# Similarity Join for Low- and High- Dimensional Data*

Dmitri V. Kalashnikov Sunil Prabhakar
Department of Computer Science, Purdue University.
Email: {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 should be processed in main memory. In this paper we develop two new spatial join algorithms (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 $\varepsilon$ parameter. We also point out the significance of the choice of this parameter for ensuring that the selectivity achieved is reasonable. For low-dimensional data both proposed algorithms clearly outperform EGO-join. For high-dimensional data, the proposed EGO*-join technique significantly outperforms the EGO-join. An analysis of the cost of Grid-join is presented and cost estimator functions are developed. These are used to choose an appropriate grid size for optimal performance and can also be used by a query optimizer to compute the estimated cost of Grid-join.

## 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 [13, 12]. The state-of-the-art algorithms for several basic data mining operations such as clustering [5], outlier detection [9], and association rule mining [10] require the processing of all pairs of points within a certain distance to each other[2]. Thus a similarity join can serve as the first step for many of these operations [1].

The problem of efficient computation of similarity joins has been addressed by several researchers. Most researchers have focussed their attention on disk-based joins for high-dimensional data. Current high-end

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

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 $\varepsilon$-joins. Given two multisets $A$ and $B$ of d-dimensional points and value $\varepsilon \in \mathbf{R}$, the goal of a join operation is to identify all pairs of points, one from each set, that are within distance $\varepsilon$ from each other, i.e. $\{(a,b) |\ a \in A,\ b \in B,$ and $\|a-b\| < \varepsilon\}$.

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: *Grid-Join* and *EGO*-Join*, and evaluate the performance of these methods alongwith the state-of-the-art algorithm (EGO-Join) [**?**] and the RSJ Join [4] which has served as a benchmark for most algorithms.

These techniques are compared through experiments using synthetic and real data. We considered the total wall-clock time for performing a join without ignoring any costs, such as pre-sorting data, building/maintaining index etc. The experimental results show that the Grid-join approach showed the best results for low-dimensional data.

Under the Grid-Join approach, the join of two sets $A$ 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 $\varepsilon$ centered at the first two coordinates of points from set $B$. The first two coordinates of points from set $A$ 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 is not accidental, it is based upon our earlier results for main memory evaluation of range queries. In [7] 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 choice of join method. The EGO*-Join that we propose in this paper is based upon the state-of-the-art EGO-Join algorihtm. The Epsilon Grid Order (EGO) join [2] algorithm was shown to outperform other techniques for spatial joins of high-dimensional data. The new 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 $\varepsilon$ is not large. The RSJ algorithm is significantly poorer than all other theree algorithms in all experiments. In order to join two sets using RSJ, an R-tree index needs to be built or maintained on both of these sets. But

unlike the case of certain approaches these indexes need not be rebuilt when the join is recomputed with a different value of ε.

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 values of ε are selected result in very small (almost no point from the first set joins with a point from the second set) or very high selectivities. In Section 4.1 we present a discussion on how to choose appropriate values of ε.

For the case of moving object environments, if the join is to be computed between a set of fixed objects and a set of moving objects, existing techniques that index both sets are not likely to perform well due to the need for repeated update to the index as the objects move [7, 14]. The Grid-join technique provides an excellent solution to this problem since the index can be built on the fixed objects requiring no updates. If both sets of objects are moving, then the index can be built on either set. Due to its simple structure, the Grid index is easier to update than other indexes such as R-trees or R*-trees.

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, and value of ε.
- 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 join using Grid-join. These functions are used to choose an optimal grid size.

The rest of this paper is organized as follows. Related work is discussed in Section 2. The new Grid-join and EGO*-join algorithms are presented in Section 3. The proposed join algorithms are evaluated in Section 4, and Section 5 concludes the paper. A sketch of the algorithm for selecting grid size and cost estimator functions for Grid-join are presented in Appendix A.

## 2 RELATED WORK

The problem of the spatial 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. [17] 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 exceed a certain threshold it is split into $\lfloor 1/\varepsilon \rfloor$ stripes[1] 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 data sets. 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 [4] 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 [6]. A cost model for spatial joins was introduced in [3]. The Multipage Index (MuX) was also introduced that optimizes for I/O and CPU cost at the same time.

In [13] 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 [12]. Shafer et al in [16] present a method of parallelizing high-dimensional proximity joins. The ε-KDB-tree is parallelized and compared with the approach of space partitioning. Koudas et al [11] 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. Both sets of points being joined are first sorted in accordance with the so called Epsilon Grid Order (EGO). The EGO-join procedure is recursive. A heuristic is utilized for determining non-joinable sequences. More details about EGO-join will be covered in Section 3.2. The EGO-join was shown to outperform other join methods in [2].

A excellent review of multidimensional index structures including grid-like and Quad-tree based structures can be found in [18]. Main-memory optimization of disk-based index structures has been explored recently for B+-trees [15] and multidimensional indexes [8]. Both studies investigate the redesign of the nodes in order to improve cache performance.

# 3 SIMILARITY JOIN ALGORITHMS

In this section we introduce two new techniques for performing an ε-join: the Grid-join and EGO*-join. The Grid-join technique is based upon a simple uniform grid and builds upon the approach proposed in [7] for evaluating continuous range queries over moving objects. The EGO*-join is based upon EGO-join proposed in [2]. We first present the Grid-join technique and an important optimization for improving the

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

cache hit-rate for Grid-join in main memory (Section 3.1). An analysis of the appropriate grid size as well as cost prediction functions for Grid-join is presented in the Appendix. The EGO*-join method is discussed in Section 3.2.

