The VLDB Journal (2013) 22:561–585
DOI 10.1007/s00778-012-0305-7

REGULAR PAPER

Super-EGO: fast multi-dimensional similarity join

Dmitri V. Kalashnikov

Received: 30 March 2012 / Revised: 20 December 2012 / Accepted: 22 December 2012 / Published online: 8 February 2013

Abstract Efficient processing of high-dimensional similarity joins plays an important role for a wide variety of data-driven applications. In this paper, we consider ε-join variant of the problem. Given two d-dimensional datasets and parameter ε, the task is to find all pairs of points, one from each dataset that are within ε distance from each other. We propose a new ε-join algorithm, called `Super-EGO`, which belongs the EGO family of join algorithms. The new algorithm gains its advantage by using novel data-driven dimensionality re-ordering technique, developing a new EGO-strategy that more aggressively avoids unnecessary computation, as well as by developing a parallel version of the algorithm. We study the newly proposed Super-EGO algorithm on large real and synthetic datasets. The empirical study demonstrates significant advantage of the proposed solution over the existing state of the art techniques.

Keywords Epsilon join · Similarity join · Multi-dimensional join · Euclidean space

1 Introduction

In the ε-join variant of similarity join, the algorithm is given two d-dimensional datasets $A, B \in \mathbb{R}^d$. The goal is to find all pairs of points (a, b), where $a \in A$ and $b \in B$ such that $||a - b|| < \varepsilon$. That is,

$$A \bowtie_\varepsilon B = \{(a, b) : ||a - b|| < \varepsilon, a \in A, b \in B\}. \quad (1)$$

Here $a = (a_1, a_2, \ldots, a_d)$ and $b = (b_1, b_2, \ldots, b_d)$ are d-dimensional points and $||a - b||$ is the distance between a and b, e.g. measured using the generic L^p norm:

$$||a - b||_p = \left[\sum_{i=1}^{d} (a_i - b_i)^p\right]^{\frac{1}{p}}, \text{ where } p = 1, 2, \ldots, \infty.$$

For instance, the focus of many ε-join techniques is often the standard Euclidean distance which corresponds to the L^2 case, where $p = 2$.

Similarity join operations play an important role in such areas as data mining, data cleaning, entity resolution, and so on [8,15,20,25]. Specifically, this join often serves as pre-processing step, also known as *blocking*, in applications that analyze similarity of objects. To find similar objects, such application would first map each object into its d-dimensional feature representation. Then, they would apply a similarity join as a crude-but-fast preprocessing step to find pairs of objects that might *potentially* be similar. The goal of this step is to quickly find a superset of the true set of similar objects.[1] Then, more advanced techniques are typically applied to this superset to remove false positives and get similar objects with higher accuracy.

The main challenge of computing $A \bowtie_\varepsilon B$ is to be able to do it *efficiently* on large datasets. For example, a simple way to compute $A \bowtie_\varepsilon B$, which we will refer to as SimpleJoin, is to implement it as two loops: one over elements of A and the other one over elements of B, inside of which performing a check on if $||a - b|| < \varepsilon$. However, the computational complexity of this simple algorithm is *quadratic* on data size $O(|A| \cdot |B| \cdot d)$. Given that the cardinality of datasets A and B can be large, this algorithm is considered to be infeasible in practice. Hence, more efficient techniques have been proposed to address the challenge [3,4,17,19,21,23,24].

D. V. Kalashnikov (✉)
Department of Computer Science, University of California, Irvine, CA, USA
e-mail: dvk@ics.uci.edu

[1] That is, the result is allowed to contain false positives (pairs of objects that are not similar) but should minimize false negatives (pairs of objects that are similar but not included in the result set).

In this paper, we present Super-EGO algorithm for efficient similarity join. As its name suggests, it belongs to the EGO family of ε-join algorithms [2] which are overviewed in Sect. 3. We show that Super-EGO is more efficient, often by very significant margins, than recent state of the art techniques such as EGO-star [14,16], CSJ [5], LSS [18], and E2-LSH [1]. The speedup is achieved by developing a dimensionality reordering technique, designing an EGO-strategy that more aggressively avoids unnecessary computations, reorganizing the simple-join part of the solution, as well as developing a scalable parallel version of the algorithm, as will be explained in Sect. 4.

We also highlight the importance of considering the *selectivity* of a join operation in assessing the performance of various join algorithms. The selectivity measures the average number of points from dataset B that joins each point from dataset A and it is controlled by ε parameter for given A and B. Given practical uses of epsilon-join (e.g., as a blocking procedure, or for finding pairs of similar objects), selectivity is expected to be within certain limits in practice. However, setting ε appropriately can be unintuitive, especially during testing for higher dimensional cases, as we discuss in Sect. 5.

This paper also contains a fairly extensive experimental evaluation in Sect. 6. It thoroughly tests the proposed approach on eight different real datasets. Furthermore, in addition to the usual types of ε-join experiments, Sect. 6.7 contains a number of empirical results that the reader might find particularly interesting:

- When $\varepsilon \geq 0.5$, Super-EGO, as any EGO-based technique, will reduce to a quadratic algorithm. However, we will see that it reduces to a "smart" quadratic algorithm that often runs much faster than the naive SimpleJoin explained above.
- Section 6.2 defines a simple quadratic baseline called O(n^2)block. Section 6.7 demonstrates that it is surprisingly competitive. We thus encourage researchers working on new ε-join solutions to compare to this baseline to demonstrate that their solutions can outperform this simple quadratic algorithm.
- We show that the join selectivity is often disregarded in various research efforts, which leads to the situations where authors draw conclusions about the performance of their techniques from *pure zero* or *very excessive* selectivity cases. Instead, we strongly suggest that the join selectivity be always presented to the readers so that they themselves can judge the performance of ε-join algorithms at various selectivity levels.

The rest of the paper is organized as follows. We first overview related work in Sect. 2. Next, we summarize the original EGO-join in Sect. 3. The new Super-EGO framework is then covered in Sect. 4. Section 5 discussed issues related to the notion of selectivity. The proposed approach is then empirically evaluated in Sect. 6 and compared to the state of the art techniques. Finally, we conclude in Sect. 7 by highlighting key insights learned from our work and suggesting future research directions.

2 Related work

The ε-join variant of similarity join has high practical significance for data mining, data cleaning, entity resolution, and other applications. Hence, many ε-join techniques have been proposed in the past [3,4,17,19,21,23,24]. We mention a few most-related approaches in more detail below.

State of the art. The database literature considers ε-joins in space $\mathbb{R}^d$, where d is typically somewhere in [2,32]. Often authors target either lower dimensional cases (e.g., spatial joins in 2-3 dimensions [5]) or higher dimensional cases (e.g., E^{2}LSH authors state that the algorithm is only meant for cases of $\simeq$ 10–20 dimensions and above [1]). Often, higher dimensional cases are considered to be more challenging due to the "dimensionality curse" discussed below.

One common ε-join solution is to build an index, such as an R-tree, on both of the datasets and then iteratively check whether MBRs, or their equivalents, are within epsilon distance from each other when performing a join [4]. This approach is known not to perform well compared to the current state of the art techniques, especially for higher dimensional cases due to (a) the need to load the data into the index first and (b) poor performance of indexes such as R-tree in higher dimensional spaces. We note that both EGO-star and Super-EGO operate with a related to MBR concept of a bounding box BB constructed on a sequence of points. The difference is that a BB is not necessarily minimal for a sequence—rather what is more important for EGO-based techniques is to be able to estimate it quickly. In addition, in BB's used by these two EGO-based techniques, last few consecutive dimensions are often unbounded, that is, they range from the minimum to maximum possible values.

A similar approach is to build an index, such as Grid, on circles of radius epsilon centered at the points of one of the datasets and then use points from the other datasets as queries to this index [14]. While this approach works well for lower dimensional cases, techniques such as EGO-join have been shown to outperform this solution for higher dimensional cases [16].

CSJ is a compact similarity join technique [5]. Its main idea is that a join algorithm might sometimes be able to detect subsets of points $A_1 \subseteq A$ and $B_1 \subseteq B$ such that each point $a \in A_1$ joins each points $b \in B_1$. According to the problem definition, ε-join is then supposed to add $|A_1| \times |B_1|$ pairs of (a, b) points to the result set R. CSJ changes the original

problem definition by allowing to simply add (A_1, B_1) to R. This results in the reduction of the physical size of R. CSJ works by employing a two-index solution. It builds indexes, such as an R-tree, on A and B and then checks if the max-distance between two MBRs is less than epsilon—in which case points inside these MBRs are outputted as groups.

LSS [18] is a recent approximate epsilon-join technique that is based on leveraging a GPU (video card) to perform a join by using NVIDIA's CUDA framework. By creating multiple space-filling curves, LSS converts a similarity join operation into the corresponding GPU sort-and-search problem. In addition to performing an ε-join, LSS can also be modified to support an approximate or exact search of the k-NNs in dataset A to all points in dataset B.

E^2LSH (*Exact Euclidean LSH*) [1] is a modification of the LSH algorithm that uses locality-sensitive hash functions to perform an approximate nearest-neighbor search. Though E^2LSH has been developed as an NN technique, its authors also view it as an approximate ε-join method. As a key motivation, the authors have tried to develop algorithm that would have sublinear query time for NN queries to beat the "curse of dimensionality" on very high-dimensional spaces. In [26], authors propose another improvement of LSH. However, E^2LSH has not performed well in our tests, frequently running orders of magnitude slower than competing solutions such as LSS or Super-EGO.

GESS is one of the earlier ε-join techniques developed by Dittrich and Seeger in [10]. It is based on associating with each feature vector x an ε-length hypercube $H(x)$ and then performing an intersection join that can involve splitting and replicating these hypercubes. Even though GESS and EGO-based joins operate with somewhat similar concepts, the two algorithms are, however, sufficiently different. EGO-join does not use the concept of hypercube $H(x)$, instead it keeps track of which virtual cell each x falls in. A single cell can contain multiple points that fall into it. EGO-join forms sequences out of adjacent cells. EGO-join does not partition the original space the way GESS does, instead it rather partitions sequences of points into subsequences. In this process, it never splits or replicates cells. EGO-join then uses geometric properties of two given sequences to check if they can join—it does not check for intersections of $H(x)$'s. EGO has been experimentally shown to be faster than GESS, often by significant margins [18].

Problem variants. Similarity joins have different variants, many unrelated to each other. For instance, in [27], the authors consider an implementation of a *set-based* variant of a similarity join using the map/reduce framework. The set-based and ε-join variants, however, are not related to each other. A set-based join uses a set-based similarity metric, such as the Jaccard similarity or edit distance for strings to compute similarity of sets based on their common members. For instance, such a join can detect that two strings "algorithm" and "algorithmic" are similar. Consequently, [27] addresses a different problem than is studied in this paper.

Parallelization. One of the important techniques we consider in this article is the parallelization of the EGO-join algorithm. Incidentally, [27] also studies parallelization, but of a different kind. The two methods have different motivation. The authors of [27] attempt to scale a set-based join operation to a large map/reduce cluster, where the latter is currently a hot topic of research. First, we deal with ε-join and not set-based join. Second, ε-join is an operation that is useful not only to computer scientists and, most often, it can be successfully performed on commodity hardware such as a regular PC. We therefore seek wide applicability of our algorithm, so that everyone can use it—not only people who have access to large map/reduce clusters. In other words, we are targeting common everyday devices.

Our motivation for a parallel version of the algorithm comes from the observation that modern computers, such as desktops and notebooks, are increasingly multi-core or even multi-processor. Hence, we want to run parallel code on a **single machine**. This can be achieved by employing the classic multi-process/multi-thread programming model, so that each thread can be executed concurrently on each parallel CPU core. However, creating parallel version of specifically EGO-join has certain challenges that are explained in Sect. 4.4.1. Section 4.4.2 explains how to successfully resolve these challenges.

In general, parallelization of regular join (but not ε-join) operations has been studied extensively in the past, for example, [22] overviews some of these techniques. Such methods would often consider issues unrelated to EGO join and its setup, such as how to partition data across machines and/or multiple disks. The work on parallelization of ε-joins is rather scarce, and we are unaware of any existing technique that deals with parallelizing specifically EGO-join.

The curse of dimensionality. The curse of dimensionality is a notion that does not have an exact definition, but which in general refers to the dramatic drop of the efficiency that different querying algorithms face when the dimensionality d of space $\mathbb{R}^d$ increases. For example, for NN queries, [1] refers to the "curse of dimensionality" to mean that the fastest way to process a given NN query becomes a naive linear-cost $O(n)$ algorithm that compares the query point to each point in the database. Hence, [1] attempts to design an approximate NN algorithm with a sublinear query cost. For ε-join $A \bowtie_\varepsilon B$, a similar definition would be that the fastest way to process the join becomes a quadratic algorithm $O(n^2)$ that compares each point $a \in A$ to each point $b \in B$. In Sect. 6.7, we will see that several state of the art techniques we test actually might not be able to overcome the curse of the dimensionality.

Miscellaneous. There have been very significant amount of research efforts on various spatial and spatio-temporal database issues and multi-dimensional data processing that are also related. Currently, we can observe a key methodological difference between spatio-temporal work and the best performing ε-join techniques. The former is often making use of creative advanced indexing (e.g., R-tree-based indexes) for lower dimensional case, for example, 2D, 3D. The modern trend for ε-join work is to look at the (more challenging) case where the data dimensionality d is high and where many standard indexing techniques stop working well. Hence, the best performing modern ε-join techniques are often not based on building indexes on data [14,18]. An example of a related work from the spatial domain is [9]. It defines the K-CPQ queries whose goal is to find K closest to each other pairs in the database, under the assumptions that R-tree indexes are maintained on data. We can notice that by dynamically increasing K and applying distance filtering, it should be possible to answer ε-join queries using K-CPQs, and vice versa. However, it should be noted that K-CPQs are often tuned and/or tested to retrieve just a few pairs, such as $K \leq 100$, whereas the number of pairs in a typical result of an ε-join is significantly higher. A similar work is [12] that considers using hierarchical indexing techniques (e.g. R-tree) to process the distance join, whose goal is to find all pairs of points that satisfy the predicate on the distance between these points. The distance join can be viewed as a generalization of the ε-join. In [7], the authors present a nice generalization of top-k pairs queries. The solution is not indexing-based and shown to outperform many existing techniques. The generalization allows the user to define (loose monotonic) local scoring functions for each attribute involved and a (monotonic) global function to combine these local values.

Our previous work. This paper builds on our previous work [14,16]. Section 3 presents a summary of that work, whereas all the other content is new. While [14,16] considered selectivity, they were based on simplified models. For instance, the models could not predict when selectivity would drop to zero and could not explain why ε can become larger than 1 for higher dimensionality cases.

