High-dimensional Similarity 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 similarity join between points in a high-dimensional domain. We present a new algorithm that utilizes a new index structure, called the ϵ-kdB tree, for fast spatial similarity joins on high-dimensional points. This index 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 index structure scales to high-dimensional data. Empirical evaluation, using synthetic and real-life datasets, shows that similarity join using the ϵ-kdB tree is 2 to an order of magnitude faster than the R^+ tree, with the performance gap increasing with the number of dimensions.

1 Introduction

Many emerging data mining applications require efficient processing of similarity joins on high-dimensional points. Examples include applications in time-series databases[1, 2], multimedia databases [9, 14, 13], medical databases [3, 21], and scientific databases[22]. 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 [2] 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 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 [7] 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 [18, 8] 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 similarity 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 similarity joins, we propose a structure

*Currently at Bell Laboratories, Murray Hill, NJ.

called the ϵ-kdB tree. This is a main-memory data structure optimized for performing similarity joins. The ϵ-kdB tree also has a very small build time. This lets the ϵ-kdB tree use the similarity distance limit ϵ as a parameter in building the tree. Empirical evaluation shows that the build plus join time for the ϵ-kdB tree is typically 3 to 35 times less than the join time for the R^+ tree [19],[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 extend the join algorithm to handle large amount of data while still using the ϵ-kdB tree.

Problem Definition We will consider two versions of the spatial similarity 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.
- **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

$$L_p = \left(\sum_1^n |X_i - Y_i|^p \right)^{1/p}, \quad 1 \leq p \leq \infty.$$

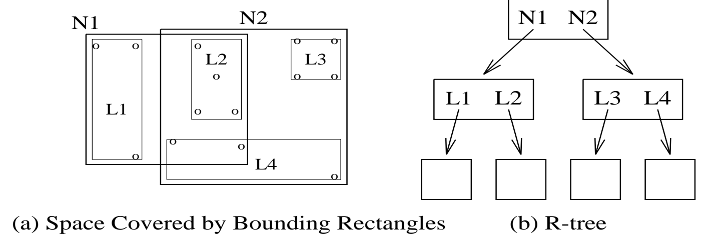
L_2 is the familiar Euclidean distance, L_1 the Manhattan distance, and L_∞ corresponds to the maximum distance in any dimension.

Paper Organization. In Section 2, we give an overview of existing spatial indices, and describe their shortcomings when used for high-dimensional similarity joins. Section 3 describes the ϵ-kdB tree and the algorithm for similarity joins. We give a performance evaluation in Section 4 and conclude in Section 5.

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 similarity joins. We also give a brief overview of other indices. We then discuss inadequacies of the current index structures.

[1] Our experiments indicated that the R^+ tree was better than the R tree [8] or the R^* tree [4] tree for high-dimensional similarity joins.



(a) Space Covered by Bounding Rectangles (b) R-tree

Figure 1. Example of an R-tree

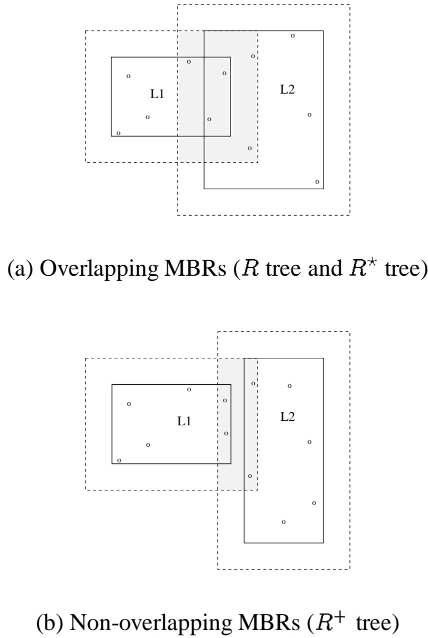
2.1 The R-tree family

R-tree [8] 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 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** [4] 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** [19] 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 [6] 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 [6] 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.



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

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

Figure 2. Screening points for join test

Similarity 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 extended boundaries. The shaded area contains screened points.

2.2 Other Index Structures

kdB tree [17] 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 [12] is similar to the kdB tree except that bounding rectangles of the children of an internal node are organized as a K-D tree [5] 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.

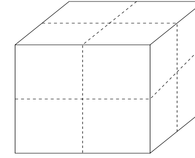


Figure 3. Number of neighboring leaf nodes.

TV-tree [10] 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 [15] 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 following inadequacies for performing similarity joins with high-dimensional points:

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 [11] (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 [16].

