Epsilon Grid Order: An Algorithm for the Similarity Join on Massive High-Dimensional Data

Christian Böhm, Bernhard Braunmüller, Florian Krebs, and Hans-Peter Kriegel
Institute for Computer Science, University of Munich
Oettingenstr. 67, D-80538 München, Germany
Phone: +49-89-2178-2191, Fax: +49-89-2178-2192
{boehm,braunmue,krebs,kriegel}@dbs.informatik.uni-muenchen.de

ABSTRACT

The similarity join is an important database primitive which has been successfully applied to speed up applications such as similarity search, data analysis and data mining. The similarity join combines two point sets of a multidimensional vector space such that the result contains all point pairs where the distance does not exceed a parameter ε. In this paper, we propose the Epsilon Grid Order, a new algorithm for determining the similarity join of very large data sets. Our solution is based on a particular sort order of the data points, which is obtained by laying an equi-distant grid with cell length ε over the data space and comparing the grid cells lexicographically. A typical problem of grid-based approaches such as MSJ or the ε-kdB-tree is that large portions of the data sets must be held simultaneously in main memory. Therefore, these approaches do not scale to large data sets. Our technique avoids this problem by an external sorting algorithm and a particular scheduling strategy during the join phase. In the experimental evaluation, a substantial improvement over competitive techniques is shown.

Keywords

Similarity join, high-dimensional space, data mining, knowledge discovery, similarity search, feature transformation.

1. INTRODUCTION

Very large sets of multidimensional vector data have become widespread to support modern applications such as CAD [Jag 91], multimedia [FBFH 94], molecular biology [KS 98b], medical imaging [KSF+ 96], and time series analysis [AFS 93]. In such applications, complex objects are stored in databases. To facilitate the search by similarity, multidimensional feature vectors are extracted from the original objects and organized in multidimensional access methods. The particular property of this *feature transformation* is that the Euclidean distance between two feature vectors corresponds to the (dis-) similarity of the original objects from the underlying application. Therefore, a similarity search can be translated into a neighborhood query in the feature space [FBFH 94].

If a user is not only interested in the properties of single data objects but also in the properties of the data set as a whole he or she is supposed to run data mining algorithms on the set of feature vectors. Data mining is the process of extracting implicit knowledge from the data set which is previously unknown and potentially useful. Standard tasks of data mining are clustering [GRS 98], i.e. finding groups of objects such that the intra-group similarity is maximized and the inter-group similarity is minimized, outlier detection [KN 98], or the determination of association rules [KH 95]. Considering these standard tasks, we can observe that many of the state-of-the-art algorithms require to process all pairs of points which have a distance not exceeding a user-given parameter ε. This operation of generating all pairs is in essence a *similarity join*: Two d-dimensional data sets $S_1 = \{p_1, \ldots, p_n\}$ and $S_2 = \{q_1, \ldots, q_m\}$ are combined such that the result set rs contains all point pairs where the distance does not exceed an user-given parameter ε, i.e. $rs = \{(p_i, q_j) \mid \|p_i - q_j\| \leq \varepsilon\}$. As a consequence, many data mining algorithms can be directly performed on top of a similarity join as proposed in [BBBK 00].

A typical example of such an algorithm is the clustering algorithm DBSCAN [SEKX 98]. This algorithm defines a point p of the database to be a *core point* with respect to the user-given parameters ε and *min_pts* if at least a number *min_pts* of the points in the database have a distance of no more than ε from p. To compute the overall cluster structure, the algorithm transitively collects all core points which have a distance not exceeding ε from each other. The original definition of the algorithm performs a range query with the radius ε for each point stored in the database. Recently, it was shown that each of the two subtasks, core point determination and cluster collection, can be performed equivalently (i.e. yielding exactly the same result) by a single run of a *similarity join* [BBBK 00]. This transformation allows great

ACM SIGMOD 2001 May 21-24, Santa Barbara, California USA

performance improvements (up to 54 times faster) using standard join algorithms. There are numerous other algorithms for knowledge discovery in databases which can be performed on top of the similarity join, for instance the outlier detection algorithm RT [KN 98], nearest neighbor clustering [HT 93], single-link clustering [Sib 73], the hierarchical cluster analysis method OPTICS [ABKS 99], proximity analysis [KN 96], spatial association rules [KH 95].

Due to the high impact of the similarity join operation, a considerable number of different algorithms to evaluate the similarity join have been proposed. However, there is no technique which efficiently scales to *very large data sets*, i.e. data sets in the order of 1 GB. This is the main focus of our paper. We propose a similarity join algorithm, denoted as *epsilon grid order*, which is based on a particular sorting order of the data points. Our algorithm applies an external sorting algorithm combined with a sophisticated scheduling strategy which allows our technique to operate with a limited cache buffer. Consequently, our algorithm exhibits a high scalability.

The remainder of our paper is organized as follows: In section 2, we review the most relevant similarity join algorithms. Then, we introduce and discuss our new approach, the epsilon grid order, in section 3. In section 4, we propose several optimization techniques which further enhance our basic similarity join algorithm. The experimental evaluation of our approach is presented in section 5 and section 6 concludes the paper.

2. RELATED WORK

In the relational data model a join means to combine the tuples of two relations R and S into pairs if a *join predicate* is fulfilled. In multidimensional databases, R and S contain points (feature vectors) rather than ordinary tuples. In a *similarity join*, the join predicate is similarity, i.e. the Euclidean distance between two feature vectors stored in R and S must not exceed a threshold value ε in order to appear in the result set of the join. If R and S are actually the same point set, the join is called a *self-join*.

2.1 Join Algorithms Using R-trees

Most related work on join processing using multidimensional index structures is based on the *spatial join*. The spatial join operation is defined for 2-dimensional polygon databases where the join predicate typically is the intersection between two objects. This kind of join predicate is prevalent in map overlay applications. We adapt the relevant algorithms to allow distance based predicates for multidimensional point databases instead of the intersection of polygons.

The most common technique is the R-tree Spatial Join (RSJ) [BKS 93] which processes R-tree like index structures built on R and S. RSJ is based on the lower bounding property which means that the distance between two points is never smaller than the distance between the regions of the two pages in which the points are stored. The RSJ algorithm traverses the indexes of R and S synchronously. When a pair of directory pages (P_R,P_S) is under consideration, the algorithm forms all pairs of the child pages of P_R and P_S having distances of at most ε. For these pairs of child pages, the algorithm is called recursively, i.e. the corresponding indexes are traversed in a depth-first order.