### 3.1 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. We treat one of the point data sets as a collection of circles of radius ε centered at each point of one of the two sets (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 data set (*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 range 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.

In earlier work [7] 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 techinique 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 [7] the choice of a good main-memory index was investigated. Several key index structures including R-tree, R*-tree, CR-tree [8], quad-tree, and 32-tree [7] 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. The Grid-index is built on circles with ε-radius. 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.

For ease of explanation assume the case of 2-dimensional data. The grid-index 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 a grid. Throughout the paper, we assume that the domain is normalized to the unit d-dimensional hyper-cube.

In this example, the domain is divided into a $10 \times 10$ grid of 100 cells, each of size $0.1 \times 0.1$. Since we have a uniform grid, given the coordinates of an object, it is easy to calculate its cell-coordinates in O(1) time. Each cell contains two lists that are identified as $full$ and *part* (see Figure 1a). The $full$ (*part*) list

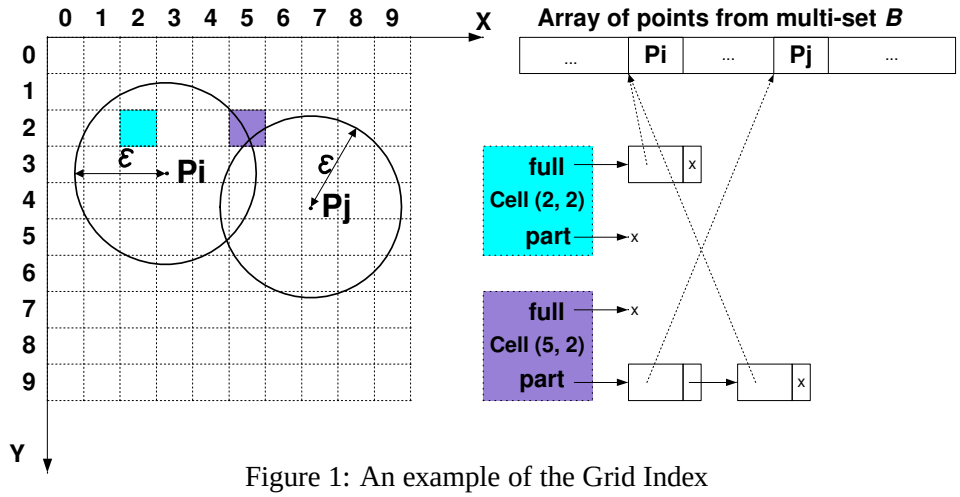



Figure 1: An example of the Grid Index

of a cell contains *pointers* to all the points from $B$ such that a circle with ε-radius around each of them fully (partially) cover the cell.

To find all points within ε-distance from a given 2-dimensional point $a$ first the cell corresponding to $a$ is retrieved. All points in $full$ list are guaranteed to be within ε-distance. 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[2] rather than lists which improves performance by roughly 40% due to the resulting clustering (and thereby reduced cache misses).

The similarity join algorithm which utilizes the grid is called the Grid-join. The Grid-join is described in Figure 2. The z-sort step applies a spatial sort to the two datasets. The need for this step is explained below.

```
grid-join(set A, set B, ε ∈ R)
{
   z-sort(A);
   z-sort(B);

   initialize grid-index;
   add circles to grid with centers in B and ε-radius;

   foreach point a ∈ A
   {
      compute a's cell C_a in grid-index;

      /* process C_a.part list (and C_a.full lists for 2D) */
      find all points {b | b ∈ C_a.part and ||a − b|| < ε};
   }
}
```

Figure 2: Grid-join procedure

The reason for two separate lists per cell for 2-dimensional points is that points in the $full$ list do not

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

need potentially costly checks for relevance since they are guaranteed to be within ε-distance.

**Case of $d$ dimensions** For the general $d$-dimensional case, 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 whether $\|a-b\| < \varepsilon$. Keeping a separate $full$ list is of little value for more than 2 dimensions since now it too needs post-processing to eliminate false positives similar to the *part* list. Therefore only one list is kept for all circles that at least partially intersect the cell in the chosen 2 dimensions. We call this the *part* list.

**Choice of grid size** The performance of grid-join 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. 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 of the join given a grid size. The cost is composed of three components: (a) initializing the empty grid; (b) loading the data (circles) into the index; and (c) processing each point of the other dataset through this index. The Appendix presents 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 perform a grid-join for given parameters or some other join.

**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) for Grid-join.

As can be seen from 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 are also 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 the Grid-index. As we will see in the Experimental section, the performance achieved with Z-sort is almost a factor of $\sim$2.5 times faster than without Z-sorting (for example see Figure 10a.

### 3.2 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. According to [2], the EGO-join algorithm is the state-of-the-art algorithm for ε-join, and was shown to outperform other methods for joining massive, high-dimensional data.

We begin by briefly describing the EGO-join technique as presented in [2] followed by our improvement of EGO-join.

**The Epsilon Grid Order:** The EGO-join is based on the so called Epsilon Grid Ordering (EGO), see [2] for details. In order to impose an EGO on set[3] *A*, a regular grid with the cell size of ε is laid over the data space. The grid is imaginary, and never materialized. By using straightforward operations, 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.

```
EGO-join(set A, set B, ε ∈ R)
{
   EGO-sort(A, ε);
   EGO-sort(B, ε);

   join_sequences(A, B);
}
```

Figure 3: EGO-join Procedure

**EGO-sort:** In order to perform an EGO-join 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. Note, for a subsequent EGO-join 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*.

