# High Performance Data Mining Using the Nearest Neighbor Join

Christian Böhm
*University for Health Informatics and Technology*
*Christian.Boehm@umit.at*

Florian Krebs
*University of Munich*
*krebs@dbs.informatik.uni-muenchen.de*

## Abstract

*The similarity join has become an important database primitive to support similarity search and data mining. A similarity join combines two sets of complex objects such that the result contains all pairs of similar objects. Well-known are two types of the similarity join, the distance range join where the user defines a distance threshold for the join, and the closest point query or k-distance join which retrieves the k most similar pairs. In this paper, we investigate an important, third similarity join operation called k-nearest neighbor join which combines each point of one point set with its k nearest neighbors in the other set. It has been shown that many standard algorithms of Knowledge Discovery in Databases (KDD) such as k-means and k-medoid clustering, nearest neighbor classification, data cleansing, postprocessing of sampling-based data mining etc. can be implemented on top of the k-nn join operation to achieve performance improvements without affecting the quality of the result of these algorithms. We propose a new algorithm to compute the k-nearest neighbor join using the multipage index (MuX), a specialized index structure for the similarity join. To reduce both CPU and I/O cost, we develop optimal loading and processing strategies.*

## 1. Introduction

KDD algorithms in multidimensional databases are often based on similarity queries which are performed for a high number of objects. Recently, it has been recognized that many algorithms of similarity search [2] and data mining [3] can be based on top of a single join query instead of many similarity queries. Thus, a high number of single similarity queries is replaced by a single run of a *similarity join*. The most well-known form of the similarity join is the distance range join $R \underset{\varepsilon}{\bowtie} S$ which is defined for two finite sets of vectors, $R = \{r_1,...,r_n\}$ and $S = \{s_1,...,s_m\}$, as the set of all pairs from $R \times S$ having a distance of no more than ε:

$$R \underset{\varepsilon}{\bowtie} S := \{(r_i,s_j) \in R \times S \mid \|p_i - q_j\| \leq \varepsilon\}$$

E.g. in [3], it has been shown that density based clustering algorithms such as DBSCAN [25] or the hierarchical cluster analysis method OPTICS [1] can be accelerated by high factors of typically one or two orders of magnitude by the range distance join. Due to its importance, a large number of algorithms to compute the range distance join of two sets have been proposed, e.g. [27, 19, 5]

Another important similarity join operation which has been recently proposed is the incremental distance join [16]. This join operation orders the pairs from $R \times S$ by increasing distance and returns them to the user either on a give-me-more basis, or based on a user specified cardinality of $k$ best pairs (which corresponds to a $k$-closest pair operation in computational geometry, cf. [23]). This operation can be successfully applied to implement data analysis tasks such as noise-robust catalogue matching and noise-robust duplicate detection [11].

In this paper, we investigate a third kind of similarity join, the $k$-nearest neighbor similarity join, short $k$-nn join. This operation is motivated by the observation that many data analysis and data mining algorithms is based on $k$-nearest neighbor queries which are issued separately for a large set of *query points* $R = \{r_1,...,r_n\}$ against another large set of *data points* $S = \{s_1,...,s_m\}$. In contrast to the incremental distance join and the $k$-distance join which choose the best pairs from the complete pool of pairs $R \times S$, the $k$-nn join combines each of the points of $R$ with its $k$ nearest neighbors in $S$. The differences between the three kinds of similarity join operations are depicted in figure 1.

Applications of the $k$-nn join include but are not limited to the following list: $k$-nearest neighbor classification, $k$-means and $k$-medoid clustering, sample assessment and sample postprocessing, missing value imputation, $k$-distance diagrams, etc. In [8] we have shown that $k$-means clustering, nearest neighbor classification, and various other algorithms can be transformed such that they operate exclusively on top of the $k$-nearest neighbor join. This transformation typically leads to performance gains up to a factor of 8.5.

Our list of applications covers all stages of the KDD process. In the preprocessing step, data cleansing algorithms are typically based on $k$-nearest neighbor queries for each of the points with NULL values against the set of complete vectors. The missing values can be computed e.g. as the weighted means of the values of the $k$ nearest neighbors. A $k$-distance diagram can be used to determine suitable parameters for data mining. Additionally, in the core step, i.e. data mining, many algorithms such as clustering and classification are based on $k$-nn queries. As such algorithms are often time consuming and have at least a linear, often $n \log n$ or even quadratic complexity they typically run on a sample set rather than the complete data set. The $k$-nn-queries are used to assess the quality of the sample set (preprocessing). After the run of the data mining algorithm, it is necessary to relate the result to the complete set of database points [10]. The typical method for doing that is again a $k$-nn-query for each of the database points with respect to the set of classified sample points. In all these algorithms, it is possible to replace a large number of $k$-nn queries which are originally issued separately, by a single run of a $k$-nn join. Therefore, the $k$-nn join gives powerful support for all stages of the KDD process.

The remainder of the paper is organized as follows: In section 2, we give a classification of the well-known similarity join operations and review the related work. In section 3, we define the new operation, the $k$-nearest neighbor join. In section 4, we develop an algorithm for the $k$-nn join which applies matching loading and processing strategies on top of the multipage index [7], an index structure which is particularly suited for high-dimensional similarity joins, in order to reduce both CPU and I/O cost and efficiently compute the $k$-nn join. The experimental evaluation of our approach is presented in section 5 and section 6 concludes the paper.

