To appear in *Information Systems Journal*, Elsevier, 2005

Fast similarity join for multidimensional data ⋆

Dmitri V. Kalashnikov *

University of California, Irvine, Department of Computer Science, 4300 Calit2 Building, Irvine, CA 92697, USA

Sunil Prabhakar

Purdue University, Department of Computer Sciences, 250 N. University Street, West Lafayette, IN 47907, USA

Abstract

The efficient processing of multidimensional 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 focused 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 in-memory spatial join algorithms, the Grid-join and EGO*-join, and study their performance. Through evaluation, we explore the domain of applicability of each approach and provide recommendations for the choice of a join algorithm depending upon the dimensionality of the data as well as the expected selectivity of the join. We show that the two new proposed join techniques substantially outperform the state of the art join algorithm, the EGO-join.

Key words: similarity join, grid-based joins
PACS:

⋆ This work was supported by NSF grants IIS-9985019, 0010044-CCR, 9972883, and an Intel Ph.D. Fellowship. A preliminary version of this work appeared in [1].
* Corresponding author.
Email addresses: dvk@ics.uci.edu (Dmitri V. Kalashnikov), sunil@cs.purdue.edu (Sunil Prabhakar).

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. Spatial joins are natural for geographic information systems and moving object environments where pairs of objects located close to each other are to be identified [2,3]. Many algorithms for several basic data mining operations such as clustering [4], outlier detection [5], and association rule mining [6] require the processing of all pairs of points within a certain distance to each other [7]. Thus a similarity join can serve as the first step for many of these operations [8].

The problem of efficient computation of similarity joins of multidimensional data has been studied extensively in the literature. Most researchers have focused 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, a self-join of 1 million 32-dimensional data points, using an algorithm similar to that of [7] (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 dataset, that are within distance ε from each other, i.e.

$$R = J(A, B, \varepsilon) = \{(a, b) : \|a - b\| < \varepsilon;\ a \in A,\ b \in B\}.$$

While several research efforts have concentrated on designing efficient high-dimensional join algorithms, the question 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 a join algorithm for low- and high-dimensional data. We propose and evaluate two new join algorithms: the *Grid-join* and *EGO*-join*. We compare them with the state of the art algorithm – EGO-join [7], and with a method which serves as a benchmark in many similar publications, the RSJ join [9].

These techniques are compared empirically using synthetic and real data. The experimental evaluation shows that the Grid-join approach is the best method for low-dimensional data. When using the Grid-join, the join of two datasets A and B is computed using an index nested loop approach: an index (i.e. specifically constructed 2-dimensional grid) is built on circles with radius ε centered at the first two coordinates of points from dataset B. The first two coordinates of points from dataset A are used as point-queries to the grid-index in order to compute the join. Although several choices are available for constructing this index, only the grid is considered in this paper. The choice of the grid index is not accidental, it is based upon our earlier results for main memory evaluation of range queries. In [10] we have shown that for range queries over moving objects, using a grid index results in an order of magnitude better performance than memory optimized R-tree, CR-tree, R*-tree, or Quad-tree.

The results for high-dimensional data show that the EGO*-join is the best join technique, unless ε is very small. The EGO*-join that we propose in this paper is based upon the EGO-join algorithm. Böhm et al. in [7] have shown that the Epsilon Grid Order (EGO) join algorithm outperforms other spatial join techniques for high-dimensional data. The new EGO*-join algorithm significantly outperforms EGO-join for all cases considered. The improvement is especially noticeable when the number of dimensions is not very high, or the value of ε is not large.

The RSJ algorithm is significantly less effective than all other tested algorithms. To join two datasets using RSJ, two R-tree indexes are built or maintained on these datasets. But unlike the case of some approaches, these indexes need not be rebuilt when the join is recomputed with different ε.

Although not often addressed in related research, the choice of the ε parameter for the join is critical for 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. Section 3.1 discusses how to choose the appropriate values of ε to achieve meaningful selectivity of the result set.

The contributions of this paper are as follows:

- Two join algorithms that achieve better performance (by a factor of 2–10) than the state of the art EGO-join algorithm.
- Recommendations for the choice of a join algorithm based upon data dimensionality d, and ε.
- Highlight the importance of the choice of ε and the corresponding selectivity for experimental evaluation.

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. Finally, Section 5 concludes the paper. A sketch of the algorithm for selecting grid size, and cost estimator functions for Grid-join, are presented in the appendix.

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 [10] for evaluating continuous range queries over moving objects. The EGO*-join is based upon EGO-join proposed in [7]. 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 the appendix. The EGO*-join method is discussed in Section 2.2.

2.1 Grid-join

Let us first assume the case of joining 2-dimensional data, the general case of d dimensions will be discussed shortly. The spatial join of two datasets, A and B, can be computed using the standard index nested loop approach as follows. We treat one of the point datasets as a collection of circles of radius ε centered at each point of one of the two datasets (say B). This collection of circles is then indexed using some spatial index structure. The join is computed by taking each point from the other dataset (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 dataset and processing a circle region query for each point from the other dataset) 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.

In earlier work [10], 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 employed 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

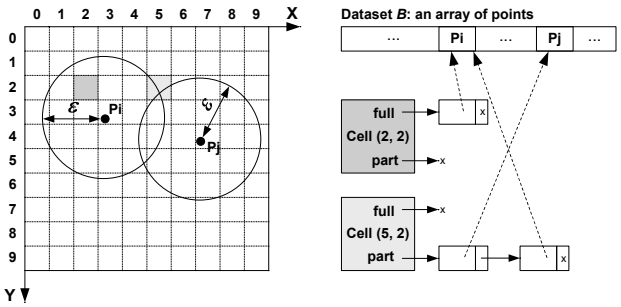


Fig. 1. An example of the Grid Index, I_G

the range queries.

In [10], the choice of a good main-memory index has been investigated. Several key index structures, including R-tree, R*-tree, CR-tree [11], quad-tree, and 32-tree [10], 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.

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.[1] The similarity join algorithm which utilizes I_G is called the Grid-join, or J_G for short.

Case of 2 dimensions Let us now consider the 2-dimensional case more formally. We will consider the problem of joining two 2-dimensional datasets, $A = \{a_0, a_1, \ldots, a_{|A|-1}\}$ and $B = \{b_0, b_1, \ldots, b_{|B|-1}\}$, using an index nested loop approach, where the grid index is built on dataset B.

The grid index 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 illustrates 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 the cell-coordinates corresponding to a point in O(1) time. Each cell in the grid contains two lists called the `full` and `part` lists, as illustrated in Figure 1. Let `circle`(p, r) denote a circle with center at point p and radius r. The `full` list of each cell C in I_G contains *pointers* to each point b_i from B such that `circle`(b_i, ε) fully covers the cell:

[1] Note however, that 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.

Input: Datasets $A = \{a_0, a_1, \ldots, a_{|A|-1}\}$, $B = \{b_0, b_1, \ldots, b_{|B|-1}\}$, and $\varepsilon \in \Re$
Output: Result set R
1. $R \leftarrow \emptyset$
2. zsort(A)
3. zsort(B)
4. initialize I_G
5. **for** $i \leftarrow 0$ **to** $|B| - 1$ **do**
   (a) $\overline{b}_i \leftarrow (b_i^0, b_i^1)$
   (b) insert $\{b_i, \texttt{circle}(\overline{b}_i, \varepsilon)\}$ into I_G
6. **for** $i \leftarrow 0$ **to** $|A| - 1$ **do**
   (a) $\overline{a}_i \leftarrow (a_i^0, a_i^1)$
   (b) $C \leftarrow$ cell in I_G corresponding to $\overline{a}_i$
   (c) **for** $j \leftarrow 0$ **to** $|C.\texttt{part}| - 1$ **do**
      i. $b \leftarrow C.\texttt{part}[j]$
      ii. **if** $(\|a_i - b\| < \varepsilon)$ **then** $R \leftarrow R \cup \{(a_i, b)\}$
   (d′) **for** $j \leftarrow 0$ **to** $|C.\texttt{full}| - 1$ **do**
      i. $b \leftarrow C.\texttt{full}[j]$
      ii. $R \leftarrow R \cup \{(a_i, b)\}$
7. **return** R

Fig. 2. Grid-join procedure, J_G

$C.\texttt{full} = \{b_i : C \subset \texttt{circle}(b_i, \varepsilon);\ b_i \in B\}$. The `part` list of each cell C contains pointers to each point b_i from B such that $\texttt{circle}(b_i, \varepsilon)$ partially covers the cell: $C.\texttt{part} = \{b_i : C \not\subset \texttt{circle}(b_i, \varepsilon)$ and $C \cap \texttt{circle}(b_i, \varepsilon) \neq \emptyset;\ b_i \in B\}$.