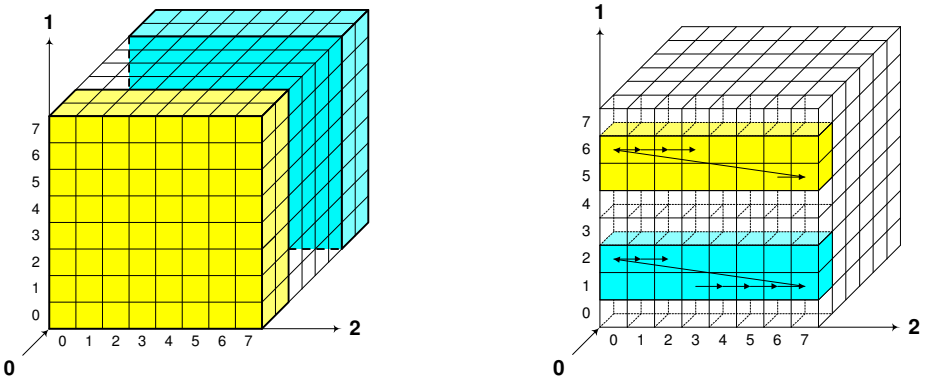
**EGO-heuristic:** A key element of EGO-join is the heuristic used to identify *non-joinable* sequences. The heuristic is based on the number of inactive dimensions, which will be explained shortly. To understand the heuristic, let us consider a simple example. For a short sequence its first and last 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,x,x). Their third coordinates differ – this correspond 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).

[3]Throughout this paper we use *set* instead of *multiset* for short.

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 you find a dimension $\in [0, min-1]$ 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 $\varepsilon$.

**New EGO*-heuristic:** The proposed EGO*-join algorithm is EGO-join with an important change to the heuristic for determining that two sequences are non-joinable. The use of the EGO*-heuristic significantly improves performance of the join, as will be seen in Section 4.

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




(a) (b)

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.

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 EGO-join.

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. Note 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 first and last cells of the sequence and produces the bounding rectangle as output. To understand getBR() algorithm, note that if first and the last cell have *n* prefix equal coordinates (e.g. (1,2,3,4) and (1,2,9,4) have two equal

```
void getBR(BR &rect, Cell &first, Cell &last)
{
   for (int i = 0; i < NUM_DIM; i++)
   {
      rect.lo[i] = first.x[i];
      rect.hi[i] = last.x[i];

      if (rect.lo[i] != rect.hi[i])
      {
         for (int j = i+1; j < NUM_DIM; j++)
         {
            rect.lo[j] = 0;
            rect.hi[j] = MAX_CELL;
         }

         return;
      }
   }
}
```

Figure 5: EGO*-join: procedure for obtaining a Bounding Rectangle of a sequence

first coordinates – (1,2,x,x) ) then all cells of the sequences have the same values in the first n coordinates (e.g. (1,2,x,x,) 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,x) and the upper bound is (1,2,9,x). In general, we cannot say anything definite about the rest of the dimensions, however the lower bound can always be set to 0 and upper bound to MAX_CELL.

```
void join_sequences(A, B)
{
   getBR(BR1, A.first, A.last);
   getBR(BR2, B.first, B.last);

   BR1.inc(); //expand BR1 by one in all directions

   if (BR1 and BR2 do not intersect)
      return;

   //-- continue as in EGO-join --
   ...
}
```

Figure 6: Beginning of EGO*-join: 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 4, EGO*-join significantly outperform EGO-join in all instances. This improvement is a direct result of the reduction of the number of sequences needed to be compared based upon the above criterion.

# 4 EXPERIMENTAL RESULTS

In this section we present the performance results for in-memory ε-join using RSJ, Grid-join, EGO-join [2], and EGO*-join. 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. The machine has 32K of level-1 cache (16K for instructions and 16K for data) and 256K level-2 cache. 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 cost of sorting the data, building or maintaining the index or cost of other operations needed for a particular implementation of ε-join are ignored. No cost is ignored in our experiments for Grid-join, EGO-join, and EGO*-join. One could argue that since for RSJ join the two indexes, once built, need not be rebuilt for different values of ε. While there are many other situations where the two indexes need to be build from scratch for RSJ, we ignore the cost of building and maintaining indexes for RSJ join, thus giving it an advantage.

## 4.1 Correlation between selectivity and ε

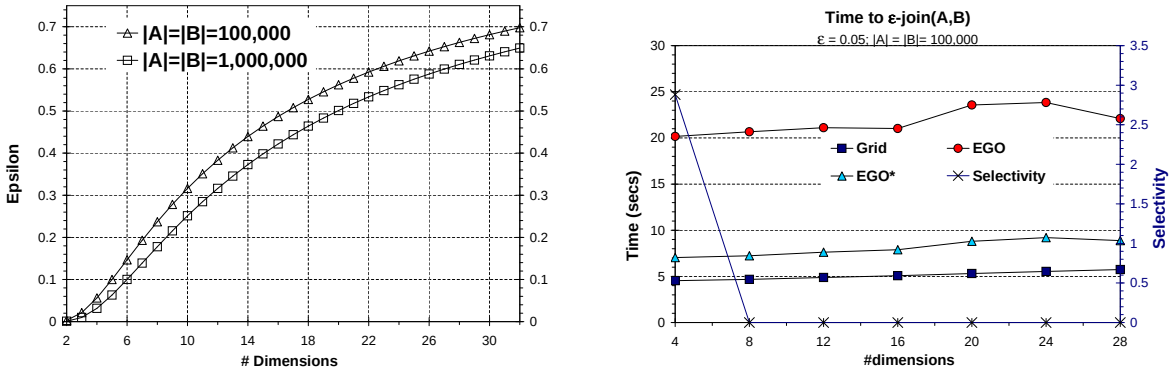
The choice of the parameter ε is critical when performing an ε-join. Little justification for choice of this parameter has been presented in related research. In fact, we present this section because we have discovered that often in similar research selected values of ε are too small.