3 Overview of the original EGO-join

`Super-EGO` framework is based on the `EGO-star` algorithm [14,16] which in turn is an improved version of `EGO-join` algorithm introduced by Böhm et al. in [2]. Both of the algorithm work with L^p norm where $||a - b||_p = \left[\sum_{i=1}^{d}(a_i - b_i)^p\right]^{\frac{1}{p}}$, where $p = 1, 2, \ldots, \infty$, though the cases where $p = 1$ and $p = \infty$ are special cases which should be considered separately. For simplicity, in the following discussion, we will assume the Euclidean space with L^2 norm, though the methods apply to L^p.

```
//   ε: a global variable visible everywhere
EGO-JOIN(A, B)
 1   LOAD(A); LOAD(B);              //1-Load phase
 2   EGO-SORT(A); EGO-SORT(B)       //2-Sort phase
 3   R ← JOIN(A, B, d_str ← 1)      //3-Join phase
 4   SAVE(R)                        //4-Save phase (optional)
 5   return R
```

Fig. 1 `EGO-join`. Original `EGO-join` does not use d_{str}

```
JOIN(A, B, d_str)
 1   if EGO-STRATEGY(A, B, d_str) = success then
 2      return ∅

 3   if |A| < t and |B| < t then // – Case 1 –
 4      return SIMPLEJOIN(A, B) // t - predefined threshold

 5   if |A| < t and |B| ≥ t then // – Case 2 –
 6      {B1, B2} ← SPLIT(B)
 7      return JOIN(A, B1, d_str) ∪ JOIN(A, B2, d_str)

 8   if |A| ≥ t and |B| < t then // – Case 3 –
 9      {A1, A2} ← SPLIT(A)
10      return JOIN(A1, B, d_str) ∪ JOIN(A2, B, d_str)

11   if |A| ≥ t and |B| ≥ t then // – Case 4 –
12      {A1, A2} ← SPLIT(A)
13      {B1, B2} ← SPLIT(B)
14      return JOIN(A1, B1, d_str) ∪ JOIN(A1, B2, d_str)∪
                ∪JOIN(A2, B1, d_str) ∪ JOIN(A2, B2, d_str)
```

Fig. 2 Recursive join procedure

Let us assume that the domain $\Omega \subseteq \mathbb{R}^d$ is normalized to d-dimensional cube $[0, 1]^d$. In EGO-based algorithms, a virtual grid G is overlaid on top of Ω. This grid is imaginary and never materialized. G is a regular grid with the cell side size of ε. It quantizes the domain Ω into regular-size cells, such that the mapping of each point into its corresponding grid coordinates can be done efficiently in O(d) time. Namely, for point $a = (a_1, a_2, \ldots, a_d)$, its grid coordinates are $c_a = (\lfloor a_1/\varepsilon \rfloor, \lfloor a_2/\varepsilon \rfloor, \ldots, \lfloor a_d/\varepsilon \rfloor)$.

To join two d-dimensional datasets A and B, EGO-based algorithms would first "EGO-sort" points in A and B, see Fig. 1. EGO-sort is a very simple procedure. It is just a regular sorting of points, except for it uses each point's d-dimensional *cell coordinates*, in lexicographical order, as the sorting key. For example, for a 3D case, point with cell coordinates (1, 2, 3) would come before points (1, 2, 4) and (2, 1, 1), but after point (1, 1, 4).

Then, the algorithm would call a recursive `EGO-join` procedure $Join(A, B)$ on A and B. `EGO-join` is a *divide and conquer* type of an algorithm which splits A and B into parts as the algorithm proceeds forward, see Fig. 2. This procedure would first apply EGO-strategy (A, B),

which returns a binary `success` or `fail` answer. Its main purpose is, for certain cases of A and B, to be able to efficiently determine that no point in A will join a point in B, in which case EGO-strategy (A, B) returns `success`. This check is done *quickly* without scanning all points in A and B. Typically it is done by analyzing only the first and last points in A and B and by leveraging the fact that A and B are EGO-sorted. For example, EGO-strategy (A, B) of `EGO-star` computes spatial bounding boxes BB_A and BB_B for points in A and B, respectively, and then checks whether there is a separation of ε between them. In general, EGO-strategy is a key component of EGO-based approaches, which implement it differently which greatly affect their efficiency.

If EGO-strategy (A, B) returns `success`, the algorithm returns empty set $R = \emptyset$ as the result of $Join(A, B)$. If EGO-strategy (A, B) returns `fail`, then there could be a point in A that joins a point in B. The algorithm then proceeds to "divide and conquer" recursively, based on four possible cases. Let t be a predefined *threshold* used to specify the bottom of recursion: the algorithm will not split sequences of length smaller than t into subsequences. Then,

Case 1: $|A| < t$ and $|B| < t$. The algorithm then checks if $|A|$ and $|B|$ are already small enough (smaller than t) and if so it applies the simple-join algorithm $R = \text{SimpleJoin}(A, B)$, described in the introduction, to compute the result by comparing each point in A to each point in B. We will explain SimpleJoin(A, B) later on in more detail.

Case 2: $|A| < t$ and $|B| \geq t$. In this case, the algorithm splits B in the middle into two equal parts B_1 and B_2 and computes the result by calling join recursively as $R = Join(A, B_1) \cup Join(A, B_2)$.

Case 3: $|A| \geq t$ and $|B| < t$. Similarly, the algorithm splits A in the middle into A_1 and A_2 and computes $R = Join(A_1, B) \cup Join(A_2, B)$.

Case 4: $|A| \geq t$ and $|B| \geq t$. Then, the algorithm splits both A and B and computes $R = Join(A_1, B_1) \cup Join(A_1, B_2) \cup Join(A_2, B_1) \cup Join(A_2, B_2)$.

4 Super-EGO framework

In this section, we present the proposed `Super-EGO` approach. We start by introducing a novel phase for EGO-based algorithms that reorders dimensions of data in Sect. 4.1. We then explain the new EGO-strategy used by `Super-EGO` in Sect. 4.2. In Sect. 4.3, we cover a smart SimpleJoin strategy that employs sampling techniques to decide the ranges of dimensions to scan. Section 4.4 then presents our solution for the parallelization of the algorithm. The space complexity of the overall approach is analyzed in Sect. 4.5. Finally, Sect. 4.6 briefly outlines miscellaneous issues related to the presented solution, including a potential extension to the dimensionality reordering algorithm, an optimization for the self-join case, and a disk-based version of the approach.

4.1 Data-driven dimensionality reordering

4.1.1 Basic technique

The default `EGO-join` algorithm analyzes dimensions in a sequential order from 1 to d. However, for higher dimensional cases, some of the dimensions might have more discriminative power than the others. Thus, there could be merit in reordering the dimensions based on their discriminative power.

The discriminative power for `EGO-join` $A \bowtie_\varepsilon B$ can be measured by applying data sampling techniques to datasets A and B. Assume that A and B are normalized to unit cube $[0, 1]^d$. Then, for dataset A, for each dimension i, we construct a histogram H_i^A with $\lceil 1/\varepsilon \rceil$ bins of size ε. The bins directly correspond to cells of the virtual grid G used by `EGO-join`. We will refer to the jth bin of H_i^A as $H_i^A[j]$.

We then sample m points from A. For each sampled point $a \in A$, we increase the count $H_i^A[j]$ by 1 if the value of a falls into jth bin in its ith dimensions. At the end of sampling, the counts in each bin are normalized by dividing them by m. Then, the procedure is repeated for dataset B and its histogram H_i^B is constructed.

The two histograms H_i^A and H_i^B are then used to compute the *fail factor* f_i for ith dimension for the given ε. This factor estimates the fraction of all (a, b) pairs of points on which EGO-strategy will fail, for the given ε, if it is allowed to analyze only the ith dimension. Specifically, EGO-strategy will fail on (a, b) in the ith dimension only if a and b are in the same or directly neighboring cells in these dimensions. Consequently, f_i for bin j is computed as

$$f_i[j] = H_i^A[j] \cdot (H_i^B[j-1] + H_i^B[j] + H_i^B[j+1]),$$

except for the marginal cases where $i, j = 0$ and $i, j = max$, which are computed accordingly. The overall f_i is then computed as $f_i = \sum_j f_i[j]$.

After computing the fail factor f_i for each dimension i for the given ε, we can compute the success factor $s_i = 1 - f_i$ for this ε. It corresponds to the fraction of pairs on which EGO-strategy will succeed if allowed to analyze only the ith dimension. We then re-order the dimension of A and B in the ascending order of their s_i so that the dimensions with the most discriminatory power will appear first. The process of reordering consists of constructing the map of re-ordering (e.g., it will tell that, say, dimension 5 should become dimension 1, and so on) and then changing each point in A and B according to this map.

Notice that after applying the re-ordering phase, the new join algorithm will work on a set of *different* points compared

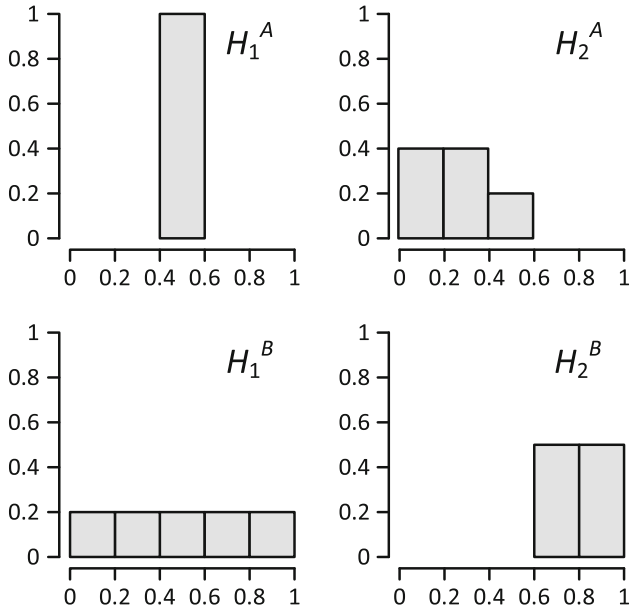

Fig. 3 Example of histograms for 2-dimensional case

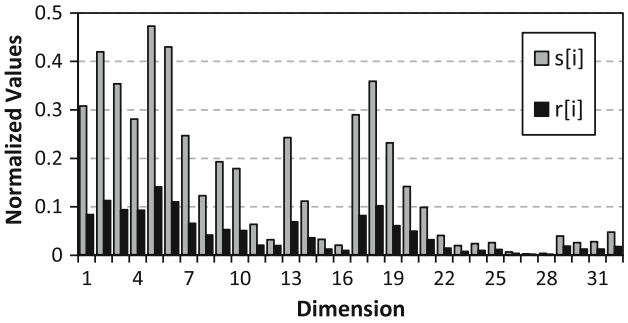

Fig. 4 Example of s_i, r_i values on a 32-dimensional dataset

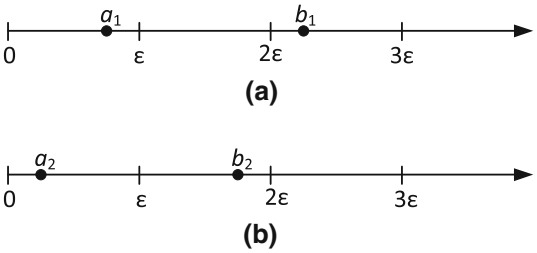

Fig. 5 Example for s_i and r_i

to the old algorithm. Even though the points are different, they are equivalent in terms of computing the distance between them, that is, $||a-b|| = ||a_{new} - b_{new}||$. However, the new algorithm will discover and then process different EGO-sequences (subsequences of A and B) from those of `EGO-join` and `EGO-star` join.

Figure 3 demonstrate an example of histograms for 2-dimensional case, where $\varepsilon = 0.2$ and thus all histograms have 5 buckets. For dimension d_1, H_1^A reveals that A's values in d_1 are located toward the middle of [0, 1], whereas according to H_1^B, the values of B in d_1 are distributed uniformly in [0, 1]. For this simple example, it is easy to see that for d_1, the fail factor is going to be $f_1 = 1 \cdot (0.2+0.2+0.2) = 0.6$, and hence, the success factor is $s_1 = 1 - f_1 = 0.4$. Similarly, for dimension d_2 histograms H_2^A and H_2^B indicate that the values of A are distributed mostly in the first three buckets in d_2, whereas the values of B are mostly in the last two. The fail factor for d_2 is $f_2 = 0.2 \cdot 0.5 = 0.1$ and the success factor is $s_2 = 0.9$. Since $s_2 > s_1$, dimensions d_1 and d_2 will be reordered.

4.1.2 Average-distance histogram

In practice, the success factor s_i for ith dimension will be strongly correlated with the average distance r_i between points in A and B in the ith dimension. See, for example, Fig. 4 which demonstrates s_i and r_i values for a 32-dimensional dataset. Like s_i, the value of r_i can also be computed by sampling points from A and B, and then, the dimensions can be re-ordered based on r_i. The value of s_i, however, provides a more direct measure into the discriminatory power than r_i, since it reflects how the points are *placed* inside cells that are used by `EGO-join`.

For example, consider points a_1, b_1, a_2, and b_2 in Fig. 5a, b. In both cases, the distance between points is the same $||a_1 - b_1|| = ||a_2 - b_2|| = 1.5\varepsilon$. However, points a_1 and b_1 in Fig. 5a will be separated by the EGO-strategy since there is a cell $[\varepsilon, 2\varepsilon]$ of size ε separating them. However, points a_2 and b_2 in Fig. 5b will not be separated by the EGO-strategy since they are in the neighboring cells $[0, \varepsilon]$ and $[\varepsilon, 2\varepsilon]$. So if we assume that sample of A consist of only $\{a_1\}$ for (a) and $\{a_2\}$ for (b), and sample of B of only $\{b_1\}$ for (a) and $\{b_2\}$ for (b), then for figure (a), $s_i = 1$, whereas for (b) $s_i = 0$.

In practice, the map for re-ordering dimensions based on s_i is often the same as that for r_i. While the cases where the two maps are different do exist, the efficiency results on the two different maps in such cases tend to be very similar. The algorithm leverages this observation for the cases where ε is very small, that is, when $\varepsilon \leq t_\varepsilon$ for a predefined threshold t_ε. When ε is small, the number of bins $\lceil 1/\varepsilon \rceil$ in histograms H^A and H^B is large and the sample size m has to be large as well to compute reliable statistics. Instead, when $\varepsilon \leq t_\varepsilon$, the algorithm reorders dimensions based on r_i instead of s_i. An additional positive effect of that is that the space complexity of extra space needed to performs dimension reordering becomes $O(d)$. This is since the size of histograms H^A and H^B is $O(\lceil 1/\varepsilon \rceil \cdot d)$, but restricting ε by a constant t_ε makes it $O(d)$. The size of the dimension reordering map is also $O(d)$.

4.1.3 Efficiency of reordering

The entire re-ordering phase has the linear computational cost of $O((|A| + |B|) \cdot d)$ and thus very efficient. It is also very efficient in terms of the space complexity, as it only requires $O(d)$ space to store two histograms and a re-ordering map. We will see in Fig. 28 in Sect. 6 that the actual execution time of this entire phase is negligible compared to the end-to-end running time of the overall algorithm. In theory, one might want to avoid reordering dimensions, for example, for the cases like uniform data. Specifically, since sampling is used, values of s_i will be slightly different for $i = 1, 2, \ldots, d$ even for uniform data, resulting in a map that might (unnecessarily) change the order of dimensions. In practice, however, avoiding re-ordering will not affect the end-to-end running time of the algorithm by any noticeable margin, see Fig. 28. But if for some reason this is still necessary, then it could be easily achieved by using standard statistical techniques, such as t-test.[2]

4.2 New EGO-strategy

At the core of `EGO-join` is its EGO-strategy, whose effectiveness determines the efficiency of the overall approach. Its task is to be able to quickly tell, for certain sequences A and B, that they will not join, without scanning A and B. Let c_1 and c_2 be the cell coordinates of the first and last points of A, respectively. Let c_a be the coordinates of any point $a \in A$. Because points in A are EGO-sorted, we know that c_1, c_2, c_a will have the form:

$$\begin{aligned} c_1 &= (v_1, v_2, \ldots, v_{i-1}, v'_i, *, *, \ldots, *) \\ c_a &= (v_1, v_2, \ldots, v_{i-1}, v_i, *, *, \ldots, *) \\ c_2 &= (v_1, v_2, \ldots, v_{i-1}, v''_i, *, *, \ldots, *). \end{aligned} \tag{2}$$

That is, they will share the same values $v_1, v_2, \ldots, v_{i-1}$ in the first zero or more dimensions, which we will call *inactive*. Then, if $i - 1 < d$, there will be ith dimension, which we will call *active*, such that $v'_i < v''_i$ and $v'_i \le v_i \le v''_i$. The values in the remaining dimensions can be anything, so they are denoted as a wildcard '$*$'.