Various optimizations of RSJ have been proposed. Huan, Jing and Rundensteiner propose the BFRJ-algorithm [HJR 97] which traverses the indexes according to a breadth-first strategy. At each level, BFRJ creates an intermediate join index and deploys global optimization strategies (e.g. ordering the pages by a space-filling curve such as the Z-order) to improve the join computation at the subsequent level. Improved cache management leads to 50% speed-up factors. Brinkhoff, Kriegel and Seeger adapted RSJ for join processing on parallel computers using shared virtual memory [BKS 96]. Their technique improves both CPU time and I/O time.

Recently, index based similarity join methods have been analyzed from a theoretical point of view. [BK 01] proposes a cost model based on the concept of the Minkowski sum [BBKK 97] which can be used for optimizations such as page size optimization. The analysis reveals a serious optimization conflict between CPU and I/O. While the CPU requires fine-grained partitioning with page capacities of only a few points per page, large block sizes of up to 1 MB are necessary for efficient I/O operations. Optimizing for CPU deteriorates the I/O performance and vice versa, and even compromises are hard to achieve, because both optima are too different. The consequence is that an index architecture is necessary which allows a separate optimization of CPU and I/O operations. Therefore, the authors propose the *Multipage Index* (*MuX*), a complex index structure with large pages (optimized for I/O) which accommodate a secondary search structure (which is optimized for maximum CPU efficiency). It is shown that the resulting index yields an I/O performance which is similar (equal, up to a small additional overhead by the more complex structure) to the I/O optimized R-tree similarity join and a CPU performance which is close to the CPU optimized R-tree similarity join.

2.2 Join Algorithms without Index

If no multidimensional index is available, it is possible to construct the index on the fly before starting the join algorithm. Usually, the dynamic index construction by repeated insert operations performs poorly and cannot be amortized by performance gains during join processing. However, several techniques for bulk-loading multidimensional index structures have been proposed [KF 94, BSW 97, BBK98].

The seeded tree method [LR 94] joins two point sets provided that only one is supported by an R-tree. The partitioning of this R-tree is used for a fast construction of the second index on the fly. The spatial hash-join [LR 96, PD 96] decomposes the set R into a number of partitions which is determined according to system parameters. Sampling is applied to determine initial buckets. Each object of R is inserted into a bucket such that bucket enlargement and bucket overlap are minimized. Then, each object

of S is inserted into every bucket having a distance not greater than epsilon from the object (object replication). If each bucket fits in main memory, a single scan of the buckets is sufficient to determine all join pairs.

A join algorithm particularly suited for similarity self joins is the ε-kdB-tree [SSA 97]. The basic idea is to partition the data set perpendicularly to one selected dimension into stripes of the width ε to restrict the join to pairs of subsequent stripes. The join algorithm is based on the assumption that the database cache is large enough to hold the data points of two subsequent stripes. In this case it is possible to join the set in a single pass. To speed up the CPU operations, for each stripe a main memory data structure, the ε-kdB-tree is constructed which also partitions the data set according to the other dimensions until a defined node capacity is reached. For each dimension, the data set is partitioned at most once into stripes of width ε. Finally, a tree matching algorithm is applied which is restricted to neighboring stripes.

It was pointed out in [BK 01] that the ε-kdB-tree has very restricting limitations when scaling to large data sets which are not main memory resident. Depending on several parameters such as the data distribution the algorithm needed large portions of the database simultaneously in main memory in order to be operational at all. For instance, an 8-dimensional artificial data set needed a constant ratio of 60% of the database in the cache, independent of the size of the database for the basic technique proposed in [SSA 97]. Having this limitation in mind, the authors of the ε-kdB-tree have also proposed an extension of their technique which does not perform a single database scan but reads parts of the database multiple times according to a complex scheduling pattern. Applying this extension, however, reduced the required cache size merely from 60% to 36% of the database size. Even for the real data experiments in [BK 01] some of the ε-stripes contained too many data points (e.g. 35% for meteorology data) and, therefore, the algorithm failed in the required configuration.

Koudas and Sevcik have proposed the Size Separation Spatial Join [KS 97] and the Multidimensional Spatial Join [KS 98a] which make use of space filling curves to order the points in a multidimensional space. Each point is considered as a cube with side-length ε in the multidimensional space. Each cube is assigned a value l (level) which is essentially the size of the largest cell (according to the Hilbert decomposition of the data space) that contains the point. The points are distributed over several *level-files* each of which contains the points of a level in the order of their Hilbert values. For join processing, each subpartition of a level-file must be matched against the corresponding subpartitions at the same level and each higher level file of the other data set. Basically, S^3J and MSJ have similar scalability limitations as the ε-kdB-tree. With increasing dimension, it becomes more and more likely that the cubes intersect with decomposition planes at a very high level. [BK 01] reported that an average of 46% of the DB size (e.g. for 8-dimensional artificial data) were needed simultaneously in main memory during the scan of the database.

3. THE EPSILON GRID ORDER

In this section, we propose our algorithm for the similarity join on massive high-dimensional data sets. Our algorithm is based on a particular order of the data set, the epsilon grid order, which is defined in the first part of this section. We will show that the epsilon grid order is a strict order (i.e. an order which is irreflexive, asymmetric and transitive). Then, we will prove a property of the epsilon grid order which is very important for join processing: We show that all join mates of some point p lie within an interval of the file. The *lower* and *upper limit* of the interval is determined by *subtracting* and *adding* the vector $[\varepsilon,\varepsilon,...,\varepsilon]^T$ to p, respectively. Therefore, we call the interval the ε-interval.

Our join algorithm exploits this knowledge of the ε-interval. Assuming a limited cache size, we have to distinguish two cases: The ε-interval of a point fits into the main memory or not. If the ε-interval *of each database point* fits into main memory, then a single scan of the database is sufficient for join processing. We call this kind of database traversal the *gallop mode*. If the ε-intervals of some points do not fit into the main memory, we have to scan the corresponding part of the database more than once. The database is traversed in the so-called *crabstep mode*. These two modes will be explained in section 3.2. Finally, we will show in section 3.3 how sequences of epsilon-grid ordered points can be joined efficiently with respect to CPU operations. Epsilon grid ordering yields the particular advantage that no directory structure needs to be constructed for this purpose. In contrast to index structures that manage main memory data structures such as MuX or ε-kdB-trees the full buffer size can be used to store point information; nearly no buffer capacity is wasted for management overhead.

3.1 Basic Properties of the Epsilon Grid Order

First we give a formal definition of the Epsilon Grid Order $(\cdot \overset{<}{_{ego}} \cdot)$. For this order, a regular grid[1] is laid over the data space, anchored in the origin, and with a grid distance of ε. We define a lexicographical order on the grid cells, i.e. the first dimension d_0 has the highest weight; for two grid cells having the same coordinates in d_0, the next dimension d_1 is considered, and so on. This grid cell order is induced to the points stored in the database: For a pair of two points p and q located in different grid cells, we let $p \overset{<}{_{ego}} q$ be *true* if the grid cell surrounding p is lexicographically lower than the grid cell surrounding q. Since we want to avoid explicit numbering of grid cells (which would be slightly clumsy unless we assume a previously limited data space), the following definition determines the order for the points directly, without explicitly introducing the grid cells:

1. Note that our grid is never materialized. It is neither necessary to determine nor to store grid cells of the data space. We use the grid cells merely as a concept to order the points, not as a physical storage container.

