Powerful Database Support for High Performance Data Mining

Habilitationsschrift im Fach Informatik
an der Fakultät für Mathematik und Informatik
der Ludwig-Maximilians-Universität München

von

Christian Böhm

2001

Powerful Database Support for High Performance Data Mining

Habilitationsschrift im Fach Informatik
an der Fakultät für Mathematik und Informatik
der Ludwig-Maximilians-Universität München

von

Christian Böhm

Tag der Einreichung:

Berichterstatter:
Professor Dr. Hans-Peter Kriegel, Ludwig-Maximilians-Universität München
Professor Dr. Stefan Conrad, Ludwig-Maximilians-Universität München
Professor Dr. Gerhard Weikum, Universität des Saarlandes

Acknowledgments

I would like to express my thanks to all people who supported me during the past years while I have been working on this thesis. My warmest thanks go to Professor Dr. Hans-Peter Kriegel. He took particular care to maintain a good working atmosphere within the group and to provide a supportive and inspiring environment. I am grateful to Professor Dr. Stefan Conrad and to Professor Dr. Gerhard Weikum from the University of Saarland who both were readily willing to act as referees to this work. This work could not have grown and matured without the discussions with my colleagues. In particular I would like to mention here Dr. Bernhard Braunmüller and Florian Krebs. Most of my publications have been done in a close collaboration with them. But also all my other colleagues from the database systems group of the University of Munich had important contributions to my work by fruitful discussions and joint publications. I would like to thank Evguenia Altareva, Dr. Mihael Ankerst, Dr. Markus Breunig, Dr. Martin Ester, Matthias Groß, Karin Kailing, Professor Dr. Daniel Keim, Martin Pfeifle, Dr. Marco Pötke, Professor Dr. Jörg Sander, Stefan Schönauer, Matthias Schubert, and Dr. Thomas Seidl. Dr. Stefan Berchtold gave me the opportunity to work as a consultant in his startup company stb ag, Augsburg. During a research visit to the AT&T research lab in Florham Park, NJ, USA, I cooperated with Professor H. V. Jagadish, Ph.D. I would like to thank all my students who supported my work: Johann Dedie, Michael Euringer, Gerald Klump, Sven Messfeld, Urs Michel, Wolfgang Mühlbauer, and Gert Unterhofer. This work could not have been completed without the administrative help of Susanne Grienberger and the technical support of Franz Krojer. Last, but not least, I want to thank my parents, all my friends and especially my girlfriend Bianca.

Christian Böhm
Munich, 2001.

Abstract

Larger and larger amounts of data are collected and stored in databases, increasing the need for efficient and effective analysis methods to make use of the information contained implicitly in the data. The extraction of such potentially useful information is called data mining.

In the thesis, it is shown that numerous data mining methods such as density based clustering, k-means clustering, outlier detection, or k-nearest neighbor classification can be based on the similarity join as a database primitive. By such a reformulation, the identical result can be achieved at a drastically improved efficiency.

The similarity join becomes an important basic operation of advanced database management systems. For a given set of feature vectors, the similarity join determines those object pairs which are similar according to some appropriate similarity measure, in SQL style

$$\text{SELECT * FROM } R, S \text{ WHERE distance } (R.\text{point}, S.\text{point}) \leq \varepsilon.$$

In this thesis, we concentrate on both aspects of the similarity join applications as well as algorithms. For the first aspect we show how typical algorithms of data analysis and data mining can be reformulated such that they are exclusively based on the similarity join. According to several example applications, we demonstrate the enormous performance potential of this database primitive.

We also introduce several different kinds of similarity join. The most important variant which corresponds to the SQL statement above is based on the range search as a join predicate. But we also introduce two similarity join operations which are based on the paradigm of the *k*-nearest neighbor search.

The main part of the thesis is dedicated to the efficient algorithms for the different kinds of similarity join. First we introduce a new cost model for index based similarity join algorithms. Starting from this cost model, we develop an innovative index architecture which takes into account that similarity join algorithms require a separate optimization of CPU and I/O cost.

Next, we develop a similarity join algorithm which is particularly suited for massive data sets. It is based on a particular sort order for high dimensional data. Then we propose a novel algorithm for the similarity join upon a nearest neighbor join condition. Finally, we present a technique for the reduction of CPU cost which is universally applicable in index based and non-index-based similarity join methods.

A perspective on future research directions in the area of database primitives for similarity search, data analysis, and data mining concludes our thesis.

Abstract (In German)

Immer größere Datenmengen werden gesammelt und in Datenbanken gespeichert. Hierdurch wird die Notwendigkeit nach effektiven und effizienten Analysemethoden, die das in den Daten implizit vorhandene Wissen nutzbar machen, immer dringender. Die Extraktion von solchen potentiell nützlichen Informationen bezeichnet man als Data Mining.

In dieser Arbeit wird gezeigt, daß sich zahlreiche Data-Mining-Verfahren wie z.B. dichtebasiertes Clustering, die Ermittlung von Ausreißern oder die simultane Klassifikation auf der Basis des Similarity Join (Ähnlichkeitsverbund) als Datenbank-Grundoperation abstützen lassen. Es wird nachgewiesen, daß sich durch eine entsprechende Umformulierung dieser Verfahren das identische Ergebnis mit einer deutlich erhöhten Effizienz erzielen läßt.

Der Similarity Join wird hierdurch zu einer wichtigen Basisoperation von Multimedia-Datenbanksystemen. Innerhalb einer Menge von Multimedia-Objekten ermittelt der Similarity Join diejenigen Objekt-Paare, die einander bezüglich eines featurebasierten Abstandsmaßes am ähnlichsten sind, in SQL-Notation

SELECT * FROM *R*, *S* WHERE distance (*R*.point, *S*.point) $\leq \varepsilon$.

Diese Arbeit konzentriert sich gleichermaßen auf beide Aspekte des Similarity Join, Anwendungen sowie Algorithmen. Für den Anwendungs-Aspekt wird gezeigt, wie typische Aufgaben der Datenanalyse und des Data Mining mit Hilfe des Similarity Join gelöst werden können. Es wird aufgezeigt, daß typische Standard-Algorithmen dieser Bereiche reformuliert werden können so daß sie ausschließlich auf dem Similarity Join aufsetzen. Anhand verschiedener Beispielanwendungen wird das enorme Effizienzpotenzial demonstriert, das sich durch die Verwendung dieser Datenbank-Grundoperation ergibt.

In dieser Arbeit beschäftigen wir uns mit verschiedenen Arten von Similarity-Join-Operationen. Die wichtigste Variante, die auch dem oben aufgeführten SQL-Kommando entspricht, basiert auf der Bereichssuche als Join-Prädikat. Es werden aber auch zwei Join-Operationen eingeführt, die auf dem Paradigma der *k*-nächsten-Nachbar-Suche beruhen.

Der Hauptteil dieser Arbeit ist den effizienten Algorithmen für die verschiedenen Arten des Similarity Join gewidmet. Zunächst wird ein Kostenmodell für die indexbasierte Anfragebearbeitung des Similarity Join eingeführt. Ausgehend von diesem Modell wird eine innovative Indexstruktur entwickelt, die einer speziellen Anforderung des Similarity Join, CPU- und I/O-Kosten getrennt zu optimieren, Rechnung trägt.

Dann wird ein Similarity Join Algorithmus entworfen, der sich speziell für massive Datenmengen eignet. Er basiert auf einer speziellen Sortierordnung für hochdimensionale Daten. Als nächstes wird ein neuartiger Algorithmus für den Similarity Join auf einer Join-Bedingung gemäß der *k*-nächsten-Nachbar-Suche vorgeschlagen. Schließlich entwickeln wir eine generische Technik zur Reduktion der CPU-Kosten, die universell bei Index-basierten wie nicht-Index-basierten Similarity-Join-Algorithmen eingesetzt werden kann.

Ein Ausblick auf mögliche zukünftige Forschungsrichtungen im Bereich Basisoperationen für Ähnlichkeitssuche, Datenanalyse und Data Mining schließt die Arbeit ab.

Table of Contents

List of Figures

10 Processing k-Nearest Neighbor Joins Using MuX

11 Optimizing the Similarity Join

12 Conclusions

References

Chapter 1
Introduction

Since the beginning of the information age, human society is facing a rapid and even accelerating growth of the available information. Both the number of databases as well as the amount of stored data per database is fast increasing, making it infeasible to evaluate the data manually. To cope with this information overkill will undoubtedly be one of the most important challenges of the 21st century.

Traditionally, data in databases are collected for dedicated applications which process the data in a relatively simple way such as booking systems, accounting and billing, storage and transportation planning etc. Companies, however, have also strong interest to exploit their databases for supporting complex decisions.

Therefore, it is necessary to find interesting patterns in the data such as clusters of similar database objects [McQ 67], outliers [KN 98], i.e. *untypical* database entries which could indicate fraudulent behavior, to assign the database objects to different meaningful classes [Mit 97], to detect associations (if-then-rules) [AS 94] etc. Such detected patterns are commonly referred to as *knowledge* and the search for these patterns is called *Knowledge Discovery in Databases* (*KDD*).

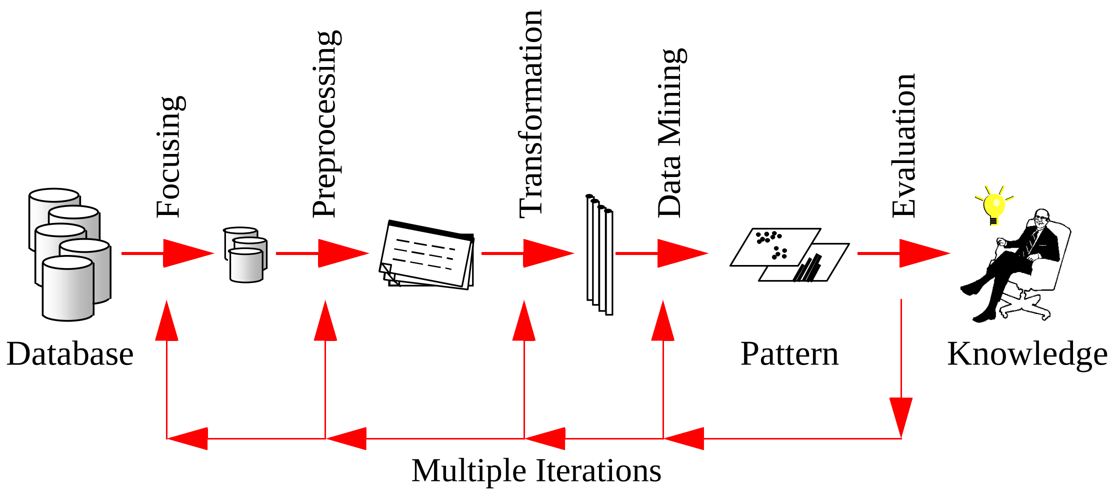


Figure 1: The KDD process

The most widespread definition for this notion has been given by Fayyad et al.:

Definition 1 Knowledge Discovery in Databases [FPS 96]

Knowledge Discovery in Databases (KDD) is the non-trivial process of identifying

- valid
- novel
- potentially useful
- and ultimately understandable

patterns in data.

The KDD process is an interactive and iterative process, involving numerous steps including preprocessing of the data set, applying a data mining algorithm to generate patterns from it, and the evaluation of the results [BA 96]:

- **Creating a target data set:**
  Selecting a subset of the data or focusing on a subset of attributes or data samples on which discovery is to be performed.

- **Data reduction:**
  Finding useful features to represent the data (i.e. dimensionality reduction) or transformation methods to reduce the number of variables under consideration or to find invariant representations of the data.
- **Data mining:**
  Searching for patterns of interest in the particular representation of the data: Classification rules or classification trees, association rules, regression, clustering etc.
- **Interpretation of results:**
  Visualization of the extracted patterns or visualization of the data giving the extracted models. Possibly the user has to return to previous steps in the KDD process if the results are unsatisfactory.

The KDD process is depicted in figure 1. The core step of knowledge discovery in databases is *data mining*. Data mining algorithms can be classified according to the kind of knowledge or pattern which is mined [FPS 96], [MCP 93], [FPM 91], [San 98], [Bre 01]:

- **Classification** (supervised learning):
  Learning a function that maps data records into one of several predefined classes which capture the common properties among a set of objects in the database. Each data record contains one dedicated attribute, the class label. The objective of classification is to analyze this training data and to construct a model for each class, which can than be used to classify newly arriving data records (not containing the class label attribute). Example algorithms include decision tree classifiers [Qui 86] and bayesian classifiers [Mit 97].
- **Clustering** (unsupervised learning):
  Identifying a finite set of clusters to describe the data, such that similar data records are assigned the same clusters and dissimilar ones different clusters. Typical algorithms are *k*-means [Sib 72] and its variants.
- **Data generalization**:
  Finding a compact description for a subset of the data. Examples of such a description are association rules, typically generated by the apriori-algorithm

[AS 94] and its variants, attribute oriented induction [CH 98], or spatial characterization [EFKS 98].

- **Regression and dependency modeling**:
  Learning a function which maps data records to real-valued variables and discovering functional relationships between variables. Well-known algorithms from the statistical domain include linear regression with error minimization.
- **Change and Deviation Detection**:
  Discovering significant changes in the data from previous or normative values. Time series analysis methods fall into this category.

1.1 High Performance Data Mining

Typically, knowledge detected in databases can be applied for

- marketing
- fraud detection
- customer segmentation and scoring
- strategic decisions including the orientation of a complete enterprise.

In these applications, the common purpose is to make important, domain-specific decisions based on the gained knowledge. For decision making, it is important, that the detected knowledge is *valid* and *accurate*. Therefore, validity and accuracy are the most important requirements of all methods of knowledge discovery and data mining. Since algorithms which do not fulfill these requirements are generally not effectively useful, we call these requirements the *effectivity requirements*.

For decision making based on knowledge from large databases, however, a second requirement is becoming of equally high importance: *Efficiency*. On the one hand, users are interested in a just-in-time analysis of their data to base their decisions on the newest available information. On the other hand, knowledge discovery involves complex algorithms for data analysis. It is difficult to gain valid and accurate knowledge fast, particularly when databases continue to rapidly increase in size.

There is a number of well-known approaches to tackle the performance problems of KDD algorithms such as

- sampling [BKKS 98]
- approximation of attributes, e.g. grid approximations [HK 98]
- dimensionality reduction [FL 95].

All these approaches serve their purpose in accelerating the applied algorithms but have also their limitations when considering the *quality* of the result (i.e. the effectivity of the algorithm) because it is not always obvious that the result of an algorithm on a reduced data set is comparable to the result on the original data set.

Therefore, the focus of our work was on approaches to accelerate data mining algorithms in a *quality preserving* way. Our objective is to reformulate standard algorithms of knowledge discovery in databases to gain efficiency but at the same time to *prove* that the result of the reformulated algorithm is identical to the result of the original algorithm. Our intention was not to propose any new data mining algorithm, because it is difficult to asses a new approach if in a comparison with competitive approaches both categories, *effectivity* and *efficiency* are different. The superiority of an approach is demonstrated in a more convincing way if it can be shown that the quality of the result is not affected at all (neither positively nor negatively) but the efficiency is considerably improved.

Lossy acceleration techniques like sampling, approximations, and dimensionality reduction can then be *additionally* applied to achieve further performance gains without quality guarantees.

1.2 Feature Databases

Our main focus is on data mining algorithms operating on feature databases which are prevalent in similarity search systems for various application domains such as multimedia [FBF+ 94, SK 97], CAD [Jag 91, GM 93, BKK 97], medical imaging [KSF+ 96], time sequence analysis [AFS 93, ALSS 95], molecular biology [KS 98b], etc. The principle of a feature transformation is demonstrated in figure 2.