To find all points within ε-distance from a given point $a_i \in A$, first the cell corresponding to a_i is retrieved. All points in the `full` list of that cell are guaranteed to be within ε-distance from a_i. The points b_j in the cell's `part` list need to be processed further to check if $\|a_i - b_j\| < \varepsilon$.[2]

Outline of the algorithm The basic steps of J_G to join two datasets A and B are outlined in Figure 2. In Steps 2 and 3, a spatial sort, called z-sort, is applied to the two datasets. The need for z-sort is explained later. A grid index I_G is initialized in Step 4. In the loop in Step 5, all points $b_i \in B$ are added to I_G one by one as follows. First $\overline{b}_i$, a 2-dimensional point constructed from the first two coordinates of b_i, is considered.[3] Then pointer to b_i is added to the

[2] The choice of data structures for the `full` and `part` lists is critical for performance. We implemented these lists as dynamic-arrays rather than lists, which improves performance by roughly 40% due to the resulting clustering in memory (and thereby reduced cache misses). A dynamic array is a standard data structure for arrays whose size adjusts dynamically.

[3] Notice, for 2-dimensional case $\overline{b}_i$ is equivalent to b_i, the difference exists only for more than two dimensions, as explained shortly.

`part` lists of each cell C in I_G that satisfies $C \cap \texttt{circle}(\overline{b}_i, \varepsilon) \neq \emptyset$.[4] 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 achieve this, point $\overline{a}_i$ is constructed from the first two coordinates of a_i and the cell corresponding to $\overline{a}_i$ in I_G, C, is determined in Steps 6(a) and 6(b). Then, in Step 6(c), the algorithm iterates through points in the `part` list of cell C and finds all points that are within ε distance from a_i. Step 6(d′) is analogous to Step 6(c) but for `full` lists and valid only for the 2-dimensional case.

Case of d dimensions In the general d-dimensional case, J_G still employs a 2-dimensional grid. Only the first two coordinates of points in A and B are utilized for all operations, exactly as in 2-dimensional case, except when processing `part` lists, in which case all d coordinates are utilized to determine if $\|a - b\| < \varepsilon$.

While in the 2-dimensional case each cell has two lists, in the d-dimensional case where $d > 2$ each cell has only one list. The reason for having two separate lists (`full` and `part`) per cell for 2-dimensional points is as follows. When processing a point $a_i \in A$, points b_j in the `full` list do not need $\|a_i - b_j\| < \varepsilon$ checks since those points are guaranteed to be within ε-distance from a_i, whereas points from the `part` list do need these checks. For more than two dimensions, keeping a separate `full` list per each cell of a 2-dimensional grid is of little value because now points from the `full` list do need $\|a_i - b_j\| < \varepsilon$ checks as well. Therefore for the d-dimensional case each cell has only one list, called the `part` list: $C.\texttt{part} = \{b : C \cap \texttt{circle}(\overline{b}, \varepsilon) \neq \emptyset;\ b \in B\}$.

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 greater the time needed to initialize the index and load the data into it. We now present a sketch of a solution for selecting appropriate grid size.

The first step is to develop a set of estimator functions that predict the cost (i.e., the execution time, in our context) 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. The appendix presents details of how each of these costs is estimated. These functions are able to achieve very accurate prediction. With the help of these functions, it is possible to estimate which grid size is optimal. Such functions can also be used by a query optimizer to

[4] This step is valid for the general d-dimensional case, where $d \geq 2$. However, for better performance in the 2-dimensional case, pointer to b_i is added to the `full` lists of each cell C in I_G that satisfies the condition $\{C \subset \texttt{circle}(b_i, \varepsilon)\}$ and it is added to the `part` lists of each cell C in I_G that satisfies the condition $\{C \not\subset \texttt{circle}(b_i, \varepsilon)$ and $C \cap \texttt{circle}(b_i, \varepsilon) \neq \emptyset\}$.

evaluate if it is more efficient to use J_G for the given parameters or another join approach.

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

In J_G, for each point in A its cell is computed, and the `full` and `part` lists (or just `part` list) of this cell are accessed, as illustrated in Figure 2. The algorithm simply processes points in sequential order in the array corresponding to dataset 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 to each other according to their first two coordinates in the 2-dimensional domain are also tend to be close to each other in the array. In this situation, data structures for a given cell (e.g., its `part` list) are likely to be reused from the cache during the processing of subsequent points from the array. The speedup 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 in 2-dimensional domain are also tend to be close in the array can easily be achieved by various methods. We choose to use a sorting based on the Z-order. We sort not only dataset A but also dataset B, which reduces the time needed to add circles to I_G. In Section 3 we will show that the gain achieved by z-sort can be quite significant. For example, approximately 2.5 times speedup is achieved by utilizing Z-sort in Figure 13 in that section.

2.2 EGO-join*

In this section we present our second algorithm called EGO*-join, which is based on EGO-join approach[7]. In [7] Böhm et al. have shown that EGO-join substantially outperforms other methods for joining massive, high-dimensional data. We will use the notation J_{EGO} to denote the EGO-join approach and J_{EGO*} for the EGO*-join approach. Before introducing J_{EGO*}, we begin by briefly describing J_{EGO} as presented in [7].

The Epsilon Grid Order: J_{EGO} is based on the so called Epsilon Grid Ordering (EGO) [7]. 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-coordinates 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)

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

EGO-sort: To join datasets A and B with a certain ε using J_{EGO}, first the points in these datasets are sorted in accordance with the EGO for the given ε. Notice, if a subsequent J_{EGO} operation is needed but with a different ε, datasets A and B must be sorted again since EGO depends on ε.

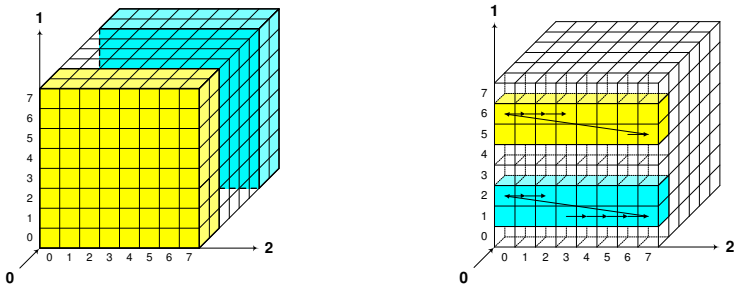
Recursive join: The procedure for joining two sequences is recursive. Each sequence is further subdivided into two roughly equal subsequences and each subsequence is joined recursively with both its counterparts. The partitioning is carried out until the length of both subsequences is smaller than a 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. We refer to such sequences as *non-joinable*.

EGO-heuristic: A key element of J_{EGO} is the heuristic to identify *non-joinable* sequences.

To understand the heuristic, let us consider a simple example. In a short EGO-sorted sequence its first and last points are likely to have the same values in the first few dimensions of their cell-coordinates. For example, points with cell-coordinates $(2, 7, 4, 1)$ and $(2, 7, 6, 1)$ have the same values in the two first dimensions $(2, 7, \times, \times)$. The values in the third dimension are different. The third dimensions is called the *active* dimension, the first two dimensions are called *inactive*. Because the sequence is EGO-sorted, in this sequence all points have '2' and '7' in their first and second dimensions of their cell-coordinates.

Given two EGO-sorted sequences, the EGO-heuristic first computes two values: the number of inactive dimensions for each of the two sequences. Then it computes another value *min*, corresponding to the minimum of the two values. It is easy to prove [7] that if there is a dimension no greater than *min* 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.

New EGO*-heuristic: The proposed J_{EGO*} (EGO*-join) algorithm is J_{EGO}



(a) (b)

Fig. 4. Two sequences with (a) 0 inactive dimensions (b) 1 inactive dimension. The EGO-heuristic fails in both cases. In both cases EGO*-heuristic correctly determines the sequences are non-joinable.

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. $BR.lo[j] \leftarrow 0$
      ii. $BR.hi[j] \leftarrow MAX\_CELL$
   (e) **break**
2. **return** BR

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

(EGO-join) with a different heuristic for determining that two sequences are non-joinable. Section 3 will demonstrate that the use of the EGO*-heuristic significantly improves performance of the join.

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

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 thus EGO-heuristic will fail to determine that they are non-joinable. In Figure 4(b), assume 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 (dimension number zero), and in dimension number zero they both have the same value of 0. Thus, the EGO-heuristic will fail again to determine that they are non-joinable.