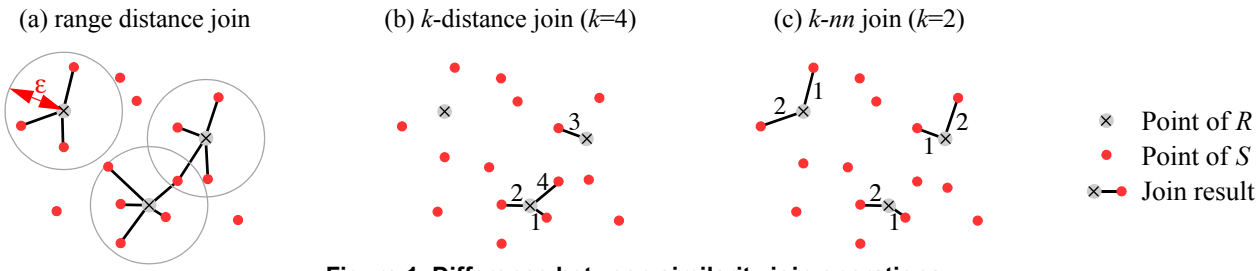



**Figure 1. Difference between similarity join operations**

## 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, e.g. the Euclidean distance between two feature vectors.

### 2.1 Distance range based similarity join

The most prominent and most evaluated similarity join operation is the distance range join. Therefore, the notions *similarity join* and *distance range join* are often used interchangeably. Unless otherwise specified, when speaking of *the similarity join*, often the distance range join is meant by default. For clarity in this paper, we will not follow this convention and always use the more specific notions. As depicted in figure 1a, the distance range join $R \underset{\varepsilon}{\bowtie} S$ of two multidimensional or metric sets $R$ and $S$ is the set of pairs where the distance of the objects does not exceed the given parameter ε:

**Definition 1** Distance Range Join (ε-Join)

The distance range join $R \underset{\varepsilon}{\bowtie} S$ of two finite multidimensional or metric sets $R$ and $S$ is the set

$$R \underset{\varepsilon}{\bowtie} S := \{(r_i, s_j) \in R \times S : \|r_i - s_j\| \leq \varepsilon\}$$

The distance range join can also be expressed in a SQL like fashion:

**SELECT** * **FROM** $R$, $S$ **WHERE** $\|R.\text{obj} - S.\text{obj}\| \leq \varepsilon$

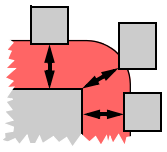
In both cases, $\|\cdot\|$ denotes the distance metric which is assigned to the multimedia objects. For multidimensional vector spaces, $\|\cdot\|$ usually corresponds to the Euclidean distance. The distance range join can be applied in density based clustering algorithms which often define the local data density as the number of objects in the ε-neighborhood of some data object. This essentially corresponds to a self-join using the distance range paradigm.

Like for plain range queries in multimedia databases, a general problem of distance range joins from the users' point of view is that it is difficult to control the result cardinality of this operation. If ε is chosen too small, no pairs are reported in the result set (or in case of a self join: each point is only combined with itself). In contrast, if ε is chosen too large, each point of $R$ is combined with every point in $S$ which leads to a quadratic result size and thus to a time complexity of any join algorithm which is at least quadratic; more exactly $o(|R| \cdot |S|)$. The range of possible ε-values where the result set is non-trivial and the result set size is sensible is often quite narrow, which is a consequence of the curse of dimensionality. Provided that the parameter ε is chosen in a suitable range and also adapted with an increasing number of objects such that the result set size remains approximately constant, the typical complexity of advanced join algorithms is better than quadratic.

Most related work on join processing using multidimensional index structures is based on the *spatial join*. 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)* [9] which processes R-tree like index structures built on both relations $R$ and $S$. RSJ is based on the lower bounding property which means that the distance between two points is never smaller than the distance (the so-called mindist, cf. figure 2) 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 proposedsuch as the *BFRJ-algorithm* [14] which traverses the indexes according to a breadth-first strategy.

Recently, index based similarity join methods have been analyzed from a theoretical point of view. [7] proposes a cost model based on the concept of the Minkowski sum [4] which can be used for optimizations such as page size optimization. The analysis reveals a serious optimization conflict between CPU and I/O time. 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. 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 (optimized for maximum CPU efficiency). It is shown that the resulting index yields an I/O performance which is similar to the I/O optimized R-tree similarity join and a CPU performance which is close to the CPU optimized R-tree similarity join.

If no multidimensional index is available, it is possible to construct the index on the fly before starting the join algorithm. Several techniques for bulk-loading multidimensional index structures



$$\text{mindist}^2 = \sum_{0 \leq i < d} \begin{cases} (R.\text{lb}_i - S.\text{ub}_i)^2 & \text{if } R.\text{lb}_i > S.\text{ub}_i \\ 0 & \text{otherwise} \\ (S.\text{lb}_i - R.\text{ub}_i)^2 & \text{if } S.\text{lb}_i > R.\text{ub}_i \end{cases}$$

**Figure 2. mindist for the similarity join on R-trees**

have been proposed [17, 12]. The *seeded tree method* [20] 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* [21, 22] decomposes the set $R$ into a number of partitions which is determined according to given system parameters.

A join algorithm particularly suited for similarity self joins is the ε-*kdB-tree* [27]. The basic idea is to partition the data set perpendicularly to one selected dimension into stripes of width ε to restrict the join to pairs of subsequent stripes. 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. Koudas and Sevcik have proposed the *Size Separation Spatial Join* [18] and the *Multidimensional Spatial Join* [19] which make use of space filling curves to order the points in a multidimensional space. An approach which explicitly deals with massive data sets and thereby avoids the scalability problems of existing similarity join techniques is the *Epsilon Grid Order (EGO)* [5]. It 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 then compares the grid cells lexicographically.

