2002

Similarity Join for Low- and High-Dimensional Data

Dmitri V. Kalashnikov

Sunil Prabhakar
Purdue University, sunil@cs.purdue.edu

Report Number:
02-017

TR #02-017

Similarity Join for Low- and High- Dimensional Data*

Dmitri V. Kalashnikov and Sunil Prabhakar
CS Dept. Purdue University
West Lafayette, IN 47907, USA
{dvk,sunil}@cs.purdue.edu

Abstract

The efficient processing of similarity joins is important for a large class of applications. The dimensionality of the data for these applications ranges from low to high. Most existing methods have focussed on the execution of high-dimensional joins over large amounts of disk-based data. The increasing sizes of main memory available on current computers, and the need for efficient processing of spatial joins suggest that spatial joins for a large class of problems can be processed in main memory. In this paper we develop two new spatial join algorithms, the Grid-join and EGO-join, and study their performance in comparison to the state of the art algorithm EGO-join and the RSJ algorithm.*

Through evaluation we explore the domain of applicability of each algorithm and provide recommendations for the choice of join algorithm depending upon the dimensionality of the data as well as the critical ε parameter. We also point out the significance of the choice of this parameter for ensuring that the selectivity achieved is reasonable.

1 Introduction

Similarity (spatial) joins are an important database operation for several applications including GIS, multimedia databases, data mining, location-based applications, and time-series analysis. The problem of efficient computation of similarity joins has been addressed by many researchers. Most researchers have focussed their attention on disk-based joins for high-dimensional data. Current high-end workstations have enough memory to handle joins even for large amounts of data. For example, the self-join of 1 million 32-dimensional data points, using an algorithm similar to that of [2] (assuming *float* data type for coordinate and *int* for point identities) requires roughly 132MB of memory (i.e. $(32 \times 4 + 4) \times 10^6 \approx 132$MB, plus memory for stack etc.). Furthermore there are situations when it is necessary to join intermediate results situated in main memory or sensor data, which is to be kept in main memory. With the availability of a large main memory cache, disk-based algorithms may not necessarily be the best choice. Moreover, for certain applications (e.g. moving object environments) near real-time computation may be critical and require main memory evaluation.

In this paper we consider the problem of main memory processing of similarity joins, also known as ε-joins. Given two datasets A and B of d-dimensional points and value $\varepsilon \in \Re$, the goal of a join operation is to identify all pairs of points, R, one from each set, that are within distance ε from each other, i.e. $R = \{(a, b) : \|a - b\| < \varepsilon;\ a \in A,\ b \in B\}$.

While several research efforts have concentrated on designing efficient high-dimensional join algorithms, the question of which method should be used when joining low-dimensional (e.g. 2–6 dimensions) data remains open. This paper addresses this question and investigates the choice of join algorithm for low- and high-dimensional data. We propose two new join algorithms: the *Grid-join* and *EGO*-join*, and evaluate their along with the state of the art algorithm EGO-join [2], and a method which serves as a benchmark in many similar publications, the RSJ join [3].

Although not often addressed in related research, the choice of the ε parameter for the join is critical to producing meaningful results. We have discovered that often in similar research the choice of values of ε yields very small selectivity, i.e. almost no point from one dataset joins with a point from the other dataset. In Section 3.1 we present a discussion on how to choose appropriate values of ε.

The contributions of this paper are as follows:

- Two join algorithms that give better performance (almost an order of magnitude better for low dimensions) than the state of the art EGO-join algorithm.
- Recommendations for the choice of join algorithm based upon data dimensionality d, and ε.

*Portions of this work was supported by an Intel PhD Fellowship, NSF CAREER grant IIS-9985019, NSF grant 0010044-CCR and NSF grant 9972883

- Highlight the importance of the choice of ϵ and the corresponding selectivity for experimental evaluation.
- Highlight the importance of the cache miss reduction techniques: spatial sortings (2.5 times speedup) and clustering via utilization of dynamic arrays (40% improvement).
- For the Grid-join, the choice of grid size is an important parameter. In order to choose good values for this parameter, we develop highly accurate estimator functions for the cost of the Grid-join. These functions are used to choose an optimal grid size.

The rest of this paper is organized as follows. The new Grid-join and EGO*-join algorithms are presented in Section 2. The proposed join algorithms are evaluated in Section 3. Related work is discussed in Section 4. Section 5 concludes the paper.

2 Similarity join algorithms

In this section we introduce two new algorithms: the Grid-join and EGO*-join. The Grid-join is based upon a uniform grid and builds upon the approach proposed in [6]. The EGO*-join is based upon EGO-join proposed in [2]. In Section 2.1 we first present the Grid-join algorithm followed by an important optimization for improving the cache hit-rate. An analysis of the appropriate grid size as well as cost prediction functions for the Grid-join is presented in [5]. The EGO*-join is discussed in Section 2.2.

2.1 Grid-join

Assume for now that we are dealing with 2-dimensional data. The spatial join of two datasets A and B can be computed using a standard Index Nested Loop approach as follows. One of the datasets, say B, is treated as a collection of circles of radius ϵ centered at each point of B. This collection of circles is then indexed using some spatial index structure. The join is computed by taking each point from A and querying the index on the circles to find those circles that contain the query point. Each point (from B) corresponding to each such circle joins with the query point (from A). An advantage of this approach (as opposed to the alternative of building an index on the points of one set and processing a circle region query for each point from the other set) is that point queries are much simpler than region queries and thus tend to be faster. For example, a region query on a quad-tree index might need to evaluate several paths while a point query is guaranteed to be a single path query. An important question is the choice of index structure for the circles.