The new heuristic being proposed is able to correctly determine that for the

two cases of Figures 4(a) and 4(b) the two sequences are *non-joinable*. To do that, the EGO*-heuristic utilizes not only inactive dimensions as EGO-heuristic, but also the active dimension.

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

The procedure takes as its input the coordinates of the first and last cells of an EGO-sorted sequence and outputs a bounding rectangle for that sequence. To understand the getBR() algorithm, note that if the first and the last points of the sequence have the same values in the first n dimensions of their cell-coordinates (e.g. $(1, 2, 3, 4)$ and $(1, 2, 9, 4)$ are equal in their first two dimensions – $(1, 2, \times, \times)$) then all points in the sequence have the same values in the first n dimensions (e.g. $(1, 2, \times, \times)$ for our example). This means that the first n dimensions of the sequence can be bounded by those values. Furthermore, the active dimension of the sequence can be bounded by the values of first and last points of that sequence in that dimension. Continuing with our example, the lower bound is now $(1, 2, 3, \times)$ and the upper bound is $(1, 2, 9, \times)$. In general, by observing only the first and the last points of the sequence, the rest of the dimensions cannot be bounded precisely, 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}

Fig. 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, then no point from the first sequence will join a point from the second sequence. The basic steps of the EGO*-heuristic are outlined in Figure 6.

As we shall see in Section 3, J_{EGO*} significantly outperforms 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 determine that two sequences are non-joinable then EGO*-heuristic will always do the same, but

the reverse is not true. The difference in CPU time needed to compute the EGO- and EGO*-heuristics, given the same two sequences, is insignificant. Thus EGO*-heuristic is more powerful.

Unlike J_G, both J_{EGO*} and J_{EGO} are disk-based joins, even though in this paper we evaluate them only in main memory. Since J_{EGO*} is identical to J_{EGO} except for the heuristic, the performance of a disk-based implementation of J_{EGO*} is expected to be better than that of J_{EGO}. J_G has been designed for main memory only and an efficient disk-based implementation of J_G is outside the scope of this paper.

3 Experimental results

In this section, we experimentally study J_{RSJ} (RSJ join), J_G, J_{EGO} [7], and J_{EGO*} approaches. In all our experiments, we have used a 1GHz Pentium III machine with 2GB of memory. All multidimensional points have been distributed on the unit d-dimensional box $[0,1]^d$. The number of points has been varied from $68,000$ to $10,000,000$. We have considered the following distributions of points:

(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 test 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 is used. Unlike uniform and skewed cases, for real data a self-join is performed.

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 a 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}, thus 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 the choice of this parameter has been presented in related research. In fact, we present this section because we have discovered that selected values of ε are often too small in similar research. [5]

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 dataset A a few points (i.e. from 0 to 10, possibly from 0 to 100, but unlikely more than 100) from dataset B need to be identified on average. The average number of points from dataset B that joins with a point from dataset A 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 [6] query ($\varepsilon = 0.1$) is 1% of a two-dimensional domain, however it is only 10^{-6}% of an eight-dimensional domain, leading to small selectivity.

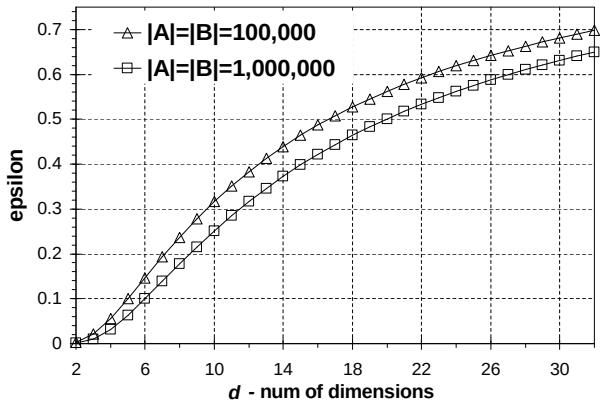


Fig. 7. Choosing ε for selectivity close to one for 10^5 (and 10^6) points uniformly distributed on $[0, 1]^d$

Let us estimate what values for ε should be considered for joining high-

[5] There is some evidence that the mistake may happen because researchers often test a *self-join* of dataset A, which is simpler than a join of two distinct datasets A and B. In a self-join, each point joins at least with itself. Thus, for the result set R of a self-join, it holds that $|R| \geq |A|$. By increasing the dimensionality d the space needed to store each data point increases. Consequently, the *space* occupied by R (e.g. in bytes) also increases as d increases, for any fixed positive ε, however small. Such an increase of the space occupied by R might be mistaken for an increase of the selectivity of the join.

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

dimensional uniformly distributed data such that a point from dataset A will join with a few (close to 1) points from dataset B. Assume that the cardinality of both datasets is m. We need to answer the question: what should the value of ε be such that m hyper-squares 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: 10^5 and 10^6. Our experimental results for various numbers of dimensions corroborate the results presented in the figure. For example, the figure predicts that in order to get a selectivity comparable to one for 32-dimensional data, the value of ε should be close to 0.65, or 0.7. If one chooses values smaller than say 0.3 instead, this will lead to zero selectivity (or close to zero) which is of little value[7]. This is in close agreement to the experimental results.

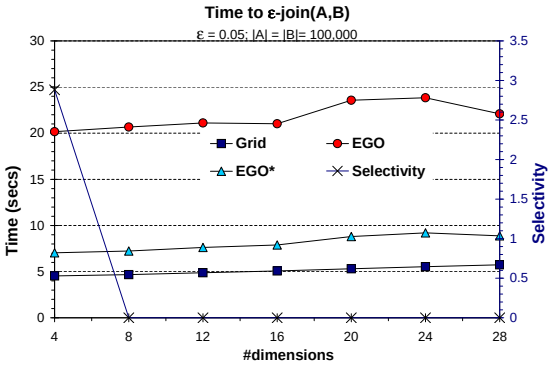


Fig. 8. Pitfall of using improper selectivity

If the domain is not normalized to the unit square, the values of ε should be scaled accordingly. For example, ε of 0.1 for $[-1, 1]^d$ domain corresponds to ε of 0.05 for our $[0, 1]^d$ domain. Figure 8 demonstrates the pitfall of using an improper selectivity. The value of ε is set such that the selectivity plunges sharply as the number of dimensions d increases: the selectivity is high for $d = 4$, it becomes very small for $d = 8$, and it is zero when $d \geq 10$. Figure 8 presumably shows that J_G is better than J_{EGO} and J_{EGO*} even for high-dimensional cases. However, the contrary is true for meaningful selectivity as will be demonstrated in Section 3.3.

Due to the importance of the selectivity, we plot its values in each experiment on the y-axis at the right end of each graph. The parameter ε is plotted on the x-axis, and the time taken by each join method is plotted on the left y-axis in seconds.

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

3.2 Low-dimensional data

In this section we study J_{RSJ}, J_{EGO}, J_{EGO*} and J_G approaches for low-dimensional data. For the experiments in this section, the value of ε is varied so as to achieve meaningful selectivity. When the *right* y-axis is present in a figure, it plots the selectivity values for each value of ε in the experiments, in actual number of matching points. As expected, the selectivity, shown by the line with the '$\times$', increases as ε increases in each graph.

In all figures in this section, the execution time of all join techniques monotonically increases as the selectivity increases, except for the low selectivity case of J_{EGO}. The reason why J_{EGO} is an exception is explained in the subsequent subsection.

