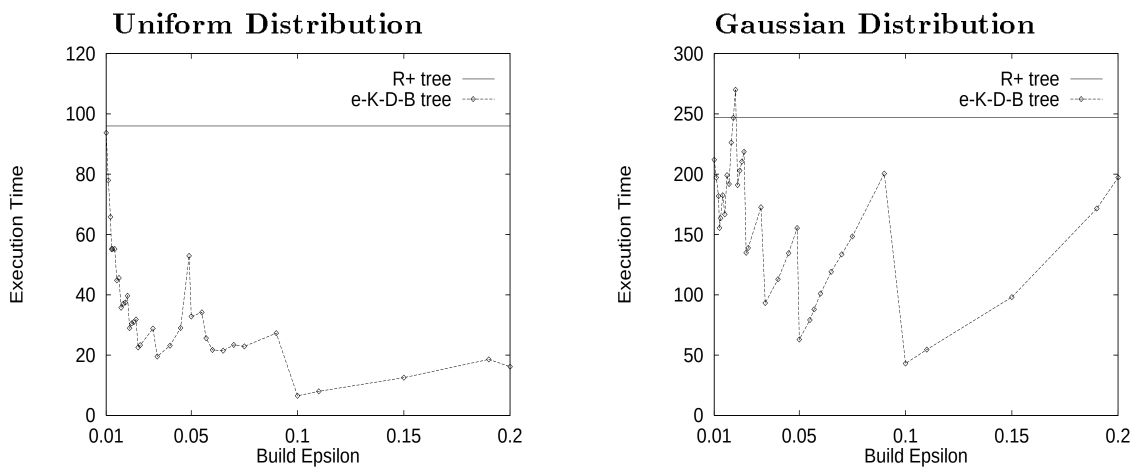


Figure 18: Effect of using different build ϵs

R trees, we decided to explore the possibility of incorporating some the ideas used in the ϵ-kd tree to the R tree family. Although we look at the R^+ tree in this section, we expect the results to also apply to other members of the R tree family.

So far, we built the ϵ-kd tree on the fly, as required. However, this is not an option for the R^+ tree since its build time is much higher. We first examine the performance of the ϵ-kd tree when built with a different ϵ value than the ϵ value used for the proximity join, to see if it still maintains its performance advantage.

6.1 Using different build and join ϵ values for the ϵ-kd tree

We looked at the result of building the ϵ-kd tree with one ϵ value, say the *build* ϵ, and doing the join with a different ϵ, say the *join* ϵ. Figure 18 shows the execution time for a fixed *join* ϵ of 0.1, for different *build* ϵ values. The horizontal line shows the time for the $R+$ tree for the same join. As expected, the best performance for the ϵ-kd tree occurs when the build ϵ is the same as the join ϵ. However, its performance is better than that for the $R+$ tree throughout a wide range of build ϵ values. There are several effects which impact the shape of the graph.

Build ϵ > 0.1 As the build ϵ increases, the number of join test increases, since the volume of the space in adjacent buckets increases. This results in a gradual increase in the overall execution time, for both distributions.

Build ϵ < 0.1 For build ϵ from 0.1 to around 0.025, the dominant factor is again the number of join tests. Assume the depth each leaf node is k. When both build and join ϵ are 0.1, each leaf node has $3^k - 1$ neighbors within ϵ distance. However, when the build ϵ value is a bit smaller than 0.1 (e.g. 0.099), there are $5^k - 1$ neighbors within ϵ distance for each leaf node. Since the number of points in each bucket is almost the same whether the build ϵ is 0.1 or 0.099, the total number of join tests shoots up when the build ϵ is decreased slightly from 0.1. As the build ϵ decreases further, the number of points in each leaf node decreases, while the number of neighbors stays the same. Hence the number of join tests, and the total execution time decrease from build ϵ of 0.099 through 0.05. When the build ϵ decreases from 0.05 to 0.049, the number of neighbors jumps again, from $5^k - 1$ to $7^k - 1$, and the cycle continues. For the uniform distribution, the average height of the tree decreases abruptly by around 1 level when the build ϵ is around 0.06. Hence the number of join tests increases. For the gaussian distribution, there are no abrupt changes in the level of the tree. Hence this pattern is clearly visible from 0.1 to around 0.02.