Skewed Data. Handling skewed data is not a problem for most current indices except 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 similarity 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 index, we can attack these problems. The problem of high-dimensional similarity joins with some distance metric and ϵ parameter has the following properties:

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

We now describe a new index structure, ϵ-kdB tree, which is a special-purpose index for this purpose.

3 The ϵ-kdB tree

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

3.1 ϵ-kdB tree definition

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

ϵ-kdB tree We assume, without loss of generality, that the co-ordinates of the points in each dimension lie between 0 and +1. We start with a single leaf node. For better space utilization, pointers to the data points are stored in leaf nodes. 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-kdB tree for two dimensional space is shown in Figure 4.

Note that for any interior node x, the points in a child y of x will not join with any points in any of the other children of x, except for the 2 children adjacent to y. This holds for any of the L_p distance metrics. Thus the same join code can be used for these metrics, with only the final test between a pair of points being metric-dependent.

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

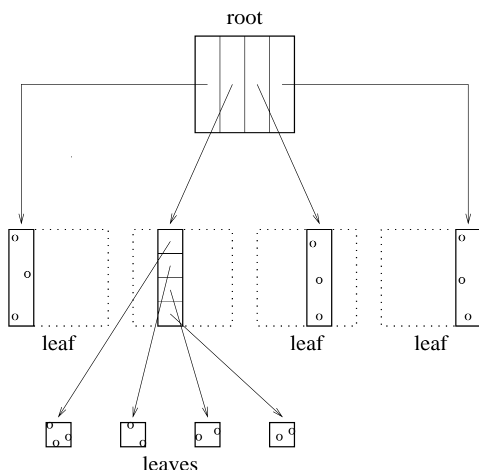


Figure 4. ϵ-kdB tree

Similarity Join using the ϵ-kdB tree Let x be an internal node in the ϵ-kdB 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-kdB 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.