Definition 1 Epsilon Grid Order ($\cdot \leq_{ego} \cdot$)

For two vectors p, q the predicate $p \leq_{ego} q$ is *true* if (and only if) there exists a dimension d_i such that the following conditions hold:

(1) $\left\lfloor \frac{p_i}{\varepsilon} \right\rfloor < \left\lfloor \frac{q_i}{\varepsilon} \right\rfloor$

(2) $\left\lfloor \frac{p_j}{\varepsilon} \right\rfloor = \left\lfloor \frac{q_j}{\varepsilon} \right\rfloor \qquad \forall j < i$

Our first lemma proves that the epsilon grid order is, indeed, an order. We have not defined the epsilon grid order as a reflexive order due to points which are located in the same grid cell. Such points are not able to fulfill the antisymmetry property which is usually required for an order. Therefore, we have defined the epsilon grid order as an irreflexive or strict order which is required to be irreflexive, asymmetric, and transitive. There are almost no consequences from a practical point of view. For instance, the usual sorting algorithms can cope with an irreflexive order without modification. In the following lemma, we prove the three required properties, one of which (transitivity) is also exploited in lemma 2 and 3.

Lemma 1. The Epsilon Grid Order is an irreflexive order.

Proof:

Irreflexifity ($\neg\, p \leq_{ego} p$):

$p \leq_{ego} p$ cannot hold, because there is no dimension d_i for which $\lfloor p_i/\varepsilon \rfloor < \lfloor p_i/\varepsilon \rfloor$;

Asymmetry ($p \leq_{ego} q \quad \neg\, q \leq_{ego} p$):

Since $p \leq_{ego} q$ holds there exists a dimension d_i with $\lfloor p_i/\varepsilon \rfloor < \lfloor q_i/\varepsilon \rfloor$ and $\lfloor p_j/\varepsilon \rfloor = \lfloor q_j/\varepsilon \rfloor$ for all $j < i$. Therefore, we know that $\lfloor q_j/\varepsilon \rfloor = \lfloor p_j/\varepsilon \rfloor$ holds but neither $\lfloor q_i/\varepsilon \rfloor < \lfloor p_i/\varepsilon \rfloor$ nor $\lfloor q_i/\varepsilon \rfloor = \lfloor p_i/\varepsilon \rfloor$ can be true, and, therefore, $q \leq_{ego} p$ is false.

Transitivity ($p \leq_{ego} q \wedge q \leq_{ego} r \quad p \leq_{ego} r$):

Since $p \leq_{ego} q$ holds there exists a dimension d_i with $\lfloor p_i/\varepsilon \rfloor < \lfloor q_i/\varepsilon \rfloor$ and $\lfloor p_j/\varepsilon \rfloor = \lfloor q_j/\varepsilon \rfloor$ for all $j < i$. Since $q \leq_{ego} r$ holds there exists a dimension $d_{i'}$ with $\lfloor q_{i'}/\varepsilon \rfloor < \lfloor r_{i'}/\varepsilon \rfloor$ and $\lfloor q_j/\varepsilon \rfloor = \lfloor r_j/\varepsilon \rfloor$ for all $j < i'$. Without loss of generality we assume $i < i'$ (the other cases are similar). We know that $\lfloor p_j/\varepsilon \rfloor = \lfloor q_j/\varepsilon \rfloor = \lfloor r_j/\varepsilon \rfloor$ for all $j < i$ and that $\lfloor p_i/\varepsilon \rfloor < \lfloor q_i/\varepsilon \rfloor = \lfloor r_i/\varepsilon \rfloor$, and, therefore, $p \leq_{ego} r$.

□

In the next two lemmata, we show that our join algorithm needs not to consider any point as a join mate of some point p which is less (according to the epsilon grid order) than the point $p - [\varepsilon,\varepsilon,...,\varepsilon]^T$ or greater than the point $p + [\varepsilon,\varepsilon,...,\varepsilon]^T$. We note without a formal proof that these bounds are in general much tighter than the bounds of the ε-kdB-tree join algorithm: While the ε-kdB-tree needs two contiguous stripes of grid cells simultaneously in the main memory, our algorithm needs only one stripe plus one additional grid cell for a similarity self join.

Lemma 2. A point q with $q \leq_{ego} p - [\varepsilon,\varepsilon,...,\varepsilon]^T$ cannot be a join mate of p or of any point p' which is **not** $p' \leq_{ego} p$.

Proof:

Following definition 1, there exists a dimension d_i such that

$$\left\lfloor \frac{q_i}{\varepsilon} \right\rfloor < \left\lfloor \frac{p_i - \varepsilon}{\varepsilon} \right\rfloor$$

The monotonicity[1] of the floor function insures that $q_i < p_i - \varepsilon$. Because both ε and $(p_i - q_i)$ are positive we can rewrite this as $(p_i - q_i)^2 > \varepsilon^2$. This specific square $(p_i - q_i)^2$ for some i cannot be smaller than the sum of all squares, which corresponds to the distance between p and q:

$$\varepsilon^2 < (p_i - q_i)^2 \leq \sum_{0 \leq j < d} (p_j - q_j)^2 = \|p - q\|^2$$

Due to the transitivity of $(\cdot \leq_{ego} \cdot)$, there exists also a dimension $d_{i'}$ such that $q_{i'} < p'_{i'} - \varepsilon$. Therefore, also $\|p' - q\|^2 > \varepsilon^2$ is valid. □

Lemma 3. A point q with $p + [\varepsilon,\varepsilon,...,\varepsilon]^T \leq_{ego} q$ cannot be a join mate of p or of any point p' which is **not** $p \leq_{ego} p'$.

Proof. Analogous to lemma 2.

3.2 I/O Scheduling Using the ε Grid Order

In the previous section we have shown that our join algorithm must consider all points between $p - [\varepsilon,\varepsilon,...,\varepsilon]^T$ and $p + [\varepsilon,\varepsilon,...,\varepsilon]^T$ to find the join mates of p. In this section we construct an algorithm which schedules the disk I/O operations for a similarity self join on a file of points which is sorted according to the epsilon grid order.