For example, consider sequence of points A whose cells coordinates are $(5, 2, 3)$, $(5, 2, 4)$, $(5, 1, 9)$, and $(5, 2, 0)$. If we EGO-sort them, they will be in the order $(5,1,9)$, $(5, 2, 0)$, $(5, 2, 3)$, $(5, 2, 4)$. Then, $c_1 = (5, 1, 9)$ and $c_2 = (5, 2, 4)$. By observing that $c_1[1] = c_2[1] = 5$, we know that dimension d_1 is inactive and that all points in A (i.e., their cell coordinates) have the same value of 5 in their first dimension. Since dimension d_2 is the first where $c_1[2] < c_2[2]$, it is the active dimension. Hence, we know that all points in A have values from 1 to 2 in their dimension d_2.

We can see that points in A are bounded by a bounding box $BB_A = [v_1, v_1] \times \cdots \times [v_{i-1}, v_{i-1}] \times [v'_i, v''_i] \times [0, M] \times \cdots \times [0, M]$, where M is the maximum possible cell number. For example, for the above sequence A, the bounding box is going to be $BB_A = [5, 5] \times [1, 2] \times [0, M]$. Similarly, points in B will be bounded by another bounding box $BB_B = [w_1, w_1] \times \cdots \times [w_{j-1}, w_{j-1}] \times [w'_j, w''_j] \times [0, M] \times \cdots \times [0, M]$, where the active dimension j for B does not have to be equal to the active dimension i for A. Now, if we can find a dimension k where intervals $BB_A[k]$, $BB_B[k]$ of BB_A, BB_B in kth dimension are separated by the distance of at least 1 cell, this will imply no point in A will join a point in B since the distance between such points will be at least ε. This is since the length of a cell side is exactly ε.

An EGO-strategy can be designed from the above observation and by noting that BB_A and BB_B can be constructed quickly, just by observing cell coordinates of the last and first points of A and B. But unlike [14,16], the new strategy will use the notion of BB only conceptually, without literally constructing and manipulating them. Furthermore, it now uses the new notion of *starting dimension* d_{str}.

Figure 6 shows the new EGO-strategy. It incrementally iterates over dimensions trying to find one where intervals $[lo_A, hi_A]$ for A and $[lo_B, hi_B]$ for B are separated by at least 1. If it finds such a dimension, it immediately returns that A and B will not join, without constructing full bounding boxes for A and B. Otherwise, it also checks whether the current dimension i is the active dimension for A or B. If it

```
EGO-STRATEGY(A, B, d_str)
 1  a_fst ← A[1]; a_lst ← A[|A|]
 2  b_fst ← B[1]; b_lst ← B[|B|]

 3  for i ← d_str to d do
 4     lo_A ← ⌊a_fst[i]/ε⌋
 5     hi_B ← ⌊b_lst[i]/ε⌋
 6     if lo_A > hi_B + 1 then
 7        return success //Successful pruning

 8     lo_B ← ⌊b_fst[i]/ε⌋
 9     hi_A ← ⌊a_lst[i]/ε⌋
10     if lo_B > hi_A + 1 then
11        return success //Successful pruning

12     if (lo_A < hi_A) or (lo_B < hi_B) then
13        //All remaining dimensions intersect
14        d_str ← i
15        return fail

16  return fail
```

Fig. 6 EGO-Strategy

[2] For instance, instead of computing s_i once, the procedure could be repeated k times, and then, s_i can be computed as average of the k observed samples of s_i's. Sorting procedures (used for dimension re-ordering), such as qsort, are defined in terms of "$<$" operation. We thus can define that $s_i < s_j$ holds for `qsort` only when both conditions hold: (1) for the averages, it holds $s_i < s_j$ and (2) the difference between s_i and s_j is statistically significant according to the t-test.

```
SimpleJoin(A, B)
1   R ← ∅ // Result set
2   for each a ∈ A do
3      for each b ∈ B do
4         s ← 0
5         for i ← 1 to d do
6            s ← s + (a_i − b_i)^2
7            if s ≥ c then // where c = ε^2
8               goto next_ab
9         R ← R ∪ {(a, b)}
10        next_ab:
11  return R
```

Fig. 7 Regular SimpleJoin procedure

```
SimpleJoin(A, B)
1   R ← ∅ // Result set
2   for each a ∈ A do
3      for each b ∈ B do
4         s ← 0
5         for i ← ⌊d/2⌋ to d do
6            s ← s + (a_i − b_i)^2
7            if s ≥ c then // where c = ε^2
8               goto next_ab
9         for i ← 1 to ⌊d/2⌋ − 1 do
10           s ← s + (a_i − b_i)^2
11           if s ≥ c then //
12              goto next_ab
13        R ← R ∪ {(a, b)}
14        next_ab:
15  return R
```

Fig. 8 An alternative SimpleJoin procedure

is, it means subsequent intervals for A (or B) could only be $[0, M]$ and they will intersect with all the remaining intervals of B (or A); hence, the strategy will not be able to prune away A and B. Furthermore, it sets starting dimension d_{str} to the current dimension i.

The purpose of setting d_{str} is that since EGO-strategy fails at that point, the algorithm will proceed by possibly splitting A and/or B into halves and applying the join procedure recursively. Let us say A is split into A_1 and A_2. We can see that if A has k inactive dimensions, then A_1 and A_2 will also have at least k inactive dimensions. Furthermore, $[lo_A, hi_A] = [lo_{A_1}, hi_{A_1}] = [lo_{A_2}, hi_{A_2}]$ for the first k dimensions. Since the algorithm has already checked that there is no distance of 1 among intervals $[lo_A, hi_A]$ and $[lo_B, hi_B]$ in these k inactive dimensions, there will not be distance of 1 in these k dimensions for A_1 and A_2 as well. Hence, there is no need to recheck the intervals in these k dimensions and the algorithm can start the checks from dimension d_{str}, saving on unnecessary computations.

4.3 New simple-join procedure

Basic intuition. As we have discussed in Sect. 3, the algorithm invokes SimpleJoin(A, B) procedure in the case the cardinality of A and B is less than the predefined threshold. SimpleJoin, illustrated in Fig. 7, iterates over each pair of points (a, b) from A and B. For each (a, b), it then iterates over dimensions from 1 to d while checking whether the (squared) partial distance s between a and b already exceeds $c = \varepsilon^2$, and, if so, it quits checking (a, b) pair early and moves on to the next pair of points.

However, SimpleJoin(A, B) can also be optimized. Recall that A and B are EGO-sorted. Also, since SimpleJoin is invoked, this means EGO-strategy failed on A and B. From these observations, it follows that A and B are too close to each other in the first few (inactive and active) dimensions. This in turn implies that when checking whether $||a-b|| < \varepsilon$ for some pair (a, b), checking it in a certain order of dimensions could speed up the algorithms e.g., as shown in Fig. 8.

New SimpleJoin procedure. To make this intuition achieve consistent improvements, we will use another data-driven strategy. From Sect. 4.1 we know that for each dimension $i = 1, 2, \ldots, d$ we can use sampling (before the join starts) to estimate the average distance r_i between points in A and B in that dimension. We also know that applying dimensionality reordering will likely result in the situation where $r_1 \geq r_2 \geq \cdots \geq r_d$, so we will assume it holds for clarity of further discussion.

Notice, when SimpleJoin is invoked for small subsequences A' and B' of A and B, they will have their own average distances $r'_1, r'_2, \ldots, r'_d$ in the corresponding dimensions where $r'_1 \geq r'_2 \geq \cdots \geq r'_d$ does not necessarily hold. Further, since $|A'|$ and $|B'|$ are already very small, it is too costly to compute $r'_1, r'_2, \ldots, r'_d$ via sampling. Nevertheless, $r'_1, r'_2, \ldots, r'_d$ could be quickly estimated from $r_1, r_2, \ldots, r_d$ and some other parameters, as explained next.

We know that points in A' and B' will have the same or neighboring cell coordinates in the first d_{str} dimensions. If points in A' and B' have the same cell-coordinate in dimension d_i, then (under the local uniformity assumption)[3] the average distance among them in dimension d_i can be estimated as $\varepsilon/3$, see;[4] for two neighboring cells—it is $2\varepsilon/3$. Thus, on average, the distance among them can be estimated as $r'_i = (\varepsilon/3 + 2\varepsilon/3)/2 = \varepsilon/2$. If, however, $r_i < \varepsilon/2$, a better estimator of the average distance r'_i for $i = 1, 2, \ldots, d_{str}$

[3] Notice, the uniformity assumption is **not** very restrictive here, especially when $\varepsilon \ll 1$. This is since while data is not uniform in general, it is often "locally uniform"—meaning it could be approximated as uniform inside small portions of space. A cell would be a good example of a small portion of space, making data in it locally uniform.

[4] This comes from the well-known fact that the average distance between two randomly placed points in [0,1] is $\frac{1}{3}$. Observe that the average distance from a given point $x \in [0, 1]$ to all points in $[0, 1]$ can be computed as a Riemann Integral $\int_0^1 |x - y| dy = x^2 - x - \frac{1}{2}$. Thus, the average for all points is $\int_0^1 (x^2 - x - \frac{1}{2}) dx = \frac{1}{3}$.

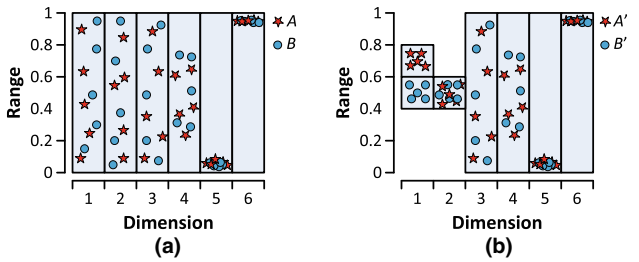

Fig. 9 Example for new SIMPLEJOIN procedure

is $r'_i = \min(r_i, \varepsilon/2)$. For dimensions $i > d_{str}$, we can use the original estimation of the average distance: $r'_i = r_i$.[5]

Hence, the new SIMPLEJOIN procedure scans dimensions over 1, 2, or 3 ranges, depending on the newly computed values r'_i for $i = 1, 2, \ldots, d$. Let $m = d_{str}$. Recall that $\min(r_1, \varepsilon/2) = r'_1 \geq r'_2 \geq \cdots \geq r'_m = \min(r_m, \varepsilon/2)$. Let k be the first dimension such that $r'_k < r'_m$, or let $k = d + 1$ if there is no such dimension. Values of m and k form three natural scanning ranges: $R_1 = [1, m-1]$, $R_2 = [m, k-1]$, $R_3 = [k, d]$. Here, R_1 is an empty range if $m = 1$, and R_3 is empty when $k = d+1$. By design, range R_3, if exists, always contains the smallest values of r'_i, so it is always scanned last. Range R_2 is scanned before R_1 only when $r'_m > r'_{m-1}$. Hence, only three situations are possible:

1. *1-Range.* The algorithm will scan $[1, d]$, which corresponds to scanning R_1, R_2, R_3.
2. *2-Ranges.* The algorithm will scan $[m, d]$ and $[1, m-1]$ for $m > 1$, which corresponds to R_2, R_1 and $R_3 = \texttt{null}$.
3. *3-Ranges.* The algorithm will scan $[m, k-1]$, $[1, m-1]$, $[k, d]$ for $m > 1$, which corresponds to R_2, R_1, R_3.

For efficiency, the algorithm precomputes these ranges right after the dimensionality re-ordering but before the join itself starts. Namely, it creates a map that maps each possible $m = 1, 2, \ldots, d$ into the corresponding scanning ranges for that m. Hence, these scanning ranges are computed only once per m and not re-computed inside SIMPLEJOIN or even JOIN.

Figure 9a demonstrates an example where initially points in A (plotted as stars) and B (plotted as circles) are distributed uniformly in [0, 1] in dimensions d_1, d_2, and d_3, uniformly in [0.2, 0.8] for d_4, uniformly in [0, 0.1] for d_5, and uniformly in [0.95, 1] for d_6. Figure 9b illustrates how the points can be distributed when SIMPLEJOIN is called for two small subsequences A' and B' of A and B, where $\varepsilon = 0.2$ and $m = 3$. Points in A' fall into the cell that corresponds to [0.6, 0.8] in d_1 and to the cell that correspond to [0.4, 0.6] in d_2. For points in B', it is [0.4, 0.6] for both d_1 and d_2. Since the cells in d_1 and d_2 for A' and B' are adjacent, the EGO-strategy failed on them prior to invoking the SIMPLEJOIN. It is easy to see that in this case $R_1 = [1, 2]$, $R_2 = [2, 4]$, $R_3 = [5, 6]$ and the algorithm with scan the three intervals in the order of R_2, R_1, R_3.

[5] While these estimations could be improved by recomputing average distances r_i that are specific to subsequences of A and B right in $Join(A, B)$ procedure (to account for possible correlation in data), experiments with such techniques have not lead to any further improvement in practice.

4.4 Algorithm parallelization

4.4.1 Challenges

Assume that we want to run some algorithm in parallel on a single machine that has n CPU cores. Then, a naive way to do that would be to try to split the task into exactly n "jobs" and run each job independently. This approach, however, rarely succeeds in practice, as it is hard to perform this split perfectly into equal-size jobs. Due to various factors, including interactions with the OS, this approach often ends up in the situation where all jobs run for vastly different length of time (and, frequently, one job running much longer than the others) leading to suboptimal performance.

This is one of the reasons of why often producer–consumer-like models are used for parallelization, where producers produce a large number of *smaller* jobs and put them into the job queue. These jobs are then extracted from the queue and processed by consumer threads, allowing them to share the load more equally and finish almost at the same time.

When it comes to creating a parallel version of specifically the `EGO-join`, we are faced with two main challenges if we want to use a produces-consumer-like model.

The first challenge of parallelizing `EGO-join` comes from the fact that there is no direct readily available "unit" of work in `EGO-join` that can serve as a "job" in a classic producer–consumer model. Instead, there are several indirect ways to define a job. Hence, we need to judiciously select one that would lead to good performance. The challenge in defining jobs is to be able to do so such that the overall processing is *load-balanced* across independent processing units (e.g., CPU cores) and be able to prevent *starvation*—a situation where a thread assigned to a CPU core periodically needs to "wait" for some time to get a job, instead of performing useful work.

Second, `EGO-join` algorithm consists of performing a large number of repetitions of a **very lightweight** processing code, see Figs. 2 and 6. Therefore, if not careful, inserting in the middle of the `EGO-join` code any extra bookkeeping procedure, or costly OS synchronization calls[6] to access semaphores/mutexes can have a significant negative impact on the performance of the overall algorithm.

[6] For example, in our testing, statement $lock(S); k = k + i; unlock(S)$ is over 10 times slower than just $k = k + i$.

```
PARALLEL-EGO-JOIN(A, B, N_thr)
1   EGO-SORT(A)
2   EGO-SORT(B)
3   d_str ← 1
4   Q ← (A, B, d_str) // Q is a global variable
5   for i ← 1 to N_thr do
6       STARTTHREAD(EGO-THREAD())
```

Fig. 10 Parallel `EGO-join` procedure

```
EGO-THREAD()
1   while [(A, B, d_str) ← GETJOB(Q))] ≠ QUIT do
2       JOIN(A, B, d_str)
```

Fig. 11 EGO- THREAD procedure

Not surprisingly, due to the above challenges, our multiple initial attempts to parallelize `EGO-join` have not succeeded: the performance would actually become slower and/or would not scale well with the increase in the amount of parallelism. In the next section, we will describe an algorithm that successfully solves this parallelization task. In Sect. 6, we will see that the proposed parallel solution outperforms the base `EGO-join` and scales relatively well with the increase in the level of parallelism.

4.4.2 Parallel solution

In order to succeed, a parallel version of `EGO-join` will need to account for the challenges identified in the previous section. The proposed parallel `EGO-join` solution starts as a regular `EGO-join` by EGO-sorting A and B, as illustrated in Fig. 10. But then, it puts a single job (A, B, d_{str}) in the priority queue Q which corresponds to joining A and B starting from the first dimension, as $d_{str} = 1$. It then creates $\mathcal{N}_{thr}$ parallel threads of execution. Figure 11 shows that each thread simply tries to extract a join job from the priority queue in a loop, until GETJOB returns `QUIT`. It then executes the extracted job by issuing the corresponding join.

