DOI 10.1007/s10115-003-0122-9
Springer-Verlag London Ltd. © 2004
Knowledge and Information Systems (2004) 6: 728–749

**Knowledge and Information Systems**

# The *k*-Nearest Neighbour Join: Turbo Charging the KDD Process

Christian Böhm, Florian Krebs

University of Munich, Germany

**Abstract.** The similarity join has become an important database primitive for supporting similarity searches and data mining. A similarity join combines two sets of complex objects such that the result contains all pairs of similar objects. Two types of the similarity join are well-known, the distance range join, in which the user defines a distance threshold for the join, and the closest pair query or *k*-distance join, which retrieves the *k* most similar pairs. In this paper, we propose an important, third similarity join operation called the *k*-nearest neighbour join, which combines each point of one point set with its *k* nearest neighbours in the other set. We discover that many standard algorithms of Knowledge Discovery in Databases (KDD) such as *k*-means and *k*-medoid clustering, nearest neighbour 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 neighbour join using the multipage index (MuX), a specialised index structure for the similarity join. To reduce both CPU and I/O costs, we develop optimal loading and processing strategies.

**Keywords:** Data mining; Knowledge discovery in databases (KDD); Similarity join; Nearest neighbour; Multimedia database; High-dimensional indexing

## 1. Introduction

Knowledge Discovery in Databases (KDD) is the non-trivial process of identifying valid, novel, potentially useful, and ultimately understandable patterns in data (Fayyad et al. 1996). The KDD process (Brachmann and Anand 1996) is an interactive and iterative process, involving numerous steps, including preprocessing of the data (data cleansing) and postprocessing (evaluation of the results). The core step of the KDD process is *data mining*, i.e. finding patterns of interest, such as *clusters*, *outliers*, *classification rules* or *trees, association rules*, and *regressions*. KDD

*Received 9 December 2002*
*Revised 7 February 2003*
*Accepted 12 May 2003*
*Published online 27 February 2004*

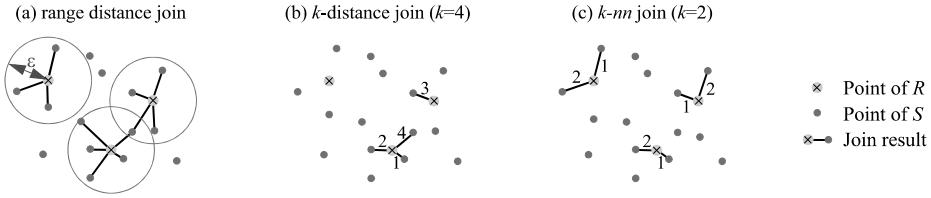



**Fig. 1.** Difference between similarity join operations.

algorithms in multidimensional databases are often based on similarity queries that are performed for a large number of objects. Recently, it has been recognised that many similarity search (Agrawal et al. 1995) and data mining (Böhm et al. 2000) algorithms can be based on top of a single join query instead of many similarity queries. Thus, a large 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 $R \bowtie_{\varepsilon} S$, which is defined for two finite sets of vectors, $R = \{r_1, \ldots, r_n\}$ and $S = \{s_1, \ldots, s_m\}$, as the set of all pairs from $R \times S$ having a distance of no more than $\varepsilon$:

$$R \bowtie_{\varepsilon} S := \{(r_i, s_j) \in R \times S | \parallel r_i - s_j \parallel \leq \varepsilon\}.$$

For example, in (Böhm et al. 2000), it has been shown that density-based clustering algorithms such as DBSCAN (Sander et al. 1998) or the hierarchical cluster analysis method OPTICS (Ankerst et al. 1999) 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. Shim et al. 1997; Koudas and Sevcik 1997; Böhm et al. 2001).

Another important similarity join operation which has been recently proposed is the incremental distance join (Hjaltason and Samet 1998). 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. Preparata and Shamos 1985)). This operation can be successfully applied to implement data analysis tasks such as noise-robust catalogue matching and noise-robust duplicate detection (Böhm 2001).

In this paper, we investigate a third kind of similarity join, the $k$-nearest neighbour similarity join, or $k$-nn join for short. This operation is motivated by the observation that many data analysis and data mining algorithms are based on $k$-nearest neighbour queries which are issued separately for a large set of *query points* $R = \{r_1, \ldots, r_n\}$ against another large set of *data points* $S = \{s_1, \ldots, s_m\}$. 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 neighbours in $S$. The differences between the three kinds of similarity join operations are depicted in Fig. 1.

Applications of the $k$-nn join include but are not limited to the following list: $k$-nearest neighbour classification, $k$-means clustering, sample assessment and sample postprocessing, missing value imputation, $k$-distance diagrams, etc. We discuss how $k$-means clustering, nearest neighbour classification, and various other algorithms can be transformed such that they operate exclusively on top of the $k$-nearest

neighbour join. This transformation typically leads to performance gains of 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 neighbour 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 neighbours. 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 linear, and 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 (Breunig et al. 2001). 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.

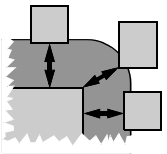
The remainder of this paper, which is an extended version of (Böhm and Krebs 2002), is organised as follows: In Sect. 2, we give a classification of the well-known similarity join operations and review related work. In Sect. 3, we define the new operation, the $k$-nearest neighbour join. Section 4 is dedicated to applications of the $k$-nn join. We show that typical data mining methods can be easily implemented on top of the join. In Sect. 5, we develop an algorithm for the $k$-nn join which applies suitable loading and processing strategies on top of the multipage index (Böhm and Kriegel 2001), an index structure which is particularly suited to high-dimensional similarity joins, in order to reduce both CPU and I/O costs and efficiently compute the $k$-nn join. The experimental evaluation of our approach is presented in Sect. 6, and Sect. 7 concludes the paper.