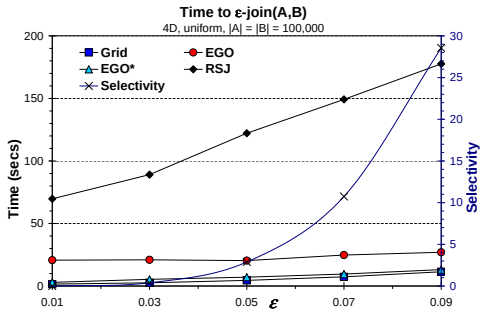


Fig. 9. Join 4D uniform data, with J_{RSJ}

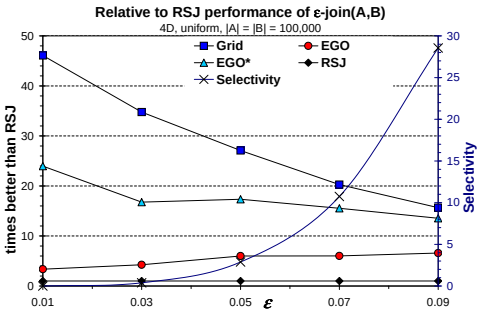


Fig. 10. Join 4D uniform data, performance relative to J_{RSJ}

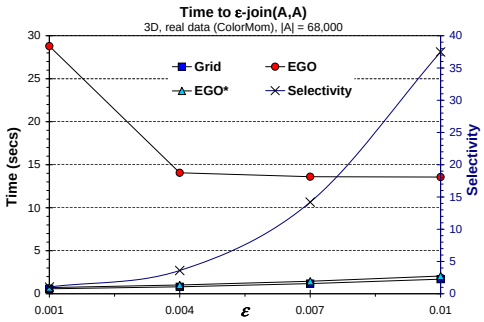


Fig. 11. Join 3D real data

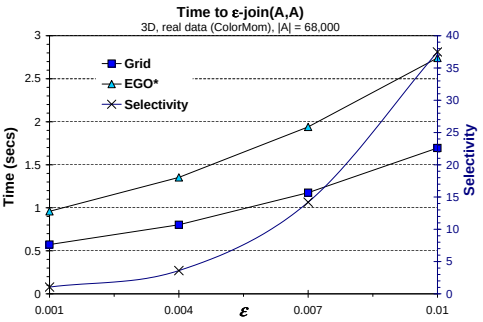


Fig. 12. Join 3D real data without J_{EGO} (for clarity)

• J_{RSJ} *technique:* In all experiments, the results of J_{RSJ} are substantially worse than the results of the other methods. Consequently, J_{RSJ} is not shown in most of the figures except for Figures 9 and 10 which have been included for the purpose of demonstrating J_{RSJ}'s poor performance. Figures 9 and 10 study the effect of varying ε on the efficiency of the join techniques for 4-dimensional uniformly distributed data where cardinality of both datasets

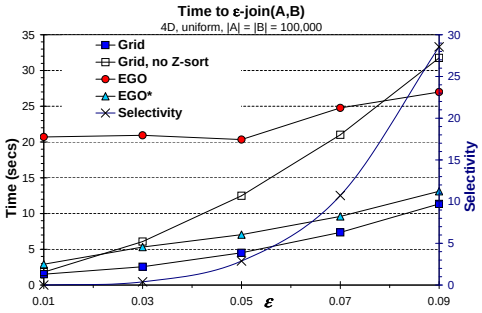


Fig. 13. 4D, uniform, 100,000 points

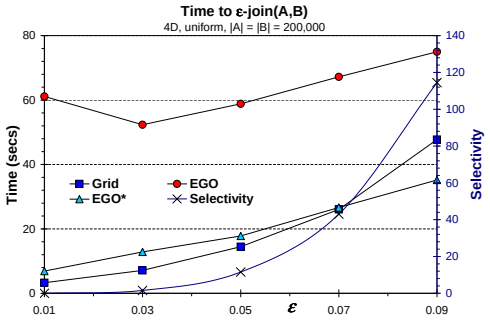


Fig. 14. 4D, uniform, 200,000 points

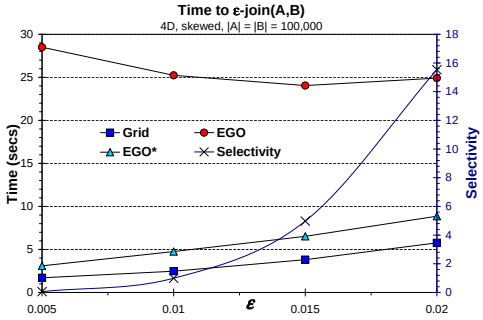


Fig. 15. Join 4D skewed data

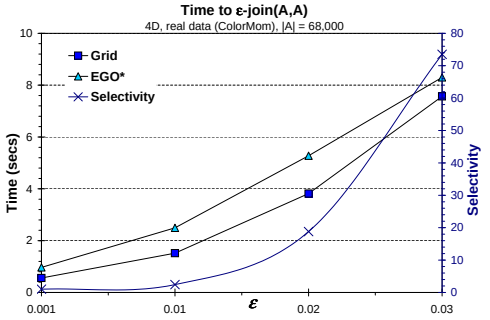


Fig. 16. Join 4D real data

being joined, A and B, is 100,000. Figure 13 is for the same experiment but with J_{RSJ} removed.

• J_{EGO} *technique:* J_{EGO} achieves 3.5–6.5 times better results than those of J_{RSJ}. This asserts J_{EGO} is a good technique for joining low-dimensional data. However, the two new techniques proposed in this paper, J_G and J_{EGO*}, are even better than J_{EGO}.

• J_{EGO*} *technique:* J_{EGO*}'s performance is *always* better than that of J_{EGO} in all experiments. This demonstrates the strength of J_{EGO*}. Because of the selectivity, the values of 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 a sequence will have an inactive dimension. In Figure 10, the performance of J_{EGO*} is 13.5–24 times better than the performance of J_{RSJ}.

Figure 14 illustrates another trend: J_{EGO*} can be better than J_G for low-dimensional data when the selectivity is high. In Figure 14, J_{EGO*} becomes a better choice than J_G for values of ε greater than approximately 0.07. This choice of ε corresponds to a high selectivity of approximately 43.

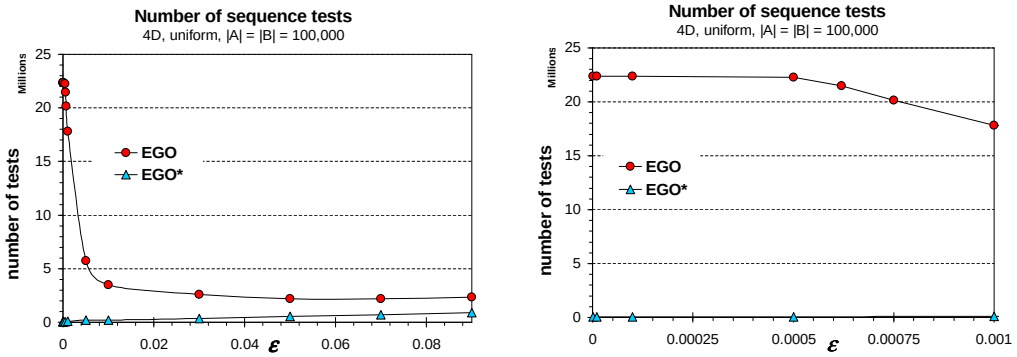
• J_G *technique:* For low-dimensional data, J_G consistently demonstrates the best results among all the tested join techniques. The results of J_G are 15.5–46

times better than the results of J_{RSJ}. They are several times better that the results of J_{EGO*}, except for high-selectivity cases as in Figure 14. Recall that the grid has been established in [10] as the best choice of index for the index nested loop approach among the main memory optimized versions of the grid, R-tree, R*-tree, CR-tree, and quad-tree indexes. Thus, the good results of J_G for low-dimensional data are not very surprising.

Figures 11 and 12 studies a self-join of real 3-dimensional data taken from the ColorMoments file. The cardinality of the dataset is 68,000. Figure 11 plots the three best schemes whereas Figure 12 omits J_{EGO} scheme due to its much poorer performance. In these figures, J_G is almost 2 times better than J_{EGO*} for small values of ε. Figures 13 and 14 study a case of joining 4-dimensional uniform data. The graph on the left is for datasets of cardinality 100,000, and that on the right is for datasets with cardinality 200,000. Figures 15 and 16 demonstrate the results for 4-dimensional skewed and real data. The trends in these figures are consistent with the trends exhibited by the join techniques for the uniform distribution.

Using spatial sorts Figure 13 emphasizes the importance of performing Z-sort on data being joined: the performance improvement is approximately 2.5 times. J_G's performance without Z-sort, in general, is better than J_{EGO} but worse than that of J_{EGO*}.

Comparing the number of sequence tests for J_{EGO*} and J_{EGO}. As discussed in Section 2.2, J_{EGO*}'s better performance over J_{EGO} is a direct result of the large reduction in the number of sequences needed to be compared. Figures 17(a) and 17(b) corroborate this assertion by showing the number of



(a) $\varepsilon \in (0, 0.09]$ (b) $\varepsilon \in (0, 0.001]$

Fig. 17. The number of sequence comparisons by J_{EGO*} and J_{EGO}

such comparisons (in millions) as a function of ε. For both EGO and EGO* heuristics, the number of tests increase as ε increases (starting from ε greater than approximately 0.5). This is because as ε increases the bounds on sequences the two methods produce get courser. Since the size of each cell is equal to ε, the bounds of the sequences are more likely to be within ε distance

from each other. [8]

The number of sequence tests performed by J_{EGO} increases as ε approaches zero. This is because it becomes more and more challenging to find sequences with at least one inactive dimension (the EGO heuristic utilizes inactive dimensions). Figures 17(b) shows that as soon as ε becomes smaller than a certain value the curve stabilizes since almost no sequence with an inactive dimension is found. The J_{EGO*} does not suffer from this drawback since it utilizes both inactive and active dimensions.