## 2.2 Closest pair queries

It is possible to overcome the problems of controlling the selectivity by replacing the range query based join predicate using conditions which specify the selectivity. In contrast to range queries which retrieve potentially the whole database, the selectivity of a ($k$-) closest pair query is (up to tie situations) clearly defined. This operation retrieves the $k$ pairs of $R \times S$ having minimum distance. (cf. figure 1b) Closest pair queries do not only play an important role in the database research but have also a long history in computational geometry [23]. In the database context, the operation has been introduced by Hjaltason and Samet [16] using the term ($k$-) *distance join*. The ($k$-)closest pair query can be formally defined as follows:

**Definition 2** ($k$-) Closest Pair Query $R \underset{k\text{-CP}}{\bowtie} S$

$R \underset{k\text{-CP}}{\bowtie} S$ is the smallest subset of $R \times S$ that contains at least $k$ pairs of points and for which the following condition holds:

$$\forall (r,s) \in R \underset{k\text{-CP}}{\bowtie} S,\ \forall (r',s') \in R \times S \setminus R \underset{k\text{-CP}}{\bowtie} S: \|r-s\| < \|r'-s'\| \quad (1)$$

This definition directly corresponds to the definition of ($k$-) nearest neighbor queries, where the single data object $o$ is replaced by the pair $(r,s)$. Here, tie situations are broken by enlargement of the result set. It is also possible to change definition 2 such that the tie is broken non-deterministically by a random selection. [16] defines the closest pair query (non-deterministically) by the following SQL statement:

**SELECT * FROM** $R$, $S$
**ORDER BY** $\|R.\text{obj} - S.\text{obj}\|$
**STOP AFTER** $k$

We give two more remarks regarding self joins. Obviously, the closest pairs of the selfjoin $R \underset{k\text{-CP}}{\bowtie} R$ are the $n$ pairs $(r_i,r_i)$ which have trivially the distance 0 (for any distance metric), where $n = |R|$ is the cardinality of $R$. Usually, these trivial pairs are not needed, and, therefore, they should be avoided in the **WHERE** clause. Like the distance range selfjoin, the closest pair selfjoin is symmetric (unless nondeterminism applies). Applications of closest pair queries (particularly self joins) include similarity queries like

- find all stock quota in a database that are similar to each other
- find music scores which are similar to each other
- noise-robust duplicate elimination in multimedia applications
- match two collections of arbitrary multimedia objects

Hjaltason and Samet [16] also define the distance semijoin which performs a GROUP BY operation on the result of the distance join. All join operations, $k$-distance join, incremental distance join and the distance semijoin are evaluated using a pqueue data structure where node-pairs are ordered by increasing distance.

The most interesting challenge in algorithms for the distance join is the strategy to access pages and to form page pairs. Analogously to the various strategies for single nearest neighbor queries such as [24] and [15], Corral et al. propose 5 different strategies including recursive algorithms and an algorithm based on a pqueue [13]. Shin et al. [26] proposed a plane sweep algorithm for the node expansion for the above mentioned pqueue algorithm [16, 13]. In the same paper [26], Shim et al. also propose the *adaptive multi-stage algorithm* which employs aggressive pruning and compensation methods based on statistical estimates of the expected distance values.

# 3. The *k*-nn-join

The range distance join has the disadvantage of a result set cardinality which is difficult to control. This problem has been overcome by the closest pair query where the result set size (up to the rare tie effects) is given by the query parameter $k$. However, there are only few applications which require the consideration of the $k$ best pairs of two sets. Much more prevalent are applications such as classification or clustering where each point of one set must be combined with its $k$ closest partners in the other set, which is exactly the operation that corresponds to our new $k$-nearest neighbor similarity join (cf. figure 1c). Formally, we define the $k$-nn join as follows:

**Definition 3** $k$-nn Join $R \underset{k\text{-nn}}{\ltimes} S$

$R \underset{k\text{-nn}}{\ltimes} S$ is the smallest subset of $R \times S$ that contains for each point of $R$ at least $k$ points of $S$ and for which the following condition holds:

$$\forall (r,s) \in R \underset{k\text{-nn}}{\ltimes} S,\ \forall (r,s') \in R \times S \setminus R \underset{k\text{-nn}}{\ltimes} S: \|r-s\| < \|r-s'\| \quad (2)$$

In contrast to the closest pair query, here it is guaranteed that each point of $R$ appears in the result set exactly $k$ times. Points of $S$ may appear once, more than once (if a point is among the $k$-nearest neighbors of several points in $R$) or not at all (if a point does not belong to the $k$-nearest neighbors of any point in $R$). Our $k$-nn join can be expressed in an extended SQL notation:

**SELECT * FROM** $R$,
( **SELECT *** FROM $S$
**ORDER BY** $\|R.\text{obj} - S.\text{obj}\|$
**STOP AFTER** $k$ )

The closest pair query applies the principle of the nearest neighbor search (finding $k$ best *things*) on the basis of the pairs. Conceptually, first all pairs are formed, and then, the best $k$ are selected. In contrast, the $k$-nn join applies this principle on a basis "per point of the first set". For each of the points of $R$, the $k$ best join partners are searched. This is an essential difference of concepts.

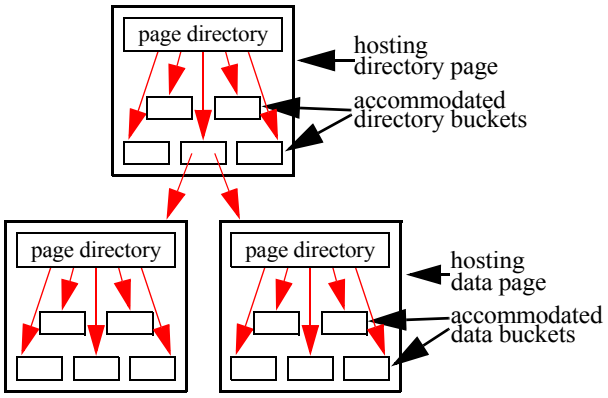



**Figure 3. Index architecture of the multipage index**

Again, tie situations can be broken deterministically by enlarging the result set as in this definition or by random selection. For the selfjoin, we have again the situation that each point is combined with itself which can be avoided using the WHERE clause. Unlike the ε-join and the *k*-closest pair query, the *k*-nn selfjoin is not symmetric as the nearest neighbor relation is not symmetric. Equivalently, the join $R \overset{k\text{-}nn}{\ltimes} S$ which retrieves the *k* nearest neighbors for each point of *R* is essentially different from $S \overset{k\text{-}nn}{\ltimes} R$ which retrieves the nearest neighbors of each *S*-point. This is symbolized in our symbolic notation which uses an *asymmetric* symbol for the *k*-nn join in contrast to the other similarity join operations.