Memory Management The value of ϵ is often given at run-time. Thus, since the value of ϵ is a parameter for building the index, it may not be possible to build a disk-based version of the index 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 $2 * \epsilon$ 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 $2 * \epsilon$, builds the ϵ-kdB tree for those points in main memory, and performs the similarity 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 $2 * \epsilon$ and $3 * \epsilon$, build the ϵ-kdB tree for these points, and performs the join procedure again. This procedure is continued until all the points have been processed. Note that we only read each point off the disk once.

```
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

This procedure works because the build time for the ϵ-kdB tree is extremely small. It can be generalized to the case where a $2 * \epsilon$ 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-kdB , perform join and so on) is instead repeated for each $4 * \epsilon^2$ chunk of the data using the additional dimension.

3.2 Design Rationale

Two distinguishing features of ϵ-kdB tree are:

- *Biased Splitting* : The dimension used in previous split is selected again for splitting as long as the length of the dimension in the bounding rectangle of each resulting leaf node is at least ϵ.

- ϵ *Sized Splitting* : When we split a node, we split the node in ϵ sized chunks.

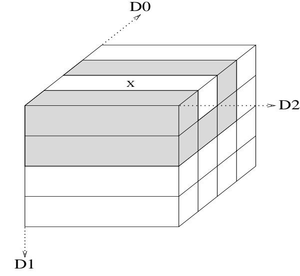
We discuss below how these features help ϵ-kdB tree solve the problems with current indices outlined in Section 2.

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-kdB 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 we split in one dimension as long as the length of the bounding rectangle of each resulting children is at least ϵ, and then start splitting in the next dimension. When a leaf node becomes full, we split the node into several children, each of size ϵ in the split dimension at once, rather than gradually, in order to reduce the build time.

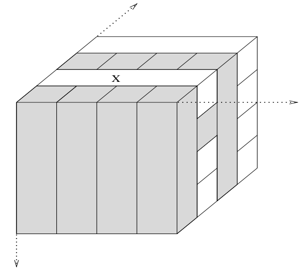
We have two alternatives for choosing the next splitting dimension: global ordering and local ordering. Global ordering uses the same split dimension for all the nodes in the same level, while local ordering chooses the split dimension based on the distribution of points in each node. Examples of these two cases are shown in Figure 6, 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 splitting dimensions, rather than local ordering.

When the number of points are so huge that the ϵ-kdB 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.



(a) Choosing splitting dimension globally



(b) Choosing splitting dimension locally

Figure 6. Global and Local Ordering of Splitting Dimensions

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 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-kdB tree will handle skewed data reasonably.

4 Performance Evaluation

We empirically compared the performance of the ϵ-kdB 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 4.1, and the datasets used in experiments in Section 4.2. Next, we show the performance of the algorithms on synthetic and real-life datasets in Sections 4.3 and 4.4 respectively.

4.1 Algorithms

ϵ-kdB tree. We implemented the ϵ-kdB 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-kdB tree include the I/O cost of reading an external sorted file containing the data points, as well as the cost of building the index. 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 similarity 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 similarity join, resulting in faster execution time.) We therefore used R^+ tree for our experiments. We used a page size of 4096 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 1 summarizes the costs included in the execution times for each algorithm.