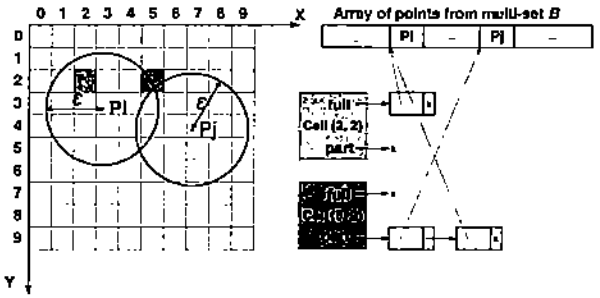

Figure 1. An example of the Grid Index, I_G

In earlier work [6] we have investigated the execution of large numbers of range queries over point data in the context of evaluating multiple concurrent continuous range queries on moving objects. The approach can also be used for spatial join if we compute the join using the Index Nested Loops technique mentioned above. The two approaches differ only in the shape of the queries which are circles for the spatial join problem and rectangles for the range queries.

In [6] the choice of a good main-memory index was investigated. Several key index structures including R-tree, R*-tree, CR-tree [7], quad-tree, and 32-tree [6] were considered. All trees were optimized for main memory. The conclusion of the study was that a *simple one-level Grid-index outperformed all other indexes* by almost an order of magnitude for uniform as well as skewed data. Due to its superior performance, in this study, we use the Grid-index for indexing the ϵ-circles.

The Grid Index While many variations exist, we have designed our own implementation of the Grid-index, which we denote as I_G. I_G is built on circles with ϵ-radius. Notice, it is not necessary to generate a new dataset consisting of these circles. Since each circle has the same radius (ϵ), the dataset of the points representing the centers of these circles is sufficient. The similarity join algorithm which utilizes I_G is called the Grid-join, or J_G for short.

Case of 2 dimensions For ease of explanation assume the case of 2-dimensional data. I_G is a 2-dimensional array of cells. Each cell represents a region of space generated by partitioning the domain using a regular grid.

Figure 1 shows an example of I_G. Throughout the paper, we assume that the domain is normalized to the unit d-dimensional hyper-cube $[0,1]^d$. In this example, the domain is divided into a 10×10 grid of 100 cells, each of size 0.1×0.1.

Since the grid is uniform, it is easy to calculate cell-coordinates of an object in O(1) time. Each cell contains two lists that are identified as $full$ and $part$, as shown in Figure 1. Let $C(p,r)$ denote a circle with center at point p and radius r. The $full$ ($part$) list of a cell contains *pointers* to all points b_i from B such that $C(b_i,\epsilon)$ fully (partially) cover the cell. That is for cell C in I_G

Input: Datasets A, B, and $\varepsilon \in \Re$
Output: Result set R
1. $R \leftarrow \emptyset$
2. z-sort(A)
3. z-sort(B)
4. Initialize I_G
5. **for** $i \leftarrow 0$ **to** $|B| - 1$ **do**
   (a) $b'_i \leftarrow (b^0_i, b^1_i)$
   (b) Insert $\{b_i, C(b'_i, \varepsilon)\}$ into I_G
6. **for** $i \leftarrow 0$ **to** $|A| - 1$ **do**
   (a) $a'_i \leftarrow (a^0_i, a^1_i)$
   (b) Let C_i be the cell in I_G corresponding to a'_i
   (c) **for** $j \leftarrow 0$ **to** $|C.part| - 1$ **do**
      i. $b \leftarrow C_i.part[j]$
      ii. **if** $(\|a_i - b\| < \varepsilon)$ **then** $R \leftarrow R \cup \{(a_i, b)\}$
   (d') **for** $j \leftarrow 0$ **to** $|C_i.full| - 1$ **do**
      i. $b \leftarrow C_i.full[j]$
      ii. $R \leftarrow R \cup \{(a_i, b)\}$
7. **return** R

Figure 2. Grid-join procedure, J_G

its *part* and *full* lists can be represented in set notation as $C.full = \{b : C \subset C(b,\varepsilon);\ b \in B\}$ and $C.part = \{b : C \not\subset C(b,\varepsilon) \wedge C \cap C(b,\varepsilon) \neq \emptyset;\ b \in B\}$.

To find all points within ε-distance from a given point a first the cell corresponding to a is retrieved. All points in $full$ list are guaranteed to be within ε-distance from a. Points in $part$ list need to be post-processed.

The choice of data structures for the $full$ and $part$ lists is critical for performance. We implemented these lists as dynamic-arrays[1] rather than lists which improves performance by roughly 40% due to the resulting clustering (and thereby reduced cache misses).

Case of d dimensions For the general d-dimensional case, 2-dimensional grid is used. The first 2 coordinates of points are used for all operations exactly as in 2-dimensional case except for the processing of $part$ lists, which uses all d coordinates to determine if $\|a - b\| < \varepsilon$.

The reason for two separate lists per cell for 2-dimensional points is that points in the $full$ list do not need potentially costly checks for relevance since they are guaranteed to be within ε-distance. Keeping a separate $full$ list is of little value for more than 2 dimensions since now, similarly to the $part$ list, it too needs post-processing. Therefore only one list is kept for all circles that at least partially intersect the cell in the chosen 2 dimensions. We call this list $part$ list: $C.part = \{b : C \cap C(b',\varepsilon) \neq \emptyset;\ b \in B,\ b' \leftarrow (b^0, b^1)\}$.

[1] A dynamic array is a standard data structure for arrays whose size adjusts dynamically.

J_G is described in Figure 2. Steps 2 and 3, the z-sort steps, apply a spatial sort to the two datasets. The need for this step is explained later. I_G is initialized in Step 4. In the loop in Step 5, all points b_i from set B are added to I_G one by one. First a 2-dimensional point b'_i constructed from the first two coordinates of b_i, is considered. Then pointer to b_i is added to $part$ lists of each cell C in I_G that satisfies $C \cap C(b'_i, \varepsilon) \neq \emptyset$.