# 4. Fast index scans for the *k*-nn join

In this section we develop an algorithm for the *k*-nn join which applies suitable loading and processing strategies on top of a multidimensional index structure, the multipage index [7], to efficiently compute the *k*-nn join. In [7] we have shown for the distance range join that it is necessary to optimize index parameters such as the page capacity separately for CPU and I/O performance. We have proposed a new index architecture (Multipage Index, MuX) depicted in figure 3 which allows such a separate optimization. The index consists of large pages which are optimized for I/O efficiency. These pages accommodate a secondary R-tree like main memory search structure with a page directory (storing pairs of MBR and a corresponding pointer) and data buckets which are containers for the actual data points. The capacity of the accommodated buckets is much smaller than the capacity of the hosting page. It is optimized for CPU performance. We have shown that the distance range join on the Multipage Index has an I/O performance similar to an R-tree which is purely I/O optimized and has a CPU performance like an R-tree which is purely CPU optimized. Although this issue is up to future work, we assume that also the *k*-nn join clearly benefits from the separate optimization (because optimization trade-offs are very similar).

In the following description, we assume for simplicity that the hosting pages of our Multipage Index only consist of one directory level and one data level. If there are more directory levels, these levels are processed in a breadth first approach according to some simple strategy, because most cost arise in the data level. Therefore, our strategies focus on the last level.

## 4.1 The fast index scan

In our previous work [6] we have already investigated fast index scans, however not in the context of a join operation but in the context of single similarity queries (range queries and nearest neighbor queries) which are evaluated on top of an R-tree like index structure, our IQ tree. The idea is to chain I/O operations for subsequent pages on disk. This is relatively simple for range queries: If the index is traversed breadth-first, then the complete set of required pages at the next level is exactly known in advance. Therefore, pages which have adjacent positions on disk can be immediately grouped together into a single I/O request (cf. figure 4, left side). But also pages which are not direct neighbors but only close together can be read without disk head movement. So the only task is to sort the page requests by (ascending) disk addresses before actually performing them. For nearest neighbor queries the trade-off is more complex: These are usually evaluated by the HS-algorithm [15] which has been proven to be optimal, w.r.t. the *number* of accessed pages. Although the algorithm loses its optimality by I/O chaining of page requests, it pays off to chain pages together which have a low probability of being pruned before their actual request is due. We have proposed a stochastical model to estimate the probability of a page to be required for a given nearest neighbor query. Based on this model we can estimate the cost for various chained and unchained I/O requests and thus optimize the I/O operations (cf. figure 4, right side).

Take a closer look at the trade-off which is exploited in our optimization: If we apply no I/O chaining or too careful chaining, then the *number* of processed pages is optimal or close to optimal but due to heavy disk head movements these accesses are very expensive. If considerable parts of the data set are needed to answer the query, the index can be outperformed by the sequential scan. In contrast, if too many pages are chained together, many pages are processed unnecessarily before the nearest neighbor is found. If only a few pages are needed to answer a query, I/O chaining should be carefully applied, and the index should be traversed in the classical way of the HS algorithm. Our probability estimation grasps this rule of thumb with many gradations between the two extremes.

## 4.2 Optimization goals of the nearest neighbor join

Shortly speaking, the trade-off of the nearest neighbor search is between (1) getting the nearest neighbor *early* and (2) limiting the cost for the single I/O operations. In this section, we will describe a similar trade-off in the *k*-nearest neighbor join. One important goal of the algorithm is to get a good approximation of the nearest neighbor (i.e. a point which is not necessarily *the* nearest neighbor


**Figure 4. The fast index scan for single range queries (l.) and for single nearest neighbor queries (r.)**

but a point which is not much worse than the nearest neighbor) for each of these active queries as early as possible. With a good conservative approximation of the *nearest neighbor distance*, we can even abstain from our probability model of the previous paragraph and handle nearest neighbor queries furtheron like range queries. Only few pages are processed too much.

In contrast to single similarity queries, the seek cost do not play an important role in our join algorithm because our special index structure, MuX, is optimized for disk I/O. Our second aspect, however, is the CPU performance which is negligible for single similarity queries but not for join queries. From the CPU point of view, it is not a good strategy to load a page and immediately process it (i.e. join it with all pages which are already in main memory, which is usually done for join queries with a range query predicate). Instead, the page should be paired only with those pages for which one of the following conditions holds:

- It is probable that this pair leads to a considerable reduction of some nearest neighbor distance
- It is improbable that the corresponding mate page will receive any improvements of its nearest neighbor distance in future

While the first condition seems to be obvious, the second condition is also important because it ensures that unavoidable workloads are done before other workloads which are avoidable. The cache is primarily loaded with those pages of which it is most unclear whether or not they will be needed in future.