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

Table 1. Costs included in the execution times.

4.2 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 2 shows the parameters for the datasets, along with their default values and the range of values for which we conducted experiments.

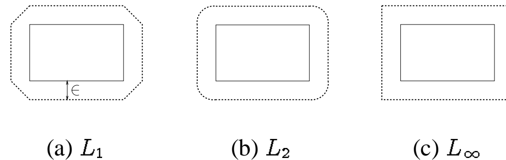
Parameter	Default Value	Range of Values
Number of Points	100,000	10,000 to 1 million
Number of Dimensions	10	4 to 28
ϵ (join distance)	0.1	0.01 to 0.2
Range of Points	-1 to +1	-same-
Distance Metric	L_2-norm	L_1, L_2, L_∞ norms

Table 2. Synthetic Data Parameters

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 7 shows the extended bounding regions for the L_1, L_2 and L_∞ norms. 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_∞.

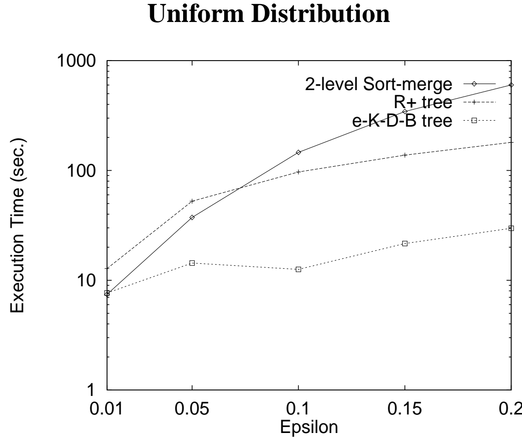
4.3 Results on Synthetic Data

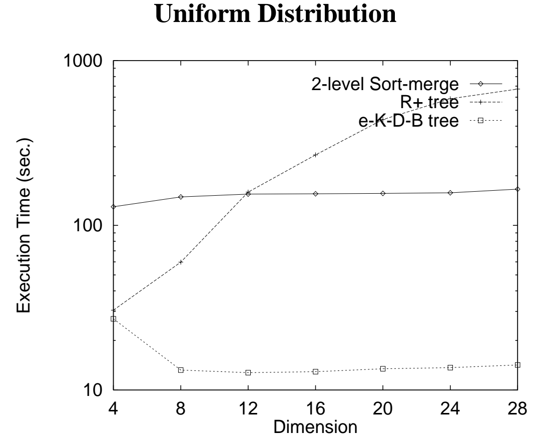
Distance Metric. We first experimented varying ϵ for the L_1, L_2 and L_∞ norms. The relative performance of the

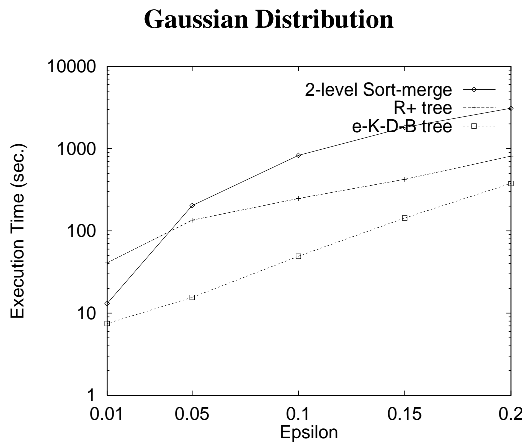


(a) L_1 (b) L_2 (c) L_∞

Figure 7. Bounding Regions extended by ϵ







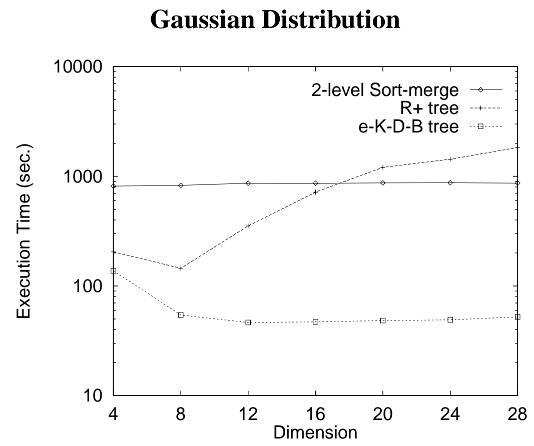


Figure 8. Performance on Synthetic Data: ϵ Value

Figure 9. Performance on Synthetic Data: Number of Dimensions

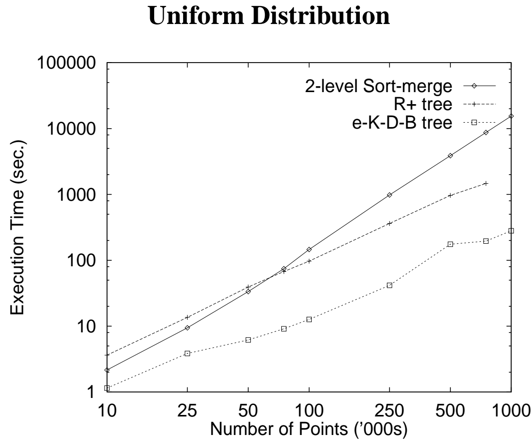
algorithms is almost identical for the three distance metrics (See [20]). We only show the results for the L_2-norm in the remaining experiments.

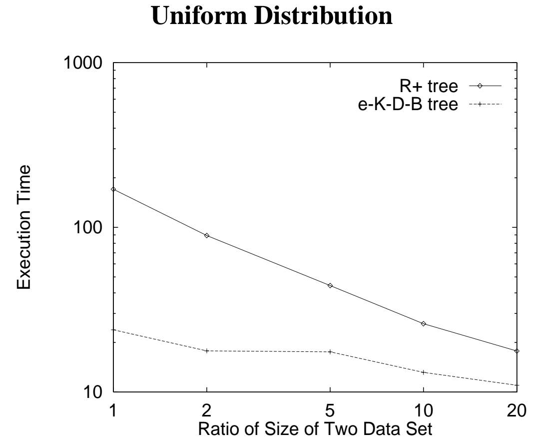
ϵ value. Figure 8 shows the results of varying ϵ from 0.01 to 0.2, for both uniform and gaussian data distributions. L_2 is used as distance metric. We did not explore the behavior of the algorithms for ϵ greater than 0.2 since the join result becomes too large to be meaningful. Note that the execution times are shown on a log scale. The ϵ-kdB 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-kdB algorithm do almost the same actions, since the ϵ-kdB will only have around 2 levels (excluding the root). For the gaussian distribution, the performance gap between the ϵ-kdB tree and the R^+ tree narrows for high values of ϵ because the join result is very large.

Number of Dimensions. Figure 9 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-kdB 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-kdB 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.[3] Even the sort-merge algorithm performs better than the R^+ tree at higher dimensions. In

[3] 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-kdB tree, for both distributions.





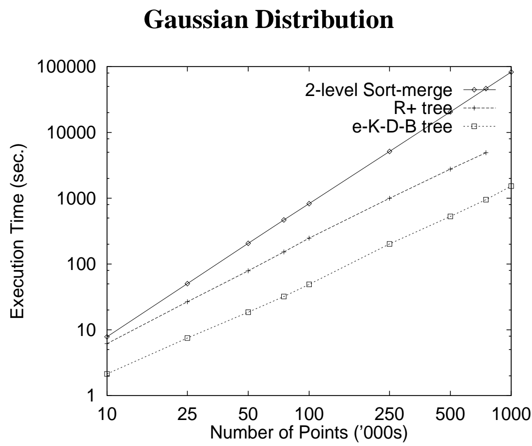


Figure 10. Performance on Synthetic Data: Number of Points

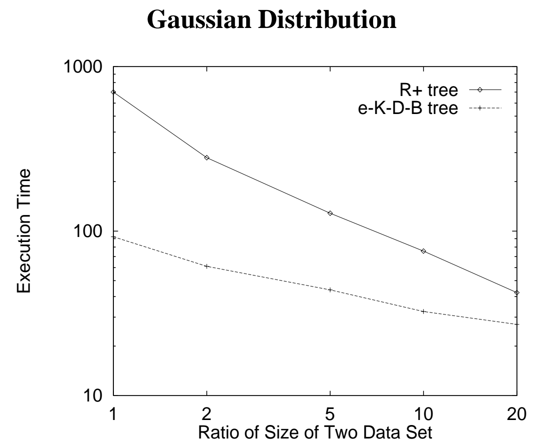


Figure 11. Non-self-joins

contrast, the execution time for the ϵ-kdB remains roughly constant as the number of dimensions increases.

Number of Points. To see the scale up of ϵ-kdB tree, we varied the number of points from 10,000 to 1,000,000. The results are shown in Figure 10. 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-kdB tree compared to the R^+ tree remains fairly constant (as a percentage). For the uniform distribution, the relative performance advantage of the ϵ-kdB tree varies since the average depth of the ϵ-kdB 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 11 shows the execution times for a similarity 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-kdB 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-kdB tree, but not for the R^+ tree.