JOIN procedure now needs to be modified. The new job-sharing logic shown in Fig. 12 should be added between Lines 4 and 5 of the original JOIN code from Fig. 2. The new code first checks whether the cardinality of A or B is sufficiently large to share this branch of recursion. If it is too small, that branch of recursion will not be shared with other threads, and the algorithm will proceed as a regular JOIN, skipping the new logic entirely.

But if it is not too small, the algorithm then locks $\mathcal{S}_{data}$ mutex that guards global variables like $\mathcal{N}_{jbls}$ (the number of jobless threads) and Q. It then checks if the size of Q exceeds the number of threads $\mathcal{N}_{thr}$ in Line 4.3. This part of the code deals with starvation: the job queue should be large enough so that whenever one or more threads need a job they do not starve and can immediately find a job in the queue, most of the time. At the same time, putting too many jobs in the queue

```
    // To be added between Lines 4 and 5 in Figure 2
4.1 if |A| > t_1 or |B| > t_1 then
4.2     lock(S_data)
4.3     if |Q| > N_thr then // Don't share jobs (enough in Q)
4.4         unlock(S_data)
4.5         goto 5 // to Line 5 in Figure 2

4.6 if |A| < t and |B| ≥ t then // Case 2'
4.7     {B_1, B_2} ← SPLIT(B)
4.8     Q ← Q ∪ (A, B_1, d_str) ∪ (A, B_2, d_str)

4.9 else if |A| ≥ t and |B| < t then // Case 3'
4.10    {A_1, A_2} ← SPLIT(A)
4.11    Q ← Q ∪ (A_1, B, d_str) ∪ (A_2, B, d_str)

4.12 else if |A| ≥ t and |B| ≥ t then // Case 4'
4.13    {A_1, A_2} ← SPLIT(A)
4.14    {B_1, B_2} ← SPLIT(B)
4.15    Q ← Q(A_1, B_1, d_str) ∪ (A_1, B_2, d_str)∪
            ∪(A_2, B_1, d_str) ∪ (A_2, B_2, d_str)

4.16    unlock(S_data)
4.17    unlock(S_need_job) // Signal new job availability
4.18 return ∅
```

Fig. 12 Sharing jobs in JOIN(A, B, d_{str})

will unnecessarily waist computational resources, preventing the algorithm from scaling well. Hence, by checking $|Q| > \mathcal{N}_{thr}$, the algorithm tries to maintain the job queue of good size of around $\mathcal{N}_{thr}$.

If $|Q| > \mathcal{N}_{thr}$, it does not share its jobs and proceeds as a regular JOIN. Otherwise, it shares its current branch of recursion with the other threads. Namely, instead of performing recursive joins on split portions of A and/or B, it puts the corresponding join jobs into the job queue. It then unlocks $\mathcal{S}_{data}$ mutex and signal to other threads that new jobs are available by unlocking $\mathcal{S}_{need\_job}$ mutex, as some threads might be waiting on $\mathcal{S}_{need\_job}$.

Notice how the new logic is guarded by a single `if` statement present in Line 4.1 of Fig. 12. It serves two purposes. The first one is that it allows to avoid frequent calls to the costly synchronization procedures *lock*() and *unlock*(). Second, it avoids unnecessary back-and-forth sharing of very small jobs among threads. In other words, it addresses the second parallelization challenge identified in Sect. 4.4.1. To address the first challenge, JOIN procedure slightly deviates from the traditional concept of producer–consumer that assumes a large pool of smaller jobs that are more or less uniform in size. Instead, in `EGO-join` algorithm, producers and consumers are the same threads, and each thread *emits* jobs as soon as it detects that $|Q| \le \mathcal{N}_{thr}$. While the jobs are non-uniform, the algorithm does ensure that the jobs are not too small.

The very important GETJOB procedure illustrated in Fig. 13 contains more synchronization logic compared to other parts of `EGO-join`. It is called by a thread when it is jobless and is trying to acquire a new job from the job queue. This is reflected in GETJOB by first acquiring access to global

```
GetJob(Q)
 1   lock(S_data)
 2   N_jbls ← N_jbls + 1
 3   goto 5

 4   lock(S_data)
 5   if |Q| = 0 then // No jobs in the queue
 6      if N_jbls ≥ N_thr then //No one is working, so quitting
 7         unlock(S_need_job) // Someone might "wait" on it
 8         unlock(S_data)
 9         return QUIT

         //– Some threads are still working –
10       unlock(S_data)
11       lock(S_need_job) //Wait till a job is posted
12       goto 4 // Try to get the new job

       //– The queue is not empty here –
13   (A, B, d_str) ← Q.Pop()
14   N_jbls ← N_jbls − 1

15   if |Q| > 0 and N_jbls > 0 then // Critical for N_thr > 2
16      unlock(S_need_job) // Signal more jobs are available

17   unlock(S_data)
18   return (A, B, d_str)
```

Fig. 13 GetJob procedure

data—by locking $\mathcal{S}_{data}$ mutex, and then increasing the number of jobless threads $\mathcal{N}_{jbls}$. The subsequent behavior of the algorithm depends on whether the job queue contains any jobs or not, which it checks in Line 5.

If there are no jobs, it checks if any thread is still working, since the working thread can still produce a new job (Line 6). If no threads are working, it means all work is done, and thus, the thread quits. But before quitting, it releases the lock on $\mathcal{S}_{need\_job}$ mutex, since some other threads might be suspended at that moment waiting for a new job, and hence waiting for the lock on $\mathcal{S}_{need\_job}$ to be released (Line 7). These threads need to be notified because otherwise they will wait indefinitely. If some threads are still working, then they might produce a new job, so the algorithm releases the lock on global data (Line 10) and suspends itself by waiting on $\mathcal{S}_{need\_job}$ mutex. Once some other thread generate a job (or decides to quit), it will release this mutex, awaking a thread waiting on it, which will try to get a job again by repeating the same procedure starting from Line 4.

If the check in Line 5 returns that the job queue is not empty, the thread will get a job from the queue and decrease the number of jobless threads $\mathcal{N}_{jbls}$ by one, since now it has a job (Line 13 and 14). Then, it does a very important step: it checks whether the queue contains more jobs and whether there are more jobless threads, and if so, it unlocks $\mathcal{S}_{need\_job}$ mutex to signal more jobs are available (Line 15 and 16). This is critical to do when the number of parallel processing cores is more than 2, as otherwise the code will not scale well beyond 2 threads. Conceptually, this is equivalent to implementing a counting semaphore (where the count corresponds to the number of available jobs) out of a (fast) binary mutex. The algorithm then unlocks the access to the global variables and returns the job obtained from the job queue.

Finally, it should be noted that each thread now maintains its own local version of the result set R, because otherwise, a single global R will need to be locked each time it is updated with a newly discovered (a, b) tuple, which is inefficient. When all threads are finished, the overall result is the union of these local result sets.

4.5 Space complexity

We know that in terms of computational complexity all exact (i.e., non-approximate) ε-join algorithms by definition have the worst case quadratic complexity of $O(|A| \cdot |B|)$. This is since by setting ε to a very large value such algorithms will be forced to output all pairs of points as their result set. Though if the problem definition is changed to allow to return groups of points instead of pairs, then CSJ-like techniques [5] could reduce the worst-time complexity.

However, ε-join approaches are vastly different in terms of their space complexity. We can observe that Super-EGO algorithm is not using any advanced index data structures and that is why it is very efficient in terms of its space complexity. Thus, even its in-memory version can be scaled to very large datasets, as we will see in Sect. 6. In fact, we are not aware of a single real dataset that has been used in the ε-join literature that could not be handled by Super-EGO entirely in-memory of a modern PC.

Super-EGO's in-memory version requires $O((|A|+|B|) \cdot d)$ space to hold datasets A and B as well as $O(|R|)$ space to hold the result set R. Depending on the desired selectivity, the size of R, of course, can be up to $|A| \times |B|$, but in many practical applications that do not require excessive selectivity, the size of R often is $O(|A|+|B|)$. As discussed in Sect. 4.1, the space complexity of the reordering phase is $O(d)$ which is subsumed by the above-mentioned $O((|A| + |B|) \cdot d)$ cost.

It should be noted that if the algorithm is supposed to save R to disk, then the $O(|R|)$ part of the spatial complexity becomes $O(1)$. The EGO-sort part of Super-EGO requires only $O(1)$ extra space, since it uses the space provided for A and B to sort. The `EGO-join` procedure is recursive and needs only $O(\log|A| + \log|B|)$ amount of space per thread, which is subsumed by the $O((|A|+|B|)\cdot d)$ cost if we assume that the number of threads is fixed. Thus, the overall space complexity is $O((|A| + |B|) \cdot d + |R|)$ if R is kept in main memory (default mode). It is $O((|A| + |B|) \cdot d)$ if R is saved to disk.[7]

[7] This is since the algorithm can save intermediate results into a fixed sized circular buffer. A separate thread can continually save the content of the buffer (in the background, concurrently with the main join algorithm) whenever the buffer is not empty. If for some reason, the saving

4.6 Miscellaneous issues

4.6.1 Extensions to dimensionality reordering

We next sketch a promising potential extension of the dimensionality reordering algorithms presented in Sect. 4.1. The extension ideas have not been implemented or tested. Observe that when reordering dimensions for the case of a self-join, a similar competitive strategy could be to reorder dimensions by using Principal Component Analysis (PCA) [13].[8] PCA is a technique that tries to find a new coordinate system in the multi-dimensional space such that the data has the most variance in the first dimension, the second most variance in the second orthogonal dimension, and so on. Applying PCA can handle the cases of linear correlation among dimensions in multi-dimensional datasets, and it could be used to reduce the dimensionality in data. Hence, for a self-join, applying PCA is anticipated to lead to better results than using the average-distance histograms. However, the challenge is to develop PCA-like techniques that could handle the generic case of a $A \bowtie_\varepsilon B$ where A and B are not the same, and thus, the dimensionality reordering and compression should happen simultaneously for A and B. Further, the criteria for selecting the best dimension should be changed from the standard one (i.e., the max variance in data) to the success factor discussed in the previous sections.

4.6.2 Disk-based version

Real datasets used in the literature to test ε-join are often rather small. We are not aware of a single one that `EGO-join` based algorithms cannot handle entirely in-memory of a regular PC with 8GB of RAM. That is, the operations with disk are limited to loading data into memory and saving the results set to disk, but the algorithm runs in-memory. Therefore, it is not very surprising that many other modern ε-join techniques, such as `LSS`, `E`^{2}`LSH`, `EGO-star`, `Grid` are either in-memory approaches, or have been tested entirely in-memory. If, however, the user wants to apply `Super-EGO` to datasets that do not fit in memory, there is a standard technique to process joins. The idea is to split A and B into contiguous sub-blocks $A_1, A_2, \ldots, A_n$ and $B_1, B_2, \ldots, B_m$ that do fit in memory and then compute $A \bowtie B$ by joining these sub-blocks $A_i \bowtie B_j$.

4.6.3 Optimizing self-join

It is easy to see that the case of a self-join $A \bowtie A$ can be optimized further. The optimization builds on the idea that for a self-join $A \bowtie_\varepsilon A$, if $a, b \in A$ and $(a, b) \in R$ then $(b, a) \in R$. Consequently, it is not necessary to perform both $A_1 \bowtie A_2$ and $A_2 \bowtie A_1$ in Case 4 as they will produce equivalent results. Instead, only one join can be performed, but when $(a, b) \in R$ is found, (b, a) should also be added to the result set. This optimization has not been used by `Super-EGO`.

5 Selectivity of join

Due to continual inconsistencies made by various research efforts that arise from disregarding the selectivity in empirical evaluations, it is desirable that this otherwise secondary issue be noted by all researchers who work on ε-joins.

5.1 Selectivity and related errors

Throughout this article, we use a new notion of "selectivity of a join *with respect to A*", though for brevity we often refer to it just as *selectivity*. Recall that the standard definition defines the selectivity of a join operation $A \bowtie B$ as $\frac{|A \bowtie B|}{|A \times B|}$ [11]. In contrast, the selectivity s_A of join $A \bowtie_\varepsilon B$ *with respect to A* is computed as the average number of points from B that join with a point from A. Let R be the result set of $A \bowtie_\varepsilon B$. Then, the selectivity w.r.t. A can be measured as follows:

$$s_A = \frac{|R|}{|A|}. \tag{3}$$

Frequently, a *self-join* is performed on a dataset, that is, $A = B$. In this case, each point joins with itself, and such trivial pairs of points $\{(a, a), a \in A\}$ are discarded when computing s_A, that is,

$$s_A = \frac{|R| - |A|}{|A|}. \tag{4}$$

When performing a join operation, parameter ε is set by the user/analyst based on particular needs of the underlying application that invokes the join. This parameter controls the selectivity of the join s_A, where setting it to lower (higher) values results in lower (higher) selectivity. However, the question arises of how to set ε during *testing* of various join techniques in research papers? Which values are reasonable, especially when the underlying application is not known?

With respect to the selectivity, the reader should expect the authors to cover a broad and reasonable range of selectivity

Footnote 7 continued
thread is not quick enough and the buffer becomes full, the main join algorithm should stop its processing to allow the saving thread to free up some space in the buffer. This technique has not been implemented in `Super-EGO`.

[8] This idea has been first suggested to the author by his colleagues. It has also been suggested by the anonymous reviewers of this article.

and explain how their ε-join algorithm behaves for different selectivity levels.

One common mistake that occurs during testing is when ε is set to values that are too small: so small that the selectivity stays at pure zero (or virtually at zero) for the *entire range* of ε values tested in some experiments. We have noticed this problem in several publications, and in fact, our own group has almost made this mistake while working on [14] but has managed to avoid it in the end. Most frequently, this issue happens for datasets with very high dimensionality. Notice, while having a few small selectivity values in a plot is reasonable and expected, the case where the selectivity is zero *everywhere* in a plot is likely to be a mistake that is both (a) unintended by the authors, and (b) unexpected by the reader. Notice that by observing only ε values, the reader cannot see the achieved selectivity, so she has to assume that the authors have chosen reasonable ε values, which might easily be not the case for ε-joins as we shall see soon.

As will be explained shortly, this error occurs because, with the increase of dimensionality, ε should actually be set to larger values (the fact that perhaps is not very intuitive), especially for uniform data. Not doing so will result in empty result set $|R| = 0$ when $A \neq B$. For a self-join where $A = B$, small ε results in $R = \{(a, a), a \in A\}$, and hence, $|R| = |A|$. Consequently, when this happens, the selectivity s_A can stay at pure zero level $s_A = 0$ for the entire tested range of ε.

A natural question is how it is even possible not to notice that $s_A = 0$, or very small, for (almost) all ε values in some experiments? One possible explanation is that frequently a self-join case of ε-join is tested, where $\{(a, a), a \in A\} \subseteq R$. A self-join always produces some non-empty result set $R \neq \emptyset$, which manifests itself as a portion of occupied memory space or a (potentially large) file on disk. Hence, without inspecting R closer, it is possible to wrongly conclude that ε-join produced a reasonable answer.

The other type of mistake is to draw conclusions about the performance of various join techniques from the cases of excessive selectivity. For example, drawing conclusions exclusively from the cases where $s_A \geq 10^4$, whereas the semantics of the domain dictates that, say, $s_A \in (0, 300)$ is more reasonable—is another type of error.

In general, the semantics of a particular join operation, that is, the end purpose of $A \bowtie_\varepsilon B$ for the particular A and B, determines what the reasonable range for s_A should be. For example, if a self-join is used to find similar images in a large image database of mostly unrelated images, then testing intervals like (0, 10], (0, 100], or even (0, 1] could be reasonable. Recall that for a self-join, selectivity s_A measures the *average* number of points that join a point of from A, except the point itself. So naturally there can be cases where some points from A join no other points, or where some points from A join many more than 100 other points, and that is why these intervals are reasonable.