The loop in Step 6 performs a nested loop join. For each point a_i in A all points from B that are within ε distance are determined using I_G. To do this, point a'_i is constructed from the first two coordinates of a_i and the cell corresponding to a'_i in I_G, C_i, is determined in Steps 6(a) and 6(b). Then, in Step 6(c), the algorithm iterates though all elements of the $part$ list of cell C_i and finds all relevant to a points. Step 6(d') is analogous to Step 6(c) and valid only for 2-dimensional case.

Choice of grid size The performance of J_G depends on the choice of grid size, therefore it must be selected carefully. Intuitively, the finer the grid the faster the processing but the slower the time needed to initialize the index and load the data into it. Due to limited space we can only present a sketch of our solution for selecting appropriate grid size, please refer to [5] for details.

The first step is to develop a set of estimator functions that predict the cost of the join given a grid size. The cost is composed of three components, the costs of: (a) initializing the empty grid; (b) loading the dataset B into the index; and (c) processing each point of dataset A through this index. In [5] we present details on how each of these costs is estimated. The quality of the prediction of these functions was found to be extremely high. Using these functions, it is possible to determine which grid size would be optimal. These functions can also be used by a query optimizer – for example to evaluate whether it would be efficient to use either J_G for the given parameters or another method of joining data.

Improving the cache hit-rate The performance of main-memory algorithms is greatly affected by cache hit rates. In this section we describe an optimization that improves cache hit rates and, consequently, the overall performance of J_G.

As shown in Figure 2, for each point, its cell is computed, and the $full$ and $part$ lists (or just $part$ list) of this cell are accessed. The algorithm simply processes points in sequential order in the array corresponding to set A. Cache-hit rates can be improved by altering the order in which points are processed. In particular, points in the array should be ordered such that points that are close together according to their first two coordinates in the 2D domain are also close together in the point array. In this situation index data for a given cell is likely to be reused from the cache during the

processing of subsequent points from the array. The speed-up is achieved because such points are more likely to be covered by the same circles than points that are far apart, thus the relevant information is more likely to be retrieved from the cache rather than from main memory.

Sorting the points to ensure that points that are close to each other also tend to be close in the array order can easily be achieved by various methods. We choose to use a sorting based on the Z-order. We sort not only set A but also set B, which reduces the time needed to add circles to I_G. As we will see in Section 3, $\sim$2.5$\times$ speedup is achieved by utilizing Z-sort, e.g. as shown in Figure 13.

2.2 EGO*-join

In this section we present an improvement of the disk-based EGO-join algorithm proposed in [2]. We dub the new algorithm the EGO*-join. We use notation J_{EGO} for the EGO-join procedure and J_{EGO*} for the EGO*-join procedure. According to [2], the state of the art algorithm J_{EGO} was shown to outperform other methods for joining massive, high-dimensional data.

We begin by briefly describing J_{EGO} as presented in [2] followed by our improvement of J_{EGO}.

The Epsilon Grid Order: J_{EGO} is based on the so called Epsilon Grid Ordering (EGO), see [2] for details. In order to impose an EGO on dataset A, a regular grid with the cell size of ε is laid over the data space. The grid is imaginary, and never materialized. For each point in A, its cell-coordinate can be determined in O(1) time. A lexicographical order is imposed on each cell by choosing an order for the dimensions. The EGO of two points is determined by the lexicographical order of the corresponding cells that the points belong to.

Input: Datasets A, B, and $\varepsilon \in \Re$
Output: Result set R
1. EGO-sort(A, ε)
2. EGO-sort(B, ε)
3. join_sequences(A, B)

Figure 3. EGO-join Procedure, J_{EGO}

EGO-sort: In order to perform J_{EGO} of two sets A and B with a certain ε, first the points in these sets are sorted in accordance with the EGO for the given ε. Notice that for a subsequent J_{EGO} operation with a different ε sets A and B need to be sorted again since their EGO values depend upon the cells.

Recursive join: The procedure for joining two sequences is recursive. Each sequence is further subdivided into two roughly equal subsequences and each subsequence is joined recursively with both its counterparts. The partitioning is carried out until the length of both subsequences is smaller than a threshold value, at which point a simple-join is performed. In order to avoid excessive computation, the algorithm avoids joining sequences that are guaranteed not to have any points within distance ε of each other. Such sequences can be termed *non-joinable*.

EGO-heuristic: A key element of J_{EGO} is the heuristic used to identify *non-joinable* sequences. The heuristic is based on the number of inactive dimensions, which will be explained shortly. To understand the heuristic, let us consider a simple example. For a short sequence its first and last points are likely to have the same first cell-coordinates. For example, points with corresponding cell-coordinates $(2,7,4,1)$ and $(2,7,6,1)$ have two common prefix coordinates $(2,7,\times,\times)$. Their third coordinates differ – this corresponds to the *active* dimension, the first two dimensions are called *inactive*. This in turn means that for this sequence all points have 2 and 7 as their first two cell-coordinates – because both sequences are EGO-sorted before being joined.

The heuristic first determines the number of inactive dimensions for both sequences, and computes min – the minimum of the two numbers. It is easy to prove that if there is a dimension between 0 and $min - 1$ such that the cell-coordinates of the first points of the two sequences differ by at least two in that dimension, then the sequences are non-joinable. This is based upon the fact that the length of each cell is ε.

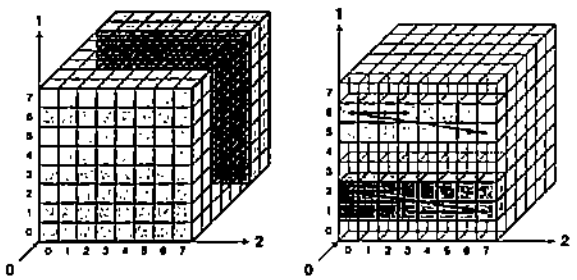

Figure 4. Two sequences with (a) 0 inactive dimensions (b) 1 inactive dimension. Unlike EGO-heuristic, in both cases EGO*-heuristic is able to tell that the sequences are non-joinable.