In our algorithm, we want to allow for *unbuffered* I/O operations on raw devices. Therefore, we assume that the block size for the I/O units is a multiple of some hardware given system constant. Generally, an I/O unit does not contain a whole number of data point records. Instead, an I/O unit is allowed to store *fragments* of point records at the beginning and at the end. Our join algorithm solves the corresponding problems by storing the fragments in separate variables. The number of points contained in an I/O unit is to some extent system given. Due to fragmentation, the number of point records per I/O unit my vary by ±1. In general, the points in an I/O unit are not perfectly aligned to rows and columns of the grid, as in the 2-dimensional example depicted in figure 1.

Figure 2 shows which pairs of I/O units must be considered for join processing. Each entry in the matrix stands for one pair of I/O units (taken from the example in figure 1), for instance, the upper left corner for the pair (1,1), i.e. the self join of "I/O-Unit 1". For the self join operation, our algorithm needs not to consider the lower left triangular matrix due to the symmetry of the pairs. The pair (x,y) is equivalent to the pair (y,x), and, therefore, the lower left half is canceled in the figure. A large, but less regular part in the upper right corner is also cancelled. The

1. $\lfloor a \rfloor < \lfloor b \rfloor$ can only be valid if also $a < b$.

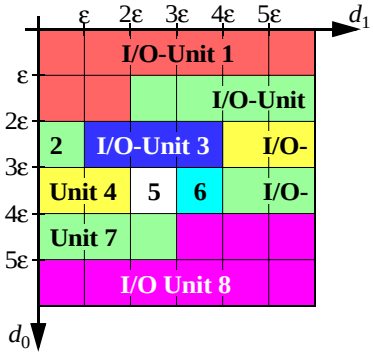

Figure 1. I/O Units in the Data Space

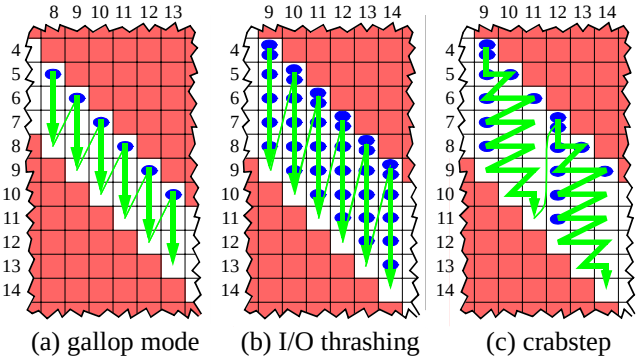

Figure 3. Scheduling Modes

corresponding pairs, for instance (1,4), are excluded from processing, because the complete I/O-Unit 1 is out of the ε-interval of I/O-Unit 4 (and vice versa, due to the symmetry of $\cdot \leqq_{ego} \cdot$).

In figure 2, a small area of pairs of I/O-Units remains (starting at the diagonal) which must be scheduled efficiently. We indicate one of the most obvious scheduling methods, column-by-column, by arrows in our running example. We start with the pair (1,1), proceed to (1,2), then (2,2), (1,3), and so on. Additionally, we mark the disk accesses caused by the schedule assuming main memory buffers for up to 3 I/O-Units which are replaced using a LRU strategy.

Our column-by-column scheduling method, which we call the gallop mode, is very efficient (even optimal, because each I/O unit is accessed only once) until the 6th column is reached. Since 4 I/O-Units which are required for processing the 6th column do not fit into main memory our scheduling turns from best case to worst case: For each scheduled pair an I/O-Unit must be loaded into main memory.

We avoid this *I/O thrashing effect* by switching into a different mode of scheduling, the *crabstep mode*. Since the ε-interval does not fit into main memory, obviously, we have to read some I/O units more than once. For those relational joins which have to form all possible pairs of I/O units or at least many of them (e.g. SELECT * FROM *A*,*B* WHERE *A.a*≠*B.b*) it is well known

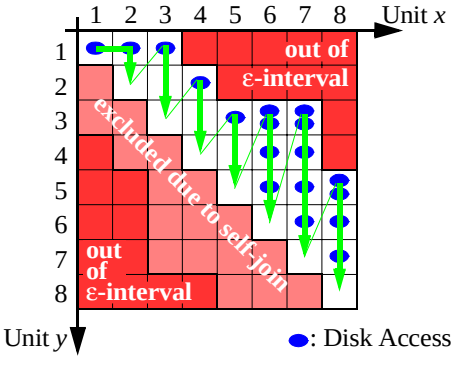

Figure 2. I/O Units in the Schedule

that the strategy of *outer loop buffering* is optimal. We adopt this strategy for the epsilon grid order where we do not have to form all possible pairs of I/O units, but only those in a common ε-interval. Our algorithm reserves in this mode only the main memory buffer for one I/O unit for the inner loop. Most of the buffer space is reserved for the outer loop, and the next I/O units from the outer loop are pinned in the buffer. The inner loop iterates over all I/O units which are in the ε-interval of any of the pinned pages. In figure 3, the two scheduling modes are visualized, assuming buffer space for up to 4 I/O units. Figure 3a shows the gallop mode where enough buffer space is available. Here, 6 disk accesses are enough to form 24 page pairs. Figure 3b shows the case where the gallop mode leads to I/O thrashing (36 disk accesses for 36 page pairs). In contrast, the crabstep mode depicted in figure 3c requires 16 disk accesses for 36 page pairs. The corresponding scheduling algorithm is shown in figure 4. Note that for a clear presentation the algorithm is simplified.

In the main loop of the algorithm, first the buffers are determined which can be discarded according to the ε-interval (code between marks [1] and [2]). If free buffers are available after this cleanup phase, we load the next I/O unit according to the strategy of the gallop mode and join the new unit immediately with the I/O units in the buffers (between marks [2] and [3]). If no buffer is free, we have to switch into the crabstep mode. In its first phase (between [3] and [4]) we discard all buffers up to one and fill them with new I/O units (which are immediately joined among each other). These new units are pinned in the cache. In the second phase (from mark [4] to the end), we iterate over the discarded I/O units, reload them, and join them with the pinned units.