Separating costs of join phases. The join procedures of J_{EGO}, J_{EGO*}, and J_G consist of several phases. All the methods first sort the points in the two arrays: J_{EGO} and J_{EGO*} use EGO-sort while J_G uses Z-sort. Thus, we will refer to the first phase of each algorithm as the *sort* phase. The next two phases

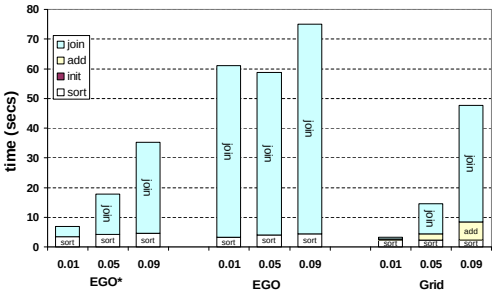


Fig. 18. Cost of join phases, 4D, uniform data , $|A| = |B| = 200,000$

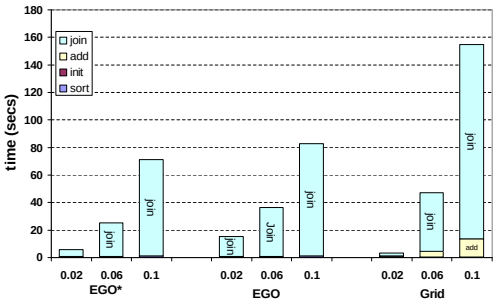


Fig. 19. Cost of join phases, 9D, real data (ColorMom), $|A| = 68,000$, self--join

of J_G are the *init* phase where the grid is initialized and the *add* phase where the grid index is built on top of the ε-radius circles. Finally, the last phase of all methods is the *join* phase. It is interesting to study the contribution of each phase to the overall cost for each method. Figures 18 and 19 plot the time spent in different phases of the join procedures for a join of 4-dimensional uniform data and for a self-join of 9-dimensional real data. The cost of the

[8] To illustrate this point consider EGO*-heuristic, $[0,1]^2$ domain and two points $a(0.05, 0.05)$ and $b(0.55, 0.55)$. The distance $d(a,b)$ is equal to $\sqrt{(0.55-0.05)^2+(0.55-0.05)^2}$ which is $\sqrt{\frac{1}{2}}$ or approximately 0.707. Assume sequence S_a consists of only one point a and sequence S_b consists of only b. If ε is 0.2 then the corresponding cells for a and b are $C_a(0,0)$ and $C_b(2,2)$. Because the cell size is ε, S_a is bounded by a rectangle $R_a = [0, 0.2] \times [0, 0.2]$ and S_b by $R_b = [0.4, 0.6] \times [0.4, 0.6]$. These rectangles are not within ε distance from each other and thus the heuristic will successfully return that the sequences are non-joinable. On the other hand, if ε is say 0.5, then the cells are $C_a(0,0)$ and $C_b(1,1)$, the corresponding rectangles are much courser (larger): $R_a = [0, 0.5] \times [0, 0.5]$ and $R_b = [0.5, 1] \times [0.5, 1]$. Now R_a and R_b are within ε distance from each other and thus the heuristic will fail to correctly determine that S_a and S_b are non-joinable.

join phase is dominant in most of the cases. The contribution of the *sort* phase is significant only for the low-dimensional case of 4-dimensional data. Notice that if we ignore the sort cost, the relative stand of the methods will not change. The cost of the *init* phase is almost nonexistent. Finally, the cost of the *add* phase of J_G, while is not negligible, is still dominated by the cost of the *join* phase.

3.3 High-dimensional data

We now study the performance of the join techniques for high-dimensional data. The results for 9-dimensional uniformly distributed data are illustrated in Figures 20 and 21. Figure 22 studies a case of joining 9-dimensional skewed data, and Figure 23 plots the results for 9-dimensional real data. Figures 24

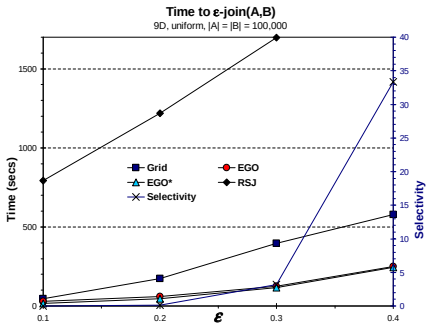


Fig. 20. Join 9D uniform data

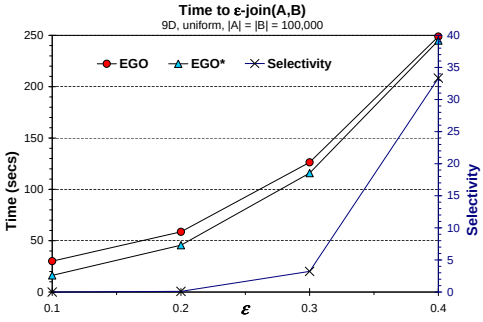


Fig. 21. Best two techniques of Figure 20

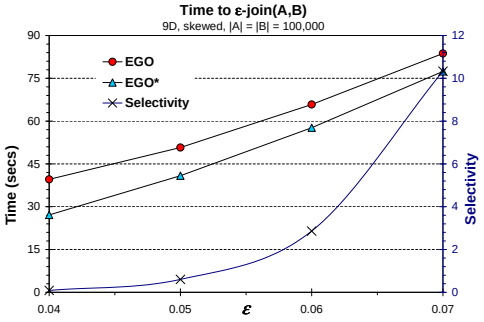


Fig. 22. Join 9D skewed data

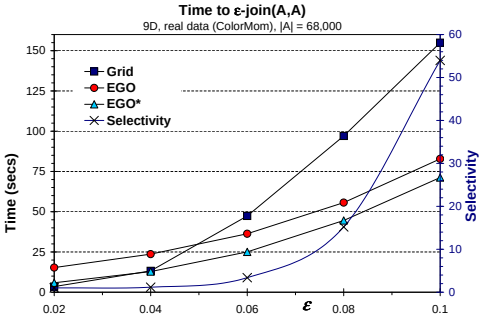


Fig. 23. Join 9D real data

and 25 show the results for the 9- and 16-dimensional real data respectively.

As in the low-dimensional data case, for all tested cases J_{RSJ} has demonstrated substantially worse results. Therefore the results of J_{RSJ} are omitted from most of the graphs, except for Figure 20.

Unlike the case of the low-dimensional data, J_{EGO} and J_{EGO*} are now better than J_G. J_G is not a good choice for the high-dimensional data, hence its

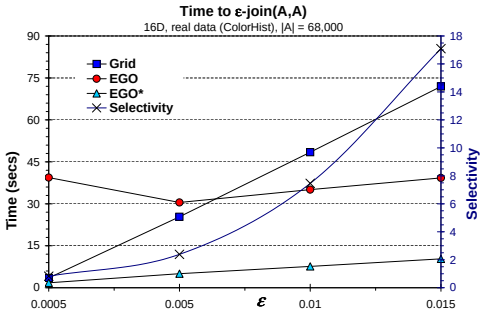


Fig. 24. Join 16D real data

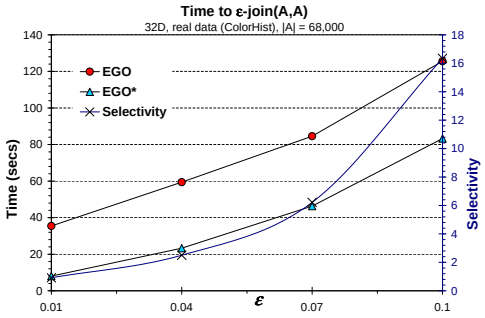


Fig. 25. Join 32D real data

results are often omitted for clarity of comparisons of the results of J_{EGO} and J_{EGO*}. A consistent trend in all experiments is that the performance of J_{EGO*} is *always* better than the performance of J_{EGO}. The difference is especially noticeable for low selectivity cases. This is a general trend: J_{EGO} does not work well for smaller epsilons, because in this case a sequence is less likely to have an inactive dimension. J_{EGO*} does not suffer from this limitation. The behavior of the J_{EGO} curve is different from that of the low-dimensional case, except for Figure 24. For the low-dimensional case the performance of J_{EGO} would increase first with ε increasing and then decrease whereas for the high-dimensional case it typically monotonically decreases. This is because with the increased dimensionality larger values of ε should be used to get reasonable selectivity. Consequently, the effect where the performance of J_{EGO} would improve first for small values of ε, is no longer present.