New EGO*-heuristic: The proposed J_{EGO*} (EGO*-join) algorithm is J_{EGO} (EGO-join) with an important change to the heuristic for determining that two sequences

Input: The first and last cells of a sequence: C_F andC_L
Output: Bounding rectangle BR
1. **for** $i \leftarrow 0$ **to** $d-1$ **do**
   (a) $BR.lo[i] \leftarrow C_F.x[i]$
   (b) $BR.hi[i] \leftarrow C_L.x[i]$
   (c) **if** $(R.lo[i] = R.hi[i])$ **then continue**
   (d) **for** $j \leftarrow i+1$ **to** $d-1$ **do**
      i. $BR.lo[j] \leftarrow 0$
      i. $BR.hi[j] \leftarrow MAX\_CELL$
   (e) **break**
2. **return** BR

Figure 5. J_{EGO*}: procedure for obtaining a Bounding Rectangle of a sequence

are non-joinable. The use of the EGO*-heuristic significantly improves performance of the join, as will be seen in Section 3.

We now present our heuristic with the help of an example for which J_{EGO} is unable to detect that the sequences are *non-joinable*.

Two sequences are shown in Figure 4(b). Assume that each sequence has many points. One sequence starts in cell (0,1,3) and ends in cell (0,2,2). The second sequence starts in cell (0,5,6) and ends in (0,6,3). Both sequences have one inactive dimension: 0. The EGO-heuristic will conclude that these two should be joined, allowing recursion to proceed. Figure 4(a) demonstrates the case when two sequences are located in two separate slabs, both of which have the size of at least two in each dimension. There are no inactive dimensions for this case and recursion will proceed further for J_{EGO}.

The new heuristic being proposed is able to correctly determine that for the cases depicted in Figures 4(a) and 4(b) the two sequences are *non-joinable*. It should become clear later on that, in essence, our heuristic utilizes not only inactive dimensions but also the active dimension.

The heuristic uses the notion of a Bounding Rectangle for each sequence. Notice that in general, given only the first and last cells of a sequence, it is impossible to compute the Minimum Bounding Rectangle (MBR) for the sequence. However, it is possible to compute a Bounding Rectangle (BR). Figure 5 describes an algorithm for computing a bounding rectangle.

The procedure takes as input the coordinates for the first and last cells of the sequence and produces the bounding rectangle as output. To understand getBR() algorithm, note that if the first and last cells have n prefix equal coordinates (e.g. $(1,2,3,4)$ and $(1,2,9,4)$ have two equal first coordinates – $(1,2,\times,\times)$) then all cells of the sequences have the same values in the first n coordinates (e.g. $(1,2,\times,\times)$ for our example). This means that the first n coordinates of the sequence can be bounded by that value. Furthermore, the active dimension can be bounded by the coordinates of first and last cell in that dimension respectively. Continuing with our example, the lower bound is now $(1,2,3,\times)$ and the upper bound is $(1,2,9,\times)$. In general, we cannot say anything precise about the rest of the dimensions, however the lower bound can always be set to 0 and upper bound to MAX_CELL.

Input: Two sequences A and B
Output: Result set R
1. $BR_1 \leftarrow$ getBR($A.first$, $A.last$)
1. $BR_2 \leftarrow$ getBR($B.first$, $B.last$)
3. Expand BR_1 by one in all directions
4. **if** $(BR_1 \cap BR_2 = \emptyset)$ **then return** $\emptyset$
5. ... // continue as in J_{EGO}

Figure 6. Beginning of J_{EGO*}: EGO*-heuristic

Once the bounding rectangles for both sequences being joined are known, it is easy to see that if one BR, expanded by one in all directions, does not intersect with the other BR, than the two sequences will not join.

As we shall see in Section 3, J_{EGO*} significantly outperform J_{EGO} in all instances. This improvement is a direct result of the large reduction of the number of sequences needed to be compared based upon the above criterion. This result is predictable since if EGO-heuristic can recognize two sequences as non-joinable than EGO*-heuristic will always do the same, but if EGO*-heuristic can recognize two sequences as non-joinable than, in general, there are many cases when EGO-heuristic will decide the sequence is joinable. Thus EGO*-heuristic is more powerful. Furthermore, the difference in CPU time needed to compute the heuristics given the same two sequences is insignificant.

3 Experimental results

In this section we present the performance results for in-memory joins using J_{RSJ} (RSJ join), J_G, J_{EGO} [2], and J_{EGO*}. The results report the actual time for the execution of the various algorithms. First we describe the parameters of the experiments, followed by the results and discussion.

In all our experiments we used a 1GHz Pentium III machine with 2GB of memory. All multidimensional points

were distributed on the unit d-dimensional box $[0,1]^d$. The number of points ranges from 68,000 to 200,000. For distributions of points in the domain we considered the following cases:

1. **Uniform**: Points are uniformly distributed.
2. **Skewed**: The points are distributed among five clusters. Within each cluster points are distributed normally with a standard deviation of 0.05.
3. **Real data**: We tested data from ColorHistogram and ColorMoments files representing image features. The files are available at the UC Irvine repository. ColorMoments stores 9-dimensional data, which we normalized to $[0,1]^9$ domain, ColorHistogram – 32-dimensional data. For experiments with low-dimensional real data, a subset of the leading dimensions from these datasets were used. Unlike uniform and skewed cases, for real data a self-join is done.

Often, in similar research, the costs of sorting the data, building or maintaining the index or costs of other operations needed for a particular implementation of join are ignored. No cost is ignored in our experiments for J_G, J_{EGO}, and J_{EGO*}. One could argue that for J_{RSJ} the two indexes, once built, need not be rebuilt for different ε. While there are many other situations where the two indexes need to be built from scratch for J_{RSJ}, we ignore the cost of building and maintaining indexes for J_{RSJ}, giving it an advantage.