## 2. Related Work

In the relational data model, a join means combining 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 Fig. 1a, the distance range join $R \overset{\varepsilon}{\bowtie} S$ of two multidimensional or metric sets $R$ and $S$ is the set of pairs for which the distance between the objects does not exceed the given parameter $\varepsilon$:



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

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

**Definition 1.** Distance Range Join ($\varepsilon$-Join)
The distance range join $R \bowtie_\varepsilon S$ of two finite multidimensional or metric sets $R$ and $S$ is the set

$$R \bowtie_\varepsilon S := \{(r_i, s_j) \in R \times S : \| r_i - s_j \| \le \varepsilon\}.$$

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

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

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 $\varepsilon$-neighbourhood of some data object. This essentially corresponds to a self-join using the distance range paradigm.

As for plain range queries in multimedia databases, a general problem of distance range joins from the user's point of view is that it is difficult to control the result cardinality of this operation. If $\varepsilon$ is chosen too small, no pairs are reported in the result set (or in the case of a self join: each point is only combined with itself). In contrast, if $\varepsilon$ 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 $\varepsilon$-values for which 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 $\varepsilon$ is chosen in a suitable range and is also adapted to 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)* (Brinkhoff et al. 1993), 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 (see Fig. 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 $\varepsilon$. For these pairs of child pages, the algorithm is called recursively, i.e. the corresponding indexes are traversed in a depth-first order. Various optimizations of RSJ have been proposed, such as the *BFRJ algorithm* (Huang et al. 1997), which traverses the indexes according to a breadth-first strategy.

Recently, index-based similarity join methods have been analysed from a theoretical point of view. Böhm and Kriegel (2001) proposed a cost model based on the concept of the Minkowski sum (Berchtold et al. 1997) which can be used for optimizations such as page size optimisation. The analysis reveals a serious optimisation 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. Optimising for CPU deteriorates the I/O performance and vice versa. The consequence is that an index architecture is necessary which allows a separate optimisation of CPU and I/O operations. Therefore, the authors proposed the *Multipage Index* (*MuX*), a complex index structure with large pages (optimised for I/O) which accommodate a secondary search structure (optimised for maximum CPU efficiency). It was shown that the resulting index yields an I/O performance which is similar to the I/O-optimised R-tree similarity join and a CPU performance which is close to the CPU-optimised 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 have been proposed (Kamel and Faloutsos 1994; van den Bercken et al. 1997). The *seeded tree method* (Lo and Ravishankar 1994) 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* (Lo and Ravishankar 1994; Patel and DeWitt 1996) decomposes the set $R$ into a number of partitions which is determined according to given system parameters.

A join algorithm particularly suited to similarity self joins is the *ε-kdB-tree* (Shim et al. 1997). The basic idea is to partition the data set perpendicularly to one selected dimension into stripes of width $\varepsilon$ to restrict the join to pairs of subsequent stripes. To speed up the CPU operations, for each stripe a main memory data structure, the $\varepsilon$-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 $\varepsilon$. Finally, a tree-matching algorithm is applied which is restricted to neighbouring stripes. Koudas and Sevcik have proposed the *Size Separation Spatial Join* (Koudas and Sevcik 1997) and the *Multidimensional Spatial Join* (Koudas and Sevcik 1998), 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)* (Böhm et al. 2001). It is based on a particular sort order of the data points which is obtained by laying an equidistant grid with cell length $\varepsilon$ 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 potentially retrieve 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 (see Fig. 1b). Closest pair queries do not only play an important role in the database research but also have a long history in computational geometry (Preparata and Shamos 1985). In the database context, the operation was introduced by Hjaltason and Samet (Hjal-

tason and Samet 1998) 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:

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

This definition directly corresponds to the definition of ($k$-)nearest neighbour 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. Hjaltason and Samet (1998) defined 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 trivially have 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 self join, the closest pair self join 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 (1998) also defined 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 distance semijoin, are evaluated using a pqueue data structure in which node-pairs are ordered by increasing distance.

The most interesting challenge in algorithms for the distance join is the strategy for accessing pages and forming page pairs. Analogously to the various strategies for single nearest neighbour queries such as those of Roussopoulos et al. (1995) and Hjaltason and Samet (1995), Corral et al. (2000) proposed 5 different strategies including recursive algorithms and an algorithm based on a pqueue. Shin et al. (2000) proposed a plane sweep algorithm for the node expansion for the above-mentioned pqueue algorithm. In the same paper, Shim et al. also proposed 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, in which the result set size (up to the rare tie effects) is given by the query parameter $k$.

However, there are only a few applications which require the consideration of the $k$ best pairs of two sets. Much more prevalent are applications such as classification or clustering in which 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 neighbour similarity join (see Fig. 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:

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

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 neighbours of several points in $R$), or not at all (if a point does not belong to the $k$-nearest neighbours 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 neighbour 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 "per point of the first set" basis. For each of the points of $R$, the $k$ best join partners are searched. This is an essential difference of concepts.

