GESS: a Scalable Similarity-Join Algorithm for Mining Large Data Sets in High Dimensional Spaces*

Jens-Peter Dittrich
Department of Mathematics and Computer Science
University of Marburg
dittrich@mathematik.uni-marburg.de

Bernhard Seeger
Department of Mathematics and Computer Science
University of Marburg
seeger@mathematik.uni-marburg.de

ABSTRACT

The similarity join is an important operation for mining high-dimensional feature spaces. Given two data sets, the similarity join computes all tuples (x, y) that are within a distance ε.

One of the most efficient algorithms for processing similarity-joins is the Multidimensional-Spatial Join (MSJ) by Koudas and Sevcik. In our previous work — pursued for the two-dimensional case — we found however that MSJ has several performance shortcomings in terms of CPU and I/O cost as well as memory-requirements. Therefore, MSJ is not generally applicable to high-dimensional data.

In this paper, we propose a new algorithm named Generic External Space Sweep (GESS). GESS introduces a modest rate of data replication to reduce the number of expensive distance computations. We present a new cost-model for replication, an I/O model, and an inexpensive method for duplicate removal. The principal component of our algorithm is a highly flexible replication engine.

Our analytical model predicts a tremendous reduction of the number of expensive distance computations by several orders of magnitude in comparison to MSJ (factor 10^7). In addition, the memory requirements of GESS are shown to be lower by several orders of magnitude. Furthermore, the I/O cost of our algorithm is by factor 2 better (independent from the fact whether replication occurs or not). Our analytical results are confirmed by a large series of simulations and experiments with synthetic and real high-dimensional data sets.

1. INTRODUCTION

An efficient support of similarity queries is of utmost importance in different novel applications like multimedia [1], text mining [9] and clustering [7]. In order to support similarity queries on complex objects, each of these objects is represented by a feature vector, which can generally be viewed as a point in a multi-dimensional space. Similarity between objects is then simply defined using a distance measure between their feature vectors.

In this paper, we address the problem of supporting similarity joins on two sets of feature vectors. For a given parameter ε, a tuple (x, y) satisfies the join predicate if the distance of x and y is not greater than ε. The conventional approach to processing a similarity join is to transform it into a d-dimensional intersection join where d is the dimension of the feature vector. The intersection join considers circles with radius $\varepsilon/2$ centered at the feature vectors.

According to the availability of indexes, the algorithms for processing intersection joins can be classified into three classes: availability of indexes on both sets, on one of the sets or on none of the sets. Though the assumption of availability of an index on a spatial data set is frequently fulfilled, it is questionable whether a useful index is available on a set of feature vectors. Moreover, an index will not be available when the feature set is delivered as a result of a preprocessing step of another query operator. For these reasons, we are primarily interested in similarity joins where pre-existing indexes are not available on both feature sets.

The Multi-dimensional Spatial Join (MSJ) [15, 16], that is a multi-dimensional extension of [14], has been shown to be among the most efficient methods for processing similarity joins. The basic idea of MSJ is to partition the data into level files. However, three major shortcomings were identfied [11, 8]:

1. **CPU cost:** [11] reported that MSJ is CPU-bound during the join-phase due to a tremendous number of distance computations and therefore, MSJ is not competitive to a simple hash-based method [20].
2. **Memory Requirements:** [8] stated that the main memory required by MSJ to run efficiently (without any swapping) is very high (46% of the input-relation for typical situations). MSJ is therefore not scalable for large data sets.
3. **I/O cost:** MSJ was found to have high I/O cost [11].

In this paper, we present solutions to these problems. We propose an efficient and scalable algorithm called GESS (Generic External Space Sweep, pron.: 'Jazz') for processing d-dimensional intersection- and similarity joins. GESS overcomes all three problems of MSJ by introducing a modest

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGKDD 2001, San Francisco, California
Copyright 2001 ACM X-XXXXX-XX-X/XX/XX ...$5.00.

*This work has been supported by grant no. SE 553/2-1 from DFG.

and controlled rate of data redundancy. Since redundancy causes undesirable duplicates in the response set, we provide an efficient and simple method to eliminate them.

In short, this paper makes the following contributions:

1. We present a cost model for redundancy and the number of distance computations required for MSJ and GESS. We validate our model by results obtained from a simulation and show that GESS is clearly superior to MSJ. In particular, our analytical model shows that a modest rate of redundancy (less than 15%) is sufficient to reduce the number of distance computations by several orders of magnitude.

2. We demonstrate that replication considerably reduces the main memory requirements. The required size of memory is generally several orders of magnitude lower for GESS in comparison to MSJ. This is important for a scalable algorithm.

3. GESS provides a new I/O strategy. We present an I/O cost analysis of our algorithm and show that our approach generally requires only half of the I/O operations compared to MSJ. This improvement is independent from data replication.

4. Finally, we present results of experiments with real high-dimensional data. Our experiments are consistent with our analytical observations and show tremendous performance improvements of GESS compared to MSJ.

GESS as well as MSJ are fully integrated in a well documented library of query processing algorithms [5, 4]. GESS will be made public available in the next release.

This paper is organized as follows. Section 2 introduces our notation and discusses related work. In Section 3, we present GESS, our new method for processing d-dimensional intersection joins. We present our new cost model for redundancy and the number of distance computations in Section 4. Section 5 gives an analysis of MSJ and GESS with respect to their I/O cost. In Section 6, we present results of an experimental performance comparison using a simulation as well as an implementation.

2. PRELIMINARIES

In this section, we introduce the notation used throughout the paper. Then, we give a brief review of existing methods for processing similarity and d-dimensional intersection joins.

2.1 Notation

In the following, we assume for sake of simplicity that the feature objects are in a d-dimensional unit hypercube $U = [0,1)^d$ and that the distance measure refers to the L_p metric.

DEFINITION 1. *Let R and S be sets of d-dimensional feature vectors. Given a distance ε, the response set of the similarity join consists of all pairs (x, y) where $x = (x_1, \ldots, x_d) \in R$ and $x = (y_1, \ldots, y_d) \in S$ such that the join predicate*

$$\left(\sum_{i=1}^{d}(x_i - y_i)^p\right)^{1/p} \leq \varepsilon$$

is satisfied.

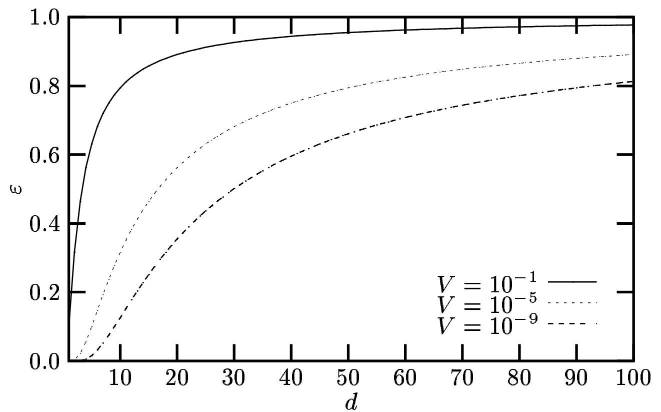


Figure 1: Length ε of a hypercube in one dimension as a function of d (total number of dimensions)

For the special case of a self join ($R = S$), the definition of a similarity join is slightly modified to exclude the trivial results (x, x) from the response set.