3.3 Joining Two I/O-Units

It is not optimal to process a pair of I/O units by direct comparisons between the points stored in the I/O units. Instead, our algorithm partitions the point set stored in each I/O unit into smaller subsets. In contrast to other partitioning approaches without preconstructed index, where partitioning requires multiple sorting of the subset according to different dimensions or the explicit construction of a space-consuming main-memory search structure, our approach exploits the epsilon grid order of the subsets

stored on the I/O units. Therefore, both sorting of the data set during the join phase as well as the explicit construction of a search structure can be avoided. Our algorithm for joining two I/O units (two sequences of epsilon-grid-ordered points) follows the divide and conquer paradigm, i.e. the algorithm divides one of the sequences in two subsequences of approximately the same number of points and performs a recursive self-call for each of the subsequences unless a minimum sequence capacity is reached or the pair of sequences does not join (distance exceeds ε). For the purpose of excluding pairs of such sequences, we introduce a concept called *inactive dimensions* of a sequence. The intuitive idea is as follows: In general, a sequence of epsilon-grid-ordered points subsumes several different grid cells. If the sequence is short, however, it is likely that all these grid cells have the same position in the dimension d_0 of highest weight. If so, with decreasing probability it is also likely that the cells also share the same position at the second and following dimensions. The leading dimensions which are common, are called the *inactive dimensions*. The name *inactive dimensions* is borrowed from the indexing domain [LJF 95] where an inactive dimension also denotes a value which is common to all items stored in a subtree.

```
algorithm ScheduleIOunits ()
    Load (0) ; JoinBuffer (0,0) ;
    i := 1 ;
    while i < NumberIOunits do
1       foreach b ∈ Buffers \ LastBuffer do
            if b.LastPoint+[ε,ε,...,ε] ≤ego LastBuffer.LastPoint
                then MakeBufferEmpty (b) ;
        if EmptyBufferAvailable then
2           (* Gallop Mode *)
            Load (i) ; i := i + 1 ;
            foreach b ∈ Buffers do
                JoinBuffer (b, LastBuffer) ;
        else
3           (* Crabstep Mode *)
            n := FirstBuffer.IOunitNumber ;
            m := i ;
            foreach b ∈ Buffers \ LastBuffer do
                MakeBufferEmpty (b) ;
                LoadAndPin (i) ; i := i + 1 ;
                foreach c ∈ PinnedBuffers do
                    JoinBuffer (b,c) ;
4           for j := n to m – 1 do
                Load (j) ;
                foreach b ∈ PinnedBuffers do
                    JoinBuffer (b, LastBuffer) ;
            UnpinAllBuffers () ;
end ;
```

Figure 4. Scheduling Algorithm

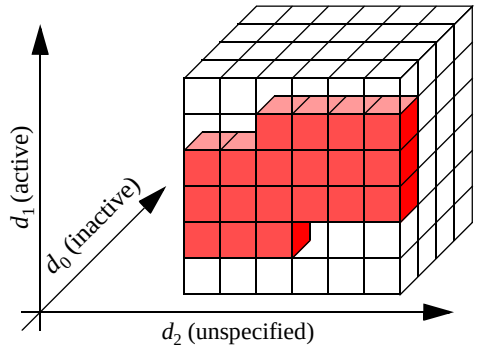

Figure 5. The active dimension of a sequence

Definition 2 (active, inactive and unspecified dimension):

For a sequence $p_1,p_2,...,p_k$ of k points which are epsilon-grid-ordered (i.e. $p_1 \leq_{ego} p_2 \leq_{ego} ... \leq_{ego} p_k$) a dimension d_i is *active* if and only if the following two conditions hold:

(1) $\left\lfloor \frac{p_{1,i}}{\varepsilon} \right\rfloor < \left\lfloor \frac{p_{k,i}}{\varepsilon} \right\rfloor$

(2) $\left\lfloor \frac{p_{1,j}}{\varepsilon} \right\rfloor = \left\lfloor \frac{p_{k,j}}{\varepsilon} \right\rfloor \qquad \forall j < i$

If an active dimension exists, all dimensions d_j with $j < i$ are called *inactive* dimensions. If no active dimension exists, all dimensions are called inactive. Dimensions which are neither active nor inactive (i.e. d_l with $i < l < d$) are *unspecified*.

The intuitive meaning of definition 2 is: The active dimension of a sequence is the first dimension where the points are extended over more than one grid cell length (if any exists). Due to the properties of the order relation, this can be decided according to the first point p_1 and the last point p_k of the sequence. Dimension d_i is the first dimension where p_1 and p_k are different after dividing and rounding.

Figure 5 shows for a 3-dimensional data space an example sequence (shaded area) where d_1 is the active dimension. The particular property of the inactive dimensions is that they can be used very effectively to determine whether two sequences $P = \langle p_1,p_2,...,p_k$ and $Q = \langle q_1,q_2,...,q_m$ of epsilon-grid-ordered points have to be joined. They need *not* be joined if for at least one of the common inactive dimensions the distance between the cells exceeds ε. Formally: If $\exists\, d_j$ such that d_j is inactive in P and d_j is inactive in Q and

$$\left| \left\lfloor \frac{p_{1,j}}{\varepsilon} \right\rfloor - \left\lfloor \frac{q_{1,j}}{\varepsilon} \right\rfloor \right| \geq 2 .$$

Active and unspecified dimensions are not used for excluding a sequence from being join mate. Figure 6 shows our recursive algorithm for the join of two sequences. It has two terminating cases: (1) the rule discussed above applies and (2) both sequences are short enough. The cases where only one sequence has more than *minlen* points are straightforward and left out in figure 6.

4. OPTIMIZATION POTENTIAL

In this section, we illustrate some of the optimization potential which is inherent to our new technique. Due to the space restrictions, we can only demonstrate two optimization concepts that integrate particularly nicely into our new technique. Further optimization techniques which are subject to future research are modifications of the sort order of the relation $\cdot \leq_{ego} \cdot$ and optimization strategies in the recursion scheme of the algorithm *join_sequences*().

4.1 Separate Optimization of I/O and CPU

It has been pointed out in [BK 01] that, for index-based processing of similarity joins, it is necessary to decouple the blocksize optimization for I/O and CPU. Therefore, a complex index structure has been proposed which utilizes large primary pages for I/O processing. These primary pages accommodate a number of secondary pages the capacity of which is much smaller and optimized for maximum CPU performance.

For our technique, the Epsilon Grid Order, a separate optimization of the size of the sequences is equally beneficial as in index based join processing. As the algorithm is based on sequences of points, ordered by a particular relation, we need no complex structure for the separate optimization. Our algorithm simply uses larger sequences for I/O processing. The length of these sequences can be optimized such that disk contention is minimized. Later, the algorithm *join_sequences* decomposes these large I/O units recursively into smaller subsequences. The size of these can be optimized for minimal CPU processing time.