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

## 4. Applications

### 4.1. $k$-Means and $k$-Medoid Clustering

The $k$-means method (cf. Han and Kamber 2000) is the most important and most widespread approach to *clustering*. For $k$-means clustering the number $k$ of clusters to be searched must be previously known. The method determines $k$ cluster centres such that each database point can be assigned to one of the centres to minimise the overall distance of the database points to their associated centre points.

The basic algorithm for $k$-means clustering works as follows: In the initialisation, $k$ database points are randomly selected as tentative cluster centres. Then, each

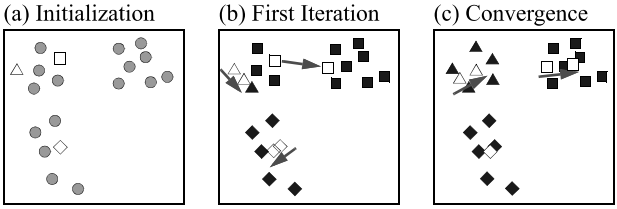



**Fig. 3.** *k*-means clustering.

database point is associated with its closest centre point and, thus, a tentative *cluster* is formed. Next, the cluster centres are redetermined as the mean point of all points of the centre, simply by forming the vector sum of all points of a (tentative) cluster. The two steps of (1) point association and (2) cluster centre redetermination are repeated until convergence (no more considerable change). It has been shown that (under several restrictions) the algorithm always converges. The cluster centres which are generated in step (2) are artificial points rather than database points. This is often not desired, and, therefore, the *k*-medoid algorithm always selects a database point as a cluster centre.

The *k*-means algorithm is visualised in Fig. 3 using $k = 3$. In Fig. 3a, $k = 3$ points (white symbols ◇□△) are randomly selected as initial cluster centres. Then, in Fig. 3b, the remaining data points are assigned to the closest centre, which is depicted by the corresponding symbols (◆■▲). The cluster centres are redetermined (moving arrows). The same two operations are repeated in Fig. 3c. If the points are finally assigned to their closest centre, no assignment changes, and, therefore, the algorithm terminates, clearly having separated the three visible clusters. In contrast to density-based approaches, *k*-means only separates compact clusters, and the number of actual clusters must be previously known.

It has not yet been recognised in the data mining community that the point association step which is performed in each iteration of the algorithm corresponds to a ($k = 1$) nearest neighbour join between the set of centre points (on the right side) and the set of database points (on the left side of the join symbol) because each database point is associated with its nearest neighbour among the centre points:

$$\text{database-point-set} \underset{\text{1-NN}}{\ltimes} \text{center-point-set}.$$

During the iteration over the cursor of the join, it is also possible to keep track of changes and to redetermine the cluster center for the next iteration. The corresponding pseudocode is depicted in the following:

```
repeat
  change := false ;
  foreach (dp,cp) ∈ database-point-set ⋉ 1-NN centre-point-set do
    if dp.centre ≠ cp.id then change := true ;
    dp.centre := cp.id ;
    cp.newsum := cp.newsum + dp.point ;
    cp.count := cp.count + 1 ;
  foreach cp ∈ centre-point-set do
    cp.point := cp.newsum / cp.count ;
until ¬ change ;
```

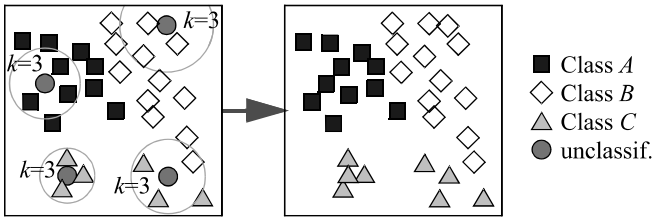



**Fig. 4.** $k$-nearest neighbour classification.

## 4.2. $k$-Nearest Neighbour Classification

Another very important data mining task is classification. Classification is somewhat similar to clustering (which is often called *unsupervised classification*). In classification, a part of the database objects is assigned to class labels (for our example of astronomy databases we have different classes of stars, galaxies, planets, etc.). For classification, a set of objects without class label (newly detected objects) is also given. The task is to determine the class labels for each of the unclassified objects by taking the properties of the classified objects into account. A widespread approach is to build up tree-like structures from the classified objects where the nodes correspond to ranges of attribute values and the leaves indicate the class labels (called *classification trees* (cf. Han and Kamber 2000)). Another important approach is $k$-nearest neighbour classification (Hattori and Torii 1993). Here, for each unclassified object, a $k$-nearest neighbour query on the set of classified objects is evaluated ($k$ is a parameter of the algorithm). The object is, e.g., assigned to the class label of the majority of the resulting objects of the query. This principle is visualised in Fig. 4. As, for each unclassified object, a $k$-nn-query on the set of classified objects is evaluated, this corresponds again to a $k$-nearest neighbour join:

$$\text{unclassified-point-set} \underset{k\text{-}nn}{\bowtie} \text{classified-point-set}\,.$$

## 4.3. Sampling-Based Data Mining

Data mining methods which are based on sampling often require a $k$-nearest neighbour join between the set of sample points and the complete set of original database points. Such a join is necessary, for instance, to assess the quality of a sample. The $k$-nearest neighbour join can give hints on whether the sample rate is too small. Another application is the transfer of the data mining result onto the original data set after the actual run of the data mining algorithm (Breunig et al. 2001). For instance, if a clustering algorithm has detected a set of clusters in the sample set, it is often necessary to associate each of the database points to the cluster to which it belongs. This can be done by a $k$-nn join with $k = 1$ between the point set and the set of sample points:

$$\text{sample-set} \underset{k\text{-}nn}{\bowtie} \text{point-set}\,.$$

The same is possible after sample based classification, trend detection, etc.