For an arbitrary similarity join, we employ the join based on the L_∞ metric as a filter step. This join is equivalent to an intersection join using sets of hypercubes $H(x) := H_{\varepsilon,d}(x)$ where ε refers to the length of the hypercube in dimension i, $1 \leq i \leq d$, and $x \in D$ is a feature vector. It is therefore sufficient to study intersection joins in the rest of the paper.

2.2 The Dimensional Impact

The most serious problem of processing high-dimensional intersection joins is the so-called *curse of dimensionality*, a problem that has been known in statistics for long [22]. Under the assumption that data is uniformly distributed, it seems to be difficult to design efficient join algorithms. In order to illustrate the problem, Figure 1 shows the length (ε) of the hypercubes as a function of the dimensionality (d) for different volumes (V) of the hypercubes. The results are obtained under the *uniformity and independence assumption.* For fifty dimensions, ε is almost as large as the unit interval. Fortunately, these assumptions do not hold for real data sets where strong correlations between different dimensions can be observed. Results obtained from different experiments with real data [12] have given a strong indication that the fractal dimension is an excellent measure for the *true* dimensionality of a data set. The analytical results obtained under the assumption of uniformity and independence are applicable and accurate for correlated data sets when d is simply replaced by the fractal dimension [2].

2.3 Review of Previous Work

We are mainly interested in methods for processing similarity joins that do not require the availability of pre-existing indexes. These methods can be classified into three categories:

- *Nested-loops/brute-force:* This class contains techniques like the VA-file [24] where scans are performed on a highly compressed database.

- *Hash-based methods:* These methods partition the input data into buckets and perform the join on pairs

of buckets in a recursive manner. Most of the methods [20, 17, 6] replicate data resulting in high replication rates (especially for high-dimensional joins). Since replication may cause duplicates in the response set, a post-processing of the results (e.g. sorting) is required to get rid of the duplicates.

Other methods like the ε-KDB-tree [23] avoids replication at the cost of extremely high main memory requirements. Each bucket of the ε-KDB-tree contains the feature vectors that are in a stripe of length ε (w.r.t. the first dimension). This method can only be efficient when the data of two adjacent stripes can be kept in memory. This however is not guaranteed.

- *Sort-based methods:* [19] is likely the first proposal of a high-dimensional intersection join where the data is sorted with respect to a space-filling curve. A similar approach has been pursued in [16] where the data is firstly partitioned into level-files. These methods are the starting point of our proposal and therefore, a detailed discussion of these methods follows.

2.4 Generic Framework

In this section, we first present a generic framework for the join methods presented in [19, 16]. Thereafter, we show how these methods can be viewed as special cases.

DEFINITION 2 (RECURSIVE PARTITIONING). *Let n be a positive integer and D be a set. Let $D_0, ..., D_{n-1}$ be a sequence of subsets of D satisfying the following conditions:*

- $D = \bigcup_{0 \le i < n} D_i$
- $D_i \cap D_j = \varnothing$ *for* $i \neq j$

Then, $P(D) = \{D\} \cup \bigcup_{0 \le i < n} P(D_i)$ is an n-ary recursive partitioning of D. $D_0, ..., D_{n-1}$ are called the direct subspaces of D.

Note that trees like kd-tries [19] and quadtrees [21] induce a recursive partitioning for $n = 2$ and $n = 2^d$, respectively, since each node represents a subspace of the underlying data space. The next lemma states some important properties of an n-ary recursive partitioning.

LEMMA 1. *Let D be a set and $P(D)$ be an n-ary recursive partitioning of D.*

1. *Then, the following conditions holds for $X, Y \in P(D)$:* $X \cap Y = \varnothing \quad \vee \quad X \supset Y \quad \vee \quad X \subseteq Y$
2. *Let $X \in P(D)$ and $D_0, ..., D_{n-1}$ be the direct subspaces of D. If $X \neq D$, there exists one and only one i such that* $X \subseteq D_i \quad \wedge \quad X \cap D_j = \varnothing$ *for* $j \neq i$.

DEFINITION 3. *Let D be a set and $P(D)$ be an n-ary recursive partitioning. Let $D_0, ..., D_{n-1}$ be the direct subspaces of D. Let $X \in P(D)$. Then,*

$$Code_D(X) := \begin{cases} < i \oplus Code_{D_i}(X) > & , \text{if } X \subseteq D_i \\ <> & , \text{otherwise } (X = D) \end{cases} \quad (1)$$

The length of $Code_D(X)$ is called the level of X.

```
 1  While (both sorted streams are not empty) {
 2        Choose minimal hypercube H(x)
             from the streams;
 3        Ensure prefix-property for Stack_R and Stack_S;
 4        If (H(x) is hypercube of R) {
 5              Stack_S.query(H(x));
 6              Stack_R.insert(H(x));
 7        }
 8        Else { //H(x) is hypercube of S
 9              Stack_R.query(H(x));
10              Stack_S.insert(H(x));
11        }
12  }
```

Figure 2: Orenstein's Merge Algorithm

The code defined above returns a string with digits from $\{0, ..., n-1\}$. In order to sort the elements in a recursive partitioning we employ the *lexicographical ordering* on the strings. Note that if a string s_1 is a prefix of s_2, s_1 will precede in the order.

A recursive partitioning $P(D)$ provides a skeleton where a hypercube $H(x)$ represented by its feature vector x is assigned to one or multiple disjoint subspaces of $P(D)$. The code of the hypercube refers to the ones of its assigned subspaces. The following join methods differ in their assignment strategies.

2.4.1 Review of Orenstein's Algorithm

In this subsection, we briefly review the join-algorithm proposed by Orenstein [18][19] (termed *ORE* in the following). The method is based on a binary recursive partitioning ($n = 2$), see Definition 2, where the binary code represents the so-called Z-ordering.

ORE assigns each hypercube of the input relations to disjoint subspaces of the recursive partitioning whose union entirely covers the hypercube. ORE sorts the two sets of hypercubes derived from the input relations (including the possible replicates) w.r.t. the lexicographical ordering of its binary code. After that, the relations are merged using two main-memory stacks *Stack_R* and *Stack_S*, see Figure 2. It is guaranteed that for two adjacent hypercubes in the stack, the prefix property is satisfied for their associated codes. This property is ensured in the third line of the algorithm. Therefore, only those hypercubes are joined (line 5 and 9) that have the same prefix code.