5.2 The effect of larger dimensionality on epsilon

Now let us consider why reasonable values for ε can increase to non-intuitively high values for higher dimensional cases. Though existing real datasets on which ε-join is performed are decidedly non-uniform, we will use a uniform case just to demonstrate the point.

A frequent case that is tested in research publications is when points in A and B are uniformly distributed in d-dimensional unit hyper-cube $\Omega = [0, 1]^d$. Observe that even though it is a "unit" cube, Euclidean distances between points can be larger than 1, for example, the length of the diagonal of this cube is $\sqrt{d}$, so if $d = 64$, it is 8. To demonstrate that reasonable ε can increase with the increase of d, let us now compute a (conservative) *lower bound* on values of ε to get $s_A = 1$ on this dataset for the given d. The meaning of this lower bound is that if ε is set to smaller values than its value, then $s_A \leq 1$.

Given that a sphere of radius ε has the volume of

$$V_d(\varepsilon) = f(d)\varepsilon^d, \text{ where } f(d) = \frac{\pi^{\frac{d}{2}}}{\Gamma(\frac{d}{2} + 1)},$$

a randomly placed sphere with its center inside Ω will occupy no more than $V_d(\varepsilon)$ portion of Ω space. That is, it will occupy exactly this portion if it is fully inside Ω, or it will occupy less if it is only partially inside. Hence, a point from A on average will join with no more than $V_d(\varepsilon)n$ points of B, where $n = |B|$. Thus, to get $s_A \geq 1$, we need to set ε *at least* such that $V_d(\varepsilon)n \geq 1$, which translates into $\varepsilon \geq [f(d)n]^{-\frac{1}{d}}$. Figures 14 and 15 plot $[f(d)n]^{-\frac{1}{d}}$ function as d is varied in [2, 32] and [2, 1024], respectively. The three curves in these plots are for the cases of $n = 10^5$, $n = 10^6$, and $n = 10^7$. Notice that while ε is small for lower dimensionality cases, it can be quite large for higher dimensionality: it can exceed 0.5 for $d \geq 16$ and can exceed 1.0 for $d \geq 32$.

Figure 16 plots average s_A for various values of d for actual experiments on synthetically generated uniform datasets

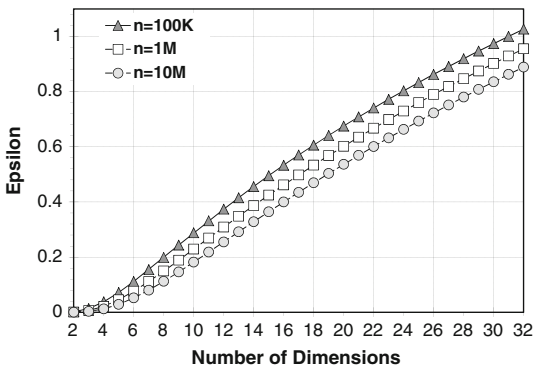

Fig. 14 If ε is less than these values, then $s_A < 1$

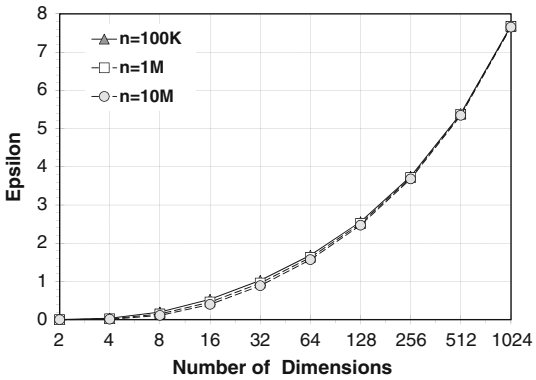

Fig. 15 If ε is less than these values, then $s_A < 1$

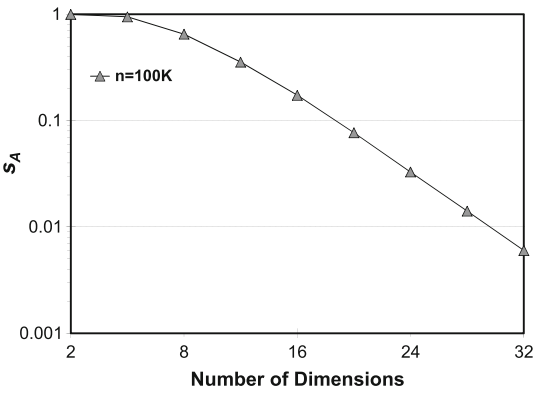

Fig. 16 Example of setting ε to $[f(d)n]^{-\frac{1}{d}}$

where ε is set its lower bound $[f(d)n]^{-\frac{1}{d}}$. As expected, s_A stays below 1. We can see that $[f(d)n]^{-\frac{1}{d}}$ is a *conservative* lower bound: s_A decreases as d increases. This is since with the increase of d, the values of ε increase as well. Hence, the volume of intersection of $V_d(\varepsilon)$ and Ω becomes much less than $V_d(\varepsilon)$ which has been used in the above calculations.

Let us assume now that the cardinality of A is also $n = |A|$. Since each point in A joins with no more than $V_d(\varepsilon)n$ points in B on average, the cardinality of the result set R on average will not exceed $|R| \leq V_d(\varepsilon)n^2$. Consequently, if $\varepsilon \leq [f(d)n^2]^{-\frac{1}{d}}$, then $V_d(\varepsilon)n^2 \leq 1$, and hence, $|R| \leq 1$ which means $s_A \simeq 0$ for such small values of ε. Figures 17 and 18 plot $[f(d)n^2]^{-\frac{1}{d}}$ function for d in $[2, 32]$ and $[2, 1024]$. Figure 19 is analogous to Fig. 16 but plots $|R|$ values for the case where ε is set to $[f(d)n^2]^{-\frac{1}{d}}$. We can see that $|R|$ stays at zero for such ε.

5.3 Consequences and suggestions

Consequence 1. Many known epsilon-join algorithms, including our own Grid, Super-EGO, and EGO-star, are simply not designed for large ε, for example, when $\varepsilon \geq 1$. For such cases, at best, they will default to the basic $O(n^2d)$ algorithm. From Fig. 14, we can see that for uniform data (a frequent testing case in the literature) when the number of points in A and B is $n = 10^5$, such algorithms will be limited to dimensionality of less than 32.

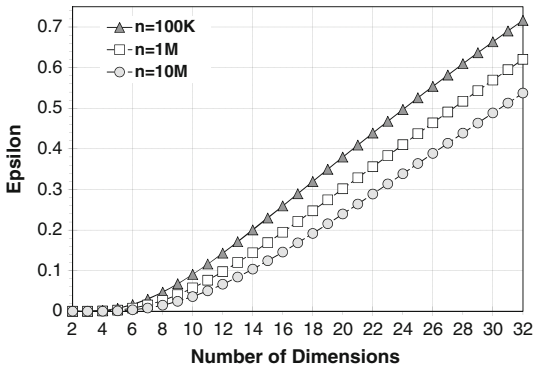

Fig. 17 If ε is less than these values, then $|R| \leq 1$

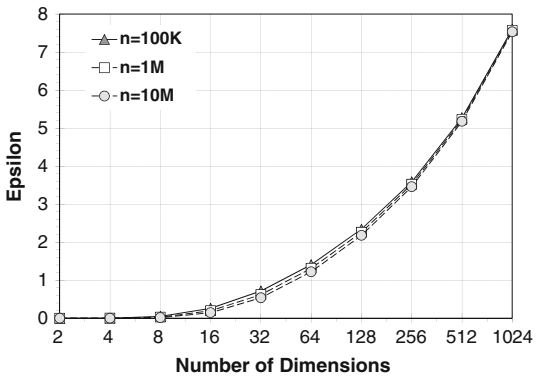

Fig. 18 If ε is less than these values, then $|R| \leq 1$

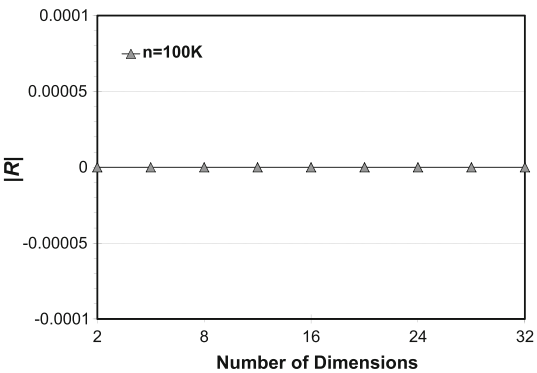

Fig. 19 Example of setting ε to $[f(d)n^2]^{-\frac{1}{d}}$

Real data tends to be skewed, resulting in smaller ε used in practice. However, a large increase in dimensionality is likely to result in an increase of ε thus limiting the applicability of modern similarity join techniques for real data as well. Hence, claims that some techniques apply to, say, $d \geq 64$ should be verified carefully.

Therefore, for high-dimensional cases, it is desirable that researchers demonstrate that their ε-join techniques beat the "dimensionality curse", by comparing them to, say, the $O(n^2)$ `block` quadratic baseline explained in Sect. 6.2.

Consequence 2. Some experiments reported in the literature disregard selectivity and we must be cautious in drawing any conclusions from them. In Sect. 6, we will show concrete examples where authors made their conclusions from cases where selectivity s_A was zero or too high. Armed with Figs. 14, 15, 16, 17 and 18, the interested reader can check (for tests on uniform data) that similar problems are not limited to the examples we show.

Consequence 3. To avoid selectivity-related mistakes in the future, it is desirable that researchers report not only ε used in their tests (which, apparently, can be deceptive to even the researchers themselves), but also the corresponding $s_A = \frac{|R|}{|A|}$, $|R|$ or similar measures. Recall that for a self-join $A \bowtie_\varepsilon A$, the metrics are $s_A = \frac{|R|-|A|}{|A|}$ and $|R|-|A|$. These measures can also serve as a *checksum* for other researchers performing tests on similar datasets. It is also desirable for researchers to analyze the concrete end goal/application of their specific join operation and then understand and explain which ranges of s_A are reasonable for that goal.

6 Experimental evaluation

In this section, we empirically evaluate our `Super-EGO` approach on several real and synthetic datasets. We compare it to several recent state of the art techniques: `CSJ` [5], `LSS` [18], `E`^{2}`LSH` [1], and `EGO-star`[16]. We are very thankful to the authors of these techniques for providing us the latest versions of their code. In addition, we present an in-depth analysis of the performance of these techniques with respect to the selectivity factor.

We start this section by first covering the experimental setup in Sect. 6.1. The setup describes the datasets used in the experiments, including which join techniques have utilized these datasets in the past in Sect. 6.1.1. The setup also introduces two quadratic-cost comparison baselines: $O(n^2)$ `naive` and $O(n^2)$ `block` in Sect. 6.2. Next, Sect. 6.3 presents basic experiments which (a) compare `Super-EGO` to its predecessor `EGO-star`, (b) study the contribution of the various join phases to the overall join cost of `Super-EGO`, (c) demonstrate the scalability of the algorithm with the increase of the parallelism, (d) test the contribution of various optimizations proposed in the paper, and (e) study the performance of the disk-based version of the algorithm. After that, Sects. 6.4, 6.5, and 6.6 compare the performance of `Super-EGO` to that of the state of the art ε-join algorithms `CSJ`, `LSS`, and `LSJ`, respectively. Finally,

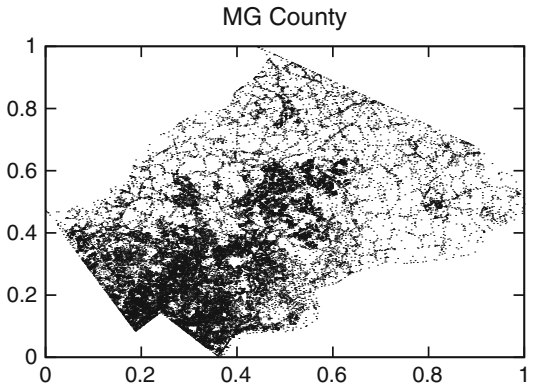

Fig. 20 Montgomery County dataset

Sect. 6.7 presents a critique of `EGO-star` as well as of some of other existing ε-join algorithms.

6.1 Experimental setup

Unless stated otherwise in the text, the experiments have been performed on a notebook with 8GB of RAM.[9]

6.1.1 Datasets

In our experiments, we compare `Super-EGO` to the state of the art techniques on the same datasets these techniques have used for their own testing:

1. **MNIST** (real, 784D, 60K) used by `J2` join explained in Sect. 6.7. This real 784-dimensional dataset consists of 28 × 28 matrix representations of handwritten 0-9 digits (gray-scale). Join can be viewed as a way to perform handwritten digits recognition by classifying images (assigning 0-9 labels) based on the labels of the closest ones, for example, by majority voting.
2. **Aerial** (real, 60D, 275 K) used by `E`^{2}`LSH`. This real dataset represent geographic map image tiles. Join corresponds to finding similar tiles.
3. **MG County** (real, 2D, 27 K) used by `CSJ`. A real dataset that represents the road network of Montgomery County, illustrated in Fig. 20.
4. **LB County** (real, 2D, 36 K) used by `CSJ`. A real dataset for the road network of Long Beach County, illustrated in Fig. 21.
5. **ColorHistogram** (real, 32D, 68 K) used by `LSS`. A real dataset of image features extracted from a Corel image

[9] The notebook has a single Intel(R) Core(TM) i7-2820QM (4-core) CPU @ 2.30 GHz. Its Geekbench score (Geekbench 2.1.13 32-bit) is 10,531. This score can be used as a means to compare different epsilon-join techniques across publications in an approximate fashion: the reported execution time results can be prorated according to this score.

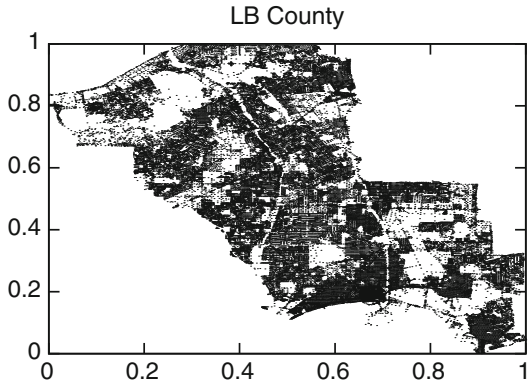

Fig. 21 Long Beach County dataset

```
BASELINE1(A, B, ε)
1  REORDERDIM(A, B)
2  SIMPLEJOIN(A, B, ε)
```

Fig. 22 Quadratic baseline O(n²) naive

collection. Histogram intersection (overlap area between color histograms of two images) can be used to measure the similarity between two images.

6. **ColorMoments** (real, 9D, 68 K) used by LSS. Also real image features, but of different type. Euclidean distance between Color Moments of two images can be used to represent the dis-similarity (distance) between two images.
7. **CoocTexture** (real, 16D, 68 K) used by LSS. Real image features, but of different type.
8. **LayoutHistogram** (real, 32D, 66 K) used by LSS. Also real image features. Histogram Intersection can be used to measure the similarity between images.
9. **Uniform** (synthetic, up to 200 Million points). A synthetic dataset where n data points are distributed uniformly inside a unit cube $[0, 1]^d$.

All datasets are normalized to fit in $[0, 1]^d$ domain.

6.2 Two quadratic baselines

To test if a given ε-join technique "beats the curse of the dimensionality" on a given dataset, we have implemented two quadratic baselines called O(n²) naive and O(n²) block.

Figure 22 provides the pseudocode for O(n²) naive. It first re-orders dimensions of A and B in the descending order of r_i, as explained in Sect. 4.1. It then calls the quadratic SIMPLEJOIN(A, B, ε) procedure explained in Fig. 7 in Sect. 4.3.