In contrast to approaches that use a directory structure such as the ε-kdB-tree [SSA 97] or the Multipage Index [BK 01] the EGO-join yields almost no space overhead for this separate optimization. For CPU, the optimal size of processing units is typically below 10 points. Therefore, the Multipage Index combines these points to an *accommodated bucket* the MBR of which must be stored in the *hosting page*. The corresponding storage overhead increases when the capacity of the accommodated buckets

```
algorithm join_sequences (Sequence s, Sequence t)
    sa := s.activeDimension() ;
    ta := t.activeDimension() ;
1   for i:=0 to min {sa,ta,d–1} do
        if |⌊s.firstPoint[i]/ε⌋ –⌊t.firstPoint[i]/ε⌋| > 2 then
            return ;
2   if s.length ≤ minlen AND t.length ≤ minlen then
        simple_join (s,t) ; return ;
    if s.length ≥ minlen AND t.length ≥ minlen then
        join_sequences (s.firstHalf, t.firstHalf) ;
        join_sequences (s.firstHalf, t.secondHalf) ;
        join_sequences (s.secondHalf, t.firstHalf) ;
        join_sequences (s.secondHalf, t.secondHalf) ;
        return ; ... (* remaining cases analogously *)
```

Figure 6. Algorithm for Joining Sequences

```
function distance_below_eps (Point p, Point q): boolean
    distance_sq := 0.0 ;
    for i:=0 to d–1 do
1       j := dimension_order [i] ;
        distance_sq := distance_sq + (p [j] – q [j])^2 ;
        if distance_sq > ε^2 then return false ;
    return true ;
```

Figure 7. Algorithm for Distance Calculations

is decreased for optimization. Therefore, the optimization potential for this structure is *a priori* limited. The ε-kdB-tree also suffers from the problem of explicitly holding a hierarchical search structure in main memory.

For Epsilon Grid Ordering, no directory is explicitly constructed. Instead, the point sequences (stored as arrays) are recursively decomposed. Therefore, the only space overhead of our technique is the recursion stack which is O (log *n*). Our technique can optimize the final size of the sequences (parameter *minlen* in figure 6) without considering any limiting overhead.

4.2 Active Dimensions and Distance Calculations

In spite of the CPU optimization proposed in section 4.1 the CPU cost is dominated by the final distance calculations between candidate pairs of points. A well-known technique to avoid a considerable number of these distance calculations is to apply the triangle inequality [BEKS 00]. In our experiments, however, the triangle inequality did not yield an improvement of the Epsilon Grid Order due to the use of small, CPU optimized sequences. A more successful way is to determine the distances between two points (dimension by dimension) and testing in each step whether the distance already exceeds ε. The corresponding algorithm is depicted in figure 7.

For this step-by-step test, it is essential that the dimensions are processed in a suitable order, depending on the inactive dimensions, because some dimensions have a rather high probability of adding large values to the distance (a high *distinguishing potential*), others not. Therefore, in the line marked with (**1**) the dimensions are taken from a lookup table which is sorted according to the *distinguishing potential*. The lookup table is filled when starting the join between two minimal sequences. In the following we will show how to estimate the distinguishing potential of the dimensions for a given pair of sequences. For the analysis in this section, we assume that the points of a sequence follow a uniform (not necessarily independent) distribution in the inactive dimensions, i.e. if d_i is inactive in sequence s and the corresponding cell extension in d_i is $[x_i \cdot \varepsilon..(x_i+1)\cdot\varepsilon]$, then for the i-th coordinate p_i of each point $p \in s$ every value between $[x_i \cdot \varepsilon..(x_i+1)\cdot\varepsilon]$ has the same probability. In the following, we determine the distinguishing potential of the inactive dimensions of a pair of sequences (i.e. the dimensions which are inactive in *both* sequences).

How large the distinguishing potential of a dimension d_i is, depends on the relative position of the two sequences in the data

space (cf. figure 8). Since we consider only the inactive dimensions (in the example both dimensions d_0 and d_1), both sequences s_j and r_j have an extension of ε in all considered dimensions. Due to the grid, the sequences are in an inactive dimension d_i either perfectly *aligned* to each other or directly *neighboring*. In figure 8, s_1 and r_1 are aligned in both dimensions; s_4 and r_4 are *neighboring* in both dimensions; s_2 and r_2 are aligned in d_0, and s_3 and r_3 are aligned in d_1, neighboring in the other dimension. Other relationships are not considered, because if the sequences are neither aligned nor neighboring, they are excluded from processing, as described in section 3.3.

A single, aligned dimension has no distinguishing power at all, because the difference between two coordinates is at most the cell length ε. It is possible that the combination of several aligned dimensions distinguishes points, but not very likely. In contrast, a dimension where the two sequences are *neighboring* has a high distinguishing power. Under the above mentioned assumptions the distinguishing power can be determined as follows, according to the sequences s_2 and r_2 in figure 8 for which we determine the distinguishing power of d_1: A point on the left boundary of s_2 cannot have any join mate on r_2 (exclusion probability 1). For points on the right boundary of s_1, no points on r_2 *can be excluded* by only considering d_1 (probability 0). Between these extremes, the exclusion probability (with respect to d_1) decreases linearly from 1 to 0 (e.g. 50% for a point in the middle of s_2). Integrating this linear function yields an overall exclusion probability of 50% for each neighboring dimension.

The distinguishing power of unspecified and active dimensions is relatively difficult to assess. It depends on the ratio between ε and the extension of the data space in the corresponding dimension and on the data distribution. Our join method generally does not require knowledge about the data space or the data distribution. Determining these parameters just for the optimization of this section would not pay off. According to our experience, the distinguishing power of unspecified dimensions is in most cases below 50% (i.e. worse than that of neighboring inactive dimensions), but also clearly better than 0 (aligned inactive dimensions).Our lookup table is filled in the following order:

- First all neighboring inactive dimensions,
- then the unspecified dimensions,
- next the active dimension(s) of the two sequences,
- and, finally, the aligned inactive dimensions.

This order reveals decreasing distinguishing powers of the dimensions and leads to an exclusion of point pairs as early as possible in the algorithm of figure 7.

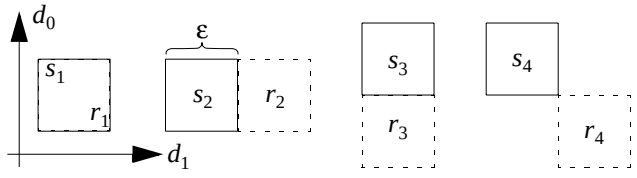

Figure 8. Distinguishing Potential of the Dimensions

5. EXPERIMENTAL EVALUATION

In order to show the benefits of our technique we implemented the EGO-algorithm and performed an extensive experimental evaluation using database sizes of well beyond 1 GB. For comparison, we applied the original source code of the Multipage Index Join [BK 01] and a similarity join algorithm based on the R-tree spatial join (RSJ) algorithm [BKS 93]. The latter join algorithm, *RSJ with Z-ordering optimization*, employs a page scheduling strategy based on Z-ordering and will be denoted as *Z-Order-RSJ*. It is very similar to the Breadth-First-R-tree-Join (BFRJ) proposed in [HJR 97]. The values for the well known nested loop join with its quadratic complexity were merely calculated and should give a reference for comparison. All algorithms were allowed to use the same amount of buffer memory (10% of the database size).