A deficiency of ORE is that the different assignment strategies examined in [19] cause substantial replication rates. This results in an increase of the problem space and hence, sorting will be very expensive. Furthermore, ORE has not addressed the problem of eliminating duplicates in the result set.

2.4.2 Review of MSJ

MSJ performs similar to ORE with two main differences: First, replication is not allowed and second, an I/O strategy based on so-called *level-files* is employed. Moreover, an n-ary recursive partitioning is used where $n = 2^d$ (quadtree-partitioning).

Let us first discuss for MSJ how a hypercube is assigned to a subspace of the recursive partitioning. Among the subspaces which cover the hypercube we choose the minimum one. This guarantees that one and only one subspace is

```
For both data sets:
        For each hypercube:
                Compute code and level l of the hypercube;
                Write hypercube into level-file l;
        For each level-file:
                Sort level-file w.r.t. the lexicographical ordering;
        Perform a synchronized linear scan through
            the level-files and
        perform Orensteins's Merge Algorithm;
```

Figure 3: The basic steps of MSJ

```
For both data sets:
        For each hypercube:
                Execute replication algorithm and
                     compute codes;
        Feed Replicates directly into sorting operator;
        Perform Orensteins's Merge Algorithm;
        For each result tuple:
                Call Reference Point Method;
```

Figure 4: The GESS-algorithm

assigned to a hypercube.

The algorithm starts by partitioning the hypercubes of the input relations into level-files according to their levels (see Figure 3). Hence, a hypercube of level l is kept in the l-th level-file. Then, the level-files are sorted w.r.t. the code of the hypercubes. Finally, the Merge algorithm of Orenstein is called.

Deficiencies of this method for high-dimensional intersection joins are that a high fraction of the input relation will be in level 0 [8]. The hypercubes in level 0, however, need to be tested against the entire input relation in a nested-loop manner. Moreover, [11] showed for two dimensions that a modest rate of replication considerably speeds up the overall execution time of MSJ.

3. GENERIC EXTERNAL SPACE SWEEP

In this section we present our scalable similarity-join algorithm called *Generic External Space Sweep (GESS)*. GESS makes use of a generic replication algorithm in order to reduce the number of expensive distance computations of feature vectors. Possible duplicates in the result set are avoided by an inexpensive removal technique. Our algorithm proceeds in three phases described in the following. Each phase is then explored in more detail in a separate subsection.

Let R and S be to sets of feature vectors. In the first phase, each vector of $R(S)$ is transformed into a hypercube that is passed to the *replication algorithm* (see Section 3.1). The replication algorithm creates codes that represent the subspaces of each hypercube. The hypercubes are then passed directly to a sorting operator which employs the lexicographical ordering (see Section 2.4) on the hypercubes. We do not partition the hypercubes physically into separate level-files but rely on the properties of the underlying sorting operator (see Section 3.2). After that, the two sorted streams are merged using Orenstein's Merge Algorithm (see Figure 2). The algorithm delivers the results to the *Reference Point Method* (see Section 3.3) that eliminates duplicates in an on-line fashion. The algorithm is summarized in Figure 4.

```
1   Predicate split_is_Allowed; //actual replication strategy

2   Procedure replicate(Hypercube H(x), SubSpace D){
3           P(D) := {D} ∪ D0 ∪ D1;
4           If( H(x) ⊆ D0) {
5                   replicate( H(x), D0 );
6           }
7           Else If( H(x) ⊆ D1) {
8                    replicate( H(x), D1 );
9           }
10          Else If ( split_is_Allowed( H(x), P(D) ) ) {
                    //Split H(x) into two replicates:
11                  replicate( H(x) ∩ D0, D0 );
12                  replicate( H(x) ∩ D1, D1 );
13          }
14          Else {
15                  return (H(x), Code_D(D));
16          }
17  }
```

Figure 5: The replication algorithm of GESS

3.1 The Replication Algorithm

For sake of simplicity, let us first assume a binary recursive partitioning ($n = 2$). Later we will then change to a quadtree partitioning which is actually used in our implementation. The replication algorithm takes a hypercube $H(x)$ and a subspace D as its input. Initially, it is called with the subspace representing the entire data space. The algorithm checks whether $H(x) \subseteq D_0 \vee H(x) \subseteq D_1$ holds. If that is the case, the algorithm is recursively called with the enclosing direct subspace. Otherwise (i.e. $H(x) \nsubseteq D_0 \wedge H(x) \nsubseteq D_1$) the algorithm calls the user-defined predicate *split_is_Allowed*. If the predicate is satisfied the hypercube $H(x)$ is split into two replicates $H(x) \cap D_0$ and $H(x) \cap D_1$. The algorithm is then invoked recursively for each replicate. If the predicate returns *false* the partitioning process stops and the code determining the actual subspace is returned. If a hypercube is replicated, multiple codes are returned by the algorithm. Figure 5 reports the algorithm in pseudo-code.

Our replication algorithm depends on a user-defined predicate *split_is_Allowed* that determines the replication strategy. The strategy is not fixed and can be changed dynamically.

Assume a quadtree-partitioning for the rest of this paper, ($P(D) = \{D\} \cup \bigcup_{0 \leq i < 2^d} P(D_i)$). We put our focus on the following important strategies:

STRATEGY 1 (MAXIMUM NUMBER OF SPLIT-LINES). *Splitting a hypercube $H(x)$ is allowed if not more than k hyperplanes are hit by $H(x)$ at the current quadtree-level.*

STRATEGY 2 (MAXIMUM SPLIT LEVEL). *Splitting a hypercube $H(x)$ is allowed if the actual level is smaller than a given bound, i.e. $l \leq msl$, where $msl \in [0, \ldots, mL]$ is the maximum split level.*

STRATEGY 3 (COMPOSITION-STRATEGY). *Splitting a hypercube $H(x)$ is allowed if Strategy 1 and Strategy 2 hold.*

3.2 Why we do not need level-files?

Important for the Merge Algorithm of Orenstein (see Figure 2) is the lexicographical ordering (see Section 2.4). A

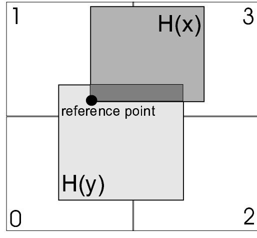


Figure 6: The Reference Point Method (RPM)

physical partitioning of data into level-files is not necessary as it is performed by MSJ. In order to make Orenstein's Merge Algorithm applicable to large data sets, we use an existing external merge-sort operator. The sorting criteria is modified to establish the lexicographical ordering on the feature vectors. Our sorting operator is based on replacement selection [13]. The final merge of the sorted streams is performed on-line, i.e., the sorted output streams are not written on disk, but merged in main memory and directly fed into the merge algorithm.