The pseudocode for the second O(n²) block baseline is provided in Fig. 23. It is similar to the first one, except for it calls SIMPLEBLOCKJOIN instead of SIMPLEJOIN, with the "block size" parameter M set to 100. SIMPLEBLOCKJOIN is

```
BASELINE2(A, B, ε)
1  REORDERDIM(A, B)
2  SIMPLEBLOCKJOIN(A, B, 100, ε)
```

Fig. 23 Quadratic baseline O(n²) block

```
SIMPLEBLOCKJOIN(A, B, M, ε)
 1  R ← ∅ // Result set
 2  N_blc ← ⌈|B| / M⌉
 3  //Let B_1, B_2, ..., B_{N_blc} be contiguous blocks of B of size M
 4  for each B_k ∈ B do
 5      for each a ∈ A do
 6          for each b ∈ B_k do
 7              s ← 0
 8              for i ← 1 to d do
 9                  s ← s + (a_i − b_i)²
10                  if s ≥ c then // where c = ε²
11                      goto next_ab
12              R ← R ∪ {(a, b)}
13              next_ab:
14  return R
```

Fig. 24 SIMPLEBLOCKJOIN procedure

shown in Fig. 24. It views (array) B as consisting of N_{blc} contiguous blocks (subarrays) of size M. The new procedure adds one extra external loop to SIMPLEJOIN that iterates over each block $B_k \in B$. The final change is that now it iterates over $b \in B_k$ instead of $b \in B$.

Initially, one might assume that O(n²) naive should be faster or about the same as O(n²) block. This is since O(n²) naive does the same comparisons as O(n²) block (though in different order) but with less code. However, we will see that O(n²) block can be significantly faster than O(n²) naive due to the reasons that will be explained in Experiment 12.

It is easy to see that these two baselines can be further sped up by factor of up to $\simeq 2$ for the case of a self-join. This is since a self-join can be implemented as a loop over $i \leftarrow 1$ to $|A|$, and then, a loop over $j \leftarrow i + 1$ to $|A|$ (instead of $j \leftarrow 1$ to $|A|$). We, however, will **not** use this optimization, but the reader can estimate its effect by dividing the reported time by 2.

6.2.1 Selectivity

We will demonstrate the importance of selectivity s_A for making a proper comparison among various join techniques. We will see that it plays a crucial role in explaining many plots. Recall from Sect. 5 that the selectivity s_A for a self-join $R = A \bowtie_\varepsilon A$ was defined as $s_A = \frac{|R|-|A|}{|A|} = \frac{|R|-n}{n}$, where $n = |A|$ is the number of points in A and R is the result set of the self-join. It tells the average number of points that joins with any given point $a \in A$, except for itself, that is, (a, a) pair is not counted. The selectivity is plotted as a dashed curve using the secondary (right) axis in each plot in this section. Unless stated otherwise, in our plots we vary ε such that s_A covers a broad range of reasonable values.

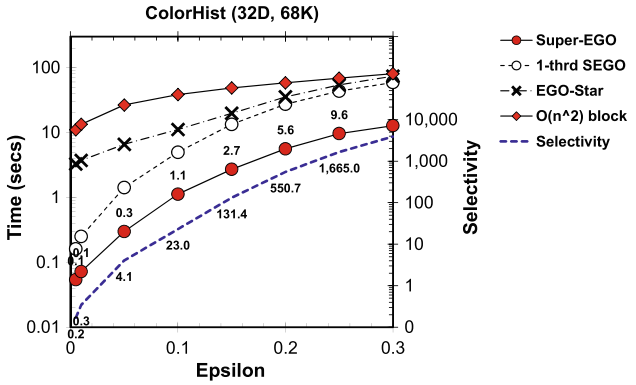

Fig. 25 `Super-EGO` versus `EGO-star` on ColorHist

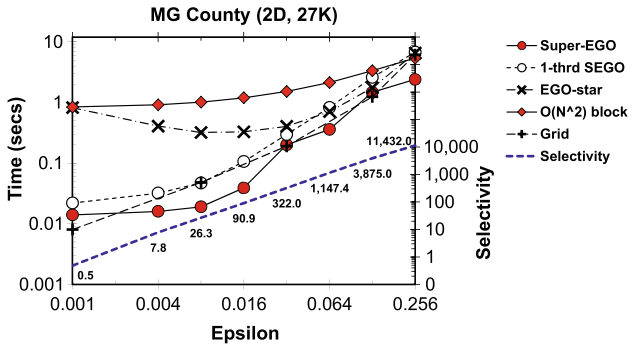

Fig. 26 `Super-EGO` versus Grid on MG County

6.2.2 Validation

As we shall see, `Super-EGO` gains major improvement over competing strategies, including our own older techniques. To validate that this is not due to an accidental error in the code, we performed frequent validations of the results across different datasets. Specifically, the result set R obtained by `Super-EGO` was compared to the results sets $R_1, R_2, \ldots, R_n$ obtained by different strategies, including (where applicable) `EGO-star`, `Grid`, `O(n^2) naive`, `O(n^2) block` and were found to be identical.

6.3 Basic experiments

Experiment 1 *(Comparing with EGO-star: high-dimensional case)* Since `Super-EGO` is based on `EGO-star` [14,16], in this experiment we compare their performance, using the original `EGO-star` code.

Figure 25 plots the execution time (for a self-join) of various techniques as a function of ε on 32-dimensional ColorHist dataset that contains 68,000 data points. This dataset was used in [14,16] for testing `EGO-star`.

We can see that choosing $\varepsilon > 0.16$ leads to very high selectivity for this image dataset. Hence, values where $\varepsilon \in [0, 0.16]$ are more likely to be used in practice. When $\varepsilon \in [0, 0.1]$, the new `Super-EGO` algorithm outperforms `EGO-star` from $\simeq 53$ to 9 times. Even 1-threaded version of `Super-EGO`, "`1-thrd SEGO`", outperforms `EGO-star` anywhere from $\simeq 21$ to 2 times when $\varepsilon \in [0, 0.1]$. The figure also shows that all the tested methods outperform the quadratic-cost baseline `O(n^2) block`, as they should, to beat the dimensionality curse on ColorHist.

Experiment 2 *(Comparing with EGO-star: low-dimensional case)* As explained in [14,16], `EGO-star` is not meant for lower dimensional (e.g., 2D) cases and instead the `Grid` technique should be used in such cases. We can see why this is the case from Fig. 26 which plots the performance of various techniques on 2D MG County dataset.

When $\varepsilon \in [0.001, 0.008]$ the selectivity reaches reasonable values of $s_A \in [0.5, 26.3]$. For such ε values `Super-EGO` is from $\simeq 59$ to 17 times faster than `EGO-star`. In Fig. 25, ε is varied in [0.001, 0.256] instead of [0.001, 0.008] because the `CSJ` technique, which we will compare to later on, uses even larger interval.

We can see that while `Super-EGO` outperforms grid, its 1-threaded version `1-thrd SEGO` is $\simeq 3$ times slower than grid when $\varepsilon = 0.001$. This means that `Grid` is still a good technique for lower dimensional data and lower selectivity cases. In [14,16], `EGO-star` was designated as a method meant for higher dimensional cases only. Now, the new EGO-join—`Super-EGO`—demonstrates reasonable performance across a wide spectrum of dimensionality and selectivity (Fig. 26).

Experiment 3 *(Contribution of various phases)* The end-to-end `Super-EGO` process can be viewed as consisting of several phases:

1. *Load.* Data is loaded from a file on disk.
2. *Reorder.* Dimensions of data are reordered.
3. *Sort.* EGO-sort is applied to data.
4. *Join.* EGO-join is applied to sorted data.
5. *Save.* Result set R is saved to disk (optional).

Figures 27 and 28 plot the relative fraction and actual time each join phase takes in the end-to-end join operation on ColorHist dataset. These figures correspond to Fig. 25.

The first phase is *loading* data. We have made no attempt to optimize this phase as its cost is traditionally ignored.[10]

[10] One of the reasons of why it is often ignored is that, as we will see later on, many other join techniques are much slower than `Super-EGO`, and in their case, the cost of loading data is negligible compared to the cost of the join itself. Another reason is that raw data comes in vastly different formats, and an ad-hoc procedure is often needed to convert it to some predefined format or an ad-hoc loader needs to be created. Furthermore, some techniques (such as `CSJ`) that, unlike `Super-EGO`, contain an index-building phase, even ignore the cost of building (R-tree) index on data, which is often quite large.

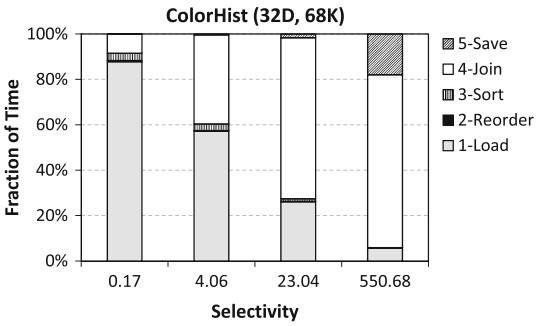

Fig. 27 Fraction of time per phase

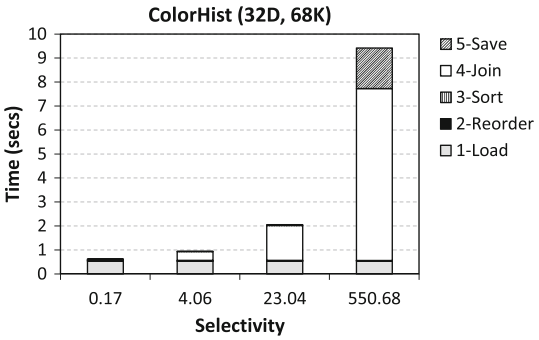

Fig. 28 Time per phase

We do not report the loading cost as well, unless we compare the end-to-end running times of algorithms, for example, when comparing with LSH. Currently, in Super-EGO, data is loaded one value at a time, and, if necessary, it is very likely that that phase can be optimized by, say, implementing a buffered read.

The second phase is *reordering of dimensions*. Its cost is so small that it is indiscernible in these two plots.

The third phase is *sorting* of data. It only plays a role for low-selectivity cases. It can be easily optimized by using a parallel sort and/or hash sort, but as we can see from the plots—that would not lead to any major performance improvement.

The fourth phase is the *join* itself and since the cost of loading data is ignored—it is the most expensive part of the overall processing even for Super-EGO.

Finally, we can see that the cost of *saving data* to disk (if that is required by the user) is also negligible—unless the selectivity of the join has to be excessively high, which is rare in practice. But even if the selectivity is high, the saving phase does not have to start after the join phase: instead it can overlap with it as the results can be buffered and then saved periodically during the join itself (not implemented in the Super-EGO). So implementing the saving phase as a separate thread that runs concurrently with the join will likely amortize the cost of that phase considerably. Notice that, by default, Super-EGO does not save data to disk.

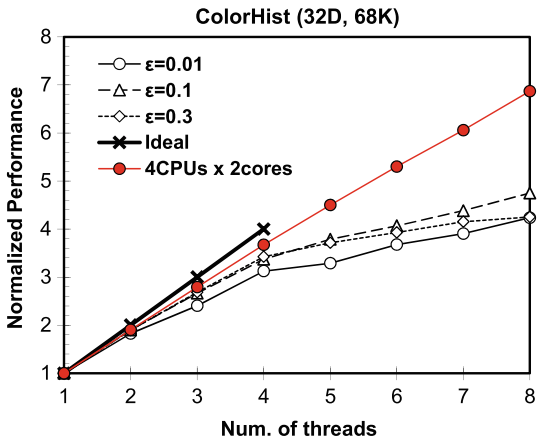

Fig. 29 Increasing parallelism: 1 CPU with 4 cores and 8 hardware threads

Overall, this picture is consistent with those of other research efforts. For example, the authors of the LSS technique state in [18]: "we found I/O times to be insignificant when compared to the actual join processing times, so they are not shown separately in our experimental results".

Experiment 4 *(Scaling with the increase of parallelism)* In this experiment, we test the scalability of the join phase of Super-EGO with the increase of the level of parallelism. We perform tests on two machines:

1. 1 CPU 4 Cores 8 Threads. This is the default notebook, released in early 2011.
2. 4 CPU's × 2 Cores. This is a slower and outdated 2004 machine. However, it has 4 independent CPU's each with 2 cores, that is, 8 CPU cores in total.

Figure 29 plots the normalized performance as a function of the number of threads for the 1 CPU machine. The normalized performance of n threads is computed as the execution time of the join phase for 1 thread, divided by that of n threads. The figure plots curves for: the "ideal" performance, for tests with various values of ε on ColorHist dataset, and a curve for the second 8-core machine as a point of reference.

We can see that the scalability depends on the values of ε. Knowing that the machine has 4 cores, we can expect that the performance should grow till 4 threads, but then should become flat (or should not grow) after that. However, curves for "$\varepsilon = val$" consist of two distinct (almost straight) growing line segments: one segment for n from 1 to 4 and another one for n from 4 to 8. The second segment shows slower growth than the first one, but it is not flat. This is since the machine has 4 cores but 8 hardware threads. According to Intel [6], hardware threads can create additional level of parallelism but only at the level of up to $\simeq 30\,\%$ of extra performance. Interestingly, the performance for these curves does

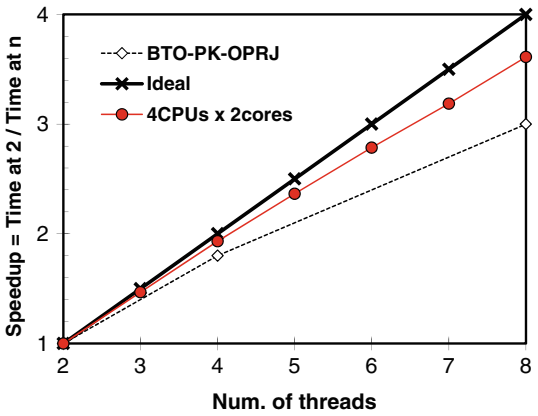

Fig. 30 Increasing parallelism: 4 CPU's × 2 cores = 8 CPU cores

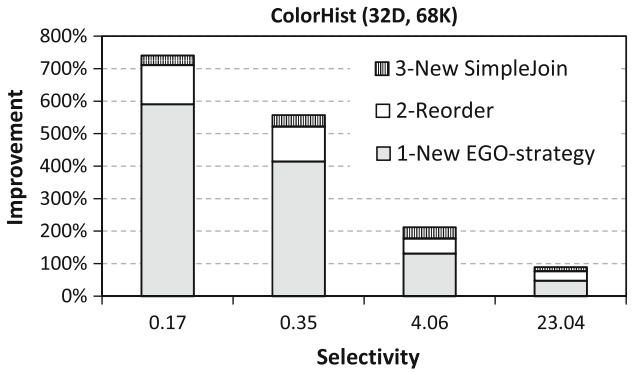

Fig. 31 The contribution of optimizations

increase by $\simeq$ 30 % as we increase the number of threads from 4 to 8. Figure 29 also includes a curve for the second 4 CPU's × 2 cores machine. We can see that with full 8 cores `Super-EGO` scales visibly better on the second machine.

To provide at least some concrete comparison yardstick, Fig. 30 plots the scalability of `Super-EGO` against that of the (unrelated set-based) similarity join approach proposed in [27] for map/reduce. The best-scaling technique in [27] was called `BTO-PK-OPRJ`, and its performance is reflected in Fig. 30. As in [27], we plot the speedup which is computed as time at 2 divided by the time at n. We can see that `Super-EGO` scales better than `BTO-PK-OPRJ`.

Experiment 5 *(Effect of optimizations)* Fig. 31 illustrates the relative effect of different optimizations on ColorHist dataset for different selectivity levels.

For example, the figure shows that applying the new EGO-strategy described in Sect. 4.2 is responsible for 5 times improvement (of $\simeq$ 400 %) of the algorithm when $s_A = 0.35$. Applying the reordering strategy outlined in Sect. 4.1 on top of that, doubles the performance (the increase of $\simeq$ 100 %). Applying the new SimpleJoin algorithms described in Sect. 4.3 on top of the previous two techniques increases the performance further by $\simeq$ 35 %. Hence, the overall improvement in this case is $\simeq 5 \cdot 2 \cdot 1.35 = 13.5$ times.