## 4.4. $k$-Distance Diagrams

The most important limitation of the DBSCAN algorithm is the difficult determination of the query radius $\varepsilon$. Sander et al. (1998) proposed a method called the $k$-

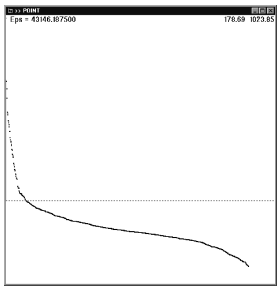


**Fig. 5.** *k*-distance diagram.

distance diagram to determine a suitable radius $\varepsilon$. For this purpose, a number of objects (typically 5–20 percent of the database) is randomly selected. For these objects, a *k*-nearest neighbour query is evaluated, where *k* corresponds to the parameter MIN_PTS which will be used during the run of DBSCAN. The resulting distances between the query points and the *k*-th nearest neighbour of each are then sorted and depicted in a diagram (see Fig. 5). Vertical gaps in that plot indicate distances that clearly separate different clusters, because there exist larger *k*-nearest neighbour distances (inter-cluster distances, noise points) and smaller ones (intra-cluster distance). As for each sample point, a *k*-nearest neighbour query is evaluated on the original point set; this corresponds to a *k*-nn join between the sample set and the original set:

$$\text{sample-set} \underset{k\text{-}nn}{\ltimes} \text{point-set}\,.$$

If the complete data set is taken instead of the sample, we have a *k*-nn self join:

$$\text{point-set} \underset{k\text{-}nn}{\ltimes} \text{point-set}\,.$$

## 5. 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 (Böhm and Kriegel 2001), to efficiently compute the *k*-nn join. We have shown for the distance range join that it is necessary to optimise 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 Fig. 6, which allows such a separate optimisation. The index consists of large pages which are optimised for I/O efficiency. These pages accommodate a secondary R-tree like a 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 optimised 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 that is purely I/O optimised and has a CPU performance like an R-tree that is purely CPU optimised. Although this issue is up to future work, we assume that the *k*-nn join also clearly benefits from the separate optimisation (because optimisation trade-offs are very similar).

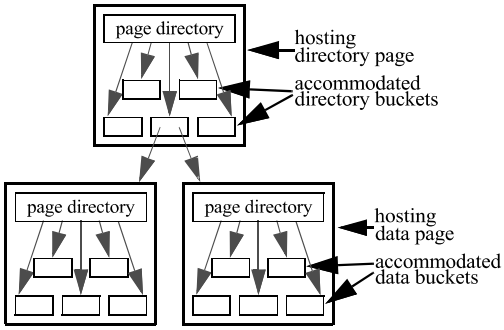



**Fig. 6.** Index architecture of the multipage index.


**Fig. 7.** The fast index scan for single range queries (l.) and for single nearest neighbour queries (r.).

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 costs arise in the data level. Therefore, our strategies focus on the last level.

## 5.1. The Fast Index Scan

In our previous work (Berchtold et al. 2000), we 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 neighbour 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 the disk can be immediately grouped together into a single I/O request (see Fig. 7, left side). But pages which are not direct neighbours but are only close together can also 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 neighbour queries the trade-off is more complex: These are usually evaluated by the HS algorithm (Hjaltason and Samet 1995), 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 stochastic model to estimate the probability of a page being required for a given nearest neighbour query. Based on this model we can estimate the cost for various chained and unchained I/O requests and thus optimise the I/O operations (see Fig. 7, right side).

Taking a closer look at the trade-off which is exploited in our optimisation: If we apply no I/O chaining or overcautious I/O 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 neighbour 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.

## 5.2. Optimisation Goals of the Nearest Neighbour Join

In brief, the trade-off of the nearest neighbour search is between (1) getting the nearest neighbour *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 neighbour join. One important goal of the algorithm is to get a good approximation of the nearest neighbour (i.e. a point which is not necessarily *the* nearest neighbour but a point which is not much worse than the nearest neighbour) for each of these active queries as early as possible. With a good conservative approximation of the *nearest neighbour distance*, we can even abstain from our probabilistic model of the previous paragraph and handle nearest neighbour queries further on, as in range queries. Only a few pages are processed too much.

In contrast to single similarity queries, the seek cost does not play an important role in our join algorithm because our special index structure, MuX, is optimised 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 the 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 neighbour distance
- It is improbable that the corresponding mate page will receive any improvements of its nearest neighbour 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 for which it is most unclear whether or not they will be needed in the future.