3.3 RPM: The Reference Point Method

Since GESS allows hypercubes to get replicated, we have to provide a method to eliminate possible duplicates from the result set. Let us consider the situation in Figure 6 where we assume that the hypercube $H(x)$ is replicated into the partitions 1 and 3 at level 1, $H(y)$ resides at level 0. Both replicates of $H(x)$ are matched against $H(y)$ during the join-phase. As a consequence the tuple (x, y) would be reported twice.

Instead of using standard techniques like hashing or sorting [20], we propose an inexpensive on-line method termed *Reference Point Method (RPM)*. This method neither allocates additional memory nor does it cause any additional I/O operations. Let $x \in R, y \in S$ be two feature vectors where $H(x) \cap H(y) \neq \emptyset$ holds. The basic idea of RPM is to define a reference point

$$rp := f_{rp}(H(x), H(y)) \in H(x) \cap H(y),$$

i.e., the reference point is contained in the section of $H(x)$ and $H(y)$. The Reference Point Method then works as follows:

DEFINITION 4 (RPM). *Let H be the hypercube with the highest level among $H(x)$ and $H(y)$. The result tuple (x, y) is reported by the RPM*

$$\Longleftrightarrow Code_D(H) \text{ is prefix of } Code_D(f_{rp}(H(x), H(y))).$$

Let us consider again the situation in Figure 6. $H = H(x)$ since it resides on a higher tree-level. The code of the reference point rp is $Code_D(f_{rp}(H(x), H(y))) =< 1.. >$. $H(x)$ was replicated and is represented by the codes $< 1 >$ and $< 3 >$. Since $< 1 >$ is a prefix of $< 1.. >$ the result is reported for that replicate. Code $< 3 >$ is not prefix of $< 1.. >$ and RPM will not report this result. The result is therefore reported exactly once.

THEOREM 1 (CORRECTNESS OF RPM). *The Reference Point Method reports each result of the join exactly once.*

PROOF. Let $P(D)$ be an n-ary recursive partitioning of D. Let (x, y) be a result tuple of the join where $H(x)$ and $H(y)$ are the corresponding hypercubes. The replication algorithm of GESS computes a set of disjoint subspaces, say $Rep(x) \subseteq P(D)$ and $Rep(y) \subseteq P(D)$ for each hypercube $H(x)$ and $H(y)$. The elements of $Rep(x)$ and $Rep(y)$ provide a conservative approximation for $H(x)$ and $H(y)$.

Since $\forall\ A, B \in Rep(x), A \neq B, A \cap B = \emptyset$ and $\forall\ A, B \in Rep(y), A \neq B, A \cap B = \emptyset$, the reference point can only be in one of the subspaces of $Rep(x)$ and $Rep(y)$. It follows that there exists exactly one pair $(A, B), A \in Rep(x), B \in Rep(y)$ for which

$$A \cap B \neq \emptyset \wedge rp \in A \cap B$$

holds. Since $A, B \in P(D)$, it follows that $A \subseteq B$ or $B \subseteq A$ holds. Let us assume $A \subseteq B$. Then,

$$\begin{aligned} & rp \in H(x) \cap H(y) \\ \Longleftrightarrow\ & rp \in A \cap B \\ \Longleftrightarrow\ & rp \in A \subseteq B \\ \Longleftrightarrow\ & \begin{cases} & Code_D(B) \text{ is prefix of } Code_D(A) \\ \wedge & Code_D(A) \text{ is prefix of } Code_D(rp) \end{cases} \end{aligned}$$

□

4. CPU COST ANALYSIS

In the following we consider the CPU cost of the intersection join for MSJ and GESS.

Both of the methods partition the input relations R and S into different levels using a recursive partitioning for $n = 2^d$ (quadtree partitioning). We use $R_i(S_i)$ to refer to the subset of elements of R that is assigned to level i. Due to replication of hypercubes, $\sum_{i=0}^{mL} R_i(S_i)$ will contain more elements than $R(S)$ for GESS. The CPU cost of the methods is clearly dominated by the number of distance computations required in the join phase.

LEMMA 2. *The average number of distance computations DC for processing an intersection join of R and S is given by*

$$DC = \sum_{i=0}^{mL} \left(|R_i| \cdot \left[\sum_{l=0}^{i} \frac{|S_l|}{n^l} + \frac{1}{n^i} \sum_{l=i+1}^{mL} |S_l| \right] \right). \qquad (2)$$

The proof of the lemma is based on the fact that every hypercube of R_i has to be checked against those hypercubes of S (including the replicated ones) which have the same prefix.

In order to compute the number of distance computations DC we need to compute how many hypercubes will be on the different partitioning levels. We assume in our analytical model that the hypercubes are entirely in the unit cube $[0, 1)^d$. This property is not satisfied for our intersection join problem yet. However, through a simple linear mapping of the feature vectors it is easily possible to transform the original join problem into an equivalent one that provides the desired properties. We transform each component x_i of a feature vector x as follows:

$$x'_i = \frac{\varepsilon}{2} + x_i \cdot (1 - \varepsilon), (1 \leq i \leq d).$$

Furthermore, we modify the join predicate of the intersection join to

$$|x_i - y_i| \leq (1 - \varepsilon) \cdot \varepsilon, (1 \leq i \leq d).$$

Due to this transformation of a similarity join into an intersection join, it is sufficient to study intersection joins in the rest of the paper. Boundary effects [3] do not occur.

4.1 Analytical Model of MSJ

Since the size of a hypercube is fixed to ε in each dimension, the maximum partitioning level is bound by $mL = \lceil -\log \varepsilon \rceil - 1$. The probability $P_{NC}(\text{"hit at } l\text{"})$ that a hypercube hypercube hits a partitioning hyperplane at level l is given by

$$P_{NC}(\text{"hit at } l\text{"}) = 1 - \left(\frac{1 - 2^{l+1}\varepsilon}{1 - 2^l \varepsilon}\right)^d. \qquad (3)$$

For levels $l \geq 1$ we have to take care only those hypercubes that passed levels $0, \ldots, l-1$. The *conditional* probability $p_l := P_C(\text{"hit at } l\text{"})$ that a hypercube hits a hyperplane at level l is then given by

$$p_l = \begin{cases} P_{NC}(\text{"hit at } l\text{"}) \cdot \left(1 - \sum_{i=0}^{l-1} P_{NC}(\text{"hit at } i\text{"})\right) & , l < mL \\ 1 - \sum_{i=0}^{l-1} P_{NC}(\text{"hit at } i\text{"}) & , l = mL \end{cases} \qquad (4)$$

4.2 Analytical Model of GESS

In this section, we study the degree of replication of GESS where the replication strategy 1 is applied for a given k, $1 \leq k \leq d$: Whenever a hypercube cuts q hyperplanes, $q \leq k$ at level l, the hypercube is divided into 2^q portions such that each portion is entirely contained in a subspace. These portions proceed up to level $l+1$. Important to our analysis is that a hypercube can never hit more than one hyperplane which belongs to the same dimension.