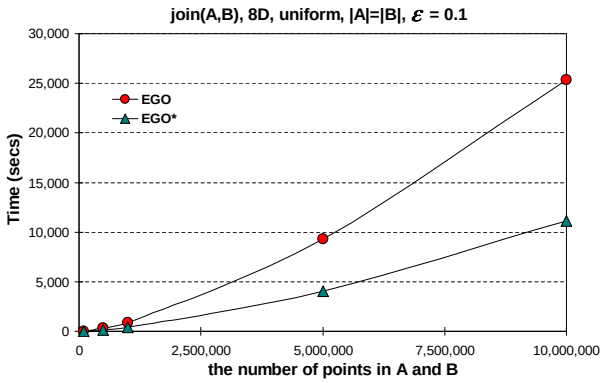


Fig. 26. Join 8D uniform, large datasets

Datasets with large cardinality. Figure 26 studies the performance of J_{EGO} and J_{EGO*} for large volumes of 8D data uniformly distributed in $[0,1]^8$. In this experiment, the cardinality of each distinct dataset A and B is not fixed as before, but varied from 100,000 points to 10,000,000 points per each dataset. The setup of this experiment is similar to that of Figure 10(left) in [7], except in [7] the maximum number of points is 40,000,000. Also, it is not very clear whether [7] studies a self-join, or a join of two distinct datasets. In the experiment illustrated in Figure 26, J_{EGO*} outperforms J_{EGO} by a factor

of approximately 2.15.

Choosing to-be-indexed dataset. When a join of two datasets is to be computed using Grid-join, an index is built on one of the two datasets. Naturally, the question of which dataset to build the index on arises. We ran experiments to study this issue. The results indicate that building the index on the smaller dataset give better results.

4 Related work

The problem of the similarity join of two datasets is to identify pairs of objects, one from each dataset, such that they satisfy a certain constraint. If both datasets are the same, this corresponds to a self-join. The most common join constraint is that of proximity: i.e. the two objects should be within a certain distance of each other. This corresponds to the ε-join where ε is the threshold distance beyond which objects are no longer considered close enough to be joined. Below we discuss some of the most prominent solutions for efficient computation of similarity joins.

Shim et. al. [14] propose to use ε-KDB-tree for performing high-dimensional similarity joins of massive data. The main-memory based ε-KDB-tree and the corresponding algorithm for similarity join are modified to produce a disk-based solution that can scale to larger datasets. Whenever the number of points in a leaf node exceeds a certain threshold, it is split into $\lfloor 1/\varepsilon \rfloor$ stripes,[9] each of width equal to or slightly greater than ε in the i^{th} dimension. If the leaf node is at level i, then the i^{th} dimension is used for splitting. The join is performed by traversing the index structures for each of the datasets. Each leaf node can join only with its two adjacent siblings. The points are first sorted with the first splitting dimension and stored in an external file.

The R-Tree Spatial Join (RSJ) algorithm [9] works with an R-tree index built on the two datasets being joined. The algorithm is recursively applied to corresponding children if their minimum bounding rectangles (MBRs) are within distance ε of each other. Several optimizations of this basic algorithm have been proposed [15]. A cost model for spatial joins was introduced in [16]. The Multipage Index (MuX) was also introduced that optimizes for I/O and CPU cost at the same time. In [2] 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 [3]. Shafer et al. in [17] present a method of

[9] Note that for high-dimensional data ε can easily exceed 0.5 rendering this approach into a brute force method.

parallelizing high-dimensional proximity joins. The ε-KDB-tree is parallelized and compared with the approach of space partitioning. Koudas et al. [18] have proposed a generalization of the Size Separation Spatial Join Algorithm, named Multidimensional Spatial Join (MSJ).

Recently, Böhm et al. [7] proposed the EGO-join. Both datasets of points being joined are first sorted in accordance with the Epsilon Grid Order (EGO). The EGO-join procedure is recursive. A heuristic is utilized for determining non-joinable sequences. More details about EGO-join are covered in Section 2.2. The EGO-join was shown to outperform other join methods in [7].

An excellent overview of multidimensional index structures including grid-like and Quad-tree based structures can be found in [19]. Main-memory optimization of disk-based index structures has been explored recently for B+-trees [20] and multidimensional indexes [11]. Both studies investigate the redesign of the nodes in order to improve cache performance. Another problem that is related to the problem of similarity joins is the problem of similarity searching. This problem has been well studied by several researchers [21–27]. Many of those solutions utilize nearest-neighbor queries.

5 Conclusions

This paper develops two novel in-memory methods for fast similarity join of multidimensional data. It demonstrates the advantage of the proposed methods, called Grid-join and EGO*-join, over the state of the art technique by evaluating them on low- and high-dimensional data. The novel Grid-join approach shows the best results for low-dimensional case, or when the value of ε is very small. The EGO*-join technique shows the best results for high-dimensional data or when the value of ε is large. The empirical and analytical evaluation demonstrates that the proposed EGO*-join technique significantly outperforms the state of the art EGO-join method. Based upon the experimental results, the recommendations for choosing a join algorithm are summarized in Table 1.

	Small ε	**Average 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
Choosing a join algorithm

The paper also studies the effect of the parameter ε on the resulting selectivity of the join and provides recommendations for choosing ε to achieve meaningful

selectivity. The paper demonstrates that improper selectivity can lead to the wrong conclusions.

Finally, the paper develops a cost-estimation function for Grid-join. Such a function can be employed for choosing the size of the grid, or by a query optimizer for selecting a query execution plan.

As future work, we plan to look into the problem of similarity join for datasets of very high dimensionality, such as 100 dimensions or more. This is an interesting challenge since many existing methods rely on the fact that ε is likely to be (much) less than 1 for $[0,1]^d$ domain. However, for data with very high dimensionality, one can expect to encounter the values of ε which are greater than (or comparable to) 1.

References

[1] D. V. Kalashnikov, S. Prabhakar, Similarity join for low- and high-dimensional data, in: Proc. of the 8th International Conference on Database Systems for Advanced Applications (DASFAA 2003), IEEE Computer Society Press, Kyoto, Japan, 2003.

[2] J. M. Patel, D. J. DeWitt, Partition based spatial-merge join, in: Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996, ACM Press, 1996, pp. 259–270.

[3] M.-L. Lo, C. V. Ravishankar, Spatial hash-joins, in: Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996, ACM Press, 1996, pp. 247–258.

[4] S. Guha, R. Rastogi, K. Shim, CURE: An efficient clustering algorithm for large databases, in: Proceedings of the ACM SIGMOD international Conference on Management of data, 1998.

[5] E. M. Knorr, R. T. Ng, Algorithms for mining distance-based outliers in large datasets, in: Proceedings of the International Conference on Very Large Data Bases, 1998.

[6] K. Koperski, J. Han, Discovery of spatial association rules in geographic information databases, in: International Symposium on Large Spatial Databases, 1995.

[7] C. Böhm, B. Braunmüller, F. Krebs, H.-P. Kriegel, Epsilon grid order: an algorithm for the similarity join on massive high-dimensional data, in: Proceedings of the 2001 ACM SIGMOD international conference on Management of data, ACM Press, 2001, pp. 379–388.

[8] C. Böhm, B. Braunmüller, M. Breunig, H.-P. Kriegel, Fast clustering based on high-dimensional similarity joins, in: Intl. Conference on Information and Knowledge Management, 2000.

[9] T. Brinkhoff, H.-P. Kriegel, B. Seeger, Efficient processing of spatial joins using R-trees, in: Proceedings of the ACM SIGMOD international Conference on Management of data, 1993.

[10] D. V. Kalashnikov, S. Prabhakar, S. Hambrusch, W. Aref, Efficient evaluation of continuous range queries on moving objects, in: Proc. of the 13th International Conference on Database and Expert Systems Applications (DEXA 2002), Aix en Provence, France, 2002.

[11] K. Kim, S. Cha, K. Kwon, Optimizing mutidimensional index trees for main memory access, in: Proc. of ACM SIGMOD Conf., Santa Barbara, CA, 2001.

[12] S. Prabhakar, Y. Xia, D. Kalashnikov, W. Aref, S. Hambrusch, Query indexing and velocity constrained indexing: Scalable techniques for continuous queries on moving objects, IEEE Transactions on Computers 51 (10) (2002) 1124–1140.

[13] D. V. Kalashnikov, S. Prabhakar, S. Hambrusch, Main memory evaluation of monitoring queries over moving objects, Distributed and Parallel Databases, An International Journal (DAPD) 15 (2) (2004) 117–135.