## 5.3. 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 neighbours are searched as the outer point set. Consequently, $S$ is the inner point set. As in Böhm and Kriegel (2001) 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 neighbour join is illustrated in Fig. 8. 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 neighbours is allocated. Candidate points are maintained

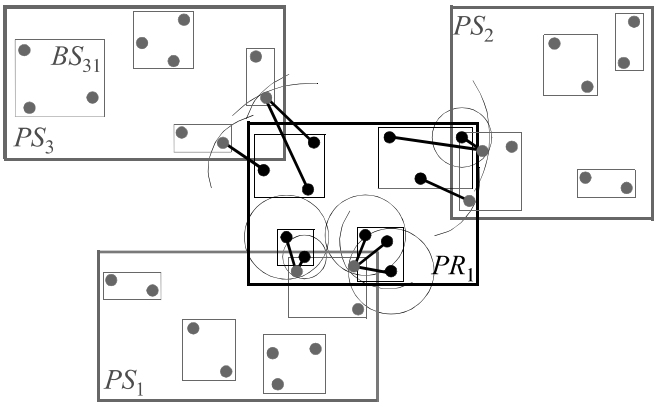



**Fig. 8.** $k$-nn join on the multipage index (here $k = 1$).

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 neighbours 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 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 neighbours, not necessarily the nearest neighbours) 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:

```
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: Line 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 neighbours (and the corresponding candidates) is allocated and initialised 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 of which hosting page to load next is implemented in the so-called loading strategy which is described in Sect. 5.4. 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 Sect. 5.5. 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.

## 5.4. Loading Strategy

In conventional similarity searches, in which the nearest neighbour 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 (Berchtold et al. 1997). For our *k*-nn *join*, we are simultaneously processing nearest neighbour 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 which 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 optimisation 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 already* has 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, tends to avoid the same bucket *BR* being 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

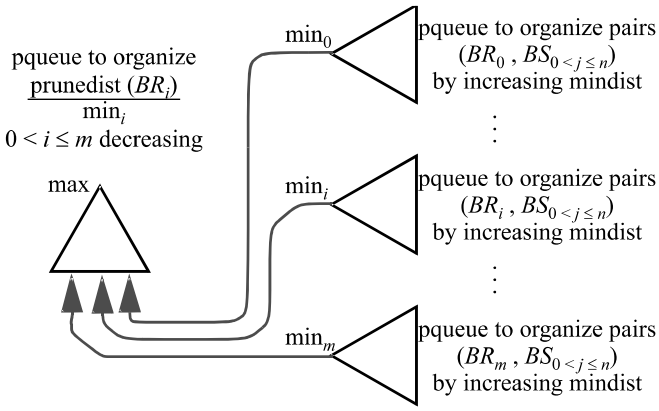



**Fig. 9.** Structure of a fractionated pqueue.

related to the current pruning distance of the buckets:

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

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

## 5.5. Processing Strategy

The processing strategy is applied in line 10. It addresses the question of the order in which 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 $\mathrm{Q}(PS)$ of a page developed in Sect. 5.4, the quality $\mathrm{Q}(BR,BS)$ of a bucket pair rewards a small distance and punishes a small pruning distance:

$$Q(BR,BS) = \frac{\mathrm{prunedist}(BR)}{\mathrm{mindist}(BS, BR)} . \tag{4}$$

We process the bucket pairs in 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 organise 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 Fig. 9. Note that this tailor-cut structure efficiently allows us (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.

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 for which 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).

## 6. Experimental Evaluation

### 6.1. Join Algorithm and Strategies

We implemented the *k*-nearest neighbour join algorithm, as described in the previous section, based on the original source code of the Multipage Index Join (Böhm and Kriegel 2001) and performed an experimental evaluation using artificial and real data sets of varying size and dimension. We compared the performance of our technique with the nested block loop join (which basically is a sequential scan optimised for the *k*-nn case) and the *k*-nn algorithm by Hjaltason and Samet (1995) 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 Pentium III 700 MHz processors and at least 128 MB of 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 were 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 both techniques 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 neighbour distance. In order to show the effects of varying the neighbouring parameter *k*, we include Fig. 10 with varying *k* (from 4-nn to 10-nn), while all other charts show results for the case of 4 nearest neighbours. In Fig. 10, we can see that, except for the nested block loop join, all techniques perform better for a smaller number of nearest neighbours and the HS algorithm starts to perform worse than the nblj if more than 4 nearest neighbours 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 preserves 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 Fig. 11 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 optimisation) 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