The choice of the values for 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 $0.1 \times$

0.1 square[4] 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.

Let us estimate what values for ε should be considered for joining high-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-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(a) 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[5]. This is in very close agreement to the experimental results.




(a) (b)

Figure 7: ε-join(A,B) (a) Choosing ε for selectivity close to one for $10^5$ (and $10^6$) points uniformly distributed on $[0,1]^d$ (b) Pitfall of using improper selectivity.

If the domain is not normalized to the unit square, such as in [11], 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 7(b) demonstrates the pitfall of using an improper selectivity. The parameters of the 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 EGO- and EGO*-joins even for high-dimensional cases. However, the contrary is true for a meaningful selectivity as will be seen in Section 4.3. Similar values for the selectivity were obtained using the setup used in [11].

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. The parameter ε is on the *x*-axis, and the time taken by each join method is plotted on the left *y*-axis in seconds.

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

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

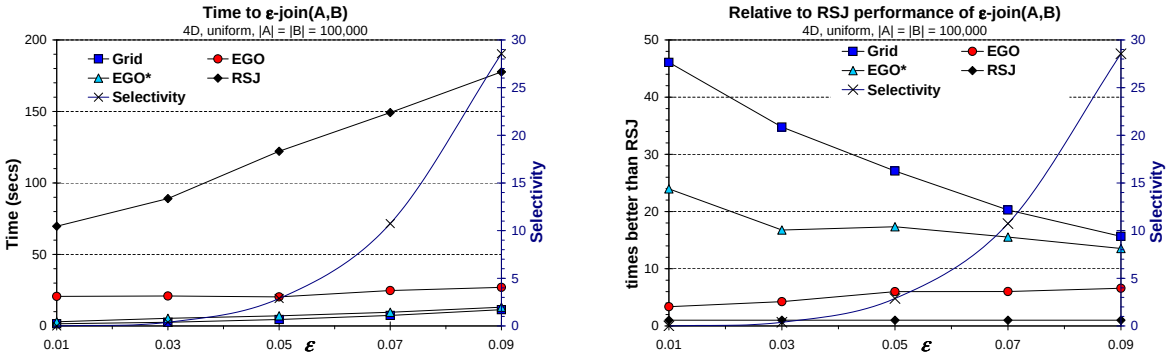
## 4.2 Low-dimensional data

We now present the performance of RSJ, EGO-join, EGO*-join and Grid-join for various settings. The cost of building indexes for RSJ is ignored, giving it an advantage.

The *x*-axis plots the values of ε, which are varied so that meaningful selectivity is achieved. Clearly, if selectivity is 0, then ε is too small and vice versa if the selectivity is more than 100.

In all but one graph the left *y*-axis represents the total time in seconds to do the join for the given settings. The right *y*-axis plots the selectivity values for each value of ε in the experiments, in actual number of matching points. As expected, in each graph the selectivity, shown by the line with the '×', increases as ε increases.

RSJ is depicted only in Figure 8 because for all tested cases it has shown much worse results than the other joins, Figure 8a depicts performance of the joins for 4-dimensional uniform data with cardinality of both sets being $10^5$. Figure 8b shows the performance of the same joins relative to that of RSJ.




(a) (b)

Figure 8: Time to do ε-join for 4D uniform data (with RSJ)

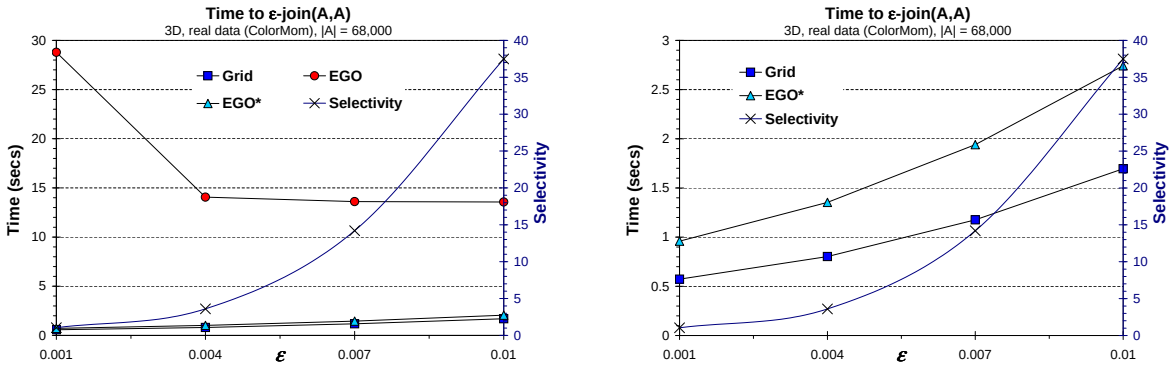
In Figure 8b, the EGO-join shows 3.5–6.5 times better results than those of RSJ, which corroborates the fact that, by itself, EGO-join is a quite competitive scheme for low-dimensional data. But it is not as good as the two new schemes.