[14] K. Shim, R. Srikant, R. Agrawal, High-dimensional similarity joins, in: Proceedings of the Thirteenth International Conference on Data Engineering, April 7-11, 1997 Birmingham U.K, IEEE Computer Society, 1997, pp. 301–311.

[15] Y.-W. Huang, N. Jing, E. A. Rundensteiner, Spatial joins using r-trees: Breadth-first traversal with global optimizations, in: M. Jarke, M. J. Carey, K. R. Dittrich, F. H. Lochovsky, P. Loucopoulos, M. A. Jeusfeld (Eds.), VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece, Morgan Kaufmann, 1997, pp. 396–405.

[16] C. Böhm, H.-P. Kriegel, A cost model and index architecture for the similarity join, in: Proceedings of the International Conference on Data Engineering, 2001.

[17] J. C. Shafer, R. Agrawal, Parallel algorithms for high-dimensional similarity joins for data mining applications, in: VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece, Morgan Kaufmann, 1997, pp. 176–185.

[18] N. Koudas, K. C. Sevcik, High dimensional similarity joins: Algorithms and performance evaluation, in: Proceedings of the Fourteenth International Conference on Data Engineering, IEEE Computer Society, 1998, pp. 466–475.

[19] J. Tayeb, Ö. Ulusoy, O. Wolfson, A quadtree-based dynamic attribute indexing method, The Computer Journal 41 (3).

[20] J. Rao, K. A. Ross, Making B^+-trees cache conscious in main memory, in: Proc. of ACM SIGMOD Conf., Dallas, TX, 2000.

[21] W. Niblack, R. Barber, W. Equitz, M. Flickner, E. Glasman, D. Petkovic, P. Yanker, The QBIC project: Querying images by content using color, texture and shape, in: Proc. of the SPIE Conf. 1908 on Storage and Retrieval for Image and Video Databases, Vol. 1908, 1993, pp. 173–187.

[22] F. Korn, N. Sidiropoulos, C. Faloutsos, E. Siegel, Z. protopapas, Fast nearest neighbor search in medical image databases, in: Proceedings of the Int. Conf. on Very Large Data Bases, Mumbai, India, 1996, pp. 215–226.

[23] S. Mehrotra, Y. Rui, M. Ortega, T. S. Huang, Supporting content-based queries over images in mars, in: Proceedings of the 4th IEEE Int. Conf. Multimedia Computing and Systems, Ottawa, Ontario, Canada, 1997, pp. 632–633.

[24] S. Mehrotra, et al, MARS project, UC Irvine, http://www-db.ics.uci.edu/pages/research/mars.shtml.

[25] A. Gionis, P. Indyk, R. Motwani, Similarity search in high dimensions via hashing, in: Proceedings of the Int. Conf. on Very Large Data Bases (VLDB), 1999.

[26] H.-P. Kriegel, S. Brecheisen, P. Kroger, M. Pfeifle, M. Schubert, Using sets of feature vectors for similarity search on voxelized cad objects, in: Proc. ACM SIGMOD Int. Conf. on Management of Data, 2003, pp. 587–598.

[27] R. Agrawal, C. Faloutsos, A. Swami, Efficient similarity search in sequence databases, in: Proc. of the 4th Intl. Conference on Foundations of Data Organization and Algorithms (FODO), 1993.

Recommended by Yannis Ioannidis, Area Editor

A Choice of grid size

In this section, we develop cost estimator functions for Grid-join. These functions can be used to determine the appropriate choice of grid size for computing the ε-join for a specific problem. The discussion focuses on the case of two dimensions, but can be generalized to any number of dimensions in a straightforward manner.

Table A.1 lists parameters needed for our analysis. All the parameters are known before the join, except for grid size n, which needs to be determined. We are interested in finding n such that the time needed for the join is minimized. Furthermore, if there are several values of n that yield minimal or close to minimal join cost, then we are interested in the smallest such n. This is because the memory requirements for the grid increase with the number of cells in the grid.

In order to determine the relationship between the join cost and the various parameters of the problem, we develop what we call estimator (or predictor) functions for the various phases of grid-join. Once the predictor functions are constructed, a suitable choice for n can be found by identifying a minimal value of the cost. For the value of n selected, the predictor functions are also

Table A.1
Parameters used for ε-join

Parameter	*Meaning*
A	first dataset for join
B	second dataset (on which the index is built)
$k = \|A\|$	cardinality of A
$m = \|B\|$	cardinality of B
c	length of side of a cell
$n = 1/c$	grid size: $n \times n$ grid
eps, ε	epsilon parameter for the join

useful in providing an estimated cost to the query optimizer which can use this information to decide whether or not Grid-join should be used for the problem.

In our analysis we assume uniform distribution of points in datasets A and B. The grid-join procedure can be divided into three phases:

(1) **init phase**: initialization of the grid pointers and lists
(2) **add phase**: loading the data into the grid
(3) **proc phase**: processing the point-queries using the grid.

Init and *add* phases collectively are called the *build index* phase. There is a tradeoff between the *build* and *proc* phases with respect to the grid size n. With fewer cells, each circle is likely to intersect fewer cells and thus be added to fewer `full` and `part` lists. On the other hand, with fewer cells the average length of the `part` lists is likely to be larger and each query may take longer to process. In other words, the coarser the grid (i.e., the smaller the value of n), the faster the *build* phase, but the slower the *proc* phase. Due to this fact, the total time needed for join is likely to be a concave downwards function of n. This has been the case in all our experiments.

Upper Bound While the general trend is that a finer grid would imply shorter query processing time (since the `part` lists would be shorter or empty), beyond a certain point, a finer grid may not noticeably improve performance. For our implementation, the difference in time needed to process a cell when its `part` list is empty, versus when its `part` list has size one, is very small. It is enough to choose grid size such that the size of `part` list is one and further partitioning does not noticeably improve query processing time. Thus we can estimate an upper bound n_{upper} for the optimal value of n, and search for the optimal value of n only among the values in the interval $[1, n_{upper}]$.

For the 2-dimensional case, if the grid is built on squares instead of circles, it

can be shown that an upper bound is given by [10]:

$$n_{upper} = \begin{cases} \lceil 4qm \rceil & \text{if } q > \frac{1}{2\sqrt{m}}; \\ \left\lceil \frac{1}{\frac{1}{\sqrt{m}} - q} \right\rceil & \text{otherwise.} \end{cases}$$

In this formula, q is the size of a side of each square. Since for ε-join the index is built on circles, the formula is reused by approximating the circle by a square with the same area, that is $q \approx \varepsilon\sqrt{\pi}$. The corresponding formula for n_{upper} is therefore:

$$n_{upper} = \begin{cases} \lceil 4\sqrt{\pi}\varepsilon m \rceil & \text{if } \varepsilon > \frac{1}{2\sqrt{\pi m}}; \\ \left\lceil \frac{1}{\frac{1}{\sqrt{m}} - \varepsilon\sqrt{\pi}} \right\rceil & \text{otherwise.} \end{cases}$$

A finer grid than that specified by the above formula will give very minor performance improvement while incurring a large memory penalty. Thus, the formula establishes the upper bound for the grid size. However, if the value returned by the formula is too large, the grid might not fit in memory. In that case n is further limited by the memory space availability.

In our experiments, the optimal value for grid size tended to be closer to 1 rather than to n_{upper}, as will be demonstrated later in this section in Figures A.4 and A.5.

Analysis For each of the phases of the Grid-join, the analysis is conducted as follows. 1) First, the parameters on which a phase depends on are determined. 2) Then, the nature of the dependence on each distinct parameter is predicted based on the algorithm and implementation of the grid. Since the grid is a simple data structure, the nature of the dependence on a parameter, as a rule, is not complicated. 3) Next, the dependence on the combination of the parameters is predicted, based on the dependence on each parameter. 4) Finally, an explanation is given on how to calibrate the predictor functions for a specific machine.