3.1 Correlation between selectivity and ε

The choice of the parameter ε is critical when performing an ε-join. Little justification for choice of this parameter has been presented in related research. In fact, we present this section because often in similar research selected values of ε are too small.

The choice of ε has a significant effect on the selectivity depending upon the dimensionality of the data. The ε-join is a common operation for similarity matching. Typically, for each multidimensional point from set A a few points (i.e. from 0 to 10, possibly from 0 to 100, but unlikely more than 100) from set B need to be identified on the average. The average number of points from set B that joins with a point from set A on the average is called *selectivity*.

In our experiments, selectivity motivated the range of values chosen for ε. The value of ε is typically lower for smaller number of dimensions and higher for high-dimensional data. For example a 0.1×0.1 square[2] query ($\varepsilon = 0.1$) is 1% of a 2-dimensional domain, however $\varepsilon^8 = 0.1^8$ is only 10^{-6}% of an eight-dimensional domain, leading to small selectivity.

Let us estimate what values for ε should be considered for joining d-dimensional uniformly distributed data such that a point from set A joins with a few (close to 1) points from set B. Assume that the cardinality of both sets is m. We need to answer the question: what should the value of ε be such that m hyper-cubes of side ε completely fill the unit d-dimensional cube? It is easy to see that the solution is $\varepsilon = \frac{1}{m^{1/d}}$. Figure 7 plots this function $\varepsilon(d)$ for two different values of m. Our experimental results for various number of dimensions corroborate the results presented in the figure. For example the figure predicts that in order to obtain a selectivity close to one for 32-dimensional data, the value of ε should be close to 0.65, or 0.7, and furthermore that values smaller than say 0.3, lead to zero selectivity (or close to zero) which is of little value[3]. This is in very close agreement to the experimental results.

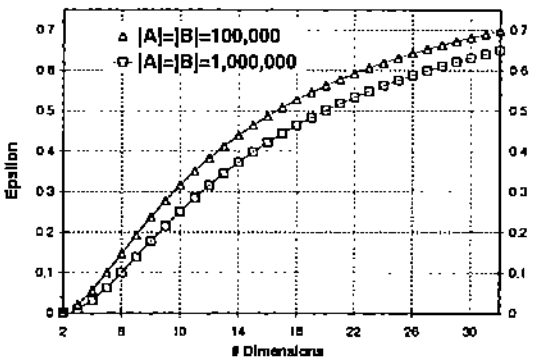

Figure 7. Choosing ε for selectivity close to one for 10^5 (and 10^6) points uniformly distributed on $[0,1]^d$

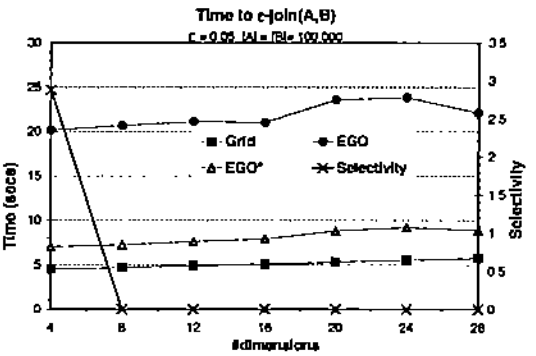

Figure 8. Pitfall of using improper selectivity

If the domain is not normalized to the unit square, such as in [8], the values of ε should be scaled accordingly. For example ε of 0.1 for $[-1,1]^d$ domain correspond to ε of 0.05 for our $[0,1]^d$ domain. Figure 8 demonstrates the pitfall of using an improper selectivity. The parameters of the

[2] A square query was chosen to demonstrate the idea, ideally one should consider a circle.

[3] For self-join selectivity is always at least 1, thus selectivity 2–100 is desirable.

experiment (distribution of data, cardinality of sets and ε (scaled)) are set to the values used in one publication. With this choice of ε the selectivity plunges to zero even for the 10-dimensional case. In fact, for our case, the figure presumably shows that the Grid-join is better than J_{EGO} and J_{EGO*} even for high-dimensional cases. However, the contrary is true for a meaningful selectivity as will be shown in Section 3.3.

3.2 Low-dimensional data

We now present the performance of J_{RSJ}, J_{EGO}, J_{EGO*} and J_G for various settings.

The x-axis plots the values of ε, which are varied so that meaningful selectivity is achieved. In all but one graph the left y-axis represents the total time in seconds to do the join for the given settings. Due to the importance of the selectivity in addition to the value of ε, we plot the resulting selectivity in each experiment. The selectivity values are plotted on the y-axis at the right end of each graph, in actual number of matching points. Clearly, if selectivity is 0, then ε is too small and vice versa if the selectivity is more than 100. As expected, in each graph the selectivity, shown by the line with the '×', increases as ε increases.

J_{RSJ} is ommitted from most of the Figures for clarity since it showed much worse results than the other joins. Figure 9 depicts performance of the joins for 4-dimensional uniform data with cardinality of both sets being 10^5. Figure 10 shows the performance of the same joins relative to that of J_{RSJ}.

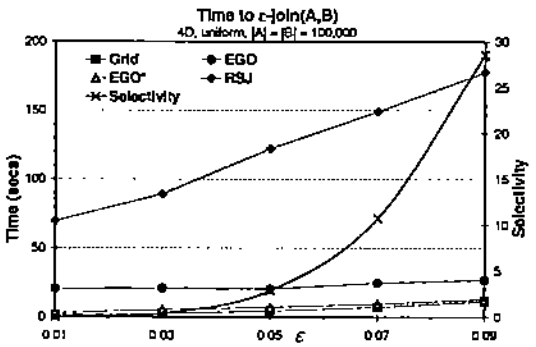

Figure 9.