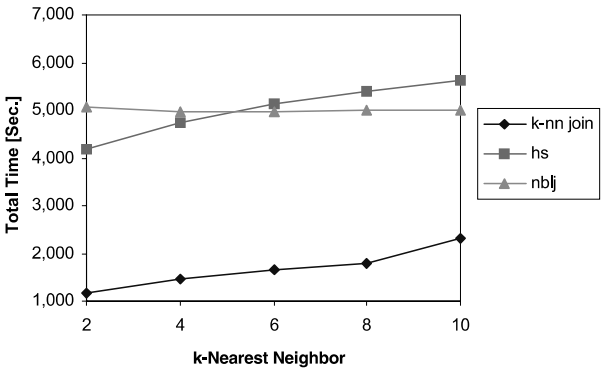



**Fig. 10.** Varying $k$ for the 8-dimensional uniform data.

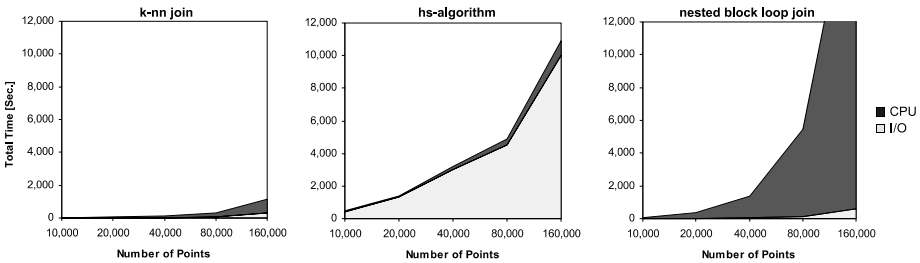



**Fig. 11.** Total time, CPU time, and I/O time for HS, $k$-nn join, and nblj for varying size of the database.

significantly less than the nblj, while it produces almost as little I/O as the nblj and as a result clearly outperforms both the HS algorithm and the nblj. This balance between the CPU and I/O costs follows the idea of MuX in optimising the CPU and I/O costs 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 with the nblj, the speed-up factor increases from 7.1 to 19.4. We can also see that the simple but optimised 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 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 Fig. 12. Obviously, the $k$-nn join scales much better with increasing size of the database than the other two techniques.

Figure 13 shows the results for the 9-dimensional weather data. The maximum speed-up of the $k$-nn join compared with the HS algorithm is 28 and the maximum speed-up compared with 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 14 shows the results for the 16-dimensional CAD data. Even for the high dimensions of the data space and the poor clustering property of the CAD data

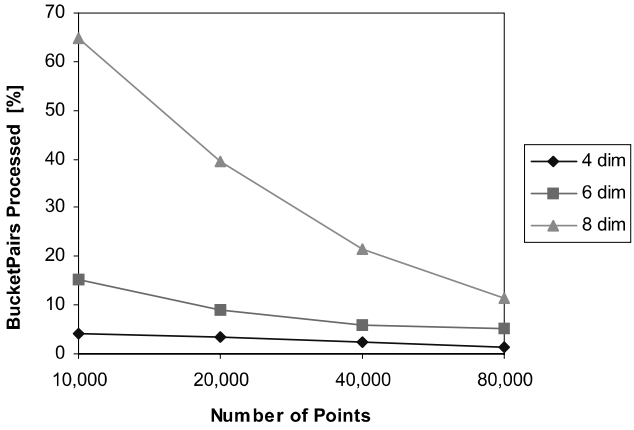



**Fig. 12.** Pruning of bucket pairs for the *k*-nn join.

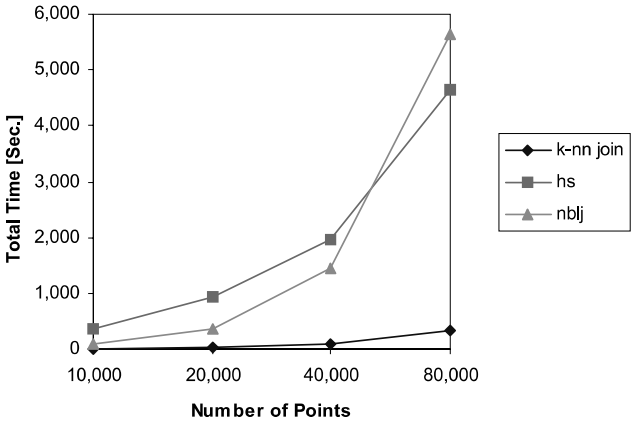



**Fig. 13.** Results for the 9-dimensional weather data.

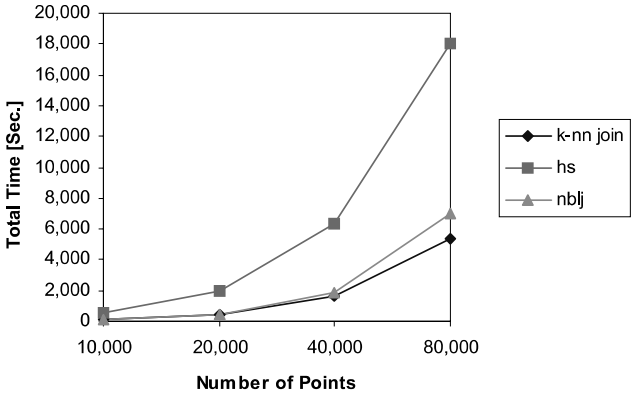



**Fig. 14.** Results for the 16-dimensional CAD data.

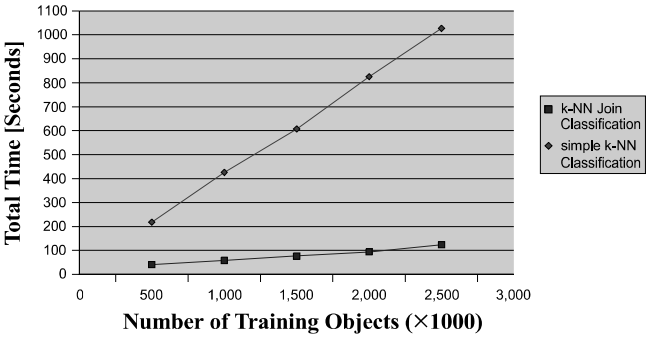



**Fig. 15.** Classification: 5-D SEQUOIA, 1000 classif. obj.

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 with the nested block loop join (which basically is a sequential scan optimised for the $k$-nn case). The speed-up factor of the $k$-nn join over the HS algorithm is greater than 3.

## 6.2. Integration into KDD Methods

We implemented a $k$-means clustering algorithm and a $k$-nearest neighbour classification algorithm in both versions traditionally with single similarity queries (nearest neighbour queries) as well as on top of our new database primitive, the similarity join. The competitive technique, the evaluation on top of single similarity queries, was also supported by the same index structure which is traversed using a variation of the nearest neighbour algorithm by Hjaltason and Samet (1995), which has been shown by Berchtold et al. (1997) to yield an optimal number of page accesses.

We allowed about 20% of the database size as cache or buffer for both techniques and included the index creation time for our $k$-nn join and the HS algorithm. We used large data sets from various application domains, in particular:

- 5-dimensional feature vectors from earth observation. These data sets were generated from the well known SEQUOIA benchmark
- 9-dimensional feature vectors from a meteorology application as in Sect. 6.1
- 16-dimensional feature vectors from a similarity search system for CAD parts as in Sect. 6.1
- 20-dimensional data from astronomy observations
- 64-dimensional feature vectors from a colour image database (colour histograms)