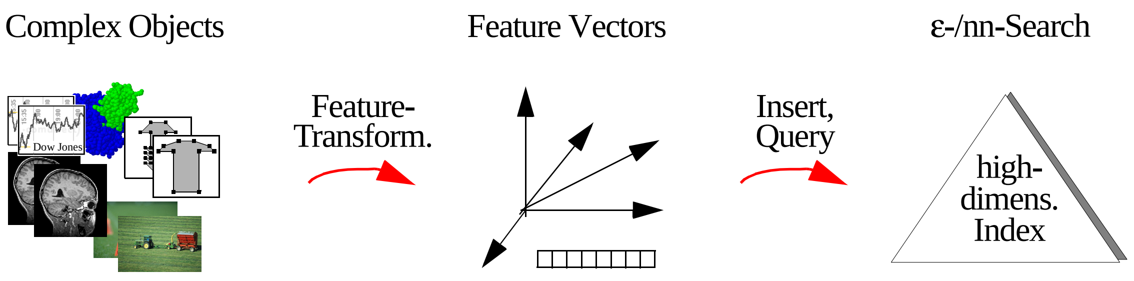


Figure 2: Basic Idea of Feature Databases

To capture similarity of complex domain-specific objects, the feature transformation extracts important, characterizing properties from the objects. Examples of such properties (for the domain of CAD objects) are the length, width, and perimeter of a CAD object but also more complex features such as parameters describing the curvature of the border of such an object. The sequence of features is interpreted as a vector from a multidimensional vector space (the feature space). These vectors can be effectively and efficiently managed using some multidimensional index structure such as an R-tree [Gut 84], [BKSS 90] or a similar structure [GG 98].

Most feature databases are high-dimensional which causes particular performance problems for usual indexing structures. Therefore, a number of dedicated index structures for high-dimensional indexing and similarity search has been proposed such as the X-tree [BKK 96], the TV-tree [LJF 95], the SS-tree [WJ 96], the pyramid technique [BBK 98b], the VA-file [WSB 98], and the IQ-tree [BBJ+ 00]. A survey of the problems of high-dimensional spaces and the most important solutions to them can be found in [Böh 98].

The most important property of a feature transformation is that *similarity* in the object space corresponds to spatial *proximity* in the feature space. I.e. whenever two of the complex application objects are similar, the associated feature vectors have a small distance according to an appropriate distance metric (often the Euclidean metric). Therefore, the similarity search is naturally translated into a neighborhood query in the feature space.

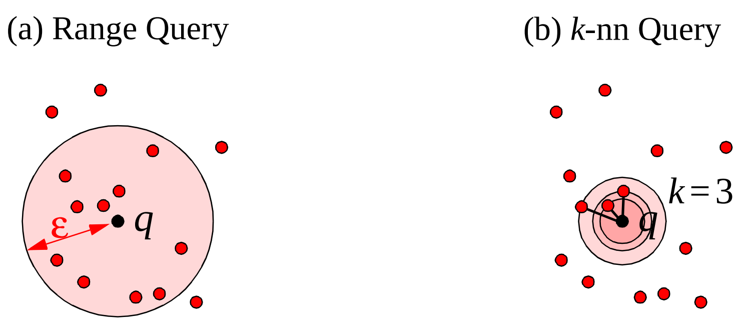


Figure 3: Usual Similarity Queries

The two most important types of neighborhood queries in feature databases are:

- Range Query:
  The user specifies a query object *q* and a query radius ε. The system retrieves all objects from the database that have a feature distance from *q* not exceeding ε.
- *k*-Nearest Neighbor Query
  The user specifies a query object *q* and the result cardinality *k*. The system retrieves those *k* objects from the database that have least distance from *q*.

Both types of queries are depicted in figure 3. Often a multi-step architecture for query processing is required. From the users' perspective, the *k*-Nearest Neighbor query is easier to handle because the result cardinality *k* is more intuitive than the query radius ε. To determine a suitable ε such that a useful set of result objects is retrieved (i.e. a set which is not empty and not almost equal to the set of *all* database objects) is sometimes difficult.

Numerous algorithms of knowledge discovery and data mining use similarity queries. Examples are the distance based outlier detection algorithm RT [KN 98], the density based outliers LOF [BKNS 00], the clustering algorithms DBSCAN [EKSX 96], DenClue [HK 98], OPTICS [ABKS 99], *k*-means [McQ 67] and *k*-medoid clustering [KR 90], nearest-neighbor clustering [HT 93], single-link clustering [JD 88], nearest neighbor classification [Mit 97], spatial association rules [KH 95], proximity analysis [KN 96], etc.

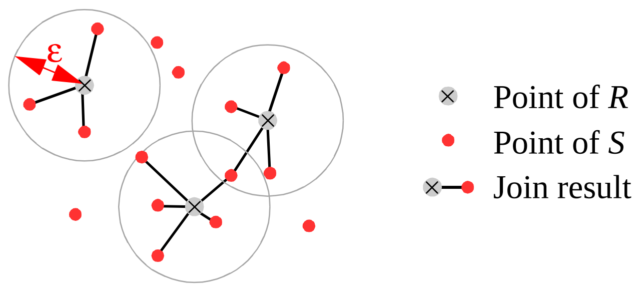


Figure 4: The Similarity Join (Distance Range Join)

Conceptually, the execution of the similarity query is performed by the database management system while the data mining algorithm itself runs as a database client application. The similarity query is provided by the database system as a *basic operation* or a *database primitive*. The advantage of this concept is *physical data independence*. The implementor of the data mining algorithm needs no knowledge about the actual algorithm for similarity search. Index structures can be seamlessly replaced by other, more efficient structures without affecting the implementation of the data mining algorithm. The similarity search algorithm is a *black box* to the data mining algorithm.

1.3 The Similarity Join

Another database primitive for feature databases which has recently gained attention is the *similarity join*. Like the relational join, the similarity join combines two data sets into one set such that the new set contains pairs of objects of the two original sets. The join condition involves some similarity predicate, i.e. some range-query based predicate or some nearest-neighbor based predicate.

Well known applications of the similarity join are e.g. catalogue matching, duplicate detection, or the search for pairwise similar items in large sets. For such applications, the similarity join has already been considered to serve as a *database primitive*.

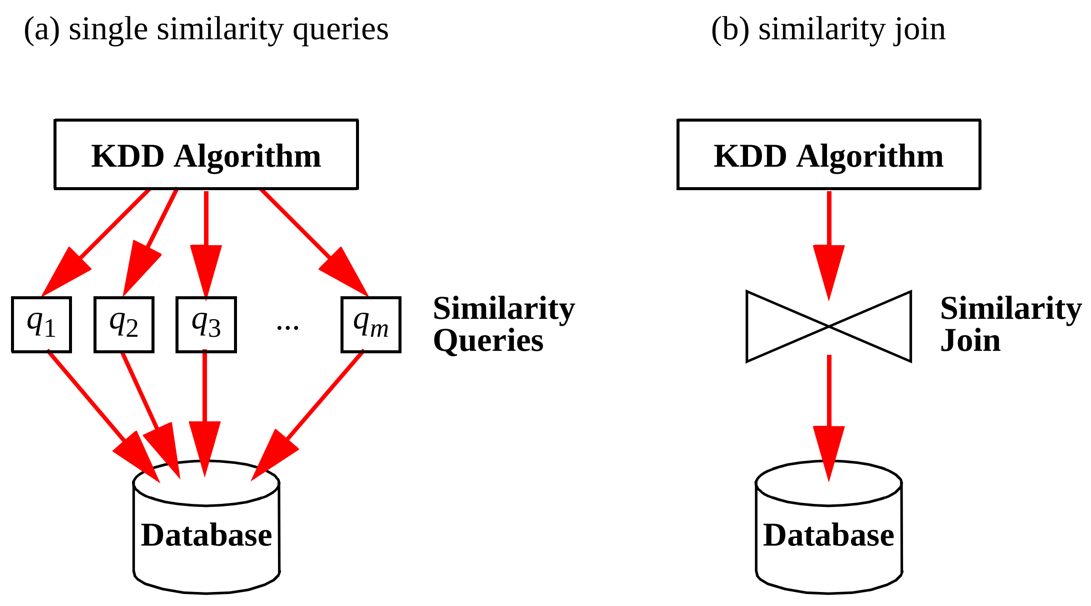


Figure 5: Database Primitives for Data Mining

For usual similarity queries, there exist several different definitions such as *range queries*, *k-nearest neighbor queries*, *inverse nearest neighbor queries*, etc. Likewise, there exist different definitions for the similarity join. The most common and most elaborated similarity join operation is the distance range join in which the user defines a query radius ε, and the system retrieves all point pairs the distance of which does not exceed ε (cf. figure 4). But there are also two further similarity join operations which are based on the principle of nearest neighbor search. We will formally introduce these definitions in chapter 2 and further elaborate later in this thesis.

1.4 High Performance Data Mining Based on the Similarity Join

The central idea of this thesis is that the similarity join is a powerful database primitive to support a number of data mining algorithms. The typical approach of many KDD algorithms is to evaluate similarity queries for a high number of query objects. Some data mining algorithms such as the clustering algorithm DBSCAN even evaluate a similarity query for each database point.

Our approach is to reformulate these KDD algorithms such that the high number of single similarity queries is replaced by a single run of the similarity join (cf. figure 5). This goal is reached for numerous important algorithms of knowledge discovery and data mining such as the density based clustering algorithms DBSCAN [EKSX 96] and OPTICS [ABKS 99] the outlier detection methods RT [KN 98] and LOF [BKNS 00], *k*-means [McQ 67] and *k*-medoid clustering [KR 90], nearest neighbor clustering [HT 93], nearest neighbor classification [Mit 97] and several others.

For some of these techniques, we show theoretically that the result of the modified algorithms is identical to the result of the original KDD algorithms. In other cases, this is obvious. In all cases, it is guaranteed that the reformulation is quality preserving.

Replacing the high number of similarity queries by a single similarity join also greatly affects the performance of these algorithms, even if relatively simple similarity join algorithms are applied. This will be shown in some of the experimental evaluations in this thesis.

Our principle of algorithm reformulation has a second advantage. The reformulated KDD algorithm operates upon a more powerful database primitive than before. Generally, the algorithm that implements a database primitive is a black box to the algorithm that uses this database primitive. Therefore, the implementing algorithm can be replaced by another solution without affecting the result (quality) of data mining.

Compared to usual similarity queries, the similarity join is a more complex and powerful database primitive. The similarity join yields more potential for performance improvements.

Our principle of powerful database primitives allows us to participate from future progress in the similarity join algorithms. Outdated algorithms can be replaced by newer ones with higher performance and different properties. E.g. new index and data structures can be used very efficiently.

1.5 Outline of the Thesis

Our thesis is built up from 4 large parts. The first part gives the motivation, introduces the similarity join formally, and reviews the related work. Our second, and largest building block is dedicated to a similarity join operation called distance range join where a query radius ε is given. We give a number of applications and show how this database primitive can be efficiently implemented. Then, the third part describes another type of similarity join which is based on a nearest neighbor join predicate. We give again numerous applications from the KDD domain and show how the basic operation of the *k*-nearest neighbor join can be efficiently implemented. The last part is dedicated to CPU optimization which can be applied to all kinds of similarity join. The remainder of our thesis consists of the following chapters:

- **Chapter 2. Defining the Similarity Join**
  In this chapter, we first introduce our basic notions. After that, we give the formal definition of a general similarity join which is a join of two multidimensional point sets based on some join predicate involving similarity of two objects. This definition leaves some degree of freedom how the similarity predicate actually looks like. After that basic definition of similarity joins, we introduce three different kinds of similarity join operations of which one is based on the paradigm of *range queries* and the others are based on the paradigm of *nearest neighbor queries*. The first operation, the *distance range join* operates on a given similarity threshold ε. As this similarity threshold is difficult to handle for the user, we also define our two nearest neighbor based similarity join operations. The first called *k*-distance join fetches those *k* pairs from the cross-product of the two point sets which have minimum distance. The other called *k*-nearest neighbor join combines each of the database points from the first set with its *k* nearest neighbors in the other point set.

- **Chapter 3. Related Work**
  This chapter is dedicated to previous approaches to the similarity join. Most of the fundamentals of the similarity join are based on the *spatial join* which is prevalent in spatial databases supporting geographic information systems (GIS).

Many algorithms for the spatial join can be adopted for the similarity join which is relatively simple. In our presentation of these approaches we show the necessary modifications to take the similarity join into account. As far as these modifications are concerned, this chapter contains also original work and not only a classification and survey over well-known techniques.

- **Chapter 4. Density Based Clustering on the Distance Range Join**
  In this chapter we show how density based clustering algorithms can be transformed such that they operate on top of the distance range join rather than on top of single similarity queries. In particular, we demonstrate such a transformation for the density based clustering method DBSCAN and for a density based analysis method for the hierarchical cluster structure of a data set called OPTICS. For these two methods, the transformation is particularly challenging because in contrast to some other methods presented in this thesis, DBSCAN and OPTICS in their original definitions enforce a certain order in which similarity queries are evaluated. Therefore it is not straightforward to replace the similarity queries by the similarity join. We propose two methods of transformation: The first, called semantic rewriting first transforms the clustering algorithm semantically to ensure that it is independent of the order in which join pairs are generated. This is done by assigning cluster IDs tentatively, and with a complex action table which handles inconsistent tentative results. The other technique is called join result materialization. The join result is predetermined prior to the run of the clustering algorithm and similarity queries are efficiently answered by lookups to the materialized join result. We can show for both techniques that the result of the clustering algorithms is identical to that of the original algorithms. Our experimental evaluation yields performance advances of up to a factor of 50 by our techniques.

- **Chapter 5. Further Applications of the Range Distance Join**
  After the complex case of the transformation of DBSCAN and OPTICS, we sketch in this chapter a few algorithms for which the evaluation on top of the similarity join is easier. The applications presented here are robust similarity search in sequence data where the join leads in particular to robustness with

respect to noise and scaling. We also present a few generalizations of this technique to similarity of multidimensional sequences (i.e. raster or voxel data) and to partial similarity. We also present applications like catalogue matching and duplicate detection.

- **Chapter 6. A Cost Model for Index Based Similarity Join Algorithms**
  This and the following chapters are dedicated to algorithms and index structures for the distance range join. We start in this chapter with a cost model for index based join evaluation. The concept used in this cost model is the Minkowski sum which is here modified to estimate the number of page pairs from the corresponding index structures which have to be considered. In contrast to usual similarity search, the concept of the Minkowski sum must be applied twice for the similarity join in order to estimate the number of page pairs which must be joined. We use this cost model to analyze the index with respect to the page capacity and show how this parameter can be optimized. Our analysis, however, reveals a serious optimization conflict between disk I/O and CPU optimization. While large pages optimize the I/O, the CPU performance benefits from small pages. This results in the observation that in traditional index structures only one of these performance factors can be optimized.

- **Chapter 7. MuX: An Index Architecture for the Similarity Join**
  This chapter is dedicated to the solution of the optimization conflict detected in the analysis of chapter 6. Our objective is to develop an index architecture which allows a separate optimization for CPU and I/O performance. Therefore, we basically need two separate page capacities, one for CPU and one for I/O. This goal is achieved by the multipage index (MuX). This index structure consists of large data and directory pages which are subject to I/O operations. Rather than directly storing points and directory records an these large pages, these pages accommodate a secondary search structure which is used to speed up the CPU operations. To facilitate an effective and efficient optimization, this secondary search structure has again an R-tree like structure with a (flat) directory and with data pages. Thus, the page capacity of the secondary search structure can be optimized by the cost functions developed in chapter 6, however, for

the CPU trade-off. We show that the CPU performance of MuX is similar (equal up to some small additional management overhead) to the CPU performance of a traditional index which is purely CPU optimized. Likewise, we show that the I/O performance resembles that of an I/O optimized traditional index. Our experimental evaluation confirms this and demonstrates the clear superiority over the traditional approaches.