Let l be the actual level. We are interested in the random variable X_l that returns the number of intersecting hyperplanes at level l for a hypercube. Note that the non-conditional probability p to hit a hyperplane in one dimension at level l is given by

$$p = 1 - \frac{1 - 2^{l+1}\varepsilon}{1 - 2^l \varepsilon}$$

The non-conditional probability to hit exactly q hyperplanes at level l follows a binomial distribution:

$$P_{NC}(X_l = q) = b_{d,p}(q) = \binom{d}{q} \cdot (1-p)^q \cdot p^{d-q}.$$

The probability to hit more than k hyperplanes is then given by

$$P_{NC}(X_l > k) = \sum_{i=k+1}^{d} P_{NC}(X_l = i).$$

For $d \to \infty$ and $d \cdot p \to \lambda$, $b_{d,p}(k)$ can be approximated by a Poisson distribution with mean value λ. We then obtain the following approximation:

$$P_{NC}(X_l > k) \approx \frac{(p \cdot d)^{k+2}}{(k+2)!} \qquad (5)$$

and hence, the probability to hit more than k hyperplanes drops exponentially with an decreasing k. Note that $p \cdot d < 1$ holds because we assume for each dimension a hypercube hits at most one hyperplane.

In order to compute the probability of a hypercube being kept at level l, only those hypercubes are taken into account that already passed levels $0, \ldots, l-1$, i.e., we consider the conditional probability. For sake of simplicity, let us first assume that replication does not increase the number of objects in the lower levels. The conditional probability to hit more than k hyperplanes at level l is given by

$$P(X_l > k) = P_{NC}(X_l > k | X_{l-1} \leq k | \ldots | X_0 \leq k). \qquad (6)$$

Since the events on the different levels are independent, the conditional probability on the right side of equation 6 can simply be expressed by a product. Overall, we obtain the following equation:

$$P(X_l > k) = \begin{cases} P_{NC}(X_l > k) \left(1 - \sum_{i=0}^{l-1} P(X_i > k)\right) & , l < mL \\ 1 - \sum_{i=0}^{l-1} P(X_i > k) & , l = mL \end{cases} \qquad (7)$$

The expected number of hypercubes at level l is then given by

$$|R| \cdot P(X_l > k). \qquad (8)$$

Next, we extend our model in such a way that the number of replicated objects are considered on the lower levels. In general, it is difficult to obtain an exact formula when replication is considered since the distributions of the different levels are not independent anymore. Therefore, we are interested in a simple and accurate estimation. We make the following two simplifying assumption: First, we assume that the replicated objects follow the same distribution of the original objects. Second, we assume that the replicated objects are of the same size as the original ones.

Let $\hat{r}_l$ be our estimation of the number of hypercubes at level l. Furthermore, let $\hat{r}_{>l}$ be our estimation of number of hypercubes that passed level l. By setting $\hat{r}_{-1} = |R|$, our estimations $\hat{r}_l$ and $\hat{r}_{>l}$ are recursively defined by the following formulas:

$$\begin{aligned} \hat{r}_l &= \hat{r}_{>l-1} \cdot P_{NC}(X_l > k) & , l < mL \\ \hat{r}_{>l} &= \hat{r}_{>l-1} \cdot \sum_{i=0}^{k} 2^i P_{NC}(X_l = i) & , l < mL \end{aligned} \qquad (9)$$

Moreover, we set $\hat{r}_{>l}$ and $\hat{r}_l = \hat{r}_{>l-1}$ for $l = mL$. In Section 6.1 we will provide results from a simulation to show that our estimations are sufficiently accurate.

5. I/O COST ANALYSIS

In this section, we first examine the I/O cost of MSJ and GESS where R and S are the input relations. The I/O cost is expressed in the number of pages transferred between disk and main memory. We assume that at most M pages are available in memory. We charge one I/O for a page transfer. The basic idea of MSJ is that R and S are partitioned into level files $R_0, R_1, \ldots$ and $S_0, S_1, \ldots$, respectively. For a partition P of a relation, $|P|$ denotes the number of pages of P. In particular, $|R|$ and $|S|$ refer to the number of pages in the base relations R and S, respectively.

5.1 I/O Cost of MSJ

THEOREM 2. *Let*

$$m_{R_i} = \begin{cases} \lceil \log_{M-1}(|R_i|/M) \rceil & , |R_i|/M > 1 \\ 0 & , |R_i|/M \leq 1 \end{cases}$$

be the number of merges for level-file R_i. The I/O cost of MSJ is

$$MSJ_{I/O\text{-}cost} = 4(|R| + |S|) + \underbrace{2 \sum_{i=0}^{mL} m_{R_i}|R_i| + m_{S_i}|S_i|}_{=:A} \tag{10}$$

(Proof in [15, 11]).

The computation of the cost for merging the runs, see term A in equation 10, requires knowledge about the distribution of the hypercubes among the different levels. With equation 4 we are in the position to compute the expected I/O cost for the merging of MSJ. As illustrated, the I/O performance decreases with a decreasing buffer size. In our experiments we observed that term A was greater 0 for data sets approximately 10 times larger than the available memory M.

5.2 I/O Cost of GESS

In this section, we examine the I/O cost of GESS. Each of the two relations R and S is directed to a sorting operator. These operators write initially sorted runs which requires $|R| + |S|$ I/Os. The total number of merges is

$$m_R = \begin{cases} \lceil \log_{M-1}(|R|/M) \rceil & , |R|/M > 1 \\ 0 & , |R|/M \leq 1. \end{cases}$$

Note that the final merge step does not write its data again on disk, but simply delivers its results to the next operator. Thus, the entire data set only has to be read in the final merge step whereas intermediate merges also have to write every block again. The number of intermediate merges is given by $m_{R,\text{intermed}} = \max(M_R - 1, 0)$. The total I/O cost of GESS is then given by

$$GESS_{\text{I/O-cost}} = 2(|R| + |S|) + \underbrace{2(m_{R,\text{intermed}}|R| + m_{S,\text{intermed}}|S|)}_{=:B}. \tag{11}$$

GESS requires at least two I/Os for each input page and additionally, I/O operations caused by intermediate merges (see B in equation 11).