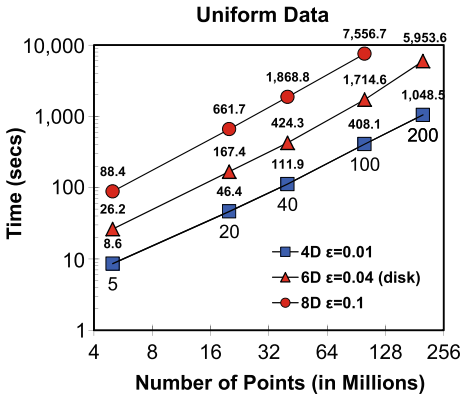

Fig. 32 Scaling to 100–200 Millions of points

The effect of these techniques decreases with the increase in selectivity. This is since with increase in s_A more pairs of points join with each other. Consequently, the optimizations, which are aimed at early detection of points that will not join, become less effective.

Experiment 6 *(Scalability w.r.t. data size)* Fig. 32 studies the scalability of `Super-EGO` on 4- and 8-dimensional uniform datasets as the cardinality of A increases. It shows that the approach can be scaled to 200 Million 4D points and 100 Million 8D points on a notebook with 8 GB of RAM. Incidentally, to the best of our knowledge, these are the highest cardinality tested for 4- and 8-dimensional data as well as the best results reported for such data on any platform.

For instance, `LSS` authors scale their approach to 4 million 16D uniform points. Even though `CSJ` is disk-based, it was only tested on 1.5 million 2D real points. One of the largest cardinality tested in the literature that we are aware of is 40 million of 8D uniform points, published in [2]. In general, the reasons why others are using smaller cardinality include (a) absence of real datasets for ε-join with such cardinality, (b) some techniques require large amounts of memory, for example, `LSH`, `Grid`, and (c) for slower techniques it takes significant amount of time for the experiments to finish on large datasets.

Figure 32 is a log-log plot, where curves are straight lines. Hence, given the slope of the curves, we can compute that the scalability for the 4D and 8D cases as $O(n^{1.30})$ and $O(n^{1.49})$, respectively, where $n = |A|$.

Experiment 7 *(Disk-based Super-EGO)* Fig. 32 also contains a curve for the end-to-end running time (including loading data) of the disk-based `Super-EGO` for 6D uniform data. It is implemented as described in Sect. 4.6.[11] Its behavior is

[11] Buffers for A_i's and B_j's have been set to include no more than 50M points.

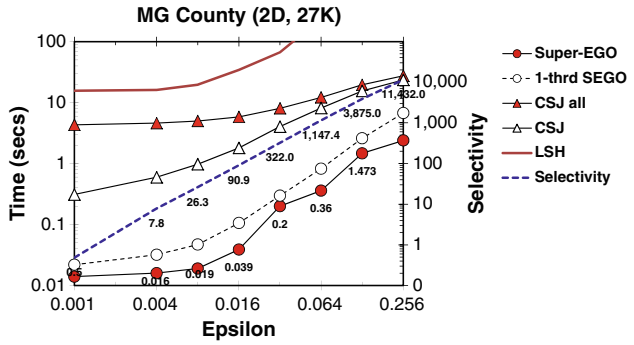

Fig. 33 MG County

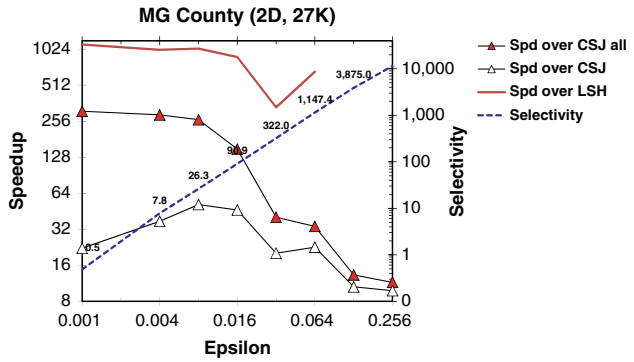

Fig. 34 MG County. Speedup over `CSJ(10)`

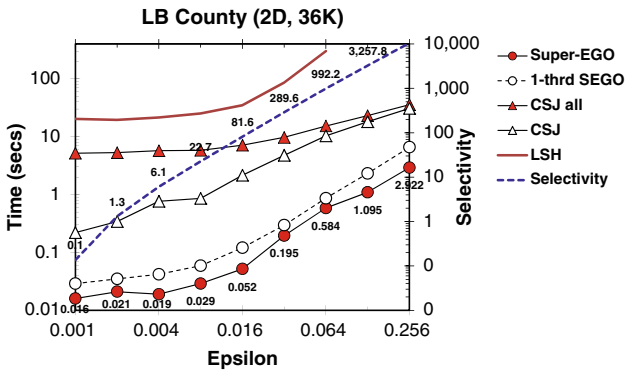

Fig. 35 LB County

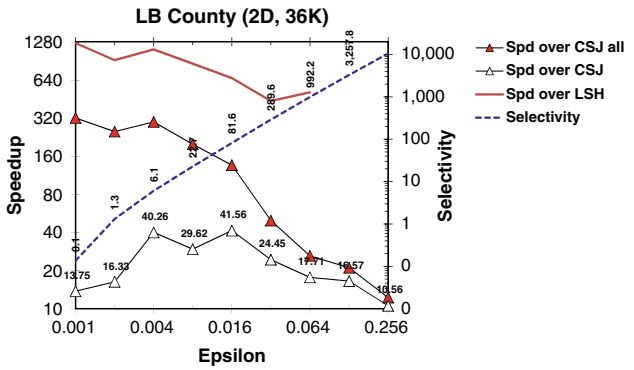

Fig. 36 LB County. Speedup over `CSJ(10)`

similar to those of 4D and 8D in-memory joins. `Super-EGO` is not meant as a disk-based strategy, and it is likely that this disk-based version of the algorithm can be improved.

6.4 Comparing with CSJ

`CSJ` is the Compact Similarity Join technique proposed in [5], where the authors show that detecting and reporting groups/cliques of points that all join with each other is a good idea. `CSJ` is the only disk-based strategy we test—the others all run in-memory. We will compare `Super-EGO` and the original `CSJ` code on 2-dimensional MG and LB County datasets. While these datasets are "real", more interesting datasets to test a spatial join would have been a POI database or twitter feeds with GPS coordinates. However, we test on MG and LB data since they have been used in [5] by `CSJ`

Experiment 8 *(`CSJ` for spatial join)* Figs. 33, 34, 35 and 36 compare performance on of `Super-EGO`, 1-threaded version of `Super-EGO`, `CSJ all` which is `CSJ` with index-building cost not ignored, and `CSJ` with index-building cost ignored. The figures demonstrate that `Super-EGO` significantly outperforms `CSJ` even if the index-building cost is ignored. The difference can be more than 2-orders of magnitude for `CSJ all` for reasonable selectivity values. This is not very surprising, as `EGO-star` (on which `Super-EGO` is based) has been shown to outperform `SSJ` (on which `CSJ` is based) by over 1 order of magnitude [14,16]

What is interesting is that in Figs. 34 and 36 the speedup over `CSJ` for $\varepsilon = 0.001$ and 0.002 is less than that for $\varepsilon = 0.016$. The explanation for it is that this is a rare case where the computational cost of the *sort phase* of `Super-EGO` becomes substantial: normally it is negligible, whereas here, it is up to 40–50 %. Hence, we see the effect of a parallel version of EGO-sort not being implemented.[12]

Figure 33 is similar to Figure 5 from [5]. Since Fig. 33 in addition plots the actual selectivity s_A for each ε, it can provide interesting insights into Figure 5 in [5]. For instance, in Figure 5 from [5], `CSJ` does not show a very major improvement in the execution time over `SSJ` for $\varepsilon \leq 0.032$. However, this is where the selectivity already reaches the very high value of $\simeq$ 300. Hence, a reasonable question could be whether users would want to run `CSJ` with $\varepsilon \geq 0.032$ on specifically MG and LB County-like datasets, and hence, whether they will see a tangible improvement of `CSJ` over `SSJ` in practice. This highlights the importance of analyzing the selectivity s_A in join operations.

[12] It was not implemented exactly because EGO-sort cost is normally just a small fraction of the overall cost.

The MG and LB county figures also include the curves for LSH. The LSH authors very clearly state that LSH is not meant for low-dimensional cases and the figures reflect that, as expected.

6.5 Comparing with LSS

LSS [18] is an ε-join algorithm that is based on an interesting idea of using GPU (video card) to perform a join by leveraging NVIDIA's CUDA toolkit. That idea, however, has its pros and cons in practice. The obvious advantage is the gain in speed from extra hardware. A disadvantage is that LSS is coded for NVIDIA GPUs, so the code simply would not work on the default early-2011 notebook we used for testing due to its video card mismatch. Hence, to test LSS we have used a 2012 notebook that has 60–70 % faster GPU but only 10 % faster CPU. Thus, we are giving LSS an advantage.

An important point about LSS is that it is already a massively parallel algorithm that runs on GPU and fully loads GPU, not CPU. The GPU is **the bottleneck** of the approach. Therefore, implementing a multi-threaded version of LSS (to run it in multiple threads on CPU) should not lead to any noticeable performance gain of that technique. Consequently, LSS should be compared to Super-EGO, not to its 1-threaded version.

The LSS authors have given us the original code. LSS runs entirely in-memory. The authors have requested to put a disclaimer that the code has been optimized for older version of CUDA and that tuning it for the current version can improve the performance of LSS.

Experiment 9 *(Comparing with* LSS*: Real Data)* Figs. 37, 38, 39 and 40 correspond to Figure 4 in [18]. They plot LSS and Super-EGO values on four real datasets that represent 4 different types of image features of the same collection of Corel images. Performing a self-join on Color Histogram and Layout Histogram loosely corresponds to finding similar images. Hence, the most reasonable selectivity s_A for these datasets is likely to be somewhere around 1, and most likely

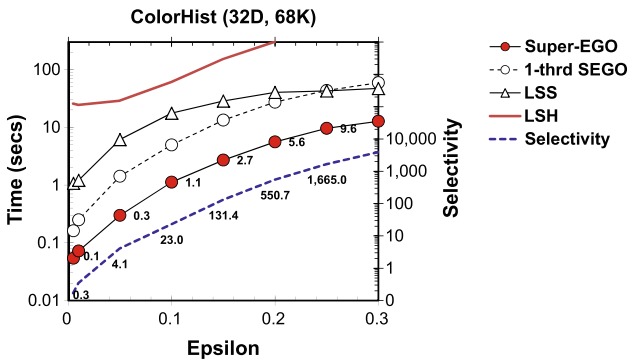

Fig. 37 Color histogram

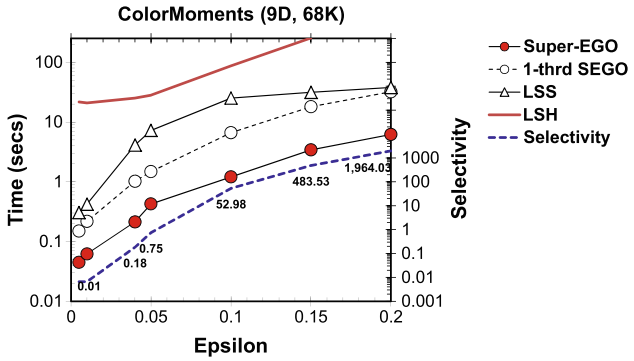

Fig. 38 Color moments

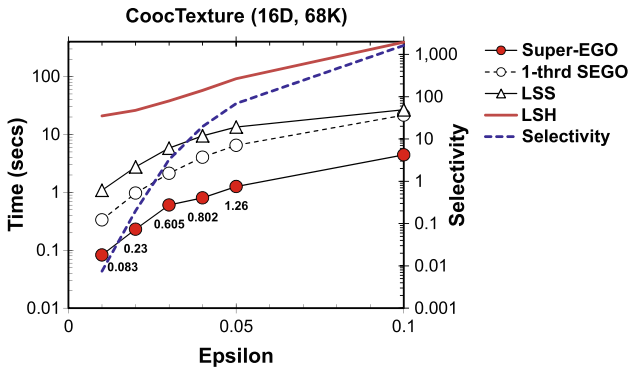

Fig. 39 Co-occurrence texture

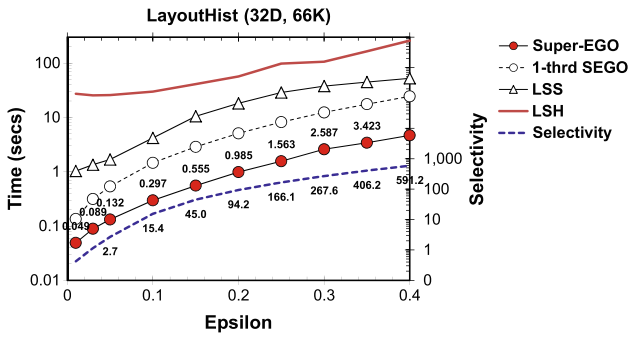

Fig. 40 Layout histogram

less than 100. A self-join on Color Moments finds pairs of the most dissimilar images—practical usefulness of which is not clear. The meaning of a self-join on Co-occurrence Texture is not apparent from the description of the dataset.

As we can see, Super-EGO tends to outperform LSS by about an order of magnitude across the board. The figures also include results for 1-threaded version of Super-EGO. Even though 1-threaded version is largely irrelevant in this context (this is since multi-threaded, LSS is unlikely to work faster) the figure demonstrate that even that version tends to be faster than LSS, except for very high selectivity cases.

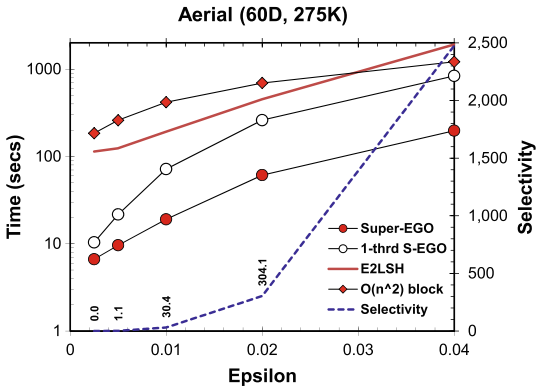

Fig. 41 Comparing to LSH on Aerial dataset

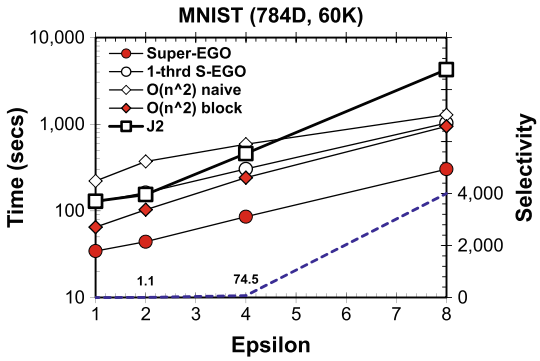

Fig. 42 Comparing to J2 on MNIST dataset

6.6 Comparing with LSH

Although E^{2}LSH has been designed as an NN algorithm [1], its authors (and the community) view it as one of the best modern epsilon-join algorithms of today. E^{2}LSH authors state that it is not meant for lower dimensional cases and that it only applies for dimensionality starting from $\simeq$ 10–20. While in the database literature ε-join is typically studied for $d \leq 32$, the preferred dimensionality of E^{2}LSH is $d \geq 60$. The authors of E^{2}LSH have given us the latest version of the code, which we will use in our tests. E^{2}LSH is an approximate join algorithm. To compare it to exact techniques, we set its "probability that nearest neighbor is not reported" to $\delta = 0.01\,\%$.

Experiment 10 *(*LSH *on real image data)* Figs. 37, 38, 39 and 40 tests E^{2}LSH on four real datasets that correspond to image features. These datasets have not been used to test LSH before. The figures demonstrate that LSH "as is" cannot compete with other techniques on these 9–32 dimensional datasets.