- **Chapter 8. Joining Massive High-Dimensional Data**
  We develop the ε Grid Order, a sort order which is founded on a virtual grid partition of the data space. This method is based on the observation that for the distance range join with a given distance parameter ε, a grid partition with a grid distance of ε is an effective means to reduce the search space for join partners of a point *p*. Due to the *curse of dimensionality,* however, the number of grid cells in which potentially joining points are contained explodes with the data space dimension ($O(3^d)$ cells). To avoid considering the grid cells one by one, we introduce the grid partition only in a virtual way as the basis of a particular sort order, the ε grid order, which orders points according to grid cell containment. The ε grid order serves as the ordering criterion in an external memory sort operator. Later, the ε grid order supports effective and efficient algorithms for CPU and I/O processing, particularly for large data sets which cannot be joined in main memory.

- **Chapter 9. k-Nearest Neighbor Joins: Turbo Charging the KDD Process**
  The next two chapters are dedicated to the *k*-Nearest Neighbor Join (*k*-nn join) which combines each point of a point set *R* with its *k* nearest neighbors in another point set *S*. This chapter gives the applications of this database primitive. Many standard tasks of data mining evaluate *k*-nearest neighbor queries for a large number of query points. Examples are clustering algorithms such as *k*-means, *k*-medoid and the nearest neighbor method, but also data cleansing and other pre- and postprocessing techniques e.g. when sampling plays a role in data mining. 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. Then, the *k*-distance diagram is a technique for a suitable parameter selection for data mining. In the core step, i.e. data mining, many algorithms such as clustering and classification are based on *k*-nn queries. 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. In this chapter, we show how some of these standard algorithms can be based on top of the *k*-nearest neighbor join.

- **Chapter 10. Processing k-Nearest Neighbor Joins Using MuX**
  In this chapter, we show how the operation of a *k*-nearest neighbor similarity join can be efficiently implemented on top of a multidimensional index structure. In chapter 6 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) (cf. chapter 7) which allows such a separate optimization. The index consists of large pages which are optimized for I/O efficiency. 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. We believe that also the *k*-nn join clearly benefits from the separate optimization, because the optimization trade-offs are very similar. We give an algorithm to efficiently compute the similarity join on MuX. This algorithms applies two strategies, the *loading* and *processing strategy*. We propose strategies that clearly optimize query processing.

- **Chapter 11. Optimizing the Similarity Join.**
  Our 11th chapter is devoted to an optimization technique which can be applied on top of all join algorithms proposed in this thesis and also on most algorithms described in the related work chapter. The most important cost factor with respect to CPU operations are the finalizing distance calculations between the feature vectors. Our optimization technique accelerates these distance calculations by selecting the dimension with the highest selectivity and sorting the

points along this optimal dimension. Therefore, we call this technique the optimal dimension order. To select an optimal dimension our technique considers the regions which are assigned to the considered partitions. It is not restricted to index based processing techniques but can also be applied on top of hashing based methods or grid based approaches such as the size separation spatial join, the ε-kdB-tree or our ε Grid Order.

Finally, chapter 12 concludes our thesis. We will summarize our contributions to the research field of applications and algorithms for the similarity join. We believe that we have illustrated this field both in its broadness as well as in its depth. Nevertheless, several research issues also remain for future work, in particular approximate join algorithms and the similarity join on non-vector metric data. We will indicate the most promising research directions.

Chapter 2
Defining the Similarity Join

In the current literature, there are several different kinds of similarity join known, founding on the concepts of range-queries and nearest neighbor queries. Further definitions may follow in future work. Therefore, we will first give an intuitive definition of what we understand to be a similarity join in general (which is informal by nature), and then give the precise formal definitions of the known approaches.

2.1 General Notion of Similarity Joins

We postulate three requirements for a similarity join. First, the similarity join is a join in the sense of the relational database model i.e. two sets R and S are combined into one such that the new set contains *pairs* of objects of R and S that fulfill a *join condition*. Every join can also be expressed as a *selection* operation (which corresponds to the *join condition*) on the *cartesian product* $R \times S$.

The second property of a similarity join is that the sets R and S are not regular relations, i.e. sets of tuples of an arbitrary record type but are either sets of points in a multidimensional vector space (or at least that some *point information* is contained in each tuple) or sets of multimedia objects with a distance metric defined upon. The third property of a general similarity join is that the *join condition* must involve the *similarity* between the objects in R and S. In the case of multidimensional point sets, the *join condition* involves the Euclidean distance or some other distance metric for vector spaces. For general multimedia objects, analogously the defined similarity metric is used. Note that, to the best of our knowledge, currently there is no publication dealing with joins on non-vector metric spaces. The concepts, however, are directly transferable and there are numerous applications of the similarity join upon non-vector multimedia objects.

The way in which similarity is involved in the join may vary. As we will see, there are join definitions which postulate that the object pairs in the result set have a distance (*dissimilarity*) not exceeding a given join parameter ε (cf. sections 2.2). Other join definitions combine exactly those objects which are *most similar* to each other (cf. section 2.3). Summarizing, we give the following

Definition 2 General Similarity Join

A similarity join $R \underset{sim}{\bowtie} S$ of two finite sets R and S has the following properties:

- the join result is a subset of the cartesian product

  $R \underset{sim}{\bowtie} S \quad \subseteq \quad R \times S$
- each tuple of the sets R and S contains either

  – *point data* from a multidimensional vector space or

  – a *multimedia object* with an associated *similarity metric*
- a vector space metric or the associated similarity metric is used in the join predicate.

2.2 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 interchangably. Unless otherwise specified, when speaking of *the similarity join*, often the distance range join is meant by default. For clarity in this thesis, we will not follow this convention and always use the exact notions.

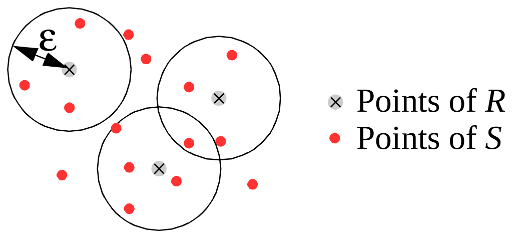


Figure 6: The Distance Range Join (ε-Join)

As depicted in figure 6, 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. Formally:

Definition 3 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** $\|R.\mathrm{obj} - S.\mathrm{obj}\| \leq \varepsilon$

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.

As we will point out later, 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. In the following we note that

Lemma 1. the distance range *self* join is symmetric i.e.

$$(r_i, r_j) \in R \underset{\varepsilon}{\bowtie} R \Leftrightarrow (r_j, r_i) \in R \underset{\varepsilon}{\bowtie} R$$

Proof follows from the symmetry of any distance metric: $||r_i - r_j|| \le \varepsilon \Leftrightarrow ||r_j - r_i|| \le \varepsilon$ for any distance metric □

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 time asymptote of advanced join algorithms is better than quadratic.

2.3 Nearest Neighbor Based Similarity Join

It is possible to overcome the problems of a selectivity which is difficult to control by replacing the range query based join predicate by a (*k*-)nearest neighbor based condition. In contrast to range queries which retrieve potentially the whole database, the selectivity of a (*k*-)nearest-neighbor query is (up to tie situations) clearly defined. There are two ways in which the concept of the nearest neighbor queries can be integrated into the similarity join. Both methods inherit from the nearest neighbor query the advantage that the size of the result set is (unless ties occur) previously known.

2.3.1 (*k*-) Closest Pair Queries

The first nearest neighbor based similarity join is the *k*-closest pair query. This operation retrieves those *k* pairs from $R \times S$ having minimum distance. Closest pair queries do not

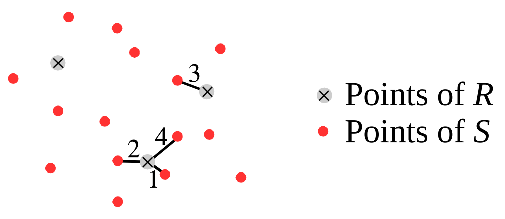


Figure 7: The *k*-Closest Pair Query (for $k = 4$)

only play an important role in the database research but have also a long history in computational geometry [PS 85]. In the database context, the operation has been introduced by Hjaltason and Samet [HS 98] using the term *distance join*. The (*k*-)closest pair query can be defined as follows:

Definition 4 (*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:

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

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 4 such that the tie is broken non-deterministically by a random selection. [HS 98] defines the closest pair query (non-deterministically) by the following SQL statement:

SELECT * **FROM** *R*, *S*
ORDER BY ||*R*.obj – *S*.obj||
STOP AFTER *k*

We give two more remarks regarding self joins. Obviously, the closest pairs of the self-join $R \underset{k\text{-CP}}{\bowtie} R$ 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

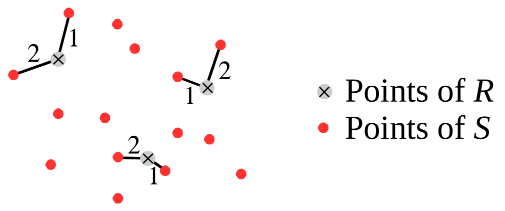


Figure 8: The k-Nearest Neighbor Join (for $k = 2$)

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 of any multimedia application

For plain similarity search in multimedia-databases, it is often useful to replace k-nearest neighbor queries by ranking queries which retrieve the first, second, third,... nearest neighbor in a one-by-one fashion. The actual number k of nearest neighbors to be searched is initially unknown. The user (or some application program on top of the database) decides according to a criterion which is unknown to the DBMS whether or not further neighbors are required. This kind of processing can also be defined on top of the closest pair query, e.g. by cancelling the STOP AFTER clause in the SQL statement above. The query results are passed to the application program using some cursor concept. It is important to avoid computing the complete ranking in the initialization phase of the cursor, because determining the complete ranking is unnecessarily expensive if the user decides to stop the ranking after retrieving only a few result points.

2.3.2 (*k*-) Nearest Neighbor Join

Even more important is the last kind of similarity join operation which does not find the best ones among all arbitrary pairs of points but rather combines each point of R with its nearest neighbor (or its k nearest neighbors) in S. In computational geometry, this operation is called the *all nearest neighbor search*. In contrast to the closest pair query, here

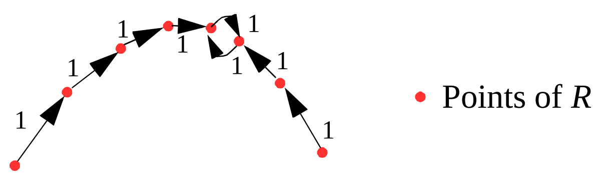


Figure 9: The NN-Join (k = 1) is not symmetric

it is guaranteed that each point of R appears in the result set exactly once (or exactly k times, respectively). Points of S may appear once, more than once (if a point is the nearest neighbor of several points in R) or not at all (if a point is the nearest neighbor of no point in R). Formally, we define the k-NN-join as follows:

Definition 5 k-NN-Join $R \underset{k\text{-NN}}{\ltimes\!\rtimes} S$

$R \underset{k\text{-NN}}{\ltimes\!\rtimes} S$ 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:

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

Here, the notion of k-nearest neighbor queries has been transformed to point sets on a basis "per point of R." Again, in this definition, tie situations are broken deterministically by enlarging the result set. Another possibility is random selection. Hjaltason and Samet define the k-NN-Join in SQL style as follows:

SELECT * **FROM** *R*, *S*
GROUP BY *R*.obj
ORDER BY ||*R*.obj – *S*.obj||
STOP AFTER *k*

For the selfjoin, we have again the situation that each point is combined with itself which can be avoided using the WHERE clause. Unlike ε-join and k-CP query, the k-NN self-join is not symmetric as the nearest neighbor relation is not symmetric (cf. the simple counterexample in figure 9). The k-NN-Join can be successfully applied in simultaneous nearest neighbor classification of a high number of query objects which is usual for domains such as

- Astronomical observation:
  A high number of newly detected objects is compared to a very high number of known reference objects
- Online customer scoring:
  Some thousand new customers are probed against some millions of known patterns.

Chapter 3
Related Work

This chapter is dedicated to the previous approaches to the similarity join. Most of the fundamentals of the similarity join base on the *spatial join* which is prevalent in spatial databases supporting geographic information systems (GIS). Many algorithms for the spatial join can be adopted for the similarity join which is relatively simple. In our presentation of these approaches we show the necessary modifications to take the similarity predicate into account.

3.1 Preliminaries: Indexing High-Dimensional Spaces

We begin with a short description of the index structures used to organize feature spaces and the corresponding query processing techniques. The interested reader is referred to more elaborate surveys of multidimensional and high dimensional indexing techniques such as [GG 98, BBK 01]. In this section, we give only a short introduction to make the needed material readily available and to make this thesis more self-contained. We concentrate on techniques which are needed later in this text. We put some emphasis on algorithms for plain similarity queries, especially nearest neighbor queries, because these algorithms will be later (cf. section 3.3 and chapter 10) used as building blocks for join algorithms upon nearest neighbor join predicates.

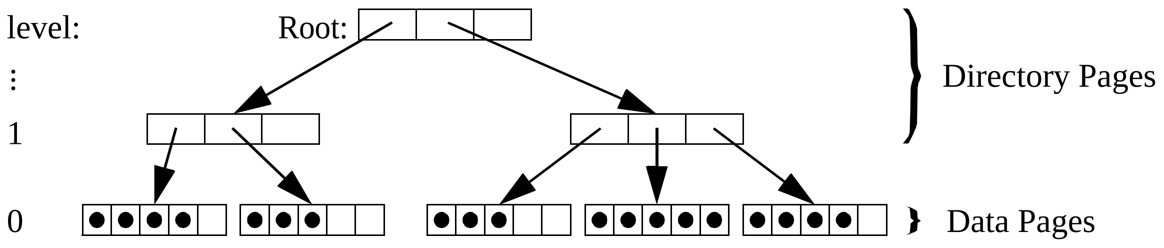


Figure 10: Hierarchical Index Structures.

3.1.1 Structure of High-Dimensional Indexes

High-dimensional indexing methods are based on the principle of hierarchical clustering of the data space. Structurally, they are similar to the B^+-tree [BM 77, Com 79]: The data vectors are stored in data nodes such that spatially adjacent vectors are likely to reside in the same node. Each data vector is stored in exactly one data node, i.e. there is no object duplication among the data nodes. The data nodes are organized in a hierarchically structured directory. Each directory node points to a set of subtrees. Usually, the structure of the information stored in data nodes is completely different from the structure of the directory nodes. In contrast, the directory nodes are uniformly structured among all levels of the index. There is a single directory node which is called the root node. It serves as an entry point for query and update processing. The index structures are height-balanced. That means, the lengths of the paths between the root and all data pages are identical, but may change after insert or delete operations. The length of a path from the root to a data page is called the *height* of the index. The length of the path from a random node to a data page is called the *level* of the node. Data pages are on level zero.

3.1.1.1 Management

The high-dimensional access methods are designed primarily for the secondary storage. Data pages have a data page capacity $C_{\text{max,data}}$, defining how many data vectors can be stored in a data page at most. Analogously, the directory page capacity $C_{\text{max,dir}}$ gives an upper limit to the number of subnodes in each directory node. The original idea was to choose $C_{\text{max,data}}$ and $C_{\text{max,dir}}$ such that data and directory nodes fit exactly into the pages of the secondary storage. However, in modern operating systems, the page size of a disk drive is considered as a hardware detail hidden from programmers and users. Even though, consecutive reading of contiguous data on disk is by orders of magnitude less expensive than reading at random positions. It is a good compromise to read data contiguously from disk in portions between a few kilobytes and a few hundred kilobytes. This is a kind of artificial paging with a user-defined logical page size.