For our new technique, EGO, we considered both CPU cost as well as I/O cost, including the sorting phase which was implemented as a mergesort algorithm on secondary storage. As in figure 4 shown, our algorithm switches between the gallop and the crabstep mode on demand.

For the index based techniques (Z-Order-RSJ and MuX-Join) we assumed that indexes are already preconstructed. To be on the conservative side, we did not take the index construction cost of our competitors into account.

All our experiments were carried out under Windows NT4.0 on Fujitsu-Siemens Celsius 400 machines equipped with a Pentium III 700 MHz processor and 256 MB main memory (128 MB available for the cache). The installed disk device was a Seagate ST310212A with a sustained transfer rate of about 9 MB/s and an average read access time of 8.9 ms with an average latency time of 5.6 ms.

We used synthetic as well as real data. Our 8-dimensional synthetic data sets consisted of up to 40,000,000 uniformly distributed points in the unit hypercube (i.e. a database size of 1.2 GB). Our real-world data set is a CAD database with 16-dimensional feature vectors extracted from geometrical parts and variants thereof.

The Euclidean distance was used for the similarity join. We determined the distance parameters ε for each data set such that they are suitable for clustering following the selection criteria proposed in [SEKX 98].

Figure 10 shows our experiments using uniformly distributed 8-dimensional point data. In the left diagram, the database size is varied from 0.5 million to 40 million points while on the right side results are compared for varying values of the ε parameter. The largest database was about 1.2 GB. For this size (as well as for the 20 million points) only the results for EGO could be obtained in reasonable time. The nested loop join has the worst performance off all the compared techniques. The Z-Order-RSJ outperforms the nested loop join by factors ranging from 30 to 140 while the MuX-Join still is at least two times faster than Z-Order RSJ. By far the best performance is obtained with our new EGO technique. EGO outperforms the best of the other techniques, the

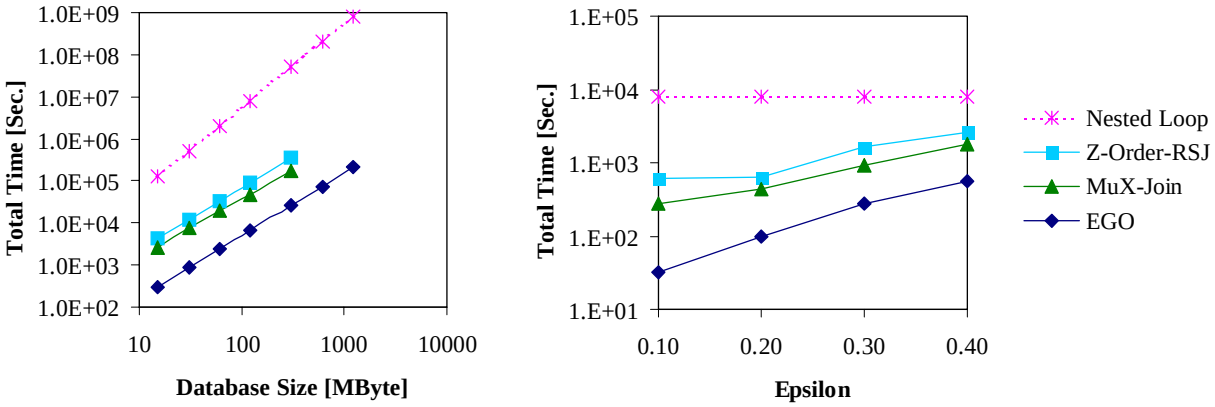

Figure 10. Experimental Results on Uniformly Distributed, 8-Dimensional Data

MuX-Join, by factors between 6 and 9, and the Z-Order-RSJ by factors between 13 and 14. The right diagram shows performance for varying distance parameter ε. Depending on its actual page boundary configuration, the Z-Order-RSJ sometimes is not as sensitive to small changes in the distance parameter as the other techniques. Again, we observe that our novel approach clearly outperforms all other techniques for all values of ε. The speedup factors were between 3.2 and 8.6 over MuX and between 4.7 and 19 over Z-Order-RSJ.

The experiments with real data are depicted in figure 9. The results for the 16-dimensional CAD data set confirm our experiments on uniform data. Again, the left diagram shows performance for varying database size while the right diagram shows performance for varying ε values. EGO was 9 times faster than the MuX-Join for the largest database size and 16 times faster than the Z-Order-RSJ. In the right diagram we can observe, that the performance of the MuX-Join and the Z-Order-RSJ converge for larger ε values while EGO still shows substantially better performance for all values of ε. The improvement factors of our technique varied between 4.0 and 10 over the Multipage Index and between 4.5 and 17 over Z-Order-RSJ.

6. CONCLUSIONS

Many different applications are based on the similarity join of very large data sets, for instance similarity search in multimedia databases, data analysis tools and data mining techniques. Unfortunately, there is no technique available which efficiently scales to very large data sets, i.e. data sets in the order of 1 GB. In this paper, we focused on this specific problem. We introduced and discussed a novel similarity join algorithm, denoted as *epsilon grid order*, which is based on a particular sorting order of the data points. This sorting order is derived by laying an equi-distant grid with cell length ε over the data space and comparing the grid cells lexicographically. We proposed to apply an external sorting algorithm combined with a sophisticated scheduling strategy which allows our technique to operate with a limited cache buffer. Additionally, we developed several optimization techniques which further enhance our method. In an experimental evaluation using data sets with sizes up to 1.2 GB we showed that our novel approach is very efficient and clearly outperforms competitive algorithms. For future work we plan a parallel version of the EGO join algorithm and the extension of our cost model for the use by the query optimizer.

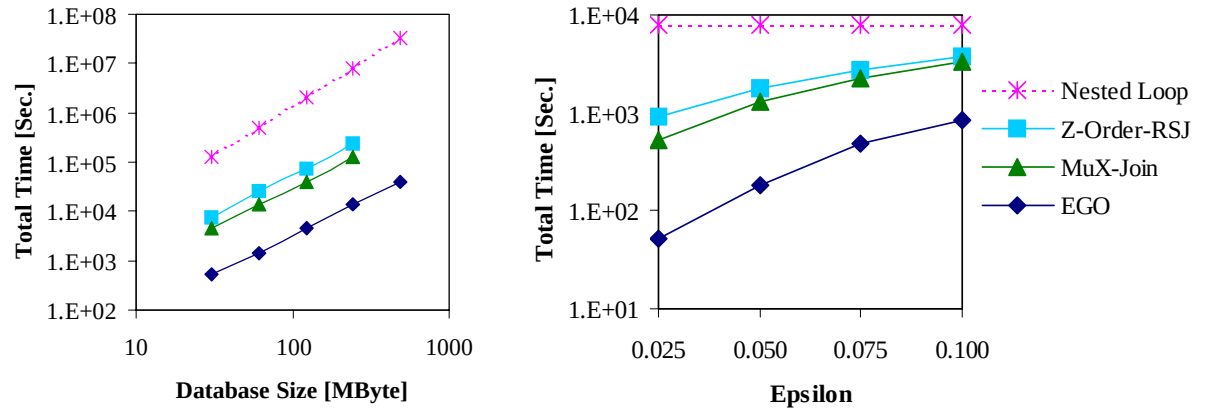

Figure 9. Experimental Results on 16-Dimensional Real Data from a CAD-Application

REFERENCES

[ABKS 99] Ankerst M., Breunig M. M., Kriegel H.-P., Sander J.: *OPTICS: Ordering Points To Identify the Clustering Structure*, ACM SIGMOD Int. Conf. on Management of Data, 1999.