Build $\epsilon \ll 0.1$ For build ϵ below 0.02, the dominant factor is the traversal cost. For the uniform distribution, the number of leaf nodes increases threefold as the build ϵ drops from 0.02 to 0.01. More importantly, the number of neighbors to look at increases from around 10^3 to 20^3. Though the number of join tests does not change dramatically, the overhead of making function calls while traversing the tree increases the overall time. For the uniform distribution, this effect is mitigated by the fact that the average height of the tree comes down as the build ϵ decreases from 0.02 to 0.015. Hence the overall time actually comes down before increasing again.

6.2 Biased Splitting for the R^+ tree

Since the ϵ-kd tree retains its performance advantage even if it is built with a different ϵ than the join ϵ, we explore using ϵ as a parameter for building the R^+ tree. We apply a *biased splitting* heuristic to the R^+ tree: the same dimension is selected repeatedly for splitting as long as the length of this dimension in the MBR is at least 2ϵ. Traditional heuristics are used to decide the split point in the split dimension. If the length of the current split dimension is less than 2ϵ, the split heuristic moves to the next dimension cyclically. We call this modified R^+ tree the *biased R^+ tree.*

We compared two versions of R^+-tree, one with a traditional splitting algorithm (that is, *unbiased* splitting) and the other with biased splitting algorithm.

Figure 19 shows that the average number of leaf nodes within ϵ distance for the R^+ tree and the biased R^+ tree, as the number of dimensions increases. Biased splitting results in around 5 to 25 times fewer intersecting pages, with the ratio increasing with the number of dimensions.

Figure 20 shows the execution times for the biased R^+ tree. As expected, the performance is mid-way between the performance of the R^+ tree and the ϵ-kd tree. There are two main reasons for the biased R^+ tree remaining slower than the ϵ-kd tree. First, the traversal cost is much higher for the biased R^+ tree since the extended regions for each leaf page still have to be checked against the MBRs in internal nodes. Second, the biased splitting heuristic stops splitting at 2ϵ, compared to ϵ for the ϵ-kd tree. We obtained similar results when varying ϵ and the number of points.

Since the biased R^+ tree only uses a few dimensions for splitting at each level, it need not store