All index structures presented here are dynamic, i.e. they allow insert and delete operations in O (log n) time. To cope with dynamic insertions, updates and deletes, the index structures allow data and directory nodes to be filled under their capacity C_{max}. In most index structures the rule is applied that all nodes up to the root node must be filled to about 40% at least. This threshold is called the *minimum storage utilization* su_{min}. The root is generally allowed to break this rule.

For B-trees, it is possible to derive an average storage utilization analytically, called the *effective storage utilization* su_{eff}. In contrast, for high-dimensional index structures, the effective storage utilization is influenced by the specific heuristics applied in insert and delete processing. Since these indexing methods are not amenable to an analytical derivation of the effective storage utilization, it has to be determined experimentally.

For comfort, we will denote the product of the capacity and the effective storage utilization as the *effective capacity* C_{eff} of a page:

$$C_{\text{eff,data}} = su_{\text{eff,data}} \cdot C_{\text{max,data}} \qquad C_{\text{eff,dir}} = su_{\text{eff,dir}} \cdot C_{\text{max,dir}} .$$

3.1.1.2 Regions

For efficient query processing it is important that the data is well clustered into the pages, i.e. that data objects which are close to each other are likely to be stored in the same data page. Assigned to each page is a so-called *page region* which is a subset of the data space. The page region can be a hypersphere, a hypercube, a multidimensional cuboid, a multidimensional cylinder or a set-theoretical combination (union, intersection) of these possibilities. For most, but not all high-dimensional index structures the page region is a contiguous and convex subset of the data space without holes. For most index structures,

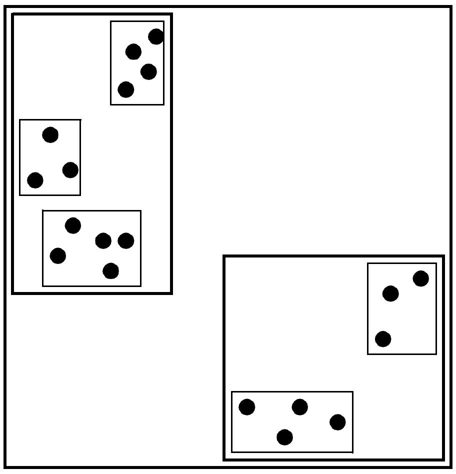


Figure 11: Corresponding Page Regions of an Indexing Structure.

regions of pages in different branches of the tree may overlap, although overlaps lead to bad performance behavior and have to be avoided if possible or at least minimized.

The regions of hierarchically organized pages always have to be completely contained in the region of their parent node. Analogously, all data objects stored in a subtree are always contained in the page region of the root page of the subtree. The page region is always a *conservative approximation* for the data objects and the other page regions stored in a subtree.

In query processing, the page region is used to exclude branches of the tree from further processing. For example, in case of range queries if a page region does not intersect with the query range, it is impossible that any region of a hierarchically subordered page intersects with the query range. Neither is it possible that any data object stored in this subtree intersects with the query range. Only pages where the corresponding page region intersects with the query have to be investigated further. Therefore, a suitable algorithm for range query processing can guarantee that no false drops occur.

For nearest neighbor queries a related but slightly different property of conservative approximations is important. Here, distances to a query point have to be determined or estimated. It is important that distances to approximations of point sets are never greater than the distances to the regions of subordered pages and never greater than the distances to the points stored in the corresponding subtree. This is commonly known as the *lower bounding property.*

Page regions have always a representation that is an invertible mapping between the geometry of the region and a set of values storable in the index. For example, spherical regions can be represented as center point and radius using $d + 1$ floating point values if d is the dimension of the data space. For efficient query processing, it is necessary that the test for intersection with a query region and the distance computation to the query point in case of nearest neighbor queries can be performed efficiently.

3.1.2 Algorithms for Insert, Delete and Update

In this section, we will present some basic algorithms on high-dimensional index structures for index construction and maintenance in a dynamic environment as well as for query processing. Although some of the algorithms are published for a specific indexing structure, here they are presented in a more general way.

Insert, delete and update are the operations which are most specific to the corresponding index structures. Even though, there are basic algorithms capturing all actions which are common to all index structures. Inserts are generally handled as follows:

- Search a suitable data page *dp* for the data object *do*.
- Insert *do* into *dp*.
- If the number of objects stored in *dp* exceeds $C_{\text{max,data}}$, then split *dp* into two data pages
- Replace the old description (the representation of the region and the background storage address) of *dp* in the parent node of *dp* by the descriptions of the new pages
- If the number of subtrees stored in the parent exceeds $C_{\text{max,dir}}$, split the parent and proceed similarly with the parent. It is possible that all pages on the path from *dp* to the *root* have to be split.

- If the root node has to be split, let the height of the tree grow by one. In this case, a new root node is created pointing to two subtrees resulting from the split of the original root.

Individual heuristics for the specific indexing structure are applied to handle the following subtasks:

- The search for a suitable data page (commonly called the *PickBranch* procedure). Due to the overlap between regions and as the data space is not necessarily completely covered by page regions, there are generally multiple alternatives for the choice of a data page in most multidimensional index structures.
- The choice of the split, i.e. which of the data objects/subtrees are aggregated into which of the newly created nodes.

Some index structures try to avoid splits by a concept named *forced re-insert*. Some data objects are deleted from a node having an overflow condition and reinserted into the index. The details are presented later in this chapter.

The choice of heuristics for insert processing may affect the effective storage utilization. For example, if a volume-minimizing algorithm allows unbalanced splitting in a 30:70 proportion, then the storage utilization of the index is decreased and the search performance is negatively affected. On the other hand, the presence of forced reinsert operations increases the storage utilization and the search performance.

Until now, few have been done to handle deletions from multidimensional index structures. Underflow conditions can generally be handled by three different actions:

- Balancing pages by moving objects from one page to another
- Merging pages
- Deleting the page and reinserting all objects into the index.

For most index structures it is a difficult task to find a suitable mate node for balancing or merging actions.

An update-operation is viewed as a sequence of a delete-operation followed by an insert-operation. No special procedure has been suggested, yet.

3.1.3 Exact Match and Range Query

Exact match queries are defined as follows: Given a query point *q*, determine whether *q* is contained in the database or not. Query processing starts with the root node which is loaded into the main memory. For all regions containing point *q* the function *ExactMatchQuery* is called recursively. Since an overlap between page regions is allowed in most index structures presented in this chapter, it is possible that several branches of the indexing structure have to be examined for processing an exact match query. The result of *ExactMatchQuery* is true if any of the recursive calls returns true. For data pages, the result is true if one of the points stored on the data page fits. If no point fits, the result is false.

The algorithm for range query processing returns a set of points contained in the query range as result to the calling function. The size of the result set is previously unknown and may reach the size of the entire database. The algorithm is formulated independently from the applied metric. Any L_p metric including metrics with weighted dimensions (ellipsoid queries, [Sei 97, SK 97]) can be applied if there exists an effective and efficient test for the predicates IsPointInRange and RangeIntersectRegion. Also partial range queries, i.e. range queries where only a subset of the attributes is specified, can be considered as regular range queries with weights (the unspecified attributes are weighted with zero). Also window queries can be transformed into range-queries using a weighted L_{max} metric.

The algorithm for the range search performs a recursive self-call for each child-page the page region of which intersects the query range. The union of the results of all recursive calls is built and passed to the caller.

3.1.4 Nearest Neighbor Query

There are two different approaches to process nearest neighbor queries on multidimensional index structures. One was published by Roussopoulos, Kelley and Vincent [RKV 95] and is in the following called *RKV algorithm*. The other algorithm (*‘HS algorithm’*), was published by Hjaltason and Samet [HS 95]. Due to their importance for our further work, these algorithms are presented in detail.

We start with the description of the RKV algorithm because it is more similar to the algorithm for range query processing in the sense that a depth-first traversal through the index is performed. RKV is an algorithm of the type “branch and bound”. In contrast, the HS algorithm loads pages from different branches and different levels of the index in an order induced by the proximity to the query point.

Unlike range query processing, there is no fixed criterion, known *a priori*, to exclude branches of the indexing structure from processing in nearest neighbor algorithms. Actually, the criterion is the nearest neighbor distance but the nearest neighbor distance is not known until the algorithm has terminated. To cut branches, nearest neighbor algorithms have to use pessimistic (conservative) estimations of the nearest neighbor distance which will change during the run of the algorithm and will approach the nearest neighbor distance. A suitable pessimistic estimation of the nearest neighbor distance is the closest point among all points visited at the current state of execution (the so-called *closest point candidate cpc*). If no point has been visited yet, it is also possible to derive pessimistic estimations from the page regions visited so far.

3.1.4.1 The RKV Algorithm

The authors of the RKV algorithm define two important distance functions, MINDIST and MINMAXDIST. MINDIST is the actual distance between the query point and a page region in the geometrical sense, i.e. the nearest possible distance of any point inside the region to the query point. The definition in the original proposal [RKV 95] is limited to R-tree like structures where regions are provided as multidimensional intervals *I* (i.e., minimum bounding rectangles, *MBR*) with

$$I = [lb_0, ub_0] \times \ldots \times [lb_{d-1}, ub_{d-1}] .$$

Then, MINDIST is defined as follows:

Definition 6 MINDIST. The distance of a point *q* to region *I*, denoted MINDIST (*q*, *I*) is:

$$\mathrm{MINDIST}^2(q, I) = \sum_{i=0}^{d-1} \left(\begin{cases} lb_i - q_i & \text{if} \quad q_i < lb_i \\ 0 & \text{otherwise} \\ q_i - ub_i & \text{if} \quad ub_i < q_i \end{cases} \right)^2$$

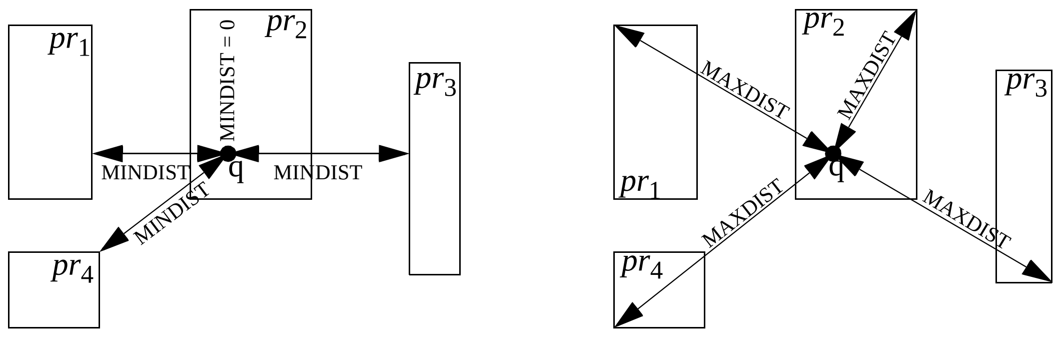


Figure 12: MINDIST and MAXDIST.

An example of MINDIST is presented on the left side of figure 12. In page regions pr_1 and pr_3, the edges of the rectangles define the MINDIST. In page region pr_4 the corner defines MINDIST. As the query point lies in pr_2, the corresponding MINDIST is 0. A similar definition can also be provided for differently shaped page regions, such as spheres (subtract the radius from the distance between center and q) or combinations. A similar definition can be given for L_1 and L_{max} metric, respectively. For a pessimistic estimation, some specific knowledge about the underlying index structure is required. One assumption which is true for all known index structures is that every page must contain at least one point. Therefore, we could define the following MAXDIST function determining the distance to the farthest possible point inside a region:

$$\text{MAXDIST}^2(q, I) = \sum_{i=0}^{d-1} \left(\begin{cases} |lb_i - q_i| & \text{if} \quad |lb_i - q_i| > |q_i - ub_i| \\ |q_i - ub_i| & \text{otherwise} \end{cases} \right)^2$$

MAXDIST is not defined in the original paper as it is not needed in R-tree like structures. An example is shown on the right side of figure 12. Being the greatest possible distance from the query point to a point in a page region, the MAXDIST is not equal to 0 even if the query point is located inside the page region pr_2.

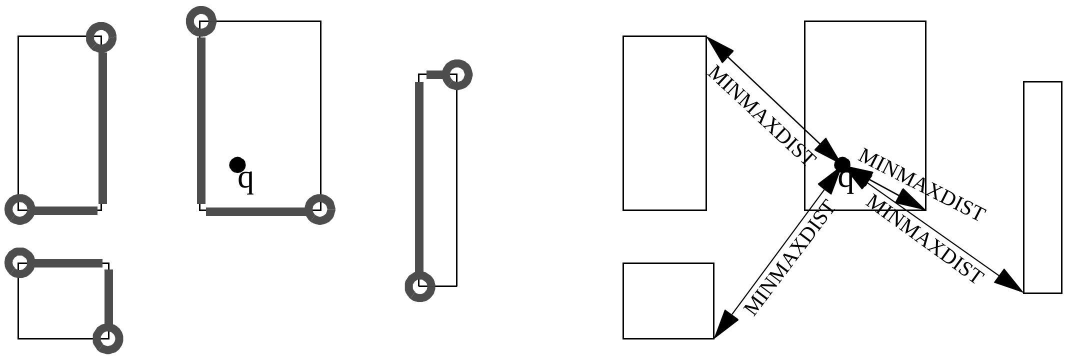


Figure 13: MINMAXDIST.

In R-trees, the page regions are minimum bounding rectangles (*MBR*), i.e. rectangular regions where each surface hyperplane contains one data point at least. The following MINMAXDIST function provides a better (i.e. lower) but still conservative estimation of the nearest neighbor distance:

$$\mathrm{MINMAXDIST}^2(q, I) = \min_{0 \le k < d} \left(|q_k - rm_k|^2 + \sum_{\substack{i \ne k \\ 0 \le i < d}} |q_i - rM_i|^2 \right) ,$$

where:

$$rm_k = \begin{cases} lb_k & \text{if } \quad q_k \le \dfrac{lb_k + ub_k}{2} \\ ub_k & \text{otherwise} \end{cases} \quad \text{and } rM_i = \begin{cases} lb_i & \text{if } \quad q_i \ge \dfrac{lb_i + ub_i}{2} \\ ub_i & \text{otherwise} \end{cases} .$$

The general idea is that every surface hyperarea must contain a point. The farthest point on every surface is determined and among those the minimum is taken. For each pair of opposite surfaces, only the nearer surface can contain the minimum. Thus, it is guaranteed that a data object can be found in the region having a distance less than or equal to MINMAXDIST (q, I). MINMAXDIST (q, I) is the smallest distance providing this guarantee. The example on figure 13 shows on the left side the considered edges. Among each pair of opposite edges of an MBR, only the edge closer to the query point is considered. The point yielding the maximum distance on each considered edge is marked with a circle. The minimum among all marked points of each page region defines the MINMAXDIST as shown on the right side of figure 13.

This pessimistic estimation cannot be used for spherical or combined regions because no property similar to the MBR property is fulfilled. In this case, MAXDIST (q, I) which is an estimation worse than MINMAXDIST has to be used. All definitions presented with the L_2-metric in the original paper [RKV 95] can easily be adapted to L_1 or L_{max} metrics as well as to weighted metrics.