Next comes EGO*-join whose performance is *always* better than that of the EGO-join in all experiments. This shows the strength of the EGO*-heuristic. 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 that a sequence has an inactive dimension. In Figure 8b EGO*-join is seen to give 13.5–24 times better performance than RSJ.

Another trend that can be observed from the graphs is that the Grid-join is better that the EGO*-join, except for high-selectivity cases (Figure 10b). EGO-join shows results several times worse than those of the Grid-join, which corroborates the choice of the Grid-index which also was the clear winner in our comparison [7] with main memory optimized versions of R-tree, R*-tree, CR-tree, and quad-tree indexes.

In Figure 8b Grid-join showed 15.5–46 times better performance than RSJ.

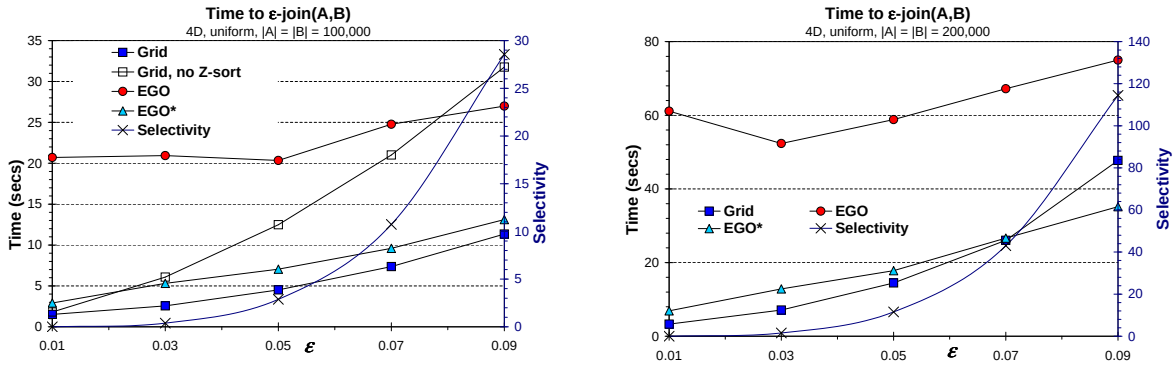
Unlike EGO-join, EGO*-join always shows results at least comparable to those of Grid-join. For all the methods, the difference in relative performance shrinks as ε (and selectivity) increases.




(a) (b)

Figure 9: Time for ε-join for 3 dimensions with real data. (a) With EGO-join (b) Without EGO-join (for clarity)

Figure 9 shows 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 the EGO-join scheme due to its much poorer performance. From these two graphs we can see that the Grid-join is almost 2 times better than the EGO*-join for small values of ε.

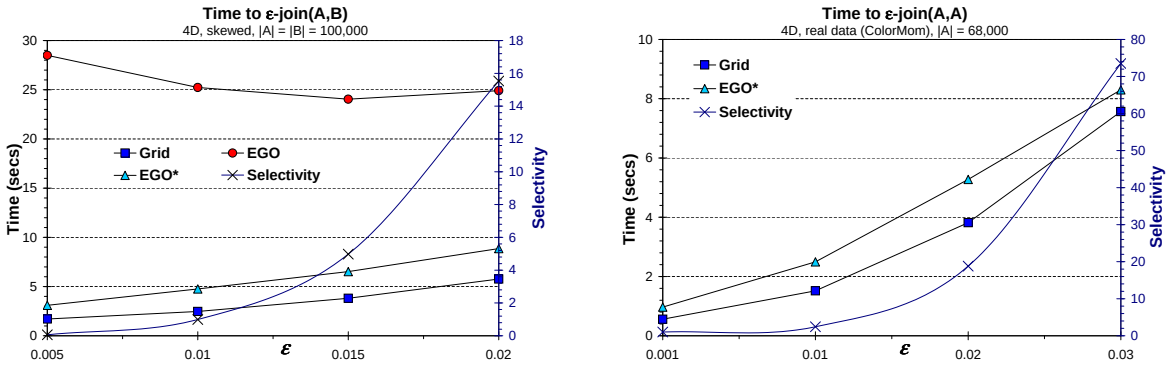



(a) (b)

Figure 10: Time to do ε-join for 4D, uniform data (a) $|A| = |B| = 100,000$ (b) $|A| = |B| = 200,000$

Figure 10 shows 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 10a emphasizes the importance of performing Z-sort on data being joined: the performance improvement is $\sim 2.5$ times. The Grid-join without Z-sort, in general, while being better than EGO-join, shows worse results than that of EGO*-join.

Figure 10b presents another trend. In this figure EGO*-join becomes a better choice than the Grid-join for values of ε greater than $\sim 0.07$. This choice of epsilon corresponds to a high selectivity of $\sim 43$. Therefore EGO*-join can be applied for joining high selectivity cases for low-dimensional data.




(a) (b)

Figure 11: Time to do ε-join for 4D (a) Skewed data (b) Real data

Figures 11 (a) and (b) 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 the EGO-join, EGO*-join, and Grid-join exhibit behavior similar to that in the previous figures with the Grid-join being the best scheme.

### 4.3 High-dimensional data

We now study the performance of the various algorithms for higher dimensions. Figures 12(a) and (b) show the results for 9-dimensional data for uniformly distributed data. Figure 13 (a) presenets the results for 9-dimensional skewed data, Figure 13 gives the results for real 9-dimensional data. Figures 14 (a) and (b) show the results with the 9- and 16-dimensional real data respectively. As with low-dimensional data, for all tested cases, RSJ had the worst results. Therefore, the performance of RSJ is ommitted from most graphs – only one representative case is shown in Figure 12a.