In Figure 10, J_{EGO} shows 3.5–6.5 times better results than those of J_{RSJ}, which corroborates the fact that, by itself, J_{EGO} is a quite competitive scheme for low-dimensional data. But it is not as good as the two new schemes.

Next comes J_{EGO*} whose performance is *always* better than that of J_{EGO} in all experiments. This shows the strength of J_{EGO*}. Because of the selectivity, the values of

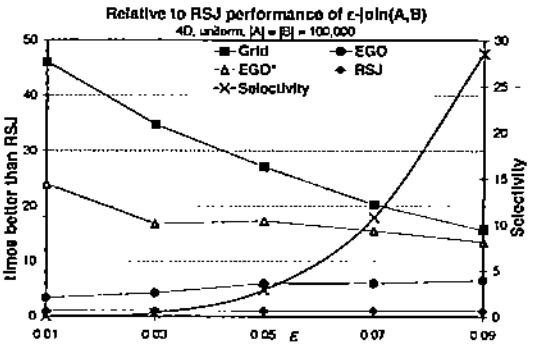

Figure 10.

ε are likely to be small for low-dimensional data and large for high-dimensional data. The EGO-heuristic is not well-suited for small values of ε. The smaller the epsilon, the less likely that a sequence has an inactive dimension. In Figure 10 J_{EGO*} is seen to give 13.5–24 times better performance than J_{RSJ}.

Another trend that can be observed from the graphs is that J_G is better that J_{EGO*}, except for high-selectivity cases (Figure 14). J_{EGO} shows results several times worse than those of J_G, which corroborates the choice of the Grid-index which also was the clear winner in our comparison [6] with main memory optimized versions of R-tree, R*-tree, CR-tree, and quad-tree indexes. In Figure 10 J_G showed 15.5–46 times better performance than J_{RSJ}.

Unlike J_{EGO}, J_{EGO*} always shows results at least comparable to those of J_G. For all the methods, the difference in relative performance shrinks as ε (and selectivity) increases.

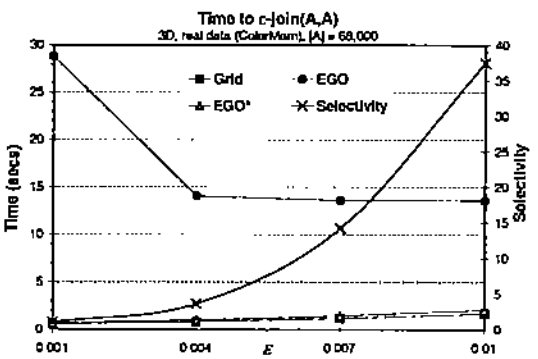

Figure 11. Join 3D real data

Figures 11 and 12 show the results for the self-join of real 3-dimensional data taken from the ColorMom file. The cardinality of the set is 68,000. The graph on the left shows the best three schemes, and the graph on the right omits J_{EGO} scheme due to its much poorer performance. From these two graphs we can see that J_G is almost 2 times better than J_{EGO*} for small values of ε.

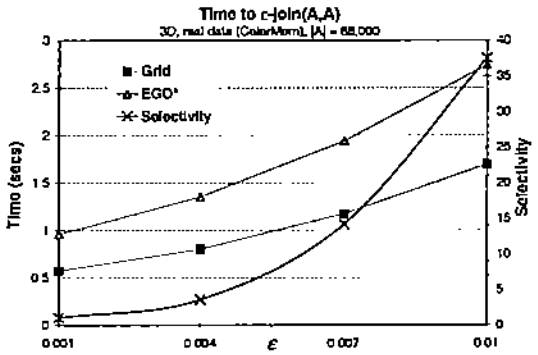

Figure 12. 3D real data, no J_{EGO} for clarity

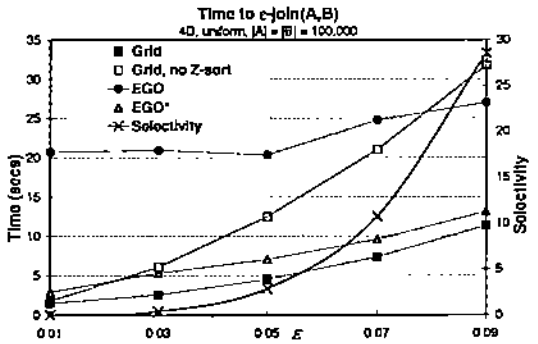

Figure 13. 4D uniform data $|A| = |B| = 100,000$

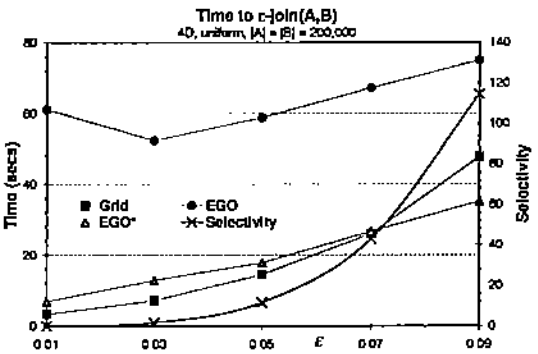

Figure 14. 4D uniform data $|A| = |B| = 200,000$

Figures 13 and 14 show the results for 4-dimensional uniform data. The graph on the left is for sets of cardinality 100,000, and that on the right is for sets with cardinality 200,000. Figure 13 emphasizes the importance of performing Z-sort on data being joined: the performance improvement is $\sim$ 2.5 times. J_G without Z-sort, in general, while being better than J_{EGO}, shows worse results than that of J_{EGO*}.

Figure 14 presents another trend. This figure shows an example where J_{EGO*} becomes a better choice than J_G for values of ϵ greater than ~ 0.07 which corresponds to a high selectivity of ~ 43.