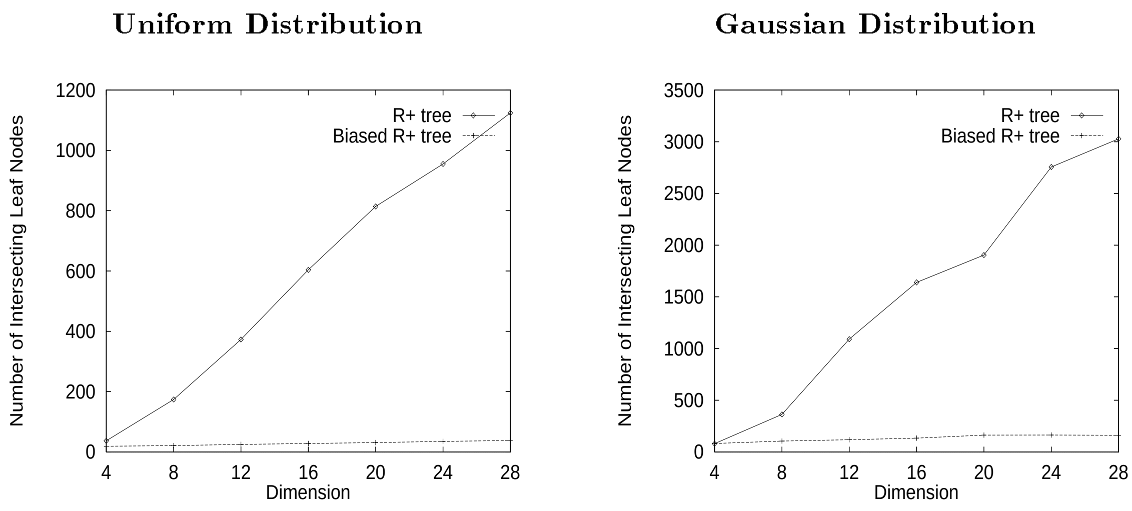


Figure 19: Average Number of Intersecting Leaf Pages

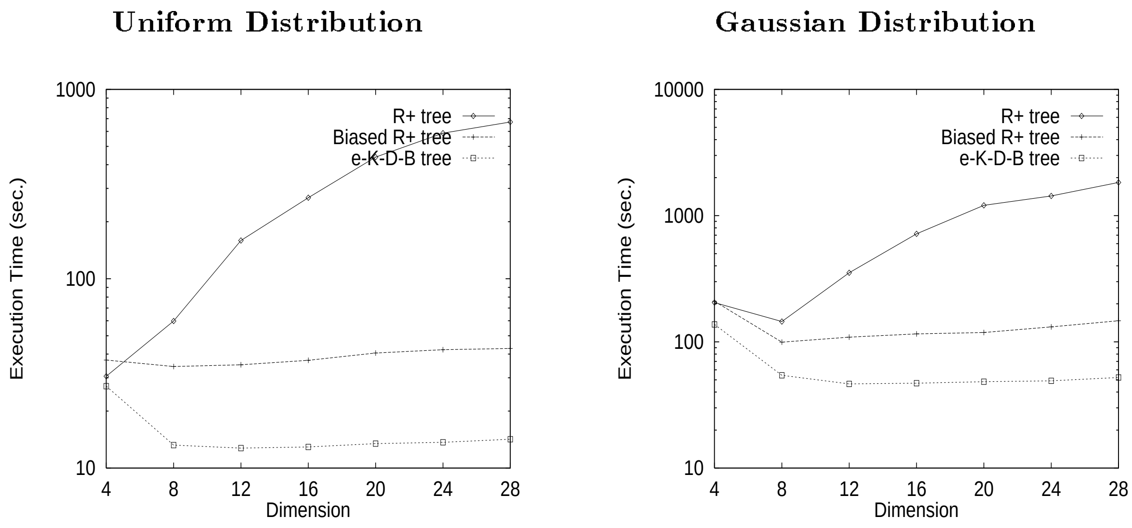


Figure 20: Performance of Biased R^+ trees

all the dimensions for the MBRs at the higher levels of the tree. This optimization, similar to (and inspired by) that in the TV-tree, should reduce both the storage cost and traversal cost. However, it will still not beat the ϵ-kd tree since traversal cost will not drop to the level of the ϵ-kd tree.

7 Conclusions

We presented a new algorithm and a data structure, called the ϵ-kd tree, for fast spatial proximity joins on high-dimensional points. Such proximity joins are needed in many emerging data mining applications. The new data structure reduces the number of neighbor leaf nodes that are considered for the join test, as well as the traversal cost of finding appropriate branches in the internal nodes. The storage cost for internal nodes is independent of the number of dimensions. Hence it scales to high-dimensional data.

We analyzed the number of join and screen tests for the ϵ-kd tree and the R^+ tree, and showed that the ϵ-kd tree will perform considerably better for high-dimensional points. The analytical results were confirmed by empirical evaluation using synthetic and real-life datasets. Proximity joins using the ϵ-kd tree were typically 2 to 40 times faster than with the R^+ tree on these datasets, with the performance gap increasing with the number of dimensions.

Given the popularity of the R-tree family of index structures, we have also studied how the ideas of the ϵ-kd tree can be grafted to the R-tree family. We found that the resulting biased R-tree performs much better than the R-tree for high-dimensional proximity joins, but the ϵ-kd tree still did better.

References

[ACF$^+$93] Manish Arya, William Cody, Christos Faloutsos, Joel Richardson, and Arthur Toga. QBISM: A prototype 3-d medical image database system. *IEEE Data Engineering Bulletin*, 16(1):38–42, March 1993.

[AFS93] Rakesh Agrawal, Christos Faloutsos, and Arun Swami. Efficient similarity search in sequence databases. In *Proc. of the Fourth Int'l Conference on Foundations of Data Organization and Algorithms*, Chicago, October 1993. Also in Lecture Notes in Computer Science 730, Springer Verlag, 1993, 69–84.