The algorithm proposed by Roussopoulos et al. performs accesses to the pages of an index in a depth-first order (“branch and bound”). A branch of the index is always completely processed before the next branch starts. Before child nodes are loaded and recursively processed, they are heuristically sorted according to their probability of containing the nearest neighbor. For the sorting order, the optimistic or pessimistic estimation or a combination thereof may be chosen. The quality of sorting is critical for the efficiency of the algorithm because for different sequences of processing the estimation of the nearest neighbor distance may approach more or less fast to the actual nearest neighbor distance. The paper [RKV 95] reports advantages for the optimistic estimation. The list of child nodes is pruned whenever the pessimistic estimation of the nearest neighbor distance changes. Pruning means to discard all child nodes having a MINDIST larger than the pessimistic estimation of the nearest neighbor distance. It is guaranteed that these pages do not contain the nearest neighbor because even the closest point in these pages is farther away than an already found point (lower bounding property). The pessimistic estimation is the lowest among all distances to points processed so far and all results of the MINMAXDIST (q, I) function for all page regions processed so far.

To extend the algorithm to *k*-nearest neighbor processing is a difficult task. Unfortunately, the authors make it easy by discarding the MINMAXDIST from path pruning, sacrificing the performance gains obtainable from the MINMAXDIST path pruning. The *k*-th lowest among all distances to points found so far must be used. Additionally required is a buffer for *k* points (the *k* closest point candidate list, *cpcl*) which allows an efficient deletion of the point with the highest distance and an efficient insertion of a random point. A suitable data structure for the closest point candidate list is a priority queue (also known as semi-sorted heap [Knu 75]).

Considering the MINMAXDIST imposes some difficulties, since the algorithm has to assure that *k* points are closer to the query than a given region is. For each region, we know that at least one point must have a distance less than or equal to MINMAXDIST. If the *k*-nearest neighbor algorithm would prune a branch according to MINMAXDIST, it would assume that *k* points must be positioned on the nearest surface hyperplane of the page region. The MBR property only guarantees one such point. We further know that *m* points must have a distance less than or equal to MAXDIST where *m* is the number of points stored in the corresponding subtree. The number *m* could be, for example, stored in the directory nodes or could be estimated pessimistically by assuming minimal storage utilization if the indexing structure provides storage utilization guarantees. A suitable extension of the RKV algorithm could use a semi-sorted heap with *k* entries. Each entry is either a *cpc* or a MAXDIST estimation or a MINMAXDIST estimation. The heap entry with the greatest distance to the query point *q* is used for branch pruning. It is called the *pruning element*. Whenever new points or estimations are encountered, they are inserted into the heap if they are closer to the query point than the pruning element. Whenever a new page is processed, all estimations based on the according page region have to be deleted from the heap. They are replaced by the estimations based on the regions of the child pages (or the contained points if it is a data page). This additional deletion implies additional complexities because a priority queue does not efficiently support the deletion of elements other than the pruning element. All these difficulties are neglected in the original paper [RKV 95].

3.1.4.2 The HS Algorithm

The problems arising from the need to estimate the nearest neighbor distance are elegantly avoided in the HS algorithm [HS 95]. The HS algorithm does not access the pages in an order induced by the hierarchy of the indexing structure such as depth-first or breadth-first. Rather, all pages of the index are accessed in the order of increasing distance to the query point. The algorithm is allowed to jump between branches and levels for processing pages.

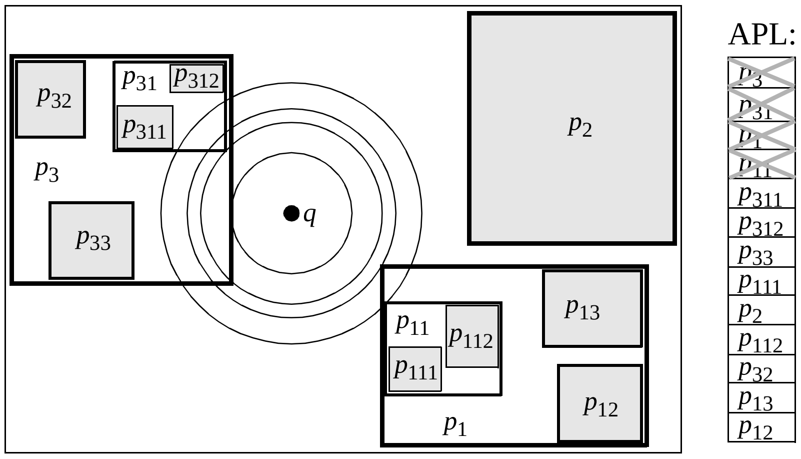


Figure 14: The HS Algorithm for Finding the Nearest Neighbor.

The algorithm manages an active page list (APL). A page is called *active* if its parent has been processed but not the page itself. Since the parent of an active page has been loaded, the corresponding region of all active pages is known and the distance between region and query point can be determined. The APL stores the background storage address of the page as well as the distance to the query point. The representation of the page region is not needed in the APL. A processing step of the HS algorithm comprises the following actions:

- Select the page *p* with the lowest distance to the query point from the APL.
- Load *p* into the main memory.
- Delete *p* from the APL.
- If *p* is a data page: Determine whether one of the points contained in this page is closer to the query point than the closest point found so far (called the *closest point candidate cpc*).
- Otherwise: Determine the distances to the query point for the regions of all child pages of *p* and insert all child pages and the corresponding distances into APL.

The processing step is repeated until the closest point candidate is closer to the query point than the nearest active page. In this case, no active page is able to contain a point closer to *q* than *cpc* due to the lower bounding property. Likewise, no subtree of any active page may contain such a point. As all other pages have already been looked upon, processing can stop. Again, the priority queue is the suitable data structure for APL.

For *k*-nearest neighbor processing, a second priority queue with fixed length *k* is required for the closest point candidate list.

3.1.4.3 Ranking Query

Ranking queries can be seen as generalized *k*-nearest neighbor queries with a previously unknown result set size *k*. A typical application of a ranking query requests the nearest neighbor first, then the second closest point, the third and so on. The requests stop according to a criterion which is external to the index-based query processing. Therefore, neither a limited query range nor a limited result set size can be assumed before the application terminates the ranking query.

In contrast to the *k*-nearest neighbor algorithm, a ranking query algorithm needs an unlimited priority queue for the candidate list of closest points (*cpcl*). A further difference is that each request of the next closest point is regarded as a phase that ends reporting the next resulting point. The phases are optimized independently. In contrast, the *k*-nearest neighbor algorithm searches all *k* points in a single phase and reports the complete set.

In each phase of a ranking query algorithm, all points encountered during the data page accesses are stored in the *cpcl*. The phase ends if it is guaranteed that unprocessed index pages cannot contain a point closer than the first point in *cpcl* (the corresponding criterion of the *k*-nearest neighbor algorithm is based on the last element of *cpcl*). Before beginning the next phase, the leading element is deleted from the *cpcl*.

It does not appear very attractive to extend the RKV algorithm for processing ranking queries due to the fact that effective branch pruning can be performed neither based on MINMAXDIST or MAXDIST estimates nor based on the points encountered during the data page accesses.

In contrast, the HS algorithm for nearest neighbor processing needs only the modifications described above to be applied as a ranking query algorithm. The original proposal [HS 95] contains these extensions.

The major limitation of the HS algorithm for ranking queries is the *cpcl*. It can be proven that the length of the *cpcl* is of the order O (n). In contrast to the APL, the *cpcl* contains the full information of possibly all data objects stored in the index. Thus, its size is bounded only by the database size questioning the applicability not only theoretically, but also practically. From our point of view, a priority queue implementation suitable for background storage is required for this purpose.

3.1.5 R-tree, R*-tree, and X-tree

The R-tree [Gut 84] uses solid minimum bounding rectangles (*MBR*) as page regions. An *MBR* is a multidimensional interval of the data space, i.e. axis-parallel multidimensional rectangles. *MBR*s are minimal approximations of the enclosed point set. There exists no smaller axis-parallel rectangle also enclosing the complete point set. Therefore, every $(d - 1)$-dimensional surface area must contain at least one data point. Space partitioning is neither complete nor disjoint. Parts of the data space may be not covered at all by data page regions. Overlapping between regions in different branches is allowed, although overlaps deteriorate the search performance especially for high-dimensional data spaces [BKK 96]. The region description of an *MBR* encompasses for each dimension a lower and an upper bound. Thus, $2\,d$ floating point values are required. This description allows an efficient determination of MINDIST, MINMAXDIST and MAXDIST using any L_p metric.

R-trees have originally been designed for spatial databases, i.e. for the management of 2-dimensional objects with a spatial extension (e.g., polygons). In the index, these objects are represented by the corresponding *MBR*. In contrast to point objects, it is possible that no overlap-free partition for a set of such objects exists at all. The same problem occurs also when R-trees are used to index data points but only in the directory part of the index. Page regions are treated like spatially extended, atomic objects in their parent nodes (no forced split). Therefore, it is possible that a directory page cannot be split without creating an overlap among the newly created pages [BKK 96].

According to our framework of high-dimensional index structures, two heuristics have to be defined to handle the insert operation: The choice of a suitable page to insert the point and the management of page overflow. When searching for a suitable page, one out of three cases may occur:

- The point is contained in exactly one page region.
  In this case, the corresponding page is used.
- The point is contained in several different page regions.
  In this case, the page region with the smallest volume is used.
- No region contains the point.
  In this case, the region is chosen which yields the smallest volume enlargement. If several such regions yield a minimum enlargement, the region with the smallest volume among them is chosen.

The insert algorithm starts with the root and chooses in each step a child node by applying the rules above. Therefore, the suitable data page for the object is found in O (log *n*) time by examining a single path of the index.

Page overflows are generally handled by splitting the page. Four different algorithms have been published for the purpose of finding the right split dimension (also called split axis) and the split hyperplane. They are distinguished according to their time complexity with varying page capacity *C*:

- The exponential algorithm [Gut 84]:
  This algorithm encounters all 2^C distributions and determines the distribution with the lowest volume.
- The quadratic algorithm [Gut 84]:
  Here, the distribution process starts with the two objects which would waste the largest volume put in one group (the *seeds*). Iteratively, two groups are built by determining the volume enlargement in group 1 and group 2 (ve_1 and ve_2, respectively) for each object not yet assigned to a group. The element where the difference between ve_1 and ve_2 reaches its maximum is assigned to the group with the smaller enlargement.

- The linear algorithm [Gut 84]:
  The linear algorithm is identical with the quadratic algorithm up to the seed determination. For each dimension, the rectangle with the smallest lower boundary and the rectangle with the highest upper boundary are chosen. The distance is normalized by the sum of the extensions of all rectangles. The pair having the largest normalized distance is used as seed.
- Greene's algorithm [Gre 89]:
  First, the split axis is chosen. Then, the objects are distributed into two equally sized groups by sorting according to the lower boundary of the object in the corresponding dimension. The choice of the split axis is handled similar to the determination of the seeds in the quadratic algorithm.

While Guttman [Gut 84] reports only slight differences between the linear and the quadratic algorithm, an evaluation study performed by Beckmann, Kriegel, Schneider and Seeger [BKSS 90] reveals disadvantages for the linear algorithm. The quadratic algorithm and Greene's algorithm are reported to yield similar search performance.

The R*-tree [BKSS 90] is an extension of the R-tree based on a careful study of the R-tree algorithms under various data distributions. In contrast to Guttman who optimizes only for a small volume of the created page regions, the authors of the R*-tree identify the following optimization objectives:

- minimize overlap between page regions
- minimize the surface of page regions
- minimize the volume covered by internal nodes
- maximize the storage utilization.

The heuristic for the choice of a suitable page to insert a point is modified in the third alternative: No page region contains the point. In this case, the distinction is made whether the child page is a data page or a directory page. If it is a data page, then the region is taken which yields the smallest enlargement of the overlap. In case of a tie, further criteria are the volume enlargement and the volume. If the child node is a directory page, the region with the smallest volume enlargement is taken. In case of doubt, the volume decides.

Like in Greene’s algorithm, the split heuristic has two phases. In the first phase, the split dimension is determined as follows:

- For each dimension, the objects are sorted according to their lower bound and according to their upper bound.
- A number of partitionings with a controlled degree of asymmetry is encountered.
- For each dimension, the surface areas of the *MBR*s of all partitionings are summed up and the least sum determines the split dimension.

In the second phase, the split plane is determined, minimizing the following criteria:

- overlap between the page regions
- in doubt, least coverage of dead space.

Splits can often be avoided by the concept of *forced re-insert*. If a node overflow occurs, a defined percentage of the objects with the highest distances from the center of the region are deleted from the node and inserted into the index again, after the region has been adapted. By this means, the storage utilization will grow to a factor between 71 % and 76 %. Additionally, the quality of partitioning improves because unfavorable decisions in the beginning of the index construction can be corrected in this way.

Performance studies report improvements between 10 % and 75 % over the R-tree. In higher-dimensional data spaces, the split algorithm proposed in [BKSS 90] leads to a deteriorated directory. Therefore, the R^*-tree is not adequate for these data spaces, rather it has to load the entire index in order to process most queries. A detailed explanation of this effect is given in [BKK 96].

The R-tree and the R^*-tree have primarily been designed for the management of spatially extended 2-dimensional objects, but also been used for high-dimensional point data. Empirical studies [BKK 96, WJ 96], however, showed a deteriorated performance of the R^*-trees for high-dimensional data. The major problem of R-tree-based index structures in high-dimensional data spaces is the overlap. In contrast to low-dimensional spaces, there exists only few degrees of freedom for splits in the directory. In fact, in most situations there exists only a single “good” split axis. An index structure that does

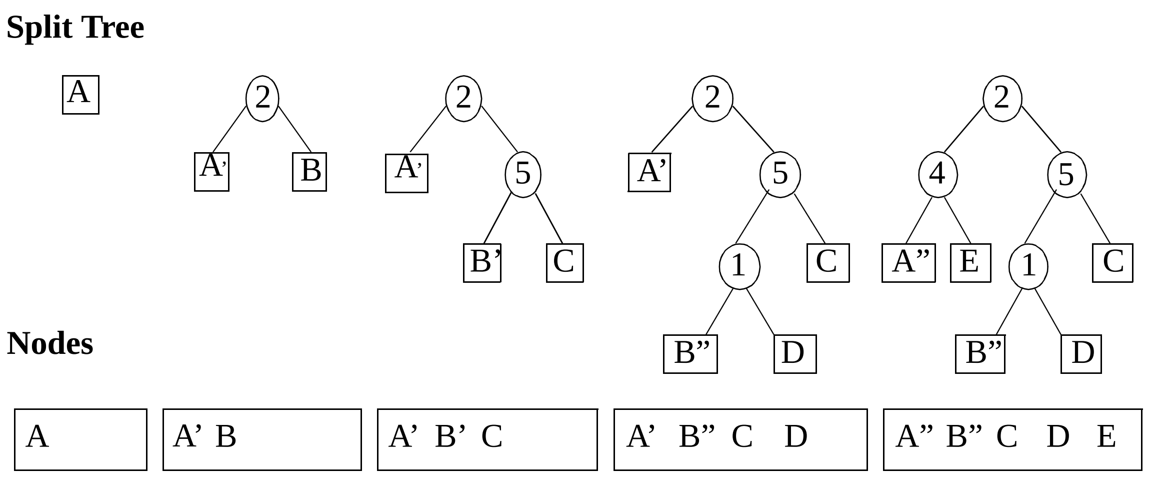


Figure 15: Example for the Split History.

not use this split axis will produce highly overlapping MBRs in the directory and thus show a deteriorated performance in high-dimensional spaces. Unfortunately, this specific split axis might lead to unbalanced partitions. In this case, a split should be avoided in order to avoid underfilled nodes.