In our first set of experiments, we tested the $k$-nearest neighbour classification method in which we varied the number of training objects (see Fig. 15) as well as the number of objects which have to be classified (see Fig. 16). The superiority of our new method becomes immediately clear from both experiments. The improvement factor over the simple $k$-nn approach is high over all measured scales. It even improves for an increasing number of training objects or classified objects, respectively, and reaches a final factor of 9.1 in Fig. 15 (factor 2.0 in Fig. 16).

Figure 17 varies over our various data sets and shows that the improvement factor also grows with increasing data space dimension. Our new database primitive outperforms the well-known approach by factors starting with 1.8 for the 5-dimensional space up to 3.2 for the 64-dimensional space.

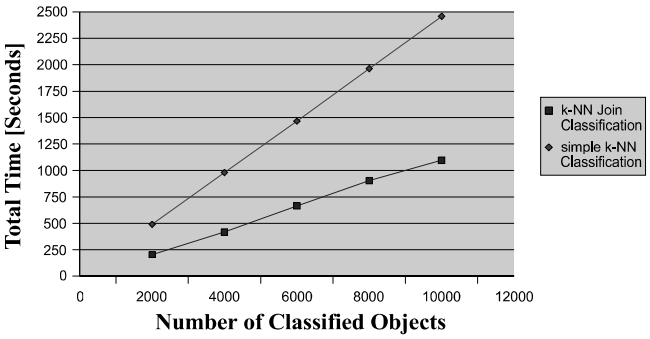



**Fig. 16.** Classification: 64-D image, 100 000 training.

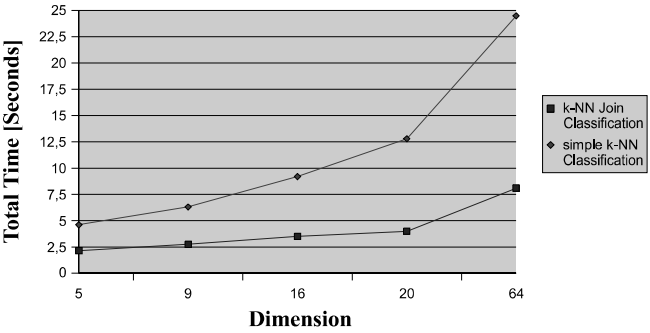



**Fig. 17.** Classification: 100 000 train. 100 classif.

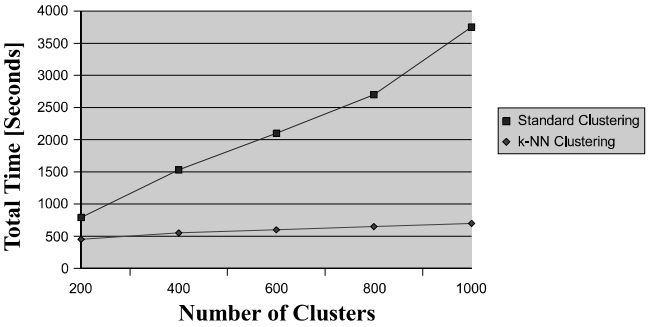



**Fig. 18.** Classification: 5-D SEQUOIA, 100 000 objects.

In our last experiment, depicted in Fig. 18, we tested the *k*-nearest neighbour clustering method. In the depicted experiment, we varied the number of clusters to be searched. Again, the improvement factor grows from 1.4 for the smallest number of clusters to 5.1 for the largest number of clusters.

## 7. Conclusions

In this paper, we have proposed an algorithm to efficiently compute the $k$-nearest neighbour 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 neighbours in another point set $S$. We have seen that the $k$-nn join can be a powerful database primitive that 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 specialised index structure for similarity join processing and applies matching loading and processing strategies in order to reduce both CPU and I/O costs. Our experimental evaluation demonstrates high performance gains compared with conventional methods.

## References

Agrawal R, Lin K, Sawhney H, Shim K (1995) Fast similarity search in the presence of noise, scaling, and translation in time-series databases. Int Conf on Very Large Data Bases (VLDB)

Ankerst M, Breunig MM, Kriegel H-P, Sander J (1999) OPTICS: ordering points to identify the clustering structure. ACM SIGMOD Int Conf on Management of Data

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

Berchtold S, Böhm C, Keim D, Kriegel H-P (1997) A cost model for nearest neighbor search in high-dimensional data space. ACM Symposium on Principles of Database Systems (PODS)

Böhm C (2001) The similarity join: a powerful database primitive for high performance data mining, tutorial. IEEE Int Conf on Data Engineering (ICDE)

Böhm C, Braunmüller B, Breunig MM, Kriegel H-P (2000) Fast clustering based on high-dimensional similarity joins. Int Conf on Information Knowledge Management (CIKM)

Böhm C, Braunmüller B, Krebs F, Kriegel H-P (2001) Epsilon grid order: an algorithm for the similarity join on massive high-dimensional data. ACM SIGMOD Int Conf on Management of Data

Böhm C, Krebs F (2002) High performance data mining using the nearest neighbor join. IEEE Int Conf on Data Mining (ICDM)

Böhm C, Krebs F (2003) Supporting KDD applications by the k-nearest neighbor join. Int Conf on Database and Expert Systems Applications (DEXA)

Böhm C, Kriegel H-P (2001) A cost model and index architecture for the similarity join. IEEE Int Conf on Data Engineering (ICDE)

Brachmann R, Anand T (1996) The process of knowledge discovery in databases. In: Fayyad et al (eds) Advances in Knowledge Discovery and Data Mining, AAAI Press

Breunig MM, Kriegel H-P, Kröger P, Sander J (2001) Data bubbles: quality preserving performance boosting for hierarchical clustering. ACM SIGMOD Int Conf on Management of Data

Brinkhoff T, Kriegel H-P, Seeger B (1993) Efficient processing of spatial joins using R-trees. ACM SIGMOD Int Conf Management of Data