## 4.3 Basic algorithm

For the $k$-nn join $R \underset{k\text{-nn}}{\ltimes} S$, we denote the data set $R$ for each point of which the nearest neighbors are searched as the outer point set. Consequently, $S$ is the inner point set. As in [7] we process the hosting pages of $R$ and $S$ in two nested loops (obviously, this is not a *nested loop join*). Each hosting page of the outer set $R$ is accessed exactly once. The principle of the nearest neighbor join is illustrated in figure 5. A hosting page $PR_1$ of the outer set with 4 accommodated buckets is depicted in the middle. For each point stored in this page, a data structure for the $k$ nearest neighbors is allocated. Candidate points are maintained in these data structures until they are either discarded and replaced by new (better) candidate points or until they are *confirmed* to be the actual nearest neighbors of the corresponding point. When a candidate is *confirmed*, it is guaranteed that the database cannot contain any closer points, and the pair can be written to the output. The distance of the last (i.e. $k$-th or worst) candidate point of each $R$-point is the pruning distance: Points, accommodated buckets and hosting pages beyond that pruning distance need not to be considered. The pruning distance of a *bucket* is the *maximum* pruning distance of all points stored in this bucket, i.e. all $S$-buckets which have a distance from a given $R$-bucket that exceeds the pruning distance of the $R$-bucket, can be safely neglected as join-partners of that $R$-bucket. Similarly, the pruning distance of a *page* is the maximum pruning distance of all accommodated buckets.

In contrast to conventional join methods we reserve only one cache page for the outer set $R$ which is read exactly once. The remaining cache pages are used for the inner set $S$. For other join predicates (e.g. relational predicates or a distance range predicate), a strategy which caches more pages of the outer set is beneficial for I/O processing (the inner set is scanned fewer times) while the CPU performance is not affected by the caching strategy. For the $k$-nn join predicate, the cache strategy affects both I/O *and* CPU performance. It is important that for each considered point of $R$ good candidates (i.e. near neighbors, not necessarily the nearest neighbors) are found as early as possible. This is more likely when reserving more cache for the inner set $S$. The basic algorithm for the $k$-nn join is given below.

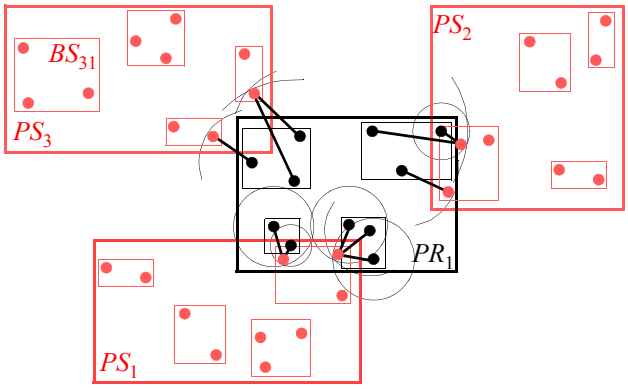



**Figure 5. *k-nn* join on the multipage index (here *k*=1)**

```
1    foreach PR of R do
2        cand : PQUEUE [|PR|, k] of point := {⊥,⊥,...,⊥} ;
3        foreach PS of S do PS.done := false ;
4        while ∃ i such that cand [i] is not confirmed do
5            while ∃ empty cache frame ∧
6                  ∃ PS with (¬PS.done ∧ ¬ IsPruned(PS)) do
7                apply loading strategy if more than 1 PS exist
8                load PS to cache ;
9                PS.done := true ;
10           apply processing strategy to select a bucket pair ;
11           process bucket pair ;
```

A short explanation: (1) Iterates over all hosting pages $PR$ of the outer point set $R$ which are accessed in an arbitrary order. For each point in $PR$, an array for the $k$ nearest neighbors (and the corresponding candidates) is allocated and initialized with empty pointers in line (2). In this array, the algorithm stores candidates which may be replaced by other candidates until the candidates are *confirmed*. A candidate is confirmed if no unprocessed hosting page or accommodated bucket exists which is closer to the corresponding $R$-point than the candidate. Consequently, the loop (4) iterates until all candidates are confirmed. In lines 5-9, empty cache pages are filled with hosting pages from $S$ whenever this is possible. This happens at the beginning of processing and whenever pages are discarded because they are either processed or pruned for all $R$-points. The decision which hosting page to load next is implemented in the so-called loading strategy which is described in section 4.4. Note that the actual page access can also be done asynchronously in a multithreaded environment. After that, we have the accommodated buckets of one hosting $R$-page and of several hosting $S$-pages in the main memory. In lines 10-11, one pair of such buckets is chosen and processed. For choosing, our algorithm applies a so-called *processing strategy* which is described in section 4.5. During processing, the algorithm tests whether points of the current $S$-bucket are closer to any point of the current $R$-bucket than the corresponding candidates are. If so, the candidate array is updated (not depicted in our algorithm) and the pruning distances are also changed. Therefore, the current $R$-bucket can safely prune some of the $S$-buckets that formerly were considered join partners.

### 4.4 Loading strategy

In conventional similarity search where the nearest neighbor is searched only for one query point, it can be proven that the optimal strategy is to access the pages in the order of increasing distance from the query point [4]. For our *k*-nn *join*, we are simultaneously processing nearest neighbor queries for all points stored in a hosting page. To exclude as many hosting pages and accommodated buckets of *S* from being join partners of one of these simultaneous queries, it is necessary to decrease all pruning distances as early as possible. The problem we are addressing now is, what page should be accessed next in lines 5-9 to achieve this goal.

Obviously, if we consider the complete set of points in the current hosting page *PR* to assess the quality of an unloaded hosting page *PS*, the effort for the optimization of the loading strategy would be too high. Therefore, we do not use the complete set of points but rather the accommodated buckets: the pruning distances of the accommodated buckets have to decrease as fast as possible.