[AFS 93] Agrawal R., Faloutsos C., Swami A. *Efficient similarity search in sequence databases.* Int. Conf. on Foundations of Data Organization and Algorithms, 1993.

[BBBK 00] Böhm C., Braunmüller B., Breunig M. M., Kriegel H.-P.: *Fast Clustering Based on High-Dimensional Similarity Joins*, Int. Conf. on Information Knowledge Management (CIKM), 2000.

[BBK 98] Berchtold S., Böhm C., Kriegel H.-P.: *Improving the Query Performance of High-Dimensional Index Structures Using Bulk-Load Operations*, Int. Conf. on Extending Database Technology (EDBT), 1998.

[BBKK 97] Berchtold S., Böhm C., Keim D., Kriegel H.-P.: *A Cost Model For Nearest Neighbor Search in High-Dimensional Data Space*, ACM Symposium on Principles of Database Systems (PODS), 1997.

[BEKS 00]Braunmüller B., Ester M., Kriegel H.-P., Sander J.: *Efficiently Supporting Multiple Similarity Queries for Mining in Metric Databases*, IEEE Int. Conf. on Data Engineering, 2000.

[BK 01] Böhm C., Kriegel H.-P.: *A Cost Model and Index Architecture for the Similarity Join*, IEEE Int. Conf on Data Engineering (ICDE), 2001.

[BKS 93] Brinkhoff T., Kriegel H.-P., Seeger B.: *Efficient Processing of Spatial Joins Using R-trees*, ACM SIGMOD Int. Conf. on Management of Data, 1993.

[BKS 96] Brinkhoff T., Kriegel H.-P., Seeger B.: *Parallel Processing of Spatial Joins Using R-trees*, IEEE Int. Conf. on Data Engineering (ICDE), 1996.

[BSW 97] van den Bercken J., Seeger B., Widmayer P.:*A General Approach to Bulk Loading Multidimensional Index Structures*, Int. Conf. on Very Large Databases, 1997.

[FBFH 94] Faloutsos C., Barber R., Flickner M., Hafner J., et al.: *Efficient and Effective Querying by Image Content*, Journal of Intelligent Information Systems, Vol. 3, 1994.

[GRS 98] Guha S., Rastogi R., Shim K.: *CURE: An Efficient Clustering Algorithm for Large Databases*, ACM SIGMOD Int. Conf. on Management of Data, 1998.

[HJR 97] Huang Y.-W., Jing N., Rundensteiner E. A.:*Spatial Joins Using R-trees: Breadth-First Traversal with Global Optimizations*, Int. Conf. on Very Large Databases (VLDB), 1997.

[HT 93] Hattori K., Torii Y.: *Effective algorithms for the nearest neighbor method in the clustering problem*. Pattern Recognition, Vol. 26, No. 5, 1993.

[Jag 91] Jagadish H. V.: *A Retrieval Technique for Similar Shapes*, ACM SIGMOD Int. Conf. on Management of Data, 1991.

[KF 94] Kamel I., Faloutsos C.: *Hilbert R-tree: An Improved R-tree using Fractals.* Int. Conf. on Very Large Databases, 1994.

[KH 95] Koperski K. and Han J.: *Discovery of Spatial Association Rules in Geographic Information Databases*, Int. Symp. on Large Spatial Databases (SSD), 1995.

[KN 96] Knorr E.M. and Ng R.T.: *Finding Aggregate Proximity Relationships and Commonalities in Spatial Data Mining*, IEEE Trans. on Knowledge and Data Engineering, 8(6), 1996.

[KN 98] Knorr E.M. and Ng R.T.: *Algorithms for Mining Distance-Based Outliers in Large Datasets*, Int. Conf. on Very Large Databases (VLDB), 1998.

[KS 97] Koudas N., Sevcik C.: *Size Separation Spatial Join*, ACM SIGMOD Int. Conf. on Management of Data, 1997.

[KS 98a] Koudas N., Sevcik C.: *High Dimensional Similarity Joins: Algorithms and Performance Evaluation*, IEEE Int. Conf. on Data Engineering (ICDE), Best Paper Award, 1998.

[KS 98b] Kriegel H.-P., Seidl T.: *Approximation-Based Similarity Search for 3-D Surface Segments*, GeoInformatica Journal, Kluwer Academic Publishers, 1998.

[KSF+ 96] Korn F., Sidiropoulos N., Faloutsos C., Siegel E., Protopapas Z.: *Fast Nearest Neighbor Search in Medical Image Databases*, Int. Conf. on Very Large Data Bases (VLDB), 1996.

[LJF 95] Lin K.-I., Jagadish H. V., Faloutsos C.: *The TV-Tree: An Index Structure for High-Dimensional Data*, VLDB-Journal Vol. 3, 1995.

[LR 94] Lo M.-L., Ravishankar C. V.: *Spatial Joins Using Seeded Trees*, ACM SIGMOD Int. Conf. Management of Data, 1994.

[LR 96] Lo M.-L., Ravishankar C. V.: *Spatial Hash Joins*, ACM SIGMOD Int. Conf. on Management of Data, 1996.

[PD 96] Patel J.M., DeWitt D.J., *Partition Based Spatial-Merge Join*, ACM SIGMOD Int. Conf. on Management of Data, 1996.

[SEKX 98] Sander J., Ester M., Kriegel H.-P., Xu X.: *Density-Based Clustering in Spatial Databases: The Algorithm GDBSCAN and its Applications*, Data Mining and Knowledge Discovery, Kluwer Academic Publishers, Vol. 2, No. 2, 1998.

[Sib 73] Sibson R.: *SLINK: an optimally efficient algorithm for the single-link cluster method*, The Computer Journal 16(1), 1973.

[SSA 97] Shim K., Srikant R., Agrawal R.: *High-Dimensional Similarity Joins*, Int. Conf. on Data Engineering (ICDE), 1997.