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

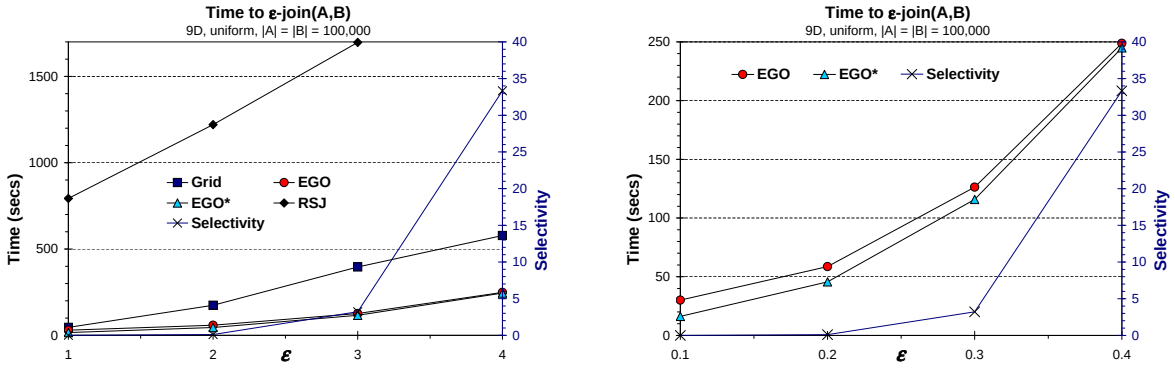



Figure 12: Performance of join for 9D uniform data (a) With RSJ and Grid (b) Only best two schemes

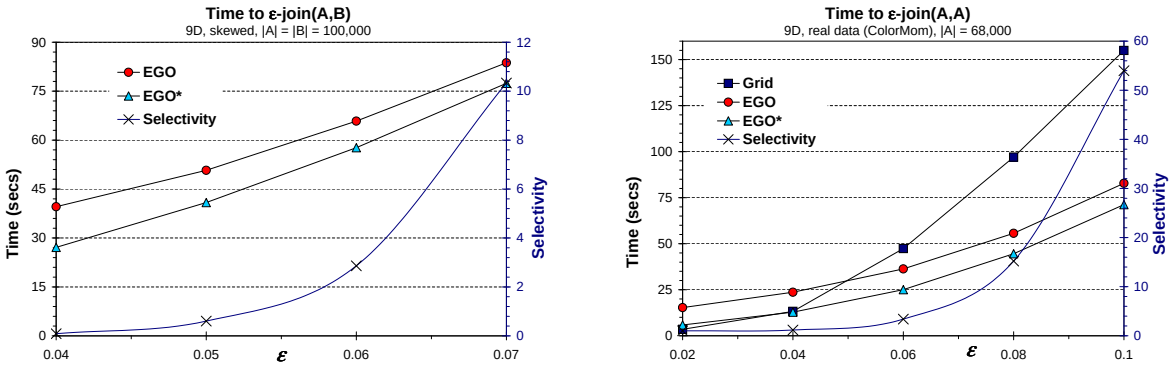


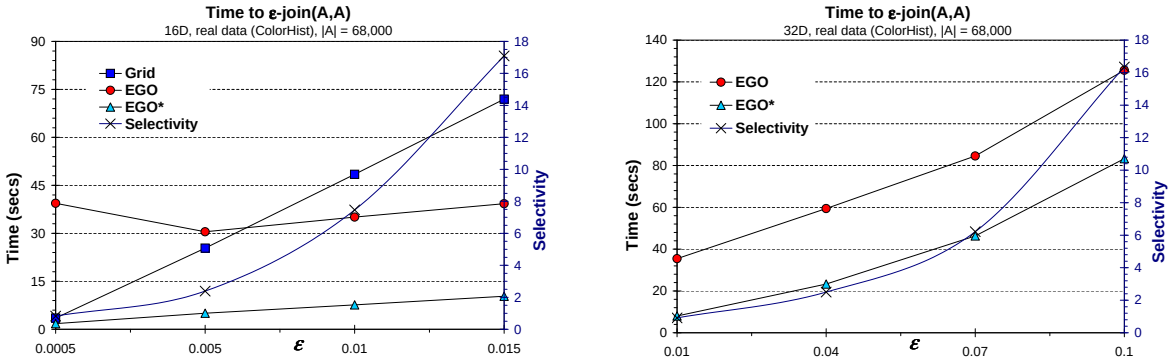

Figure 13: Performance of join for 9D data (a) Skewed data (b) Real data

**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.

# 5 CONCLUSIONS

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 algorihtms 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.




(a) (b)

Figure 14: Performance of join (a) 16D, Real data (b) 32D, Real data

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.

An analytical study has been presented for selecting the grid size. As a side effect of the study the cost-estimating function for the Grid-join has been developed. This function can be used by a query optimizer for selecting the best execution plan.

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

| | **Low ε** | **High ε** |
|---|---|---|
| **Low Dimensionality** | Grid-join | Grid-join/EGO*-join(very large ε's) |
| **High Dimensionality** | EGO*-join/Grid-join(very small ε's) | EGO*-join |

Table 1: Choice of Join Algorithm

# References

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

[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 *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 379–388. ACM Press, 2001.

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

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

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

[6] Y.-W. Huang, N. Jing, and 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, and M. A. Jeusfeld, editors, *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 396–405. Morgan Kaufmann, 1997.