In order for a page *PS* to be good, this page must have the power of *considerably improving* the pruning distance of at least one of the buckets *BR* of the current page *PR*. Basically there can be two obstacles that can prevent a pair of such a page *PS* and a bucket *BR* from having a high improvement power: (1) the distance (mindist) between this page-bucket pair is large, and (2) the bucket *BR* has *already* a small pruning distance. Condition (1) corresponds to the well-known strategy of accessing pages in the order of increasing distance to the query point. Condition (2), however, intends to avoid that the same bucket *BR* is repeatedly processed before another bucket *BR*' has reached a reasonable pruning distance (having such buckets *BR*' in the system causes much avoidable effort).

Therefore, the *quality* Q(*PS*) of a hosting page *PS* of the inner set *S* is not only measured in terms of the distance to the current buckets but the distances are also related to the current pruning distance of the buckets:

$$Q(PS) = \max_{BR \in PR} \left\{ \frac{\text{prunedist}(BR)}{\text{mindist}(PS, BR)} \right\} \qquad (3)$$

Our *loading strategy* applied in line (7) is to access the hosting pages *PS* in the order of decreasing quality Q(*PS*), i.e. we always access the unprocessed page with the highest quality.

### 4.5 Processing strategy

The processing strategy is applied in line (10). It addresses the question in what order the accommodated buckets of *R* and *S* that have been loaded into the cache should be processed (joined by an in-memory join algorithm). The typical situation found at line (10) is that we have the accommodated buckets of *one* hosting page of *R* and the accommodated buckets of *several* hosting pages of *S* in the cache. Our algorithm has to select a pair of such buckets (*BR*,*BS*) which has a high quality, i.e. a high potential of improving the pruning distance of *BR*. Similarly to the quality Q(*PS*) of a page developed in section 4.4, the quality Q(*BR*,*BS*) of a bucket pair rewards a small distance and punishes a small pruning distance:

$$Q(BR, BS) = \frac{\text{prunedist}(BR)}{\text{mindist}(BS, BR)} \qquad (4)$$

We process the bucket pairs in the order of decreasing quality. Note that we do not have to redetermine the quality of every bucket pair each time our algorithm runs into line (10) which would be prohibitively costly. To avoid this problem, we organize our current bucket pairs in a tailor-made data structure, a fractionated pqueue (half sorted tree). By *fractionated* we mean a pqueue of pqueues, as depicted in figure 6. Note that this tailor-cut structure allows efficiently (1) to determine the pair with maximum quality, (2) to insert a new pair, and in particular (3) to update the prunedist of $BR_i$ which affects the quality of a large number of pairs.

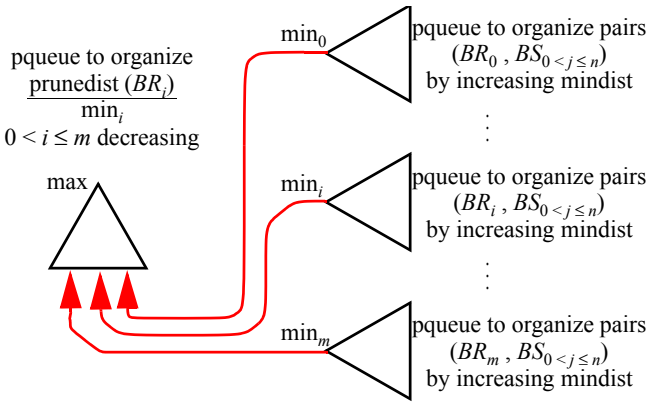



**Figure 6. Structure of a fractionated pqueue**

Processing bucket pairs with a high quality is highly important at an early stage of processing until all *R*-buckets have a sufficient pruning distance. Later, the improvement power of the pairs does not differ very much and a new aspect comes into operation: The pairs should be processed such that one of the hosting *S* pages in the cache can be replaced as soon as possible by a new page. Therefore, our processing strategy switches into a new mode if the last *c* (given parameter) processing steps did not lead to a considerable improvement of any pruning distance. The new mode is to select one hosting *S*-page *PS* in the cache and to process all pairs where one of the buckets *BS* accommodated by *PS* appears. We select that hosting page *PS* with the fewest active pairs (i.e. the hosting page that causes least effort).

## 5. Experimental evaluation

We implemented the k-nearest neighbor join algorithm, as described in the previous section, based on the original source code of the Multipage Index Join [7] and performed an experimental evaluation using artificial and real data sets of varying size and di-