The X-tree [BKK 96] is an extension of the R*-tree which is directly designed for the management of high-dimensional objects and based on the analysis of problems arising in high-dimensional data spaces. It extends the R*-tree by two concepts:

- overlap-free split according to a split-history
- supernodes with an enlarged page capacity

If one records the history of data page splits in an R-tree based index structure, this results in a binary tree: The index starts with a single data page *A* covering almost the whole data space and inserts data items. If the page overflows, the index splits the page into two new pages *A'* and *B*. Later on, each of these pages might be split again into new pages. Thus, the history of all splits may be described as a binary tree, having split dimensions (and positions) as nodes and having the current data pages as leave nodes. Figure 15 shows an example for such a process. In the lower half of the figure, the according directory node is depicted. If the directory node overflows, we have to divide the set of data pages (the MBRs *A”*, *B”*, *C*, *D*, *E*) into two partitions. Therefore, we have to choose a split axis first. Now, what are potential candidates for split axis in our example? Say, we choose dimension 5 as a split axis. Then, we had to put *A”* and *E* into one of the partitions. However, *A”* and *E* have never been split according to dimension 5. Thus, they span almost the whole data space in this dimension. If we put *A”* and *E* into one of the partitions, the MBR of this partition in turn will span the whole data space. Obviously, this leads to a high overlap with the other partition, regardless of the shape of the other partition. If one looks at the example in figure 15, it becomes clear that only dimension 2 may be used as a split dimension. The X-tree generalizes this observation and uses always the split dimension with which the root node of the particular split tree is labeled. This guarantees an overlap free directory. However, the split tree might be unbalanced. In this case it is advantageous not to split at all because splitting would create one underfilled node and another almost overflowing node. Thus, the storage utilization in the directory would decrease dramatically and the directory would degenerate. In this case the X-tree does not split and creates an enlarged directory node instead – a supernode. The higher the dimensionality, the more supernodes will be created and the larger the supernodes become. To operate on lower-dimensional spaces efficiently, the X-tree split algorithm also includes a geometric split algorithm. The whole split algorithm works as follows: In case of a data page split, the X-tree uses the R^*-tree split algorithm or any other topological split algorithm. In case of directory nodes, the X-tree first tries to split the node using a topological split algorithm. If this split would lead to highly overlapping MBRs, the X-tree applies the overlap-free split algorithm based on the split history as described above. If this leads to a unbalanced directory, the X-tree simply creates a supernode.

The X-tree shows a high performance gain compared to the R^*-trees for all query types in medium-dimensional spaces. For small dimensions, the X-Tree shows a behavior almost identical to the R-trees, for higher dimensions the X-tree also has to visit such a large number of nodes that a linear scan is less expensive. It is impossible to provide the exact values here because many factors such as the number of data items, the dimensionality, the distribution, and the query type have a high influence on the performance of an index structure.

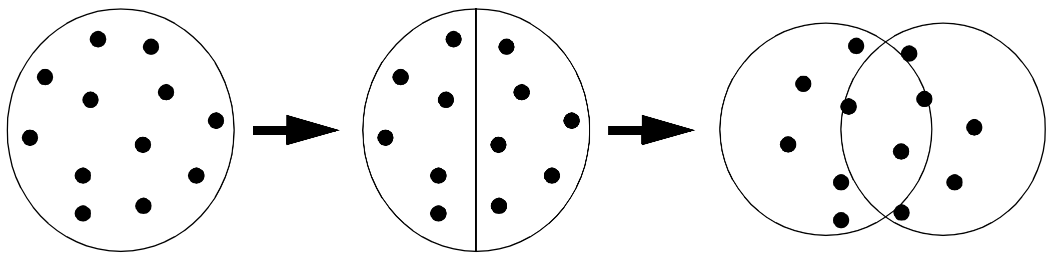


Figure 16: Situation in the SS-tree where no Overlap-Free Split is Possible.

3.1.6 SS-tree and TV-tree

In contrast to all previously introduced index structures, the SS-tree [WJ 96] uses spheres as page regions. For efficiency, the spheres are not minimum bounding spheres. Rather, the centroid point (i.e. the average value in each dimension) is used as center for the sphere and the minimum radius is chosen such that all objects are included in the sphere. Therefore, the region description comprises the centroid point and the radius. This allows an efficient determination of the MINDIST and the MAXDIST, but not of the MINMAXDIST. The authors suggest using the RKV algorithm, but they do not provide any hints how to prune the branches of the index efficiently.

For insert processing, the tree is descended choosing the child node whose centroid is closest to the point, regardless of volume or overlap enlargement. Meanwhile, the new centroid point and the new radius is determined. When an overflow condition occurs, a *forced reinsert* operation is raised, like in the R*-tree. 30% of the objects with the highest distances from the centroid are deleted from the node, all region descriptions are updated, and the objects are reinserted into the index.

The split determination is merely based on the criterion of variance. First, the split axis is determined as the dimension yielding the highest variance. Then, the split plane is determined by encountering all possible split positions which fulfill the space utilization guarantees. The sum of the variances on each side of the split plane is minimized.

The general problem of spheres is that they are not amenable to an easy, overlap-free split as depicted in figure 16. Therefore, the SS-tree outperforms the R^*-tree by a factor of 2, however, it does not reach the performance of the LSD^h-tree and the X-tree.

The TV-tree [LJF 95] is designed especially for real data that is subject to the Karhunen-Loève-Transform (also known as principal component analysis), a mapping which preserves distances and eliminates linear correlations. Such data yield a high variance and therefore, a good selectivity in the first few dimensions while the last few dimensions are of minor importance for query processing. Indexes storing KL-transformed data tend to have the following properties:

- The last few attributes are never used for cutting branches in query processing. Therefore, it is not useful to split the data space in the corresponding dimensions.
- Branching according to the first few attributes should be performed as early as possible, i.e. in the topmost levels of the index. Then, the extension of the regions of lower levels (especially of data pages) is often zero in these dimensions.

Regions of the TV-tree are described by so-called Telescope Vectors (TV), i.e. vectors which may be dynamically shortened. A region has k inactive dimensions and α active dimensions. The inactive dimensions form the greatest common prefix of the vectors stored in the subtree. Therefore, the extension of the region is zero in these dimensions. In the α active dimensions, the region has the form of an L_p-sphere where p may be 1, 2 or ∞. The region has an infinite extension in the remaining dimensions which are supposed either to be active in the lower levels of the index or to be of minor importance for query processing. Figure 17 depicts the extension of a telescope vector in space.

The region description comprises α floating point values for the coordinates of the center point in the active dimensions and one float value for the radius. The coordinates of the inactive dimensions are stored in higher levels of the index (exactly in the level where a dimension turns from active into inactive). To achieve a uniform capacity of directory nodes, the number α of active dimensions is constant in all pages. The concept of telescope vectors increases the capacity of the directory pages. It was experimentally

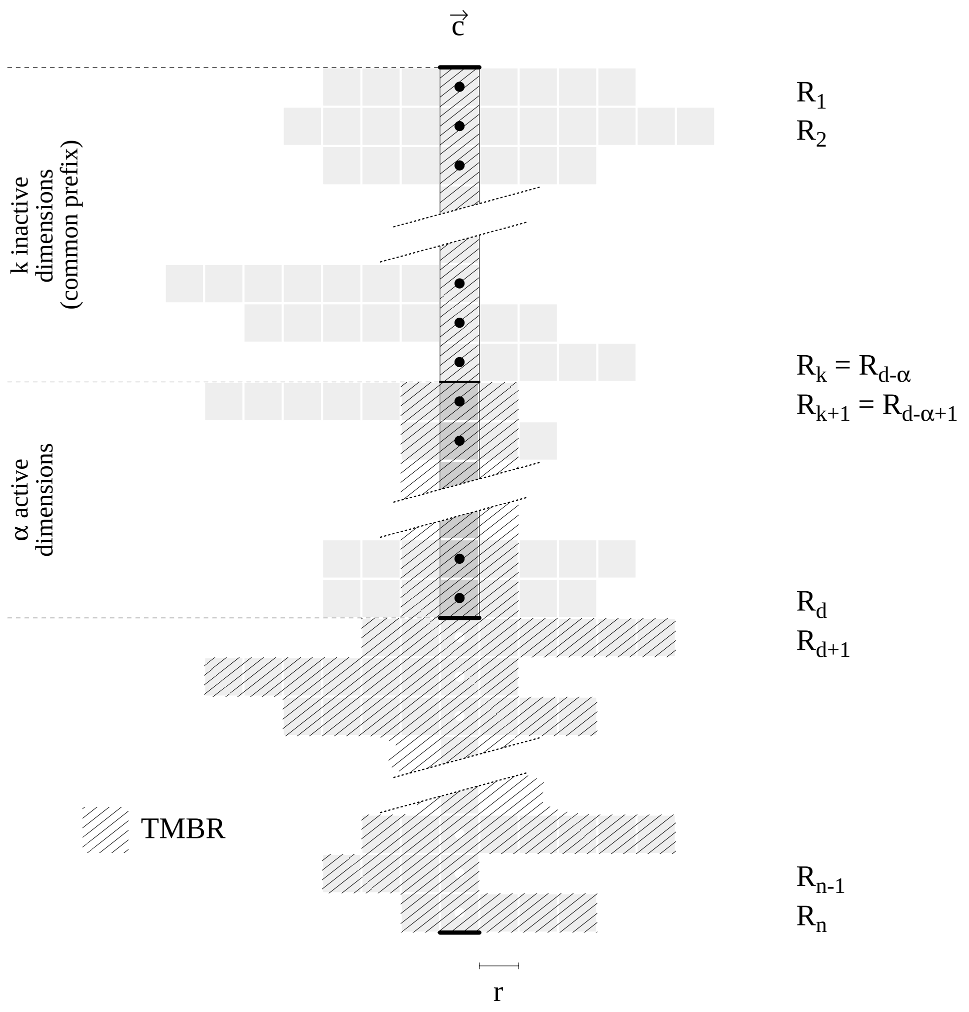


Figure 17: Telescope Vectors

determined that a low number of active dimensions ($\alpha = 2$) yields the best search performance.

The insert-algorithm of the TV-tree chooses the branch to insert a point according to the following criteria (with decreasing priority):

- minimum increase of the number of overlapping regions
- minimum decrease of the number of inactive dimensions
- minimum increase of the radius
- minimum distance to the center.

To cope with page overflows, the authors propose to perform a re-insert operation, like in the R^*-tree. The split algorithm determines the two seed-points (seed-regions in case of a directory page) which have the least common prefix or (in case of doubt) the maximum distance. The objects are then inserted into one of the new subtrees using the above criteria for the subtree choice in insert processing while the storage utilization guarantees are considered.

The authors report a good speed-up in comparison to the R^*-tree when applying the TV-tree to data that fulfills the precondition stated in the beginning of this section. Other experiments [BKK 96] however show that the X-tree and the LSD^h-tree outperform the TV-tree on uniform or other real data (not amenable to the KL transformation).

3.2 Algorithms for the Distance Range Join

After this short introduction to index structures for the *usual similarity search* we can turn ourselves to the *similarity join* algorithms. First we start with a few very simple algorithms following the *nested loop* paradigm. Due to their simplicity such algorithms can be applied to the vast majority of join predicates. Then we introduce the more sophisticated approaches applying index structures. Most of these algorithms have not been proposed for the similarity join but for the *spatial join* which is prevalent in the map overlay operation of a geographical information system. We show how these algorithms can be transformed for high-dimensional data spaces and for distance based join predicates rather than the polygon intersection.

3.2.1 Nested Loop Join

In relational join processing, the simplest approaches are several algorithms following the nested loop approach [Ull 89]. Due to their simplicity these algorithms can also be used for complex join predicates such as distance range joins, and also for k-closest pair queries and k-nearest neighbor joins.

Pure nested loop joins generate the complete set of point pairs (the cartesian product $R \times S$) and evaluate the join predicate for each point pair $(r,s) \in R \times S$. Both point sets are not organized by index structures, hashing or similar concepts but are stored in flat files without any specific order. Nested loop joins are thus the analogon of the sequential scan for simple similarity queries.

From the point of view of CPU, these algorithms are quite similar, as each point pair is generated and evaluated (i.e. the distance between the points is computed and compared to ε). A few optimizations of the CPU operations are possible but most optimizations of the nested loop join are concerned with I/O processing. Nested loop joins can be distinguished according to the strategy of the traversal of the two files.

3.2.1.1 The Simple Nested Loop Join

The simple nested loop join iterates in a first loop over all elements of *R* (therefore called the *outer* relation/point set) and in a second loop, nested in the first one, over all elements of *S*:

```
foreach $r \in R$ do
    read ($r$) ;
    foreach $s \in S$ do
        read ($s$) ;
        if $\|r - s\| \leq \varepsilon$ then output ($r,s$) ;
```

Thus, the inner point set *S* is scanned |*R*| times where |*R*| denotes the cardinality of the set *R*. This is usually not acceptable. Although most similarity join algorithms are clearly CPU bound on today's architectures, the simple nested loop join is I/O bound as reading of one point of *S* is usually more expensive than the corresponding distance calculation. An improvement of the simple nested loop join is described in the following.

3.2.1.2 Nested Block Loop Join

Rather than reading the outer set point by point and scanning the inner relation for each *R*-point, we can reserve a large block of the cache for the outer set *R*, read the outer set blockwise and scan the inner set *S* for each such block. This corresponds to the following algorithm:

```
foreach block $rb \subseteq R$ do
    read ($rb$) ;
```

```
foreach block $sb \subseteq S$ do
    read ($sb$) ;
    foreach $r \in rb$ do
        foreach $s \in sb$ do
            if $\|r - s\| \leq \varepsilon$ then output ($r$,$s$) ;
```

In contrast to the simple nested loop join the nested block loop join scans the inner data set S only $|R| / |rb|$ times. As we will show in chapter 7 it is beneficial to optimize the block capacities $|rb|$ and $|sb|$ carefully. Generally, the block capacity $|rb|$ of the outer set should be chosen larger (or in case of standard page sizes, rb should consist of more physical/logical pages) than the inner block capacity, because the larger the outer block is the fewer scans of S are due. However, if the remaining block capacity of sb is too small, the corresponding accesses become too expensive as for each access we have to take a rotational delay of the disk drive into account.

From a CPU point of view, the nested block loop join is equally expensive as the simple nested loop join as the number of distance calculations also corresponds to $|R| \cdot |S|$ and the management overhead is negligible. Typically, the nested block loop join is CPU bound. It may be competitive or even superior to more sophisticated methods whenever the selectivity of the join result is bad or whenever the applied indexing or sorting methods yield a bad performance (bad index selectivity) due to the *curse of dimensionality*).

3.2.1.3 Indexed Nested Loop Join

The indexed nested loop join needs a multidimensional index structure for the inner point set S. Therefore, it can also be classified as a join method upon preconstructed indexes (section 3.2.2). The term *nested loop* is additionally misleading as we have only one loop iterating over the points of R and performing an index based similarity query (in this case, range query) which does not simply correspond to a loop but is a more complex database primitive. However, due to its simplicity and flexibility (it can be easily transformed into an algorithm for other join predicates such as *k*-closest pairs or *k*-nearest neighbors) we describe it here. The corresponding algorithm is given below:

```
foreach $r \in R$ do
    read ($r$) ;
    $sresult$ := RangeQuery ($r, S, \varepsilon$) ;
    foreach $s \in sresult$ do
        output ($r,s$) ;
```

As the outer set is not ordered, typically a cache (which could be applied in the range query processor) does not exhibit any locality. Depending on the index structure and algorithm which is applied to process the range queries, the indexed nested loop join is typically I/O bound as the range query is I/O bound. Typically the indexed nested loop join is not competitive with other more sophisticated methods (including nested block loops). The only exception are situations where the index yields a very good performance and the outer point set is small, such that simultaneous processing of several similarity queries cannot bear any improvement, anyway.