[7] D. V. Kalashnikov, S. Prabhakar, W. Aref, and S. Hambrusch. Efficient evaluation of continuous range queries on moving objects. In *DEXA 2002, Proc. of the 13th International Conference and Workshop on Database and Expert Systems Applications*, Aix en Provence, France, September 2–6 2002.

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

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

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

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

[12] M.-L. Lo and 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*, pages 247–258. ACM Press, 1996.

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

[14] 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. In *IEEE Transactions on Computers, Special Issue on DBMS and Mobile Computing*, 2002. To appear.

[15] J. Rao and K. A. Ross. Making $B^+$-trees cache conscious in main memory. In *Proc. of ACM SIGMOD Conf.*, Dallas, TX, May 2000.

[16] J. C. Shafer and 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*, pages 176–185. Morgan Kaufmann, 1997.

[17] K. Shim, R. Srikant, and R. Agrawal. High-dimensional similarity joins. In *Proceedings of the Thirteenth International Conference on Data Engineering, April 7-11, 1997 Birmingham U.K*, pages 301–311. IEEE Computer Society, 1997.

[18] J. Tayeb, Ö. Ulusoy, and O. Wolfson. A quadtree-based dynamic attribute indexing method. *The Computer Journal*, 41(3), 1998.

## Appendix 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 straight-forward manner.

Table 2: Parameters used for ε-join

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

Table 2 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 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 set $A$ 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 length of the part lists is likely to be longer and each query may take longer to process. In other words, the coarser (i.e. smaller $n$) the grid 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 vs. 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 for $n$ and search only for number of cells in the interval $[1, n_{upper}]$.

For example, for 2-dimensional square data, it can be shown that the upper bound is given by [7]:

$$n = \begin{cases} 4qm & \text{if } q > \frac{1}{2\sqrt{m}}; \\ \frac{1}{\frac{1}{\sqrt{m}} - q} & \text{otherwise.} \end{cases}$$

In this formula $q$ is the size of each square. Since for $\varepsilon$-join we are adding circles, the formulas is reused by approximating the circle by a square with the same area ($\Rightarrow q \approx \varepsilon\sqrt{\pi}$). The corresponding formula for $n$ is therefore:

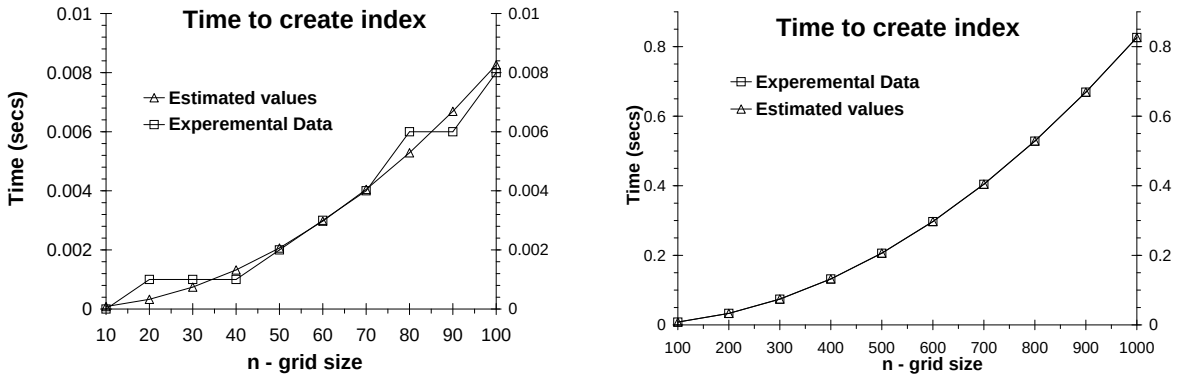
$$n = \begin{cases} \lceil 4\sqrt{\pi}\varepsilon m \rceil & \text{if } \varepsilon > \frac{1}{2\sqrt{\pi m}}; \\ \lceil \frac{1}{\frac{1}{\sqrt{m}} - \varepsilon\sqrt{\pi}} \rceil & \text{otherwise.} \end{cases}$$

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 grid size domain.

However, if the value returned by the formula is too large, the grid might not fit in memory. In that case $n$ can be further limited by memory space availability.

In our experiments the optimal value for grid size tended to be closer to 1 rather than to $n_{upper}$, as in Figure 17.

**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 are determined. 2) Then the nature of dependence on each parameter separately is predicted based on the algorithm and implementation of the grid. Since the Grid is a simple data structure, 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 for each parameter. 4) Finally, an explanation is given on how the calibration of predictor functions can be achieved for a specific machine.




(a) (b)

Figure 15: Time to initialize index (a) $n \in [10, 100]$ (b) $n \in [100, 1000]$

**Estimating *init* Phase:** The time to initialize the index depends only on the grid size $n$. The process of index initialization can be described in $O(1)$ operation followed by the initialization of $n^2$ cells. Thus the index initialization time is expected to be a polynomial of degree two over $n$ such as: $P_{init}(n) = an^2 + bn + c$, for some coefficients $a$, $b$, and $c$. This value 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 for other values of $n$ not used for the calibration. The result is shown in Figure 15 ($a = 8.26 \times 10^{-7}$, $b = 0$, and $c = 0$). The two graphs shown are for different ranges of $n$: on the left $n$ varies from 10 to 100, on the right $n$ varies from 100 to 1000. The graphs show the actual times measured for different values of $n$ as well as the time predicted by the estimator function. As can be seen, the estimator gives very good approximation of the actual initialization times. This is especially true for larger values of $n$.

Figure 15 shows 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.