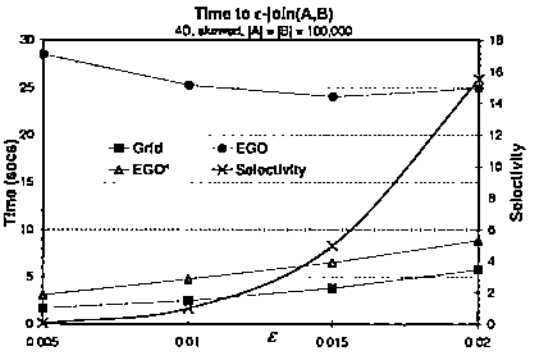

Figure 15. Join 4D skewed data

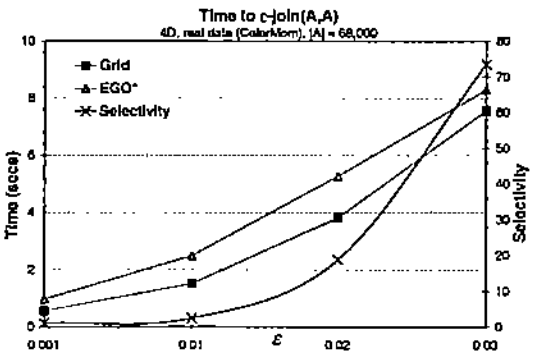

Figure 16. Join 4D real data

Figures 15 and 16 show the results for 4-dimensional skewed and real data. Note that the values of ϵ are now varied over a smaller range than that of the uniformly distributed case. This is so because in these cases points are closer together and smaller values of ϵ are needed to achieve the same selectivity as in uniform case. In these graphs J_{EGO}, J_{EGO*}, and J_G exhibit behavior similar to that in the previous figures with J_G being the best scheme.

3.3 High-dimensional data

We now study the performance of the various algorithms for higher dimensions. Figures 17 and 18 show the results for 9-dimensional uniformly distributed data. Figure 19 presents the results for 9-dimensional skewed data, Figure 20 gives the results for real 9-dimensional data. Figures 21 and 22 show the results with the 9- and 16-dimensional real data respectively. As with low-dimensional data, for all tested cases, J_{RSJ} had the worst results. Therefore, the performance of J_{RSJ} is omitted from

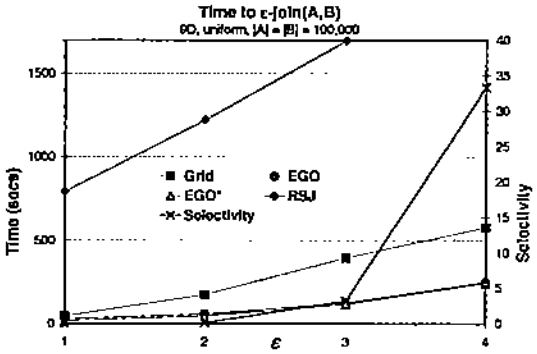

Figure 17. Join 9D uniform data

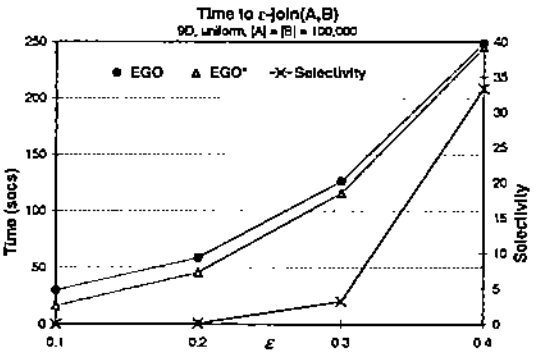

Figure 18. Join 9D uniform data, the best two techniques

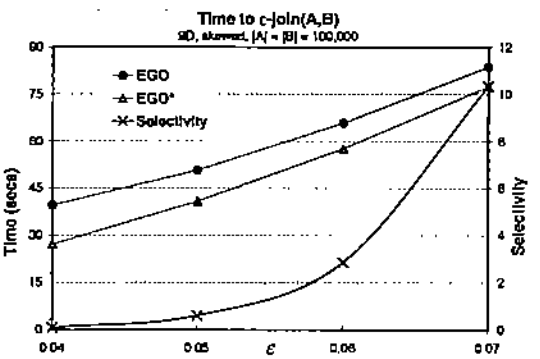

Figure 19. Join 9D skewed data

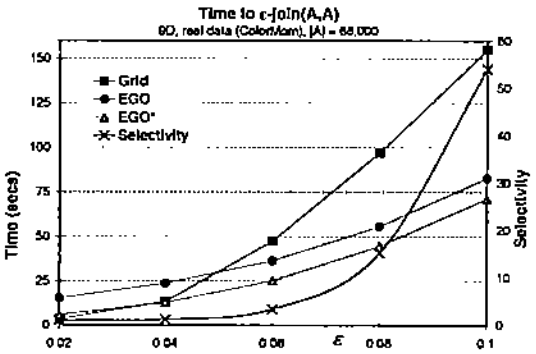

Figure 20. Join 9D real data

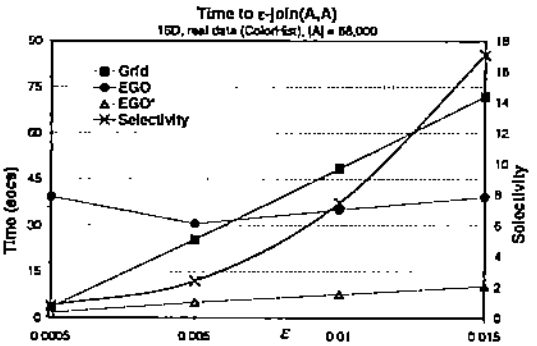

Figure 21. Join 16D real data

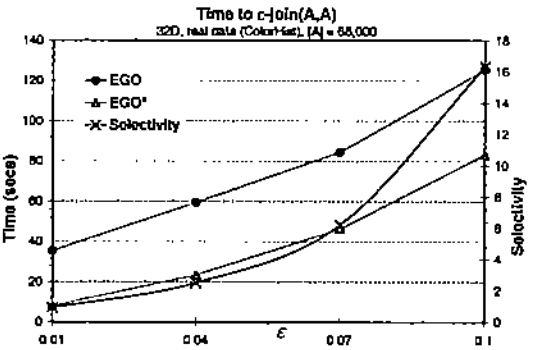

Figure 22. Join 32D real data

most graphs – only one representative case is shown in Figure 17.