In the following we show that term B is likely to be zero for most practical cases. GESS consists of two sorting operators and the partitions that have to be kept in memory. Recall, that during the creation of initial runs and intermediate merges external sorting may occupy the entire memory of the join. Our operator is based on replacement selection [13] which produces on average sorted runs of size $2M$. During the final merge, the sorting operators have to share the available memory. Let then $M_{\text{Sorter R}}$ ($M_{\text{Sorter R}} + M_{\text{Sorter S}} < M$) be the main memory (in pages) available to the sorting operator for data set R during the next-phase of the join. The maximal fan-in of the on-line merge is $M_{\text{Sorter R}} - 1$. If we assume sorted input runs of average size $2M$ we conclude that a data set of size

$$max_{data} = (M_{\text{Sorter R}} - 1)\, 2M$$

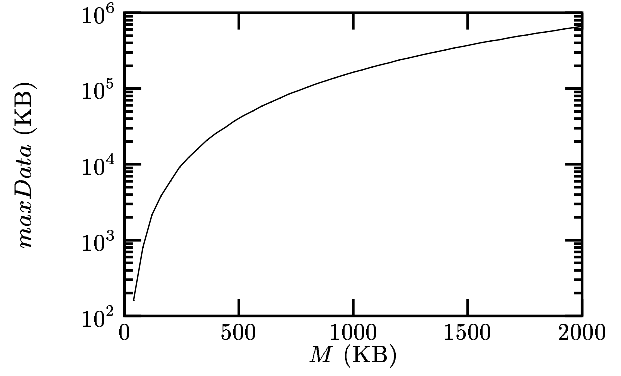


Figure 7: Maximum size of data sets if no intermediate merges are used

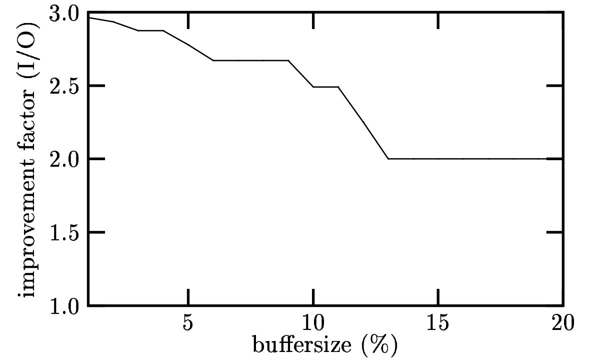


Figure 8: Improvement of GESS over MSJ in terms of I/O

can be sorted without using intermediate merges. Figure 7 depicts max_{data} as a function of M ($M_{\text{Sorter R}} = M/3$, page size = 4K). The graph shows that even for small values of M very large data sets can be sorted without the need of intermediate merges, e.g. data sets as large as 690 MB can be processed using only 2 MB memory. We conclude that in most practical cases GESS does not need any intermediate merges. For these cases, the I/O cost of GESS is

$$GESS_{\text{I/O-cost}} \approx 2(|R| + |S|) \tag{12}$$

independent from the fact whether replication occurs or not. Figure 8 depicts the improvement of GESS over MSJ. GESS performs at least two times better than MSJ for all buffer sizes. If 10% buffer is available GESS is 2.5 times better. For smaller buffer sizes the improvement is 2.7 and better.

6. EXPERIMENTS

6.1 Simulations

In order to confirm the analytical results we have implemented a simulator in Java based on the random engine of the COLT library [10]. Our simulator generates uniformly distributed hypercubes and computes the codes and the corresponding levels for each hypercube.

The results depicted in Figure 9 show an excellent agreement of the result obtained from our simulation and the ones of the analytical model ($d = 10$, $\varepsilon = 10^{-4}$). The graphs provide the relative occupation of hypercubes among the

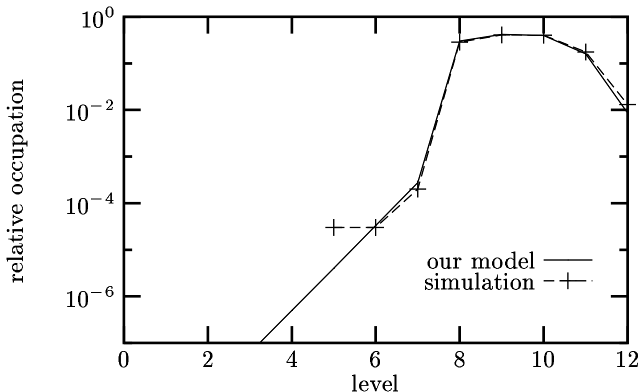


Figure 9: Comparison of the analytical model and the simulation

different level-files.

6.1.1 Distance Computations

Let IF be the improvement factor of GESS compared to MSJ w.r.t. distance computations. Figure 10 depicts IF for a self-join for uniformly distributed data sets. Figure 10a shows IF for $d = 10$, $msl = 2$ as a function of k (log-scale, $k = 0$ corresponds to the strategy of MSJ). For $\varepsilon = 10^{-4}$ ($k = 1$) IF is close to 10^3 and $1.2 \cdot 10^8$ ($k = 2$). IF increases with an increasing value of k up to several orders of magnitude. It however decreases for large ε. We observed modest replication for $\varepsilon = 10^{-3}$ and $\varepsilon = 10^{-4}$ (less than 1%). For $\varepsilon = 10^{-2}$ the replication was still less than factor 2. Since the I/O improvement of GESS is better than factor 2 (cf. chapter 5), GESS still needs less I/O operations for these parameter settings than MSJ. For larger values of ε we observed a higher replication rate.

In order to keep the replication rate low we used a different replication strategy in Figure 10b where $msl = 0$, i.e. splits were only allowed at level 0. For $k \geq 3$, the graph shows an IF of about 500 for $\varepsilon = 10^{-4}, 10^{-3}$ and 10^{-2} that remains constant for higher values of k. This can be explained by the fact that for $k = 1$ and $\varepsilon = 10^{-4}, 10^{-3}$ level 0 is already almost empty and only few hypercubes hit more than one hyperplane. For larger values of ε, however, the probability to hit multiple hyperplanes increases. For these cases, k has to be increased in order to empty level 0. In our experiments, the replication rates were less than 11% for all settings. For $\varepsilon = 10^{-1}$ the replication rose up to 2.85. The improvement IF however is still 95.

6.1.2 Memory Requirements

In this section we will take a look at the memory requirements of GESS. The size of the memory required for GESS can be estimated by $\hat{M} = \sum_{i=0}^{mL}(|R_i| + |S_i|)/n^i$ where $\hat{M} = \sum_{i=0}^{mL}|R_i|/n^i$ for a self-join. Figure 11 reports the main memory requirements $\hat{M}$ normalized to 1 ($\hat{M} = 1$ means that the entire input has to be kept in memory) of GESS for the experiments of the last section. Figure 11a shows that with an increasing k the memory requirements of GESS decrease exponentially. The main memory required for the join phase is by a factor 10^{-6} lower in comparison to MSJ for $k = 1$ and $\varepsilon = 10^{-4}$. For higher values of k this improvement even increases. For large values of ε the improvement is less but still exponentially depending on k.

data set	description	# points	d	size
CAD	fourier-vectors of CAD-parts	1,312,173	16	160 MB
CAD10	sample of CAD data set	131,217	16	16 MB

Table 1: Description of the data sets used in the experiments

data set	MSJ		GESS	
ε	10^{-3}	10^{-4}	10^{-3}	10^{-4}
CAD	35,093	3,821	772	275
CAD10	398	56	46	41

Table 2: Execution times in seconds for GESS and MSJ for a self-join

Figure 11b shows the results of the same experiment for $msl = 0$. The memory requirements in the graph first decrease exponentially and then remain constant. The latter effect can be explained by our observation that for high values of k level 0 is almost empty.