Estimating the *init* phase: The time to initialize the index depends only on the grid size n. The process of index initialization can be viewed as $O(1)$ operations, followed by the initialization of n^2 cells. Thus, the index initialization time is expected to be a polynomial such that the degree of n is 2: $P_{init}(n) = an^2 + bn + c$, for some coefficients a, b, and c. The values of the coefficients depend upon the particular machine on which the initialization is performed. They can be determined through a calibration step. To validate the correctness of this estimator, we calibrated it for a given machine. The corresponding estimator function was then used to predict the performance

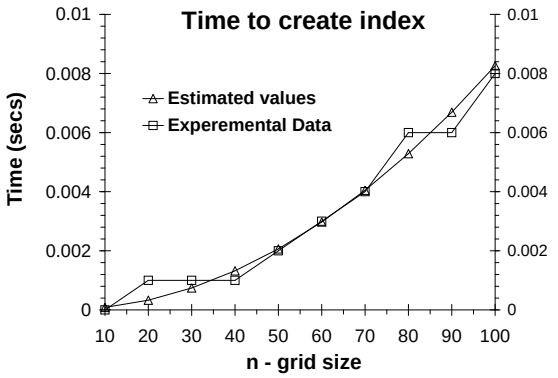


Fig. A.1. Time to initialize index $n \in [10, 100]$

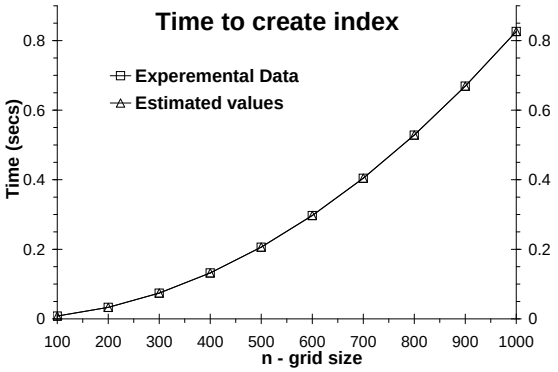


Fig. A.2. Time to initialize index $n \in [100, 1000]$

for other values of n not used for the calibration. The result is illustrated in Figures A.1 and A.2 ($a = 8.26 \times 10^{-7}$, $b = 0$, and $c = 0$). These two figures are for different ranges of n, that is, n varies from 10 to 100 in Figure A.1 and it varies from 100 to $1,000$ in Figure A.2. The figures plot the actual grid initialization time, measured for different values of n, and the initialization time predicted by the estimator function. Let us observe that the estimator function computes very accurate approximations of the actual values, especially for larger values of n.

Figures A.1 and A.2 show that the time needed for index initialization phase can be approximated well with a simple polynomial. Any numerical method can be used for calibrating the coefficients a, b, and c for a particular machine.

Estimating the *add* phase: This phase is more complicated than the init phase because it depends on three parameters: n – the grid size , m – the cardinality of the indexed dataset B, and ε. By analyzing the dependence on each parameter separately, we estimate that the overall function can be represented as a polynomial $P_{add}(n, m, \varepsilon) = a_{17}n^2\varepsilon^2 m + \cdots + a_1 m + a_0$ such that the degree of n and ε is 2 and the degree of m is 1. The next step is

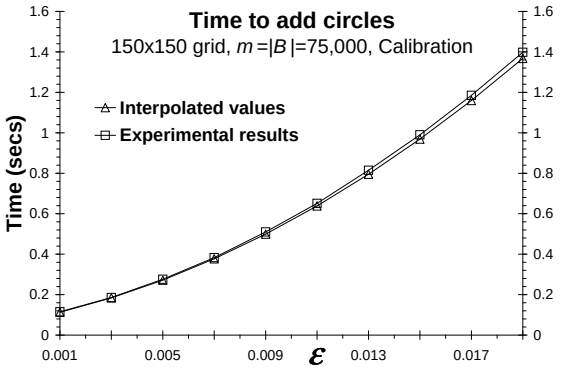


Fig. A.3. Estimation with polynomial for add phase

to calibrate the coefficients a_i's. This can be done by solving a system of 18 linear equations. These equations can be obtained by choosing three different values of n, three values of ε, and two values of m ($3 \times 3 \times 2 = 18$).

The combinations of the following calibration points have been examined in order to get the coefficients: $n_0 = 10$, $n_1 = 100$, $n_2 = 200$; $\varepsilon_0 = 0.001$, $\varepsilon_1 = 0.01$, $\varepsilon_2 = 0.02$; $m_0 = 50$, and $m_1 = 100$. The choice of values implies we assume that typically $n \in [10, 200]$, $\varepsilon \in [0.001, 0.02]$, and $m \in [50, 100]$. The linear system was solved using the Gaussian elimination with pivoting method. Figure A.3 demonstrates the time needed for the add phase for various values of ε when $n = 150$ and $m = 75$ and the other curve represents our interpolation polynomial. Again, we observe that the estimator function is highly accurate. In fact, we never encountered more than a 3% relative error in our experiments.

Estimating the *proc* phase: The processing phase depends on all parameters: n – grid size, $k = |A|$, $m = |B|$, and ε. Thankfully, the dependence on k is linear since each point is processed independently of other points. Thus, once the solution for some fixed k_0 is known, the solution for an arbitrary k can be computed by scaling. However, there is a small complication: the average lengths of the `full` and `part` lists are given by different formulae depending upon whether cell size c is greater than $\sqrt{\pi}\varepsilon$ or not (see [10], in our case query side size q is replaced by $\sqrt{\pi}\varepsilon$).

Consequently, the *proc* phase cost can be estimated by two polynomials (depending on whether $\sqrt{\pi}\varepsilon \geq c$ or not), which we denote as $P_{proc,\sqrt{\pi}\varepsilon \geq c}(c, \varepsilon, m, k_0)$ and $P_{proc,\sqrt{\pi}\varepsilon < c}(c, \varepsilon, m, k_0)$. Each of them has the type $P(c, \varepsilon, m, k_0) \equiv a_{17}c^2\varepsilon^2 m + \cdots + a_1 m + a_0$ such that the degree of c and ε is 2 and the degree of m is 1. Once again, the calibration can be done by solving a system of 18 linear equations for each of the two cases.

Estimating the total time: The estimated total time needed for Grid-join is the sum of estimated time needed for each phase. Figures A.4 and A.5 demonstrate estimation of time needed for Grid-join when $\varepsilon = 0.001$,

$m = 20,000$, $k = 10,000$ as a function of grid size n. The estimator functions of each phase were calibrated using different values than those shown in the graph.

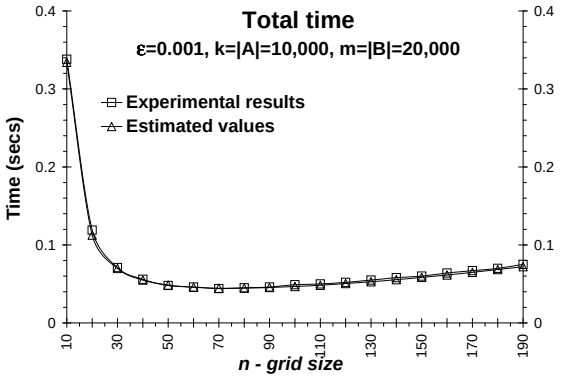


Fig. A.4. Estimation of total join time, $n \in [10, 190]$

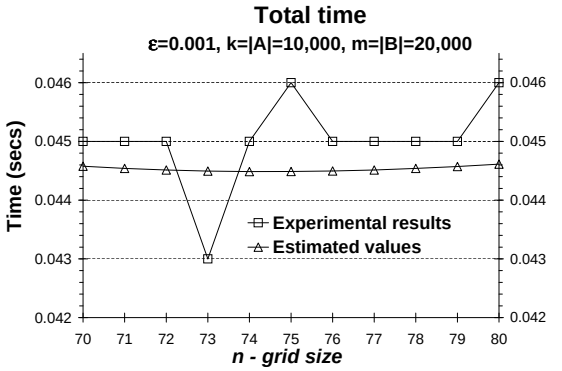


Fig. A.5. Estimation of total join time, $n \in [70, 80]$

A simple *bisection method* was employed to get the estimated optimal value of n, as computed by the estimator function $f_{est}(n)$ for the total execution time. This method assumes that it is given a concave downwards function $f_{est}(n)$, defined on $[a, b]$. The function $f_{est}(n)$ has been concave downwards in all our experiments, however in future work we plan to prove that the estimator function is always concave downwards for various combinations of parameters. The objective of the bisection method is to find the leftmost minimum of $f_{est}(n)$ on the interval $[a, b]$. The method first computes $c = (a + b)/2$. If $f(c-1) \leq f(c+1)$, then it makes the new b equal c and repeats the procedure, otherwise it makes the new a equal c and repeat the procedure. The process is repeated until $(b - a) < 2$.

The bisection method for the example in Figures A.4 and A.5 returns '74' as an estimated optimal value for n. Experimentally, we found that the actual optimal value for n was 73. The difference in execution time for the grid-join with 73×73 grid and with 74×74 grid is just two milliseconds for the given settings. This illustrates the high accuracy of the estimator functions. Let us

observe that the results of interpolation look even better if they are rounded to the closest millisecond values.