An interesting change in the relative performance of J_G is observed for high-dimensional data. Unlike the case of low-dimensional data, J_{EGO} and J_{EGO*} give better results than J_G. J_G is not competitive for high-dimensional data, and its results are often omitted for clear presentation of J_{EGO} and J_{EGO*} results. A consistent trend in all graphs is that J_{EGO*} results are *always* better than those of J_{EGO}. The difference is especially noticeable for the values of ε corresponding to low selectivity. This is a general trend: J_{EGO} does not work well for smaller epsilons, because in this case a sequences is less likely to have an inactive dimension. J_{EGO*} does not suffer from this limitation.

Set Cardinality When the join of two sets is to be computed using Grid-join, an index is built on one of the two sets. Naturally, the question of which set to build the index on arises. We ran experiments to study this issue. The results indicate that building the index on the smaller dataset always gave better results.

4 Related work

Below we discuss some of the most prominent solutions for efficient computation of similarity joins. This section was reduced, see [5] for details. Shim et. al. [13] propose to use ε-KDB-tree for performing high-dimensional similarity joins of massive data. The R-Tree Spatial Join (RSJ) algorithm [3] works with an R-tree index built on the two datasets being joined. Several optimizations of this basic algorithm have been proposed [4]. In [10] Patel et. al a plane sweeping technique is modified to create a disk-based similarity join for 2-dimensional data. The new procedure is called the Partition Based Spatial Merge join, or PBSM-join. A partition based merge join is also presented in [9]. Shafer et al in [12] present a method of parallelizing high-dimensional proximity joins. Koudas et al [8] have proposed a generalization of the Size Separation Spatial Join Algorithm, named Multidimensional Spatial Join (MSJ). Recently, Böhm et al [2] proposed the EGO-join. More details about EGO-join are in Section 2.2. The EGO-join was shown to outperform other join methods in [2]. Grid-join is based on [6, 11, 1].

5 Conclusions

	Small ε	Avg ε	Large ε
Low Dim	J_G	J_G	J_G or J_{EGO*}
High Dim	J_G or J_{EGO*}	J_{EGO*}	J_{EGO*}

Table 1. Choice of Join Algorithm

In this paper we considered the problem of similarity join in main memory for low- and high-dimensional data. We propose two new algorithms: *Grid-join* and *EGO*-join* that were shown to give superior performance than the state-of-the-art technique (EGO-join) and RSJ.

The significance of the choice of ε and recommendations for a good choice for testing and comparing algorithms with meaningful selectivity were discussed. We demonstrated an example with values of ε too small for the given dimensionality where one methods showed the best results over the others whereas with more meaningful settings it would show the worst results.

While recent research has concentrated on joining high-dimensional data, little attention was been given to the choice of technique for low-dimensional data. In our experiments, the proposed Grid-join approach showed the best results for low-dimensional case or when values of ε are very small. The EGO*-join has demonstrated substantial improvement over EGO-join for all the cases considered and is the best choice for high-dimensional data or when values of ε are large. The results of the experiments with RSJ proves the strength of Grid-join and EGO*-join.

Based upon the experimental results, the recommendation for choice of join algorithm is summarized in Table 1.

References

[1] PLACE, Pervasive Location Aware Computing Environment. http://www.cs.purdue.edu/place.

[2] C. Böhm, B. Braunmüller, F. Krebs, and H.-P. Kriegel. Epsilon grid order: an algorithm for the similarity join on massive high-dimensional data. In *SIGMOD*, 2001.

[3] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient processing of spatial joins using R-trees. In *SIGMOD'93*.

[4] Y.-W. Huang, N. Jing, and E. A. Rundensteiner. Spatial joins using r-trees: Breadth-first traversal with global optimizations. In *VLDB'97*.

[5] D. V. Kalashnikov and S. Prabhakar. Similarity joins for low- and high- dimensional data. Technical Report TR 02-025, Purdue University, Oct. 2002.

[6] D. V. Kalashnikov, S. Prabhakar, S. Hambrusch, and W. Aref. Efficient evaluation of continuous range queries on moving objects. In *DEXA'02*.

[7] K. Kim, S. Cha, and K. Kwon. Optimizing mutidimensional index trees for main memory access. In *Proc. of ACM SIGMOD Conf.*, May 2001.

[8] N. Koudas and K. C. Sevcik. High dimensional similarity joins: Algorithms and performance evaluation. In *ICDE'98*.

[9] M.-L. Lo and C. V. Ravishankar. Spatial hash-joins. In *Proc. of ACM SIGMOD Conf.*, 1996.

[10] J. M. Patel and D. J. DeWitt. Partition based spatial-merge join. In *Proc. of ACM SIGMOD Conf.*, 1996.

[11] S. Prabhakar, Y. Xia, D. Kalashnikov, W. Aref, and S. Hambrusch. Query indexing and velocity constrained indexing: Scalable techniques for continuous queries on moving objects. *IEEE Transactions on Computers, Special section on data management and mobile computing*, 51(10):1124–1140, Oct. 2002.

[12] J. C. Shafer and R. Agrawal. Parallel algorithms for high-dimensional similarity joins for data mining applications. In *Proc. of VLDB Conf.*, 1997.

[13] K. Shim, , R. Srikant, and R. Agrawal. High-dimensional similarity joins. In *Proc. of ICDE Conf.*, 1997.