3.2.1.4 Multiple Queries

The general idea of the Multiple Queries approach [BEKS 00] is to select a number of points of *R* and to evaluate the corresponding queries simultaneously while traversing the index structure (R-tree) constructed for *S*. This way, many different types of similarity joins can be implemented, the distance range join as well as join operations with nearest neighbor based join predicates. The authors of the multiple queries technique were not conscious about the fact that this technique can efficiently support the similarity join operation. Rather, they directly supported algorithms of similarity search and data mining by the simultaneous execution of queries. From this point of view, the multiple queries technique is a competitor of the similarity join as a database primitive for high performance data mining. A comparative evaluation [Bra 00] of this aspect had the result that it is often easier to implement data mining algorithms on top of the multiple queries paradigm than on top of the similarity join. The efficiency potential of the multiple queries technique, however, is much more limited.

Besides the general idea to evaluate similarity queries simultaneously while traversing the index structure, [BEKS 00] also proposes a technique to accelerate and partially avoid the distance computations between point pairs. The idea is to exploit the triangle inequality to avoid distance calculations between feature vectors.

The following algorithm in pseudocode describes the general idea of the implementation of the distance range join using the multiple queries paradigm. In contrast to the algorithm in section 3.2.1.3 (indexed nested loop join) the outer loop iterates over the blocks of *R*. Instead of the usual range query, the multiple query version is called.

```
foreach block $rb \subseteq R$ do
    read ($rb$) ;
    $sresult$ := MultipleRangeQuery ($rb, S, \varepsilon$) ;
    foreach ($r,s$) $\in$ $sresult$ do
        output ($r,s$) ;
```

To evaluate the multiple range query, we give a recursive schema which performs a depth-first index traversal similarly to a single range search. In detail we list only the recursive part (directory pages):

```
function MultipleRangeQuery ($rb, pg, \varepsilon$): set of pair of Point
    var $match$: bool:= false ;
    var $result$: set of pair of Point := $\emptyset$;
    foreach $r \in rb$ do
        if mindist ($r, pg$) $\leq \varepsilon$ then
            $match$ := true ;
    if $match$ then
        read ($pg$) ;
        if IsDirectoryPage ($pg$) then
            foreach $p \in pg$.children do
                $result$ := $result \cup$ MultipleRangeQuery ($rb, p, \varepsilon$) ;
        else
            ....
```

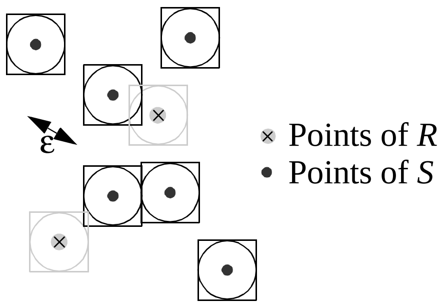


Figure 18: Cube approximation of points

3.2.2 Algorithms upon Preconstructed Indexes

In this section, we describe the R-tree spatial join (*RSJ*) [BKS 93] and a few variants of this algorithm. Generally, RSJ is not an algorithm for the similarity join but for the spatial join which is primarily used in geographical applications for the map overlay operation. There, the joined objects are 2D polygons and the join predicate is the intersection of the polygons. However, RSJ can be easily generalized to higher dimensions and to distance based predicates for points rather than intersection predicates for spatially extended objects.

The most obvious way of this generalization is to approximate the points of the two sets by hypercubes of side length ε (cf. figure 18). This way, we get a conservative approximation, because if two points have a distance of no more than ε, the associated hypercubes must intersect. In higher dimensions, the selectivity of this filter step deteriorates because a cube is a quite coarse approximation. Moreover, if the distance parameter ε changes, it would be necessary to construct a new index.

A better idea is to store the points without approximation in a multidimensional index and to consider the distance predicate in join processing. In the leaf level of the index, obviously the distances between points must be computed as usual. More complex is the determination of the distance between page regions to decide whether or not a pair of pages must be considered. We need a distance measure which considers a pair of pages

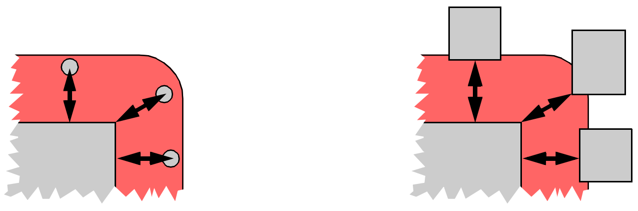


Figure 19: MINDIST for similarity search (l.) and join (r.)

if and only if it is possible to contain a pair of matching points. For plain similarity search, a suitable distance measure for that purpose is the MINDIST between the query point and the page region which is defined below and depicted on the left side of figure 19. This distance measure forms a sum over the dimensions. In each dimension, the distance between the corresponding coordinate of the query point q_i and the interval of the page region $[R.\mathrm{lb}_i, R.\mathrm{ub}_i]$ is determined:

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

This principle can be generalized for the distance between two page regions by summing up the squared differences between the intervals. The MINDIST between two page regions is visualized on the right side of figure 19.

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

3.2.2.1 R-tree Spatial Join

The R-tree Spatial Join (RSJ) [BKS 93] performs a strict depth first traversal of the two index structures. Both trees are descended simultaneously, in the most basic form of RSJ without any prioritization. Provided that both indexes are of equal height, the algorithm proceeds as follows: First the two roots are considered. If the two page regions associated with the roots have a distance (MINDIST) of no more than ε, then the roots are loaded from disk and the algorithm considers all pairs of child pages. For each pair of child pages, the MINDIST is determined, and, provided this distance does not exceed ε, the algorithm is called recursively. The corresponding pseudocode is given below.

```
procedure rtree_similarity_join ($R$, $S$: page)
    load ($R$) ;
    load ($S$) ;
    if is_data_page ($R$) and is_data_page ($S$) then
        foreach $r \in R$ do
            foreach $s \in S$ do
                if $\|r - s\| \le \varepsilon$ then output ($r,s$) ;
    else (* both pages directory pages *)
        foreach $r \in R$ do
            foreach $s \in S$ do
                if MINDIST ($r$, $s$) $\le \varepsilon$ then
                    rtree_similarity_join ($r$, $s$) ;
```

As in general, each page is paired with several pages of the other set, it is often necessary to load a page more than once from disk. A cache is intended to avoid or shadow many of these redundant accesses. In order to exhibit a high cache hit ratio, it is possible to apply some strategy to determine the order in which the pairs of child pages are generated. The authors of RSJ propose the local application of a plane sweep algorithm instead of the two simple nested loops in the else-branch of the algorithm above (cf. figure 20). It has been shown by Huang et al. [HJR 97] that global optimization yields a higher optimization potential than the local plane sweep algorithm (cf. section 3.2.2.3).

3.2.2.2 Parallel RSJ

The parallel version of RSJ [BKS 96] consists of three phases,

- task creation (non parallel),
- task assignment to processors (non parallel), and

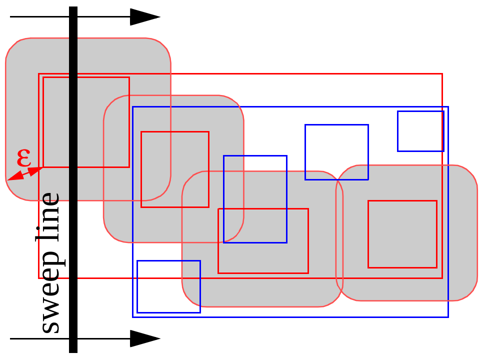


Figure 20: Plane sweep as a local optimization of RSJ

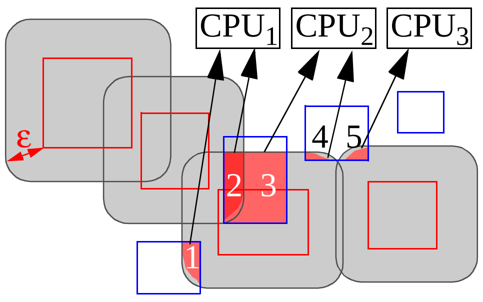


Figure 21: Task definition and static range assignment

- task execution (completely executed in parallel).

The notion of a task is here defined as a pair of (joining) subtrees which are generated at a very high level, i.e. typically the first or second level underneath of the root. The level is chosen such that the number of tasks is high enough to allow good load balancing but not too high to keep the management overhead for task creation and assignment low. In the following example depicted in figure 21 we have 5 tasks corresponding to 5 pairs of regions which are associated for instance with the nodes of the first tree level following the root.

The tasks are generated by a plane-sweep algorithm as shown before in figure 20 to preserve locality among subsequent tasks. The authors define several strategies for the assignment of tasks to processors:

- Static range assignment (cf. figure 21):
  Each of the *p* processors receives $n / p \pm 1$ subsequent tasks where *n* is the number of tasks. This strategy preserves most locality for the processors, i.e. least effort for data redistribution.
- Static round robin assignment:
  Tasks are dispatched in a round robin fashion. Locality is not preserved, but a better load balancing is achieved.
- Dynamic task assignment:
  The processors request a new task (i.e. basically round robin) whenever they are idle. Best load balancing.

In the experiments, the authors report for spatial data a near-linear speed-up if the number of disks is scaled up together with the number of processors.

3.2.2.3 Breadth-First R-tree Join

Like RSJ, the breadth-first R-tree join (BFRJ) [HJR 97] has been proposed for spatial databases and an intersection join predicate but can be generalized for similarity predicates in a straightforward way. The authors address shortcomings of RSJ which are caused by its strict depth-first traversal. As in a depth-first strategy each pair of tree branches must be processed to its end before a new pair of branches can be started it is not possible to apply global optimization strategies for ordering of the pairs. It is merely possible to apply local ordering strategies which are applied when determining the pairs of child pages for a single pair of parent pages. The optimization potential, however, is higher if in the access strategy a larger set is considered, e.g. the set of pairs of a complete R-tree level.

To organize a breadth-first traversal of a tree index structure, the authors propose a so-called *intermediate join index* (*IJI*), a list of all joining page pairs of an index level. The algorithm starts with IJI = $\langle(root_R, root_S)\rangle$. Then in each step, all entries of the IJI are expanded, i.e. the corresponding pages are retrieved from disk and replaced by the join-

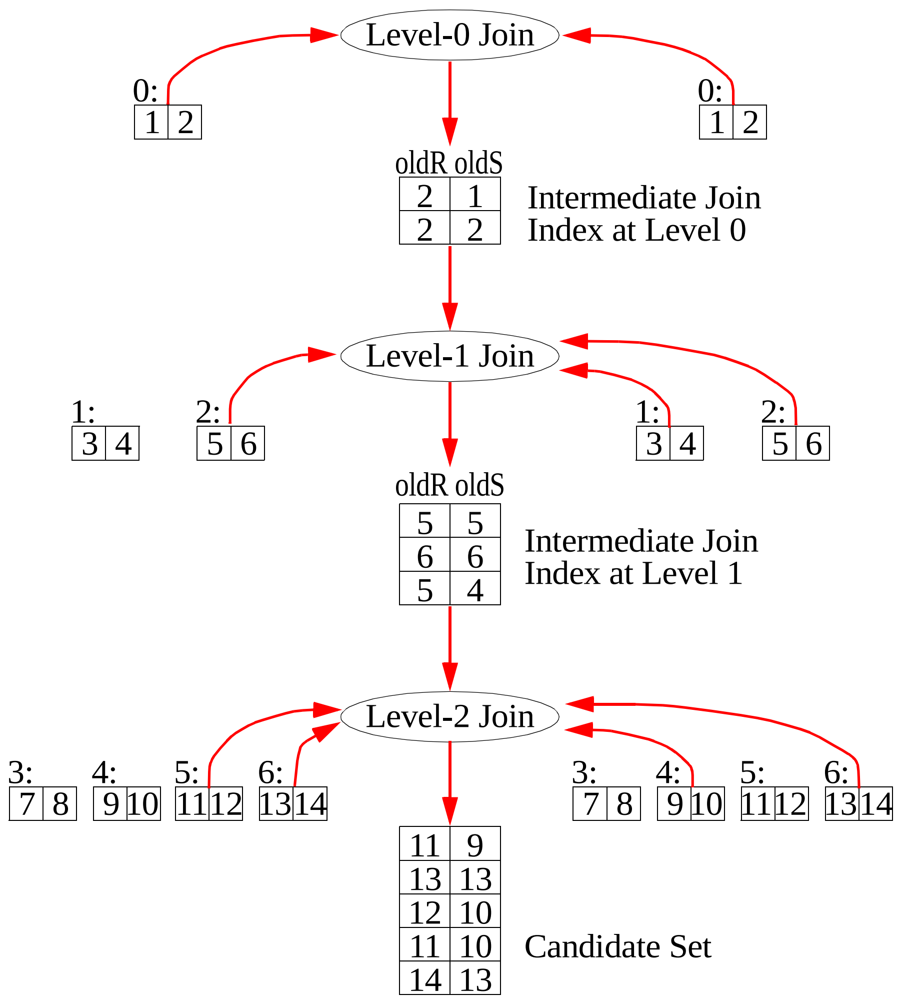


Figure 22: Breadth-first traversal and intermediate join index

ing pairs of child pages until the leaf level of both indexes is reached. This concept is visualized in figure 22 for the first three levels of the R-trees: First, the two roots (with ID=0) are joined (level-0 join). The result is the intermediate join index with the two pairs (2,1) and (2,2) which is the basis for the level-1 join.

Global optimization strategies are applied by sorting the entries of the IJI. For suitable ordering criteria, the corresponding MBRs of the pages can be used. The authors propose 5 different strategies:

- No particular order
  Note that this does not correspond to a random order but is influenced by the hierarchical order of the tree. Nodes in a common branch of the tree are always adjacent in the IJI without any particular order.
- Lower *x*-coordinate ($r.\mathrm{lb}_0$) of the nodes of *R*
- The sum of the centers of the *x*-coordinates of *R* and *S*:
  $(r.\mathrm{lb}_0 + r.\mathrm{ub}_0) / 2 + (s.\mathrm{lb}_0 + s.\mathrm{ub}_0) / 2$
- The *x*-coordinate of the center of the common MBR of *R* and *S*:
  $(\min \{r.\mathrm{lb}_0, s.\mathrm{lb}_0\} + \max \{r.\mathrm{ub}_0, s.\mathrm{ub}_0\}) / 2$
- The Hilbert-value of the center of the common MBR of *R* and *S*.

The authors present only experiments on 2d polygon data which are not representative for high-dimensional similarity joins. In these experiments, the simple strategy of ordering only according to the lower *x*-coordinate of *R* is the winner. The more sophisticated strategies are even outperformed by the strategy *no particular order* which takes the child pairs in the order in which they are generated. The standard RSJ algorithm is outperformed but only by a small factor.

3.2.3 Index Construction on-the-Fly

If no preconstructed index for the two joining sets exist, it is possible to construct the corresponding index structures temporarily for the join. The usual R-tree construction methods by repeated call of the INSERT operation for each data object, however, turns out to be too expensive. During the last years a few methods for a fast bottom-up construction of R-tree like index structures have been proposed which do not in every case generate an index of high quality (e.g. Hilbert-R-trees are typically outperformed by original R-trees or R*-trees). In most cases, the quality of bottom-up constructed index structures is high enough to speed up join operations such that the index construction cost is amortized. Well-known index construction methods are:

- Hilbert R-trees [KF 94] sort the data points according to a space-filling curve (the Hilbert curve) and pack sequences of adjacent points into pages which are then grouped into directory pages. For each page, the MBR is determined.