Experiment 11 *(*LSH *on Aerial data)* Fig. 41 compares E^{2}LSH and Super-EGO on Aerial 60-dimensional dataset of 275K points that has been used by LSH in the past.

The points represent features of map tiles, and the join is used to find similar tiles. E^{2}LSH outperforms O(n^2) block quadratic-cost baseline, but Super-EGO and its 1-threaded version 1-thrd S-EGO demonstrates better performance than E^{2}LSH.

6.7 Miscellaneous experiments

This section demonstrates examples of several issues that are present in some of ε-join research efforts. Namely, it shows instances of cases where the selectivity stayed at the zero (or very low) level in entire plots. It also demonstrate examples where ε-join techniques could not outperform our quadratic baseline O(n^2) block, or its parallel version where appropriate.

We note that the issues themselves are more important than the names of the concrete techniques where they are present. Thus, we will anonymize the names of the actual techniques tested and refer to them only as J1, J2, J3, and J4. Some of them can be the same as tested above, some of them can be different. We should note, however, that J1–J4 are *not* some marginal outliers: they are well-known recent state of the art methods developed by prominent research groups.

Experiment 12 *(*J2 *on MNIST data)* This is probably the most interesting experiment in this paper. The main dataset used by J2 is the 784-dimensional MNIST dataset that contains 60 K points. A point in a dataset represents a handwritten digit from 0 to 9, mapped into 28 $\times$ 28 gray-scale matrix. Hence, the ε-join can be used for recognizing written digits—by assigning a label based on the labels of points that join with a given point, for example, by using majority voting.

The original purpose of this test of Super-EGO on MNIST dataset was to show where Super-EGO should **fail**. This is since, as we now know from Sect. 5, with 784 dimensions the "right" value of ε is likely to be more than 0.5, in which case Super-EGO should degrade to a quadratic strategy. But the outcome of this experiment has led to a few completely unexpected surprises discussed next.

Figure 42 is a log-lin plot of the results on the MNIST dataset. It shows that, yes, ε has to be larger than 1 to get meaningful selectivity. The first surprise is that Super-EGO is still the fastest technique, even though it is supposed to be *quadratic* for $\varepsilon \in [1, 8]$. Furthermore, even its one threaded version, 1-thrd SEGO, is either about the same for $\varepsilon \in [1, 2]$, or faster than J2 for $\varepsilon \in [3, 8]$.

Because of the above, we have implemented our first quadratic baseline O(n^2) naive which is nothing more that the dimensionality reordering procedure (Sect. 4.1) followed by a quadratic SIMPLEJOIN (Sect. 4.3). The second surprise was that O(n^2) naive got the same result set R

as Super-EGO, but was visibly *slower* than 1-threaded Super-EGO, see Fig. 42.

The question is how is that even possible? Intuitively, Super-EGO will do the same comparisons of each point to each point as O(n^2) naive. However, it has all that extra code from Sect. 4 to do that. Therefore, from purely theoretical point of view, one might initially think it should be less efficient than O(n^2) naive. But a closer look at Super-EGO reveals that when it performs a SIMPLEJOIN it always joins a small contiguous *block* of points from A with a *block* from B. Therefore, it takes advantage of the CPU cache, whereas O(n^2) naive "AS IS" actively purges points from the CPU cache.

Armed with these observations, we have implemented our second quadratic baseline O(n^2) block. Like Super-EGO, O(n^2) naive also takes advantage of the CPU cache by using blocks in its procedure. Naturally, it computes the same result R as Super-EGO and O(n^2) naive. We can see that the new O(n^2) block outperforms both J2 and 1-thrd SEGO on MNIST data. This is important since any good join algorithm is supposed to be faster than any quadratic strategy. In a way, this experiment show that the "curse of dimensionality" is not addressed for MNIST dataset by any of the existing ε-join algorithms.

Experiment 13 *(J2 as an Approximate Join)* We know that Super-EGO and J2, when they have been invoked as exact joins, failed to beat the dimensionality curse on MNIST 784-dimensional data. An interesting question to study is whether J2, as an approximate join, could beat the curse.

Thus, we test J2 on MNIST data while setting its "probability of success" p parameter to $p = 50\%$. To level the play-field, we run an "approximate" version of O(n^2) block. That version is the same as before, except for it has the following line added to the pseudocode in Fig. 24 (as Line 6.5): **if** $rnd() > 0.5$ **then continue**.

Figure 43 demonstrates that the approximate version J2, p = 0.5 still cannot beat the quadratic baseline O(n^2) block, p = 0.5.

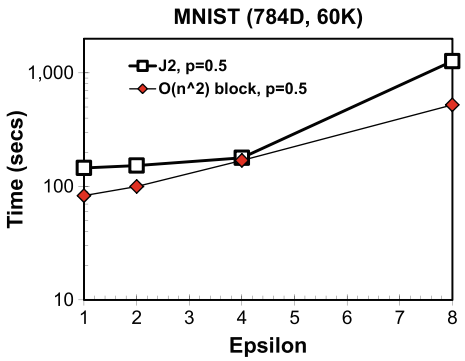

Fig. 43 J2 as an approximate join

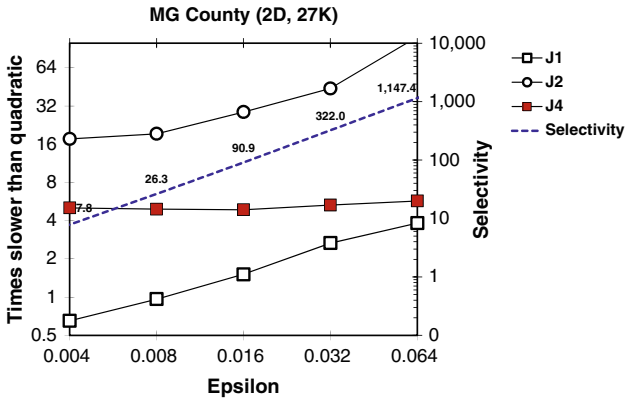

Fig. 44 Time of J1, J2, J4, divided by time of O(n^2) block

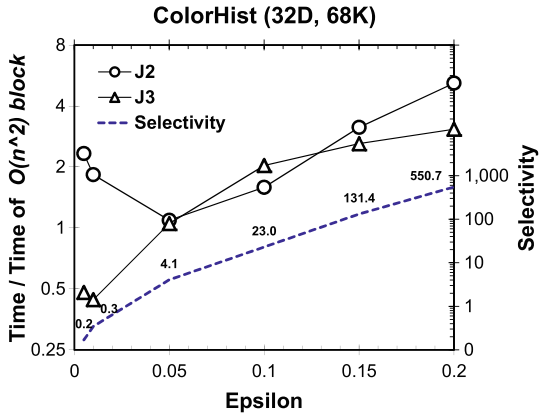

Fig. 45 Time of J2 and J3 divided by time of O(n^2) block

Experiment 14 *(Comparing with Quadratic Baseline)* Experiment 12 has made us develop O(n^2) block quadratic baseline. It has become interesting to know how other state of the art techniques would fair against it. What we have discovered is that J1–J4 would often have difficulty outperforming it. We note that while our own EGO-star tends to outperform O(n^2) block, the difference between them is often not very drastic—especially if O(n^2) block is further optimized by a factor of 2 for the self-join case. Hence, in a way, EGO-star suffers from the same issue. Among the datasets, we have used for testing, 1-threaded Super-EGO has always outperformed O(n^2) block on all datasets except for the 784-dimensional MNIST dataset.

Figures 44 and 45 plot how much J1–J4 are slower than the quadratic-cost baseline O(n^2) block on MG County and ColorHist datasets. This metric is computed as the time of a J divided by the time of O(n^2) block for the given ε. We tested J1–J4 and several other datasets as well, and we will summarize the results below.

Figures 44 and 45 demonstrate that J2 and J4 cannot outperform O(n^2) block on these datasets. J2 have not performed will in our tests on many other datasets as well. In

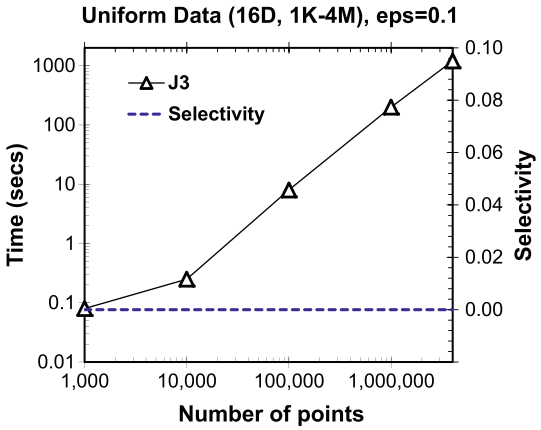

Fig. 46 "Scaling to 4M" plot in J3. Selectivity stays at zero

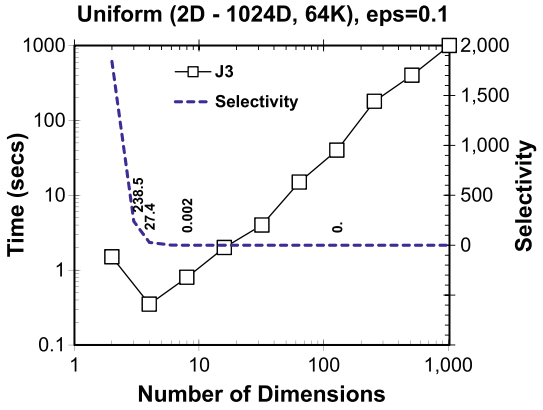

Fig. 47 "Time versus d" plot in J3. Here, $s_A = 0$ for d[9, 1024]

fact, it could not outperform O(n^2) block on all but one dataset. This was quite surprising since J2 is a very famous technique.

Figure 44 shows that J1 is better than O(n^2) block only for lower selectivity cases. Interestingly, J1 is a technique that makes more sense for higher selectivity cases. Figure 45 demonstrates that J3 outperforms O(n^2) block only for lower selectivity cases and by a factor of $\simeq 2$ at most. Recall that O(n^2) block could be further optimized by a factor of $\simeq 2$ for the self-join case, in which case the advantage of J1 and J3 should be less than what is currently shown.

Experiment 15 *(Zero-Selectivity Tests)* While the CSJ authors draw some of their conclusions from very large selectivity cases, some authors go to the other extreme and draw their conclusions from plots where selectivity stays at the zero level everywhere in a plot or where it is zero in very large portions of their plots. In fact, our own group has almost made this mistake in the past for uniform data while working on [14], but we have managed to avoid it in the end. Analyzing the selectivity should help prevent this type of error.

For instance, Fig. 46 demonstrates a plot taken from J3 paper, but with the actual selectivity values added. We can see that the selectivity stays at zero in the entire plot. From Fig. 17 from Sect. 5 we know that to get nonzero selectivity even for 10M point 16D data the value of ε should have been set to at least $\simeq 0.15$ whereas in this experiment it was set to only 0.10 by the J3 authors.

Figure 47 is another plot from the J3 paper, except for we added the actual selectivity values. The figure tests the performance of J3 as the dimensionality d increases from 2 to 1,024. It draws attention because of such a grand dimensionality used. We can see that the selectivity quickly plunges to zero with the increase of d, and the plot is not very interesting already for $d \geq 9$—the result set contains no pairs except for the trivial (a, a) pairs for each $a \in A$. From Fig. 18 from Sect. 5, we know that ε should have been set to higher values.

7 Conclusion

In this paper, we have proposed Super-EGO ε-join algorithm. We have demonstrated that it performs well compared to several prominent state of the art techniques on a variety of real and synthetic datasets. We have highlighted the importance of the selectivity factor in comparing various join algorithms. We strongly encourage all developers of ε-join techniques to report selectivity in their experiments and to compare to the O(n^2) block quadratic baseline introduced in this article. As future work we plan to look into different ε-join methods that can apply to high-dimensional cases where reasonable values of ε are expected to be larger than 1.

References

1. Andoni, A., Indyk, P.: Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In FOCS, (2006)
2. Böhm, C., Braunmüller, B., Krebs, F., Kriegel, H.-P.: Epsilon grid order: an algorithm for the similarity join on massive high-dimensional data. In SIGMOD, (2001)
3. Böhm, C., Kriegel, H.-P.: A cost model and index architecture for the similarity join. In ICDE, (2001)
4. Brinkhoff, T., Kriegel, H.-P., Seeger, B:. Efficient processing of spatial joins using R-trees, In SIGMOD (1993)
5. Bryan, B., Eberhardt, F., Faloutsos, C.: Compact similarity joins, In ICDE (2008)
6. Casey, S.D.: How to determine the effectiveness of hyper-threading technology with an application. Intel Technol. J. 6(1), (2009)
7. Cheema, M.A., Lin, X., Wang, H., Wang, J., Zhang, W.: A unified approach for computing top-k pairs in multidimensional space, In ICDE, pp. 1031–1042 (2011)
8. Chen, Z.S., Kalashnikov, D.V., Mehrotra, S.: Exploiting context analysis for combining multiple entity resolution systems, In SIGMOD (2009)
9. Corral, A., Manolopoulos, Y., Theodoridis, Y., Vassilakopoulos, M.: Closest pair queries in spatial databases, In SIGMOD (2000)
10. Dittrich, J.-P., Seeger, B.: Gess: a scalable similarity-join algorithm for mining large data sets in high dimensional spaces. In KDD, (2001)
11. Elmasri, R., Navathe, S.B.: Fundamentals of Database Systems, 3rd edn. Addison-Wesley, Longman (2000)

12. Hjaltason, G.R., Samet, H.: Incremental distance join algorithms for spatial databases. In SIGMOD, (1998)
13. Jolliffe, I.: Principal component analysis. Encyclopedia of Statistics in, Behavioral Science, (2005)
14. Kalashnikov, D., Prabhakar, S.: Similarity join for low- and high-dimensional data, pp. 26–28. In DASFAA, Mar (2003)
15. Kalashnikov, D.V., Mehrotra, S.: Domain-independent data cleaning via analysis of entity-relationship graph. ACM Trans. Database Syst. (ACM TODS) **31**(2), 716–767 (2006)
16. Kalashnikov, D.V., Prabhakar, S.: Fast similarity join for multi-dimensional data. Inf. Syst. J. **32**(1), 160–177 (2007)
17. Koudas, N., Sevcik, K.C.: High dimensional similarity joins: algorithms and performance evaluation. In ICDE, (1998)
18. Lieberman, M.D., Sankaranarayanan, J., Samet, H.: A fast similarity join algorithm using graphics processing units, In ICDE (2008)
19. Lo, M.-L., Ravishankar, C.V.: Spatial hash-joins. In SIGMOD, (1996)
20. Nuray-Turan, R., Kalashnikov, D.V., Mehrotra, S., Yu, Y.: Attribute and object selection queries on objects with probabilistic attributes. ACM Trans. Database Syst. (ACM TODS), 37(1), Feb. (2012)
21. Patel, J.M., DeWitt, D.J.: Partition based spatial-merge join. In SIGMOD, (1996)
22. Schneider, D.A., DeWitt, D.J.: A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In SIGMOD, (1989)
23. Shafer, J.C., Agrawal, R.: Parallel algorithms for high-dimensional similarity joins for data mining applications. In VLDB, (1997)
24. Shim, K., Srikant, R., Agrawal, R.: High-dimensional similarity joins, In ICDE (1997)
25. Tan, P.-N., Steinbach, M., Kumar, V.: Introduction to Data Mining. Addison-Wesley Longman Publishing Co., Inc., Boston (2005)
26. Tao, Y., Yi, K., Sheng, C., Kalnis, P.: Efficient and accurate nearest neighbor and closest pair search in high-dimensional space. ACM Trans. Database Syst., 35(3), (2010)
27. Vernica, R., Carey, M.J., Li, C.: Efficient parallel set-similarity joins using mapreduce, In SIGMOD (2010)