Corral A, Manolopoulos Y, Theodoridis Y, Vassilakopoulos M (2000) Closest pair queries in spatial databases. ACM SIGMOD Int Conf on Management of Data

Fayyad UM, Piatetsky-Shapiro G, Smyth P (1996) From data mining to knowledge discovery: an overview. In: Fayyad UM, Piatetsky-Shapiro G, Smyth P, Uthurusamy R (eds) Advances in knowledge discovery and data mining. AAAI/MIT Press, Menlo Park, CA

Han J, Kamber M (2000) Data Mining: Concepts and Techniques. Morgan Kaufmann, San Francisco, CA

Hattori K, Torii Y (1993) Effective algorithms for the nearest neighbor method in the clustering problem. Pattern Recognit 26(5)

Hjaltason GR, Samet H (1995) Ranking in spatial databases. Int Symp on Large Spatial Databases (SSD)

Hjaltason GR, Samet H (1998) Incremental distance join algorithms for spatial databases. SIGMOD Int Conf on Management of Data

Huang Y-W, Jing N, Rundensteiner EA (1997) Spatial joins using R-trees: breadth-first traversal with global optimizations. Int Conf on Very Large Databases (VLDB)

Kamel I, Faloutsos C (1994) Hilbert R-tree: an improved R-tree using fractals. Int Conf on Very Large Databases

Koudas N, Sevcik K (1997) Size separation spatial join. ACM SIGMOD Int Conf on Management of Data

Koudas N, Sevcik K (1998) High dimensional similarity joins: algorithms and performance evaluation. IEEE Int Conf on Data Engineering (ICDE), Best Paper Award

Lo M-L, Ravishankar CV (1994) Spatial joins using seeded trees. ACM SIGMOD Int Conf

Lo M-L, Ravishankar CV (1996) Spatial hash joins. ACM SIGMOD Int Conf on Management of Data

Patel JM, DeWitt DJ (1996) Partition based spatial-merge join. ACM SIGMOD Int Conf

Preparata FP, Shamos MI (1985) Computational Geometry. Springer

Roussopoulos N, Kelley S, Vincent F (1995) Nearest neighbor queries. ACM SIGMOD Int Conf

Sander J, Ester M, Kriegel H-P, Xu X (1998) Density-based clustering in spatial databases: the algorithm GDBSCAN and its applications. Data Mining and Knowledge Discovery 2(2). Kluwer Academic Publishers

Shin H, Moon B, Lee S (2000) Adaptive multi-stage distance join processing. ACM SIGMOD Int Conf

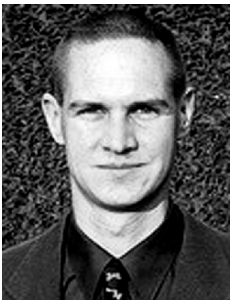
Shim K, Srikant R, Agrawal R (1997) High-dimensional similarity joins. IEEE Int Conf on Data Engineering

Ullman JD (1989) Database and Knowledge-Base Systems, Vol II. Computer Science Press, Rockville

van den Bercken J, Seeger B, Widmayer P (1997) A general approach to bulk loading multidimensional index structures. Int Conf on Very Large Databases

## Author Biographies

**Christian Böhm** is an associate professor of computer science at the University of Munich, Germany. From 1988 to 1994, he studied computer science at the Technical University Munich. He finished his PhD thesis in 1998 and his habilitation thesis in 2001 at the Ludwig Maximilians University in Munich, Germany. From 2002 to 2003, he was associate professor at the University for Health Informatics and Technology Tyrol (UMIT) in Innsbruck, Austria. His research interests are content-based similarity searches, knowledge discovery in databases, cost modelling, high-dimensional and metric spaces, as well as database applications of biomedical computer science such as genomics, proteomics, and metabolomics. Böhm has approximately 40 publications in international journals and reviewed conference proceedings.



**Florian Krebs** studied computer science at the Ludwig Maximilians University in Munich, Germany, from 1994 to 2000. Since 2002, he has been leader of the software engineering department at Symplex GmbH, Munich, Germany. He is working on his PhD thesis at the University of Munich. His major research interest is the mutual dependency of high-dimensional indexing and data mining methods.

*Correspondence and offprint requests to*: Christian Böhm, University of Munich, Oettingenstr. 67, 80538 München, Germany. Email: boehm@informatik.uni-muenchen.de