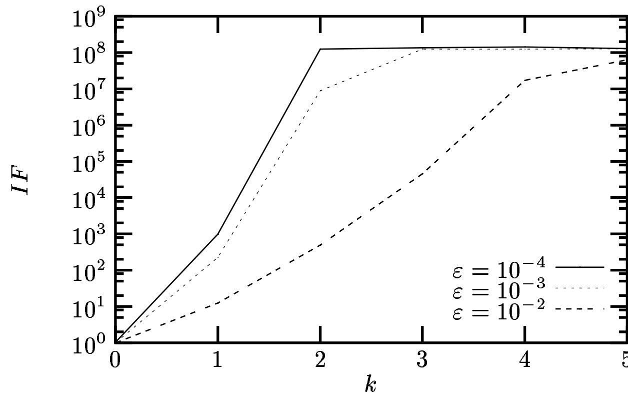
6.2 Implementations

In this section, we present results of an experimental evaluation of GESS and MSJ. The experiments were performed on an AthlonTB 700 with 256 MB main memory and 40 GB hard disk. All algorithms were implemented in Java 1.2 on top of the XXL-library [5, 4] using Suns just-in-time compiler. The buffer available to the algorithms was set to 10 MB. We run the experiments on real world data sets that have already been used in other experiments [8]. Table 1 describes important properties of the data sets.

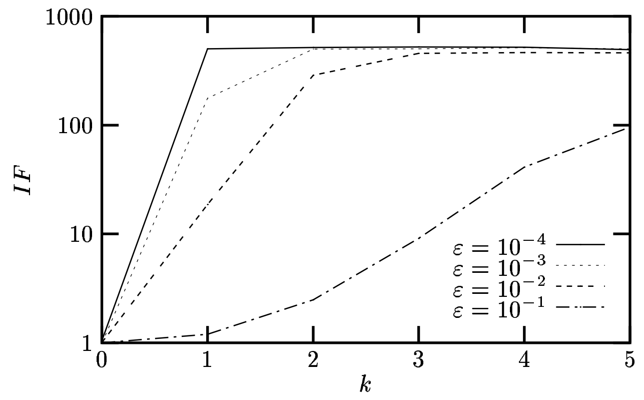
The charts in Table 2 show the execution time for GESS. Our new algorithm takes 772 seconds to compute the join for $\varepsilon = 0.001$ and 275 seconds for $\varepsilon = 0.0001$. For MSJ, however, the corresponding runtimes were 3,821 seconds and 35,093 seconds $\approx$ 10 hours, respectively. This means that the speedup of GESS is 14 and 45 for $\varepsilon = 0.0001$ and $\varepsilon = 0.001$, respectively. The size of the result set for these joins is $4.4 \cdot 10^5$ and $4.1 \cdot 10^6$ tuples. Because of this result we created a sample of the CAD data set where we only considered every tenth hypercube of the original data set for the join. Table 2 depicts the result where GESS performs the entire join in only 41 seconds ($\varepsilon = 0.0001$) whereas MSJ performs the join on the sample in 56 seconds. For $\varepsilon = 0.001$, our algorithm takes 46 seconds, whereas the runtime of MSJ rises to 398 seconds. Overall, GESS is 8 times faster than MSJ for that experiment.

The number of results for a self-join on CAD for $\varepsilon = 0.001$ is already 4.1 million which is surprisingly large. Under the uniformity and independent assumption, the number of expected tuples would be only $1.31 \cdot 10^{-42} \ll 1$. This is a strong indication that the true dimensionality [12] of the data is much lower.

In order to illustrate the dramatic performance improvements of GESS, we varied the replication strategy of our algorithm. Figure 12 shows the runtime as a function of the parameter k. Recall that hypercubes are split when at most k hyperplanes are hit. For $k > 3$ the curve remains almost constant (770 seconds). For $k = 2$ however the curve rises to 1,010 seconds. For $k = 1$ the execution time was

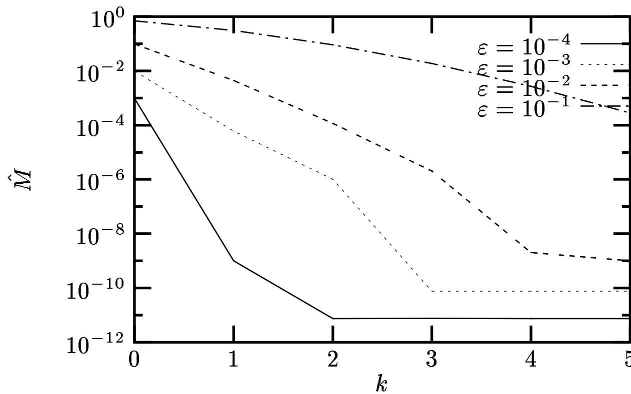


(a) replication allowed at levels $l \leq 2$

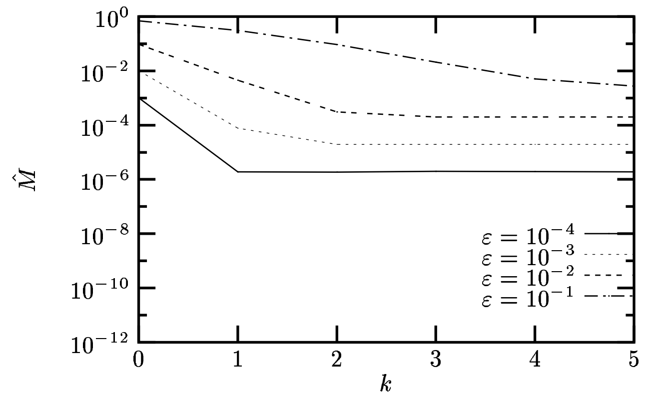


(b) replication only allowed at level $l = 0$

Figure 10: Distance-computations: Improvement-factor IF for GESS over MSJ



(a) replication allowed at levels $l \leq 2$



(b) replication only allowed at level $l = 0$

Figure 11: Memory requirements of GESS as a function of k ($k = 0$: MSJ)

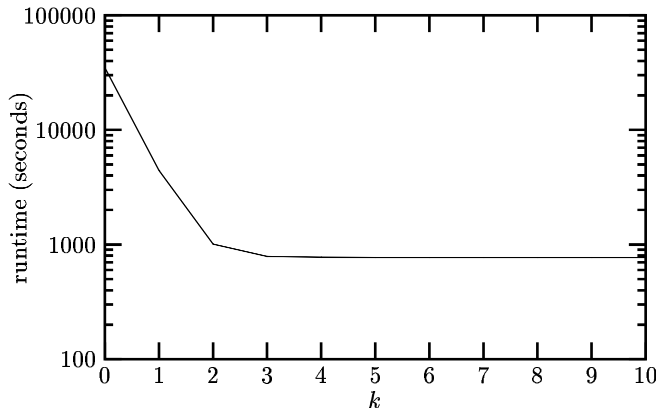


Figure 12: Self-join on CAD for various k ($k = 0$: MSJ)

4,427 seconds. This demonstrates that a modest kind of replication dramatically improves the performance of the algorithm. Overall, the replication rate was always less than 14%.