[ALSS95] Rakesh Agrawal, King-Ip Lin, Harpreet S. Sawhney, and Kyuseok Shim. Fast similarity search in the presence of noise, scaling, and translation in time-series databases. In *Proc. of the 21st Int'l Conference on Very Large Databases*, pages 490–501, Zurich, Switzerland, September 1995.

[Ben75] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communication of ACM*, 18(9), 1975.

[BKK96] S. Berchtold, D.A. Kiem, and H. Kriegel. The x-tree: An index structure for high-dimensional data. In *Proc. of the 22nd Int'l Conference on Very Large Databases*, Bombay, India, September 1996.

[BKSS90] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R^*-tree: an efficient and robust access method for points and rectangles. In *Proc. of the ACM SIGMOD Conference on Management of Data*, pages 322–331, Atlantic City, NJ, May 1990.

[FL95] Christos Faloutsos and King-Ip Lin. Fastmap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In *Proc. of the ACM SIGMOD Conference on Management of Data*, pages 163–174, San Jose, CA, June 1995.

[Gut84] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proc. of the ACM SIGMOD Conference on Management of Data*, pages 47–57, Boston, Mass, June 1984.

[Jag94] H. V. Jagadish. A retrieval technique for similar shapes. In *Proc. of the ACM SIGMOD Conference on Management of Data*, pages 208–217, Denver, May 1994.

[LJF94] King-Ip Lin, H. V. Jagadish, and Christos Faloutsos. The TV-Tree: An index structure for high-dimensional data. *VLDB Journal*, 3(4):517–542, 1994.

[LR94] Ming-Ling Lo and Chinya V. Ravishankar. Spatial joins using seeded trees. In *Proc. of the ACM SIGMOD Conference on Management of Data*, pages 209–220, May 1994.

[LS09] D. Lomet and B. Salzberg. The hB-tree: A multiattribute indexing method with good guaranteed performance. *ACM Transactions on Database Systems*, 15(4), 1909.

[NBE+93] Wayne Niblack, Ron Barber, Will Equitz, Myron Flickner, Eduardo Glasman, Dragutin Petkovic, Peter Yanker, Christos Faloutsos, and Gabriel Taubin. The qbic project: Querying images by content using color, texture and shape. In *SPIE 1993 Int'l Symposium on Electronic Imaging: Science and Technology, Conference 1908, Storage and Retrieval for Image and Video Databases*, Feb 1993. Also available as IBM Reseach Report RJ 9203 (81511), Feb 1, 1993, Computer Science.

[NC91] A Desai Narasimhalu and Stavros Christodoulakis. Multimedia information systems: the unfolding of a reality. *IEEE Computer*, 24(10):6–8, Oct 1991.

[NHS84] J. Nievergelt, H. Hinterberger, and K.C. Sevcik. The grid file: an adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, 9(1):38–71, 1984.

[PD96] Jignesh M. Patel and David J. DeWitt. Partition Based Spatial-Merge Join. In *Proc. of the ACM SIGMOD Conference on Management of Data*, pages 259–270, Montreal, Canada, June 1996.

[Rob81] J. T. Robinson. The k-D-B-tree: A search structure for large multidimensional dynamic indexes. In *Proc. of the ACM SIGMOD Conference on Management of Data*, pages 10–18, Ann Arbor, MI, April 1981.

[Sam89] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1989.

[SRF87] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R^+ tree: a dynamic index for multi-dimensional objects. In *Proc. 13th Int'l Conference on Very Large Databases*, pages 507–518, Brighton, England, 1987.

[TBS90] A. W. Toga, P. K. Banerjee, and E. M. Santori. Warping 3d models for interbrain comparisons. *Neurosc. Abs. 16:247*, 1990.

[Vas93] Dimitris Vassiliadis. The input-state space approach to the prediction of auroral geomagnetic activity from solar wind variables. In *Int'l Workshop on Applications of Artificial Intelligence in Solar Terrestrial Physics*, Sept 1993.