4.4 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 [2]

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 [2] for the rationale behind this approach.)

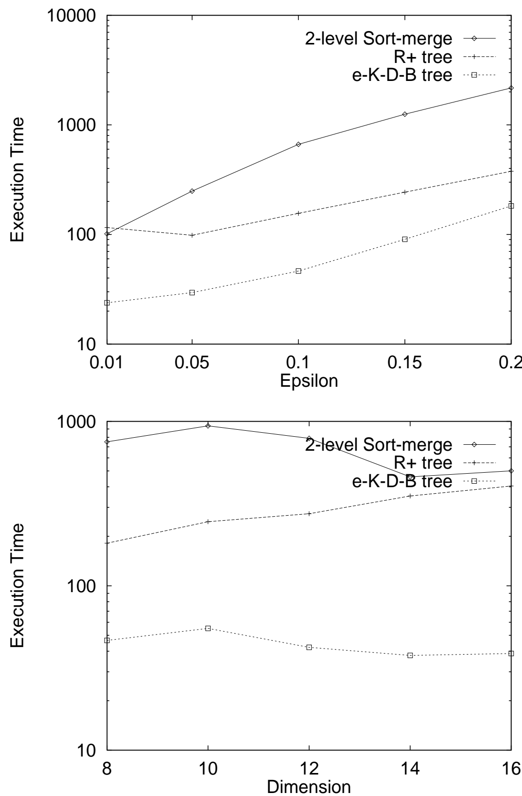


Figure 12. Performance on Mutual Fund Data

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 12 shows the resulting execution times for the three algorithms. The results are quite similar to those obtained on the synthetic dataset, with the ϵ-kdB tree outperforming the other two algorithms.

4.5 Summary

The ϵ-kdB 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-kdB tree was still faster than the R^+ tree. But the performance gap narrowed since we include the build time for the ϵ-kdB 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.

5 Conclusions

We presented a new algorithm and an index structure, called the ϵ-kdB tree, for fast spatial similarity joins on high-dimensional points. Such similarity joins are needed in many emerging data mining applications. The new index 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 studied the performance of ϵ-kdB tree using both synthetic and real-life datasets. The join time for the ϵ-kdB tree was 2 to an order of magnitude less than the join time for the R^+ tree on these datasets, with the performance gap increasing with the number of dimensions. We have also analyzed the number of join and screen tests for the ϵ-kdB tree and the R^+ tree. The analysis showed that the ϵ-kdB tree will perform considerably better for high-dimensional points. This analysis can be found in [20].

Given the popularity of the R-tree family of index structures, we have also studied how the ideas of the ϵ-kdB 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 similarity joins, but the ϵ-kdB tree still did better. The details of this study can be found in [20].

References

[1] R. Agrawal, C. Faloutsos, and A. 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.

[2] R. Agrawal, K.-I. Lin, H. S. Sawhney, and K. 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.

[3] M. Arya, W. Cody, C. Faloutsos, J. Richardson, and A. Toga. QBISM: A prototype 3-d medical image database system. *IEEE Data Engineering Bulletin*, 16(1):38–42, March 1993.

[4] 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.

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

[6] S. Berchtold, D. 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.

[7] C. Faloutsos and K.-I. 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.

[8] 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.

[9] 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.

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

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

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

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

[14] W. Niblack, R. Barber, W. Equitz, M. Flickner, E. Glasman, D. Petkovic, P. Yanker, C. Faloutsos, and G. 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.

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

[16] J. M. Patel and D. 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.

[17] 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.

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

[19] 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.

[20] K. Shim, R. Srikant, and R. Agrawal. The ϵ-kdb tree: A fast index structure for high-dimensional similarity joins. Research Report, IBM Almaden Research Center, San Jose, California, 1996. Available from `http://www.almaden.ibm.com/cs/quest`.

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

[22] D. 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.