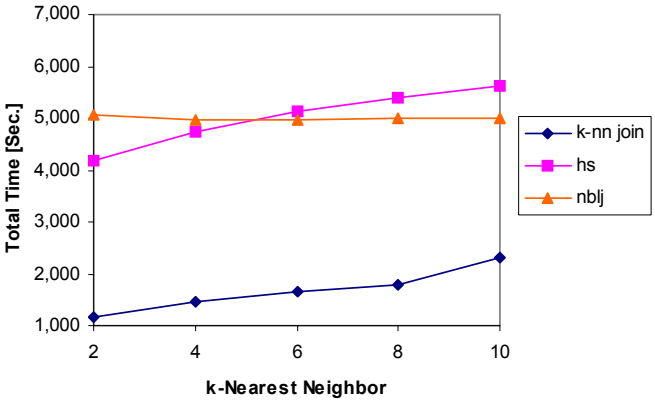



**Figure 7. Varying k for 8-dimensional uniform data**

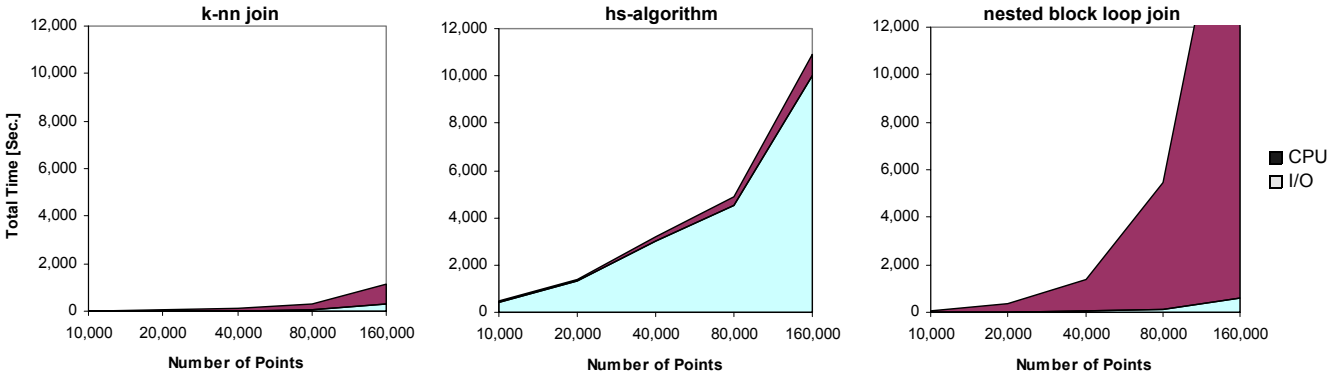



**Figure 8. Total time, CPU-time and I/O-time for hs, *k-nn* join and nblj for varying size of the database**

mension. We compared the performance of our technique with the nested block loop join (which basically is a sequential scan optimized for the *k*-nn case) and the *k*-nn algorithm by Hjaltason and Samet [15] as a conventional, non-join technique.

All our experiments were carried out under Windows NT4.0 SP6 on Fujitsu-Siemens Celsius 400 machines equipped with a Pentium III 700 MHz processor and at least 128 MB main memory. 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. The synthetic data sets consisted of 4, 6 and 8 dimensions and contained from 10,000 to 160,000 uniformly distributed points in the unit hypercube. Our real-world data sets are a CAD database with 16-dimensional feature vectors extracted from CAD parts and a 9-dimensional set of weather data. We allowed about 20% of the database size as cache resp. buffer for either technique and included the index creation time for our *k*-nn join and the hs-algorithm, while the nested block loop join (nblj) does not need any preconstructed index.

The Euclidean distance was used to determine the k-nearest neighbor distance. In order to show the effects of varying the neighboring parameter k we include figure 7 with varying k (from 4-nn to 10-nn) while all other charts show results for the case of the 4-nearest neighbors. In figure 7 we can see, that except for the nested block loop join all techniques perform better for a smaller number of nearest neighbors and the hs-algorithm starts to perform worse than the nblj if more than 4 nearest neighbors are requested. This is a well known fact for high dimensional data as the pruning power of the directory pages deteriotates quickly with increasing dimension and parameter k. This is also true, but far less dramatic for the *k*-nn join because of the use of much smaller buckets which still perserve pruning power for higher dimensions and parameters k. The size of the database used for these experiments was 80,000 points.

The three charts in figure 8 show the results (from left to right) for the hs-algorithm, our *k*-nn join and the nblj for the 8-dimensional uniform data set for varying size of the database. The total elapsed time consists of the CPU-time and the I/O-time. We can observe that the hs-algorithm (despite using large block sizes for optimization) is clearly I/O bound while the nested block loop join is clearly CPU bound. Our *k*-nn join has a somewhat higher CPU cost than the hs-algorithm, but significantly less than the nblj while it produces almost as little I/O as nblj and as a result clearly outperforms both, the hs-algorithm and the nblj. This balance between CPU and I/O cost follows the idea of MuX to optimize CPU and I/O cost independently. For our artificial data the speed-up factor of the *k*-nn join over the hs-algorithm is 37.5 for the small point set (10,000 points) and 9.8 for the large point set (160,000 points), while compared to the nblj the speed-up factor increases from 7.1 to 19.4. We can also see, that the simple, but optimized nested block loop join outperforms the hs-algorithm for smaller database sizes because of its high I/O cost.

One interesting effect is, that our MUX-algorithm for *k*-nn joins is able to prune more and more bucket pairs with increasing size of