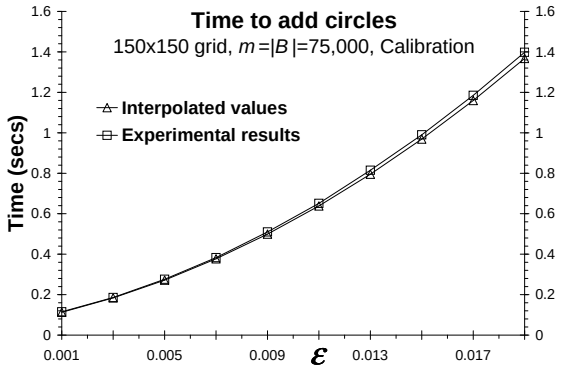



Figure 16: Estimation with polynomial for add phase

**Estimating *add* Phase:** This phase is more complicated then the init phase because it depends on three parameters: $n$ – grid size , $m$ – cardinality of indexed set B, and $\varepsilon$. By analyzing the dependence on each parameter separately, we estimate that the overall function can be represented as a polynomial $P_{add}(n,m,\varepsilon) = a_{17}n^2\varepsilon^2 m + \cdots + a_1 m + a_0$ with degrees of $n$ and $\varepsilon$ no greater than two and degree of $m$ no greater than one. The next step is 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 $\varepsilon$, and two values of $m$ $(3 \times 3 \times 2 = 18)$.

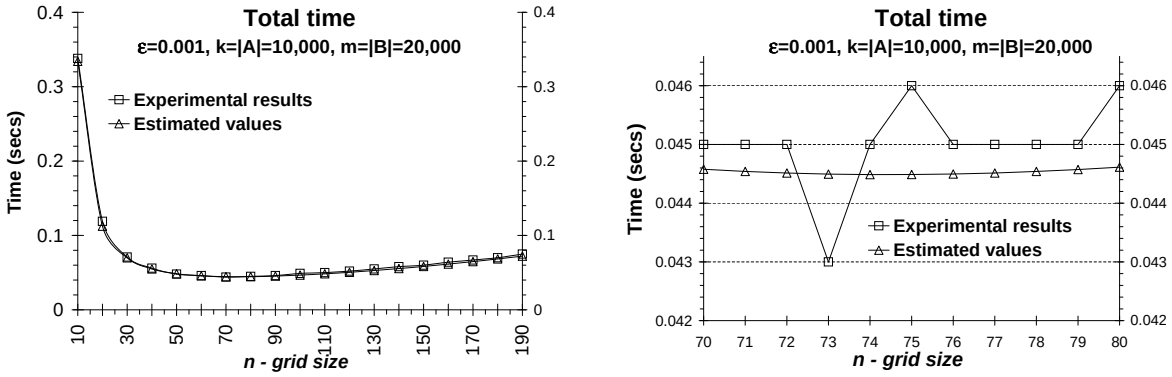
The combinations of the following calibration points have been examined in order to get the coefficients: $n_0 = 10$, $n_1 = 100$, $n_2 = 200$; $\varepsilon_0 = 0.001$, $\varepsilon_1 = 0.01$, $\varepsilon_2 = 0.02$; $m_0 = 50$, and $m_1 = 100$. The choice of values implies we assume that typically $n \in [10, 200]$, $\varepsilon \in [0.001, 0.02]$, and $m \in [50, 100]$. The linear system was solved using Gaussian elimination with pivoting method. Figure 16 demonstrates time needed for add phase for various values of $\varepsilon$ when $n = 150$ and $m = 75$ and another curve is 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 *proc* Phase:** The processing phase depends on all parameters: $n$ – grid size, $k = |A|$, $m = |B|$, and $\varepsilon$. Thankfully, dependence on $k$ is linear since each point is processed independent of other points. Once the solution for some fixed $k_0$ is known, it is easy to compute for an arbitrary $k$. 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 [7], 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): $P_{proc,\sqrt{\pi}\varepsilon \geq c}(c,\varepsilon,m,k_0)$ and $P_{proc,\sqrt{\pi}\varepsilon < c}(c,\varepsilon,m,k_0)$ each of type $P(c,\varepsilon,m,k_0) \equiv a_{17}c^2\varepsilon^2 m + \cdots + a_1 m + a_0$ with degrees of $c$ and $\varepsilon$ no greater than two and degree of $m$ no greater than one. Once again the calibration can be done by solving a system of 18 linear equations for each of the two cases.

**Estimating Total Time:** The estimated total time needed for Grid-join is the sum of estimated time needed for each phase. Figure 17 demonstrates 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.




(a) (b)

Figure 17: Estimation of total time needed for ε-join (a) $n \in [10,190]$ (b) $n \in [70,80]$

A simple *bisection method* for finding the optimal value of *n* was used. This method assumes that it is given a concave downwards function, defined on $[a,b]$. The function 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 bisection method in this context works as follows. The goal is to find the leftmost minimum on the interval $[a,b]$. Compute $c = (a+b)/2$. If $f(c-1) \leq f(c+1)$ then make new *b* be equal *c* and repeat the process, otherwise make new *a* be equal *c* and repeat the process. The process is repeated until $(b-a) < 2$.

The bisection method for the example in Figure 17 gives an estimated optimal value for *n* as 74. Experimentally, we found that the actual optimal value for *n* was 73. The difference between time needed for the grid-join with $73 \times 73$ grid and $74 \times 74$ grid is just two milliseconds for the given settings. These numbers show the high accuracy of the estimator functions. the Notice that the results of interpolation look even better if they are rounded to the closest millisecond values.