The overhead of the reference point method was irrelevant in most experiments. In the worst case of the experiments, 15% of the total runtime was spent for duplicate removal.

7. CONCLUSIONS

In this paper, we discussed different methods for processing similarity joins in high-dimensional spaces. We presented a new method (GESS) which employs a modest degree of replication to improve its overall runtime. The performance of GESS is compared to MSJ, analytically and experimentally. In our analytical model as well as in our experiments we observed that GESS clearly outperforms MSJ.

Our experiments show the tremendous potential of our approach. As we have seen, the replication strategy used for GESS plays an important role to achieve a good tradeoff between replication and the improvement factor IF. For practical purposes it is useful to find an 'optimal' strategy that determines the replication strategy dynamically. We are currently investigating dynamic and adaptable replica-

tion strategies that guarantee a lower bound for replication. These strategies also exploit the actual data distribution of the input relations and can easily be integrated in GESS.

Our future work also examines the applicability of compressed representations of feature vectors [24]. Furthermore, we are interested in integrating the fractal dimension in our cost models.

8. ACKNOWLEDGMENTS

We would like to acknowledge Christian Böhm for making available the data sets used in the experiments. Furthermore, we thank Björn Blohsfeld for discussions on this topic.

9. REFERENCES

[1] J. Ashley, M. Flickner, J. Hafner, D. Lee, W. Niblack, and D. Petkovic. The Query By Image Content (QBIC) System. In *ACM SIGMOD*, page 475, 1995.

[2] A. Belussi and C. Faloutsos. Self-Spacial Join Selectivity Estimation Using Fractal Concepts. In *TOIS*, volume 16, pages 161–201, 1998.

[3] S. Berchtold, C. Böhm, D. Keim, and H.-P. Kriegel. A Cost Model For Nearest Neighbor Search in High-Dimensional Data Space. In *PODS*, pages 78–86, 1997.

[4] J. Bercken, B. Blohsfeld, J.-P. Dittrich, J. Krämer, T. Schäfer, and B. Seeger. XXL — A Library Approach to Supporting Efficient Implementations of Advanced Database Queries. In *VLDB*, 2001.

[5] J. Bercken, J.-P. Dittrich, and B. Seeger. javax.XXL: a prototype for a library of query processing algorithms. In *ACM SIGMOD*, page 588, 2000.

[6] J. Bercken, M. Schneider, and B. Seeger. Plug&Join: An easy-to-use Generic Algorithm for Efficiently Processing Equi and Non-Equi Joins. In *EDBT*, pages 495–509, 2000.

[7] C. Böhm, B. Braunmüller, M. M. Breunig, and H.-P. Kriegel. High Performance Clustering Based on the Similarity Join. In *CIKM*, pages 298–313, 2000.

[8] C. Böhm and H.-P. Kriegel. A Cost Model and Index Architecture for the Similarity Join. In *ICDE*, pages 411–420, 2001.

[9] W. Cohen. Data integration using similarity joins and a word-based information representation language. In *TOIS*, volume 18, pages 288–321, 2000.

[10] http://tilde-hoschek.home.cern.ch/~hoschek-/colt/index.htm.

[11] J.-P. Dittrich and B. Seeger. Data Redundancy and Duplicate Detection in Spatial Join Processing. In *ICDE*, pages 535–546, 2000.

[12] C. Faloutsos, B. Seeger, A. J. M. Traina, and C. Traina Jr. Spatial Join Selectivity Using Power Laws. In *ACM SIGMOD*, pages 177–188, 2000.

[13] D. Knuth. *The Art of Computing, vol. 3 – Sorting and Searching.* Addison-Wesley, 1973.

[14] N. Koudas and K. Sevcik. Size Separation Spatial Join. In *ACM SIGMOD*, pages 324–335, 1997.

[15] N. Koudas and K. Sevcik. High Dimensional Similarity Joins: Algorithms and Performance Evaluation. In *ICDE*, pages 466–475. (Best Paper Award), 1998.

[16] N. Koudas and K. Sevcik. High Dimensional Similarity Joins: Algorithms and Performance Evaluation. *TKDE*, 12:3–18, 2000.

[17] M.-L. Lo and C. V. Ravishankar. Spatial Hash-Joins. In *ACM SIGMOD*, pages 247–258, 1996.

[18] J. Orenstein. Spatial Query Processing in an Object-Oriented Database System. In *ACM SIGMOD*, pages 326–336. ACM Press, 1986.

[19] J. Orenstein. An Algorithm for Computing the Overlay of k–Dimensional Spaces. In *SSD*, pages 381–400, 1991.

[20] J. Patel and D. DeWitt. Partition Based Spatial-Merge Join. In *ACM SIGMOD*, pages 259–270, 1996.

[21] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Weseley, 1990.

[22] D. Scott. *Multivariate Density Estimation.* Wiley-Interscience, 1992.

[23] K. Shim, R. Srikant, and R. Agrawal. High-Dimensional Similarity Joins. In *ICDE*, pages 301–313, 1997.

[24] R. Weber, H.-J. Schek, and S. Blott. A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces. In *VLDB*, pages 194–205, 1998.

APPENDIX

A. NOTATION

Symbol	Definition
d	dimension
ε	query distance
U	unit space: $U = [0,1)^d$
$H_{\varepsilon,d}(x)$	d-dimensional hypercube of length ε in each dimension for feature vector x
$H(x)$	$= H_{\varepsilon,d}(x)$
msl	maximum split level
n	number of direct subspaces (for an n-ary recursive partitioning)
$P(D)$	recursive partitioning of D
$Code_D(X)$	code of a subspace $X \in P(D)$
l	recursion level of the recursive partitioning (entire space: $l = 0$)
mL	maximum level of the recursive partitioning
k	maximum number of hyperplanes a hypercube is allowed to intersect
R, S	name of input relations
$\|R\|, \|S\|$	number of pages of relation R (S)
R_i, S_i	subset of elements of data set R (S) that is assigned to level i
M	number of pages available in main memory
DC	number of distance computations for a join
IF	improvement factor for GESS over MSJ in terms of distance computations

Table 3: Notation used in this paper