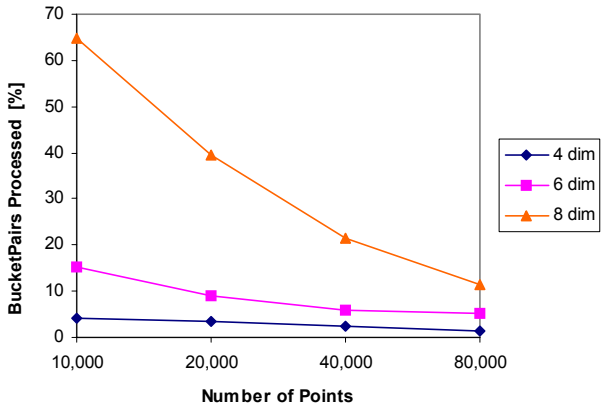



**Figure 9. Pruning of bucket pairs for the *k-nn* join**

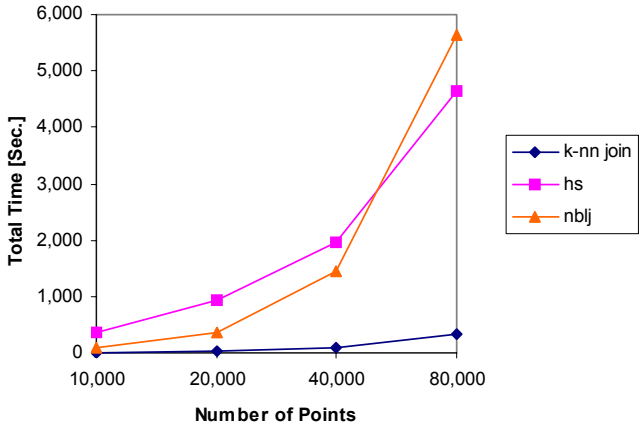



**Figure 10. Results for 9-dimensional weather data**

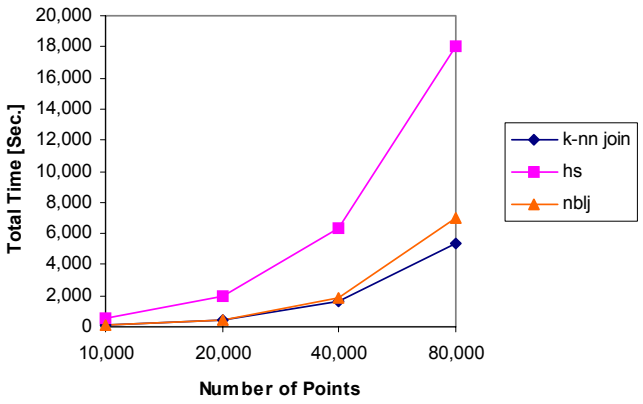



**Figure 11. Results for 16-dimensional CAD data**

the database i.e. the percentage of bucket pairs that can be excluded during processing increases with increasing database size.We can see this effect in figure 9. Obviously, the *k*-nn join scales much better with increasing size of the database than the other two techniques.

Figure 10 shows the results for the 9-dimensional weather data. The maximum speed-up of the *k*-nn join compared to the hs-algorithm is 28 and the maximum speed-up compared to the nested block loop join is 17. For small database sizes, the nested block loop join outperforms the hs-algorithm which might be due to the cache/buffer and I/O configuration used. Again, as with the artificial data, the *k*-nn join clearly outperforms the other techniques and scales well with the size of the database.

Figure 11 shows the results for the 16-dimensional CAD data. Even for this high dimension of the data space and the poor clustering property of the CAD data set, the *k*-nn join still reaches a speed-up factor of 1.3 for the 80,000 point set (with increasing tendency for growing database sizes) compared to the nested block loop join (which basically is a sequential scan optimized for the *k*-nn case). The speed-up factor of the *k*-nn join over the hs-algorithm is greater than 3.

## 6. Conclusions

In this paper, we have proposed an algorithm to efficiently compute the *k*-nearest neighbor join, a new kind of similarity join. In contrast to other types of similarity joins such as the distance range join, the *k*-distance join (*k*-closest pair query) and the incremental distance join, our new *k*-nn join combines each point of a point set *R* with its *k* nearest neighbors in another point set *S*. We have seen that the *k*-nn join can be a powerful database primitive which allows the efficient implementation of numerous methods of knowledge discovery and data mining such as classification, clustering, data cleansing, and postprocessing. Our algorithm for the efficient computation of the *k*-nn join uses the Multipage Index (MuX), a specialized index structure for similarity join processing and applies matching loading and processing strategies in order to reduce both CPU and I/O cost. Our experimental evaluation proves high performance gains compared to conventional methods.

## References


[1] 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.

[2] Agrawal R., Lin K., Sawhney H., Shim K.: *Fast Similarity Search in the Presence of Noise, Scaling, and Translation in Time-Series Databases*, Int. Conf on Very Large Data Bases (VLDB), 1995.

[3] 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.

[4] 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.

[5] Böhm C., Braunmüller B., Krebs F., Kriegel H.-P.: *Epsilon Grid Order: An Algorithm for the Similarity Join on Massive High-Dimensional Data*, ACM SIGMOD Int. Conf. on Management of Data, 2001.

[6] Berchtold S., Böhm C., Jagadish H. V., Kriegel H.-P., Sander J.: *Independent Quantization: An Index Compression Technique for High Dimensional Data Spaces*, IEEE Int. Conf. on Data Engineering (ICDE), 2000.

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

[8] Böhm C., Krebs F.: *The k-Nearest Neighbor Join: Turbo Charging the KDD Process*, submitted.

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

[10] Breunig M. M., Kriegel H.-P., Kröger P., Sander J.: *Data Bubbles: Quality Preserving Performance Boosting for Hierarchical Clustering*, ACM SIGMOD Int. Conf. on Management of Data, 2001.

[11] Böhm C.: *The Similarity Join: A Powerful Database Primitive for High Performance Data Mining*, tutorial, IEEE Int. Conf. on Data Engineering (ICDE), 2001.

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

[13] Corral A., Manolopoulos Y., Theodoridis Y., Vassilakopoulos M.: *Closest Pair Queries in Spatial Databases*, ACM SIGMOD Int. Conf. on Management of Data, 2000.

[14] 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.

[15] Hjaltason G. R., Samet H.: *Ranking in Spatial Databases*, Int. Symp. on Large Spatial Databases (SSD), 1995.

[16] Hjaltason G. R., Samet H.: *Incremental Distance Join Algorithms for Spatial Databases*, SIGMOD Int. Conf. on Management of Data, 1998.

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

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

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

[20] Lo M.-L., Ravishankar C. V.: *Spatial Joins Using Seeded Trees*, ACM SIGMOD Int. Conf., 1994.

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

[22] Patel J.M., DeWitt D.J., *Partition Based Spatial-Merge Join*, ACM SIGMOD Int. Conf., 1996.

[23] Preparata F. P., Shamos M. I.: *Computational Geometry*, Springer 1985.

[24] Roussopoulos N., Kelley S., Vincent F.: *Nearest Neighbor Queries*, ACM SIGMOD Int. Conf., 1995.

[25] 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.

[26] Shin H., Moon B., Lee S.: *Adaptive Multi-Stage Distance Join Processing*, ACM SIGMOD Int. Conf., 2000.

[27] Shim K., Srikant R., Agrawal R.: *High-Dimensional Similarity Joins*, IEEE Int. Conf. on Data Engineering, 1997.