- Buffer trees [BSW 97] are a generic technique for any kind of index structure. The idea is to delay insert operations by additional buffers for data records which are associated to non-leaf nodes of the tree. Points are propagated to the next deeper level on buffer overflows.
- Repeated partitioning [JW 96, BBK 98] sort the data set according to different dimensions and partition thus the data set until the data page capacity is reached.

In contrast to these methods for the fast construction of general purpose index structures for similarity search and similar applications, also a few methods have been proposed that construct *specialized* index structures which are particularly suited for spatial joins and similarity joins. These approaches are introduced in this section.

3.2.3.1 Seeded Trees

The general assumption of the idea of seeded trees [LR 94] is that only one of the two joined sets (say *R*) is supported by an index. If neither of the two sets is indexed, the index for *R* can be constructed bottom-up. The idea is then to exploit the knowledge about the partitioning of *R* to construct an index for *S* which can be efficiently matched with the other tree. An obvious observation is that two index structures can be particularly efficiently matched if both partition the data set in a similar way (cf. figure 23). On the left side, we show the *partitions* of the index of set *R* denoted by R_1 to R_4 and a few *objects* of the set *S* (small squares). In the middle, a typical partitioning for *S* is shown which is generated by a dead space minimization of the R*-tree. In this scenario, each of the R-partitions is joined with two partitions of *S* (assuming a spatial intersection join or a similarity join with a small parameter ε). On the right side, *S* is partitioned such that the *R*-partitions are used as templates. Although these *S*-partitions cover more space than the partitioning depicted in the middle, considerably fewer pairs must be formed.

The idea of seeded trees is, therefore, to use the first few levels of the index for *R* as a template for the index *S*. Instead of beginning the index construction with an empty root node, we begin with a template tree the leaf nodes of which are empty. For this purpose, the first levels of *R* are simply copied (*seeding phase*). The template tree with the empty leaves is called the *seed level* of the seeded tree. The empty leaves which are associated with a minimum bounding rectangle but not stored on disk (represented by empty point-

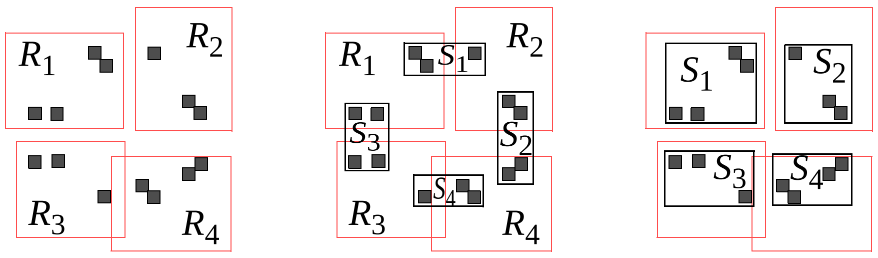


Figure 23: Matching of similar and dissimilar partitions

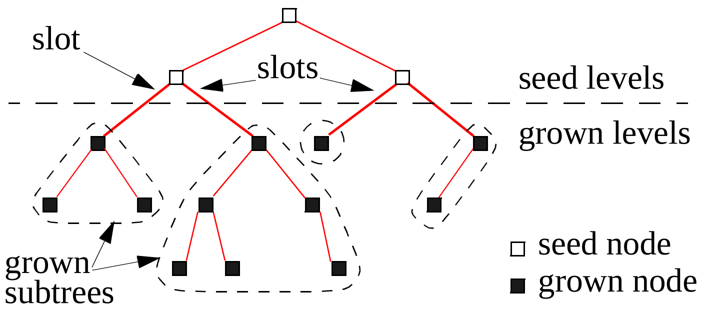


Figure 24: The seeded tree

ers) are called *slots*. During the tree construction (the *growing phase*) this seed level is not changed. The points are inserted to the tree by applying the choose_subtree strategy of the R-tree. Whenever an inserted object reaches an empty slot, a new node is generated which is further treated like the root node of a usual R-tree, i.e. it is not forced to have a certain storage utilization and upon an overflow of this node, a new "root" is allocated and the complete subtree grows by one level. As depicted in figure 24, the corresponding subtrees are called *grown subtrees* and the levels of the tree are called *grown levels*. Although each grown subtree is balanced the seeded tree as a whole is unbalanced. Note that the property of balance is actually not needed for join processing.

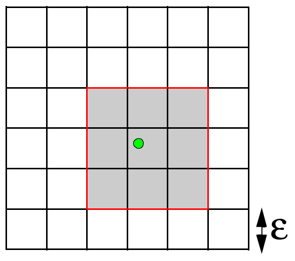


Figure 25: Grid partitioning of the ε-kdB-tree

3.2.3.2 Epsilon-kdB-tree

The general idea of the ε-kdB-tree [SSA 97] is to apply a grid partitioning of the data space where the distance of the partitioning planes exactly corresponds to the query parameter ε (cf. figure 25). The general advantage of such a grid approach is that the part of the data space in which a join partner of a point may be positioned is well restricted to neighboring grid cells (shaded area). With increasing dimension, however, the number of neighboring grid cells increases drastically to $3^d - 1$. Therefore, it is no good idea to consider the grid cells one by one or even to retrieve them separately from disk. Grid based approaches have to apply more sophisticated ideas to avoid this problem. The approach of the ε-kdB-tree is to use only a part of the dimensions for partitioning. As we will describe later, it uses as many dimensions for partitioning as are needed to achieve a suitable number of points per cell.

In order to do I/O processing, even only one dedicated dimension is used for partitioning. The authors assume that the data file is already sorted according to this dedicated dimension. Therefore, for this first partitioning step, no further sorting is necessary. The file can be partitioned into stripes by simply reading as many points until the next stripe boundary is reached (cf. figure 26). As the authors further assume that each pair of neighboring stripes fits into main memory, no *external* sorting step is needed at all.

Once a pair of stripes has been loaded into main memory, for each of the stripes a main memory data structure called ε-kdB-tree is constructed (cf. figure 27). The ε-kdB-

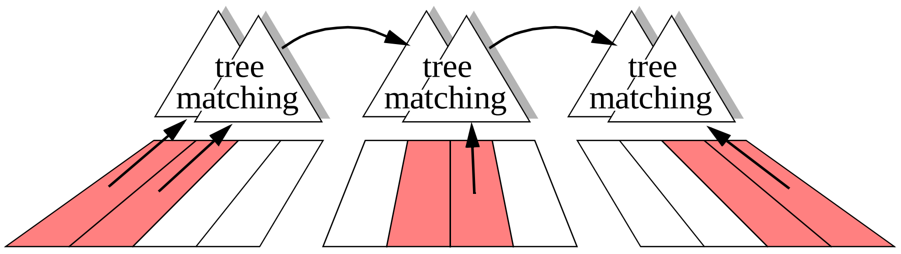


Figure 26: Join algorithm of the ε-kdB-tree

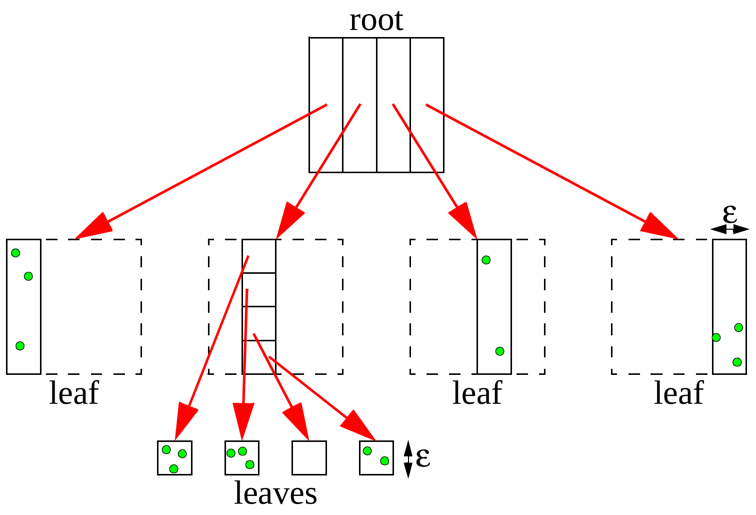


Figure 27: Structure of the ε-kdB-tree

tree is an unbalanced tree where each inner node partitions the data set according to a selected dimension. The fan-out of a node corresponds to 1/ε provided that the data space is normalized to $[0..1]^d$. Some of the child pointers may be NULL if the corresponding grid cell does not contain any data point. Leaf nodes are data nodes and have a defined capacity to store data points. The ε-kdB-tree is constructed top-down by repeatedly sorting and partitioning of the data set until the defined node capacity is reached. Each ε-kdB-tree must be matched with itself and with the two trees for the neighboring stripes. This is done in a straightforward way. Matching of ε-kdB-trees performs very efficiently and is a filter step with a relatively good selectivity. After processing a pair of stripes, the data structure of the first stripe is discarded and replaced by the tree for the next stripe. The file is processed in a strictly linear way never accessing any part more than once.

The most important limitation of the approach is the assumption that any pair of stripes fits into the main memory, which may be unrealistic for skewed and high-dimensional data sets. Skew may lead to the situation that one data stripe contains considerably more points than the rest. But even for uniformly distributed points it has been shown [BK 01a] that $\varepsilon = 0.3$ is a typical situation for high-dimensional data spaces. Therefore, a pair of stripes contains about 60% of all database points. For such situations, the ε-kdB-tree approach is not really scalable. To solve this problem, the authors propose to partition the data space according to more than one dimension in the first step. This solution, however, removes most of the advantages of the approach. The file must be explicitly sorted according to two dimensions at the beginning. And later, there is no strictly linear access pattern but a complex one which reads parts of the file multiple times. Finally, the problem of a high memory requirement is not really solved, because for our running example ($\varepsilon = 0.3$) our algorithm must simultaneously hold 4 adjacent cells which contain approximately $0.3^2 = 9\%$ of the data points each. A total of 36% of the data must be held simultaneously in main memory.

3.2.3.3 Parallel ε-kdB-trees

Shafer & Agrawal [SA 97] have also proposed a parallel version of the ε-kdB-tree. As both the construction of the ε-kdB-tree as well as matching of two trees are expensive, both is parallelized. The assumption is that the data set is randomly distributed over all processors each of which has approximately N / p points. Each processor constructs an ε-kdB-tree of its own set. During this construction the processors exchange information about their splits to enforce that all constructed trees have the same structure as depicted in figure 28 where the gray lines indicate splits which are introduced by this *split broadcast*. Finally a union operation for all trees of the processors is executed and if node sizes are still too large, the leaf nodes of the resulting tree are further split. Then, the workload

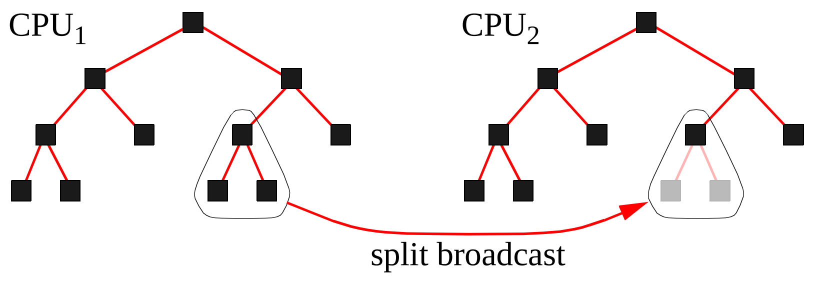


Figure 28: Enforcing equal structures for all ε-kdB-trees

for tree matching is statically distributed according to the estimated cost. The cost of a leaf node is estimated by the following formulas:

- $|r| \cdot |s|$ for the join of different leaves
- $|r| \cdot (|r| + 1) / 2$ for the self join of a leaf.

The join units are clustered to preserve locality and minimize the redistribution effort and replication. Data redistribution is performed asynchronously to avoid network flooding.

3.2.3.4 Plug & Join

Plug & Join [BSS 00] is a generic technique which is suitable for several different join operations including spatial join and similarity join. The idea is to construct a main memory R-tree template from a sample of *R*. Instead of data nodes, the leaves are associated with partitions on the disk and with main memory buffers. After construction of the tree template, the set R is partitioned by inserting points into the tree (i.e. actually, only the choose_subtree procedure is called). At the leaf level, the points are inserted into the buffers which are flushed to disk on overflow (cf. figure 29). After this first partitioning phase, some of the partitions are flushed to disk, others may still lie completely in the buffer. In the next phase, *S* is also partitioned according to the template R-tree. The difference, however, is now that partitioning of *S* introduces object replication. Each object is dispatched to every joining partition. If the joining partition has never been flushed to disk, then the corresponding pairs can be immediately answered from the

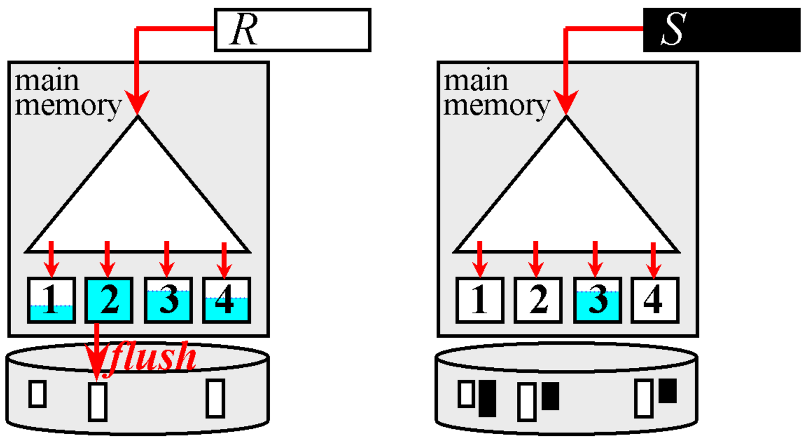


Figure 29: Plug & Join

buffer. Otherwise, the S-object is stored in the buffer. Upon buffer overflow, all buffered objects are joined with the corresponding partition on disk.

3.2.4 Join Algorithm Based on Object Replication

In 1996 and 1997, two groups proposed spatial join algorithms based on spatial hashing. Both of approaches involve object replication.

3.2.4.1 Spatial Hash Join

The general method proposed by Lo and Ravishankar in [LR 96] is to partition the set *R* without any replication. Then, the object set *S* is partitioned according to the buckets of *R*. This second step involves object replication whenever an *S*-object intersects more than one bucket region of *R*. Finally, only pairs of corresponding buckets must be joined.

To generate the initial partitioning of *R*, the idea of the *seeded tree* [LR 94] is reused (cf. section 3.2.3.1). The initial process of generating the slots of the seed level is called *bootstrap seeding*: A suitable number *ns* of slots is determined. Then, the set *R* is sampled with a sample size of *c* times the number of slots with some small constant *c*. Using some simple clustering method, in the set *R* a number of *ns* cluster centers are determined which are used as slots in the seeded tree (initially without spatial extension).

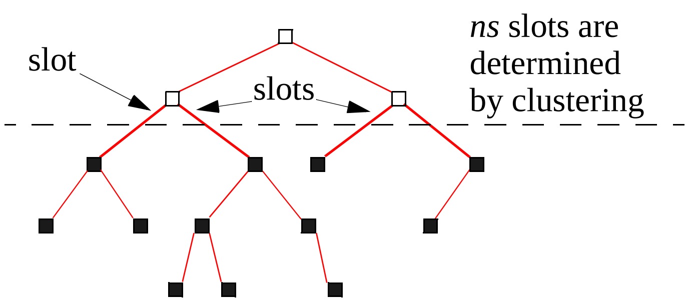


Figure 30: Bootstrap Seeding of the Spatial Hash Join

Then the seeded tree grows by applying the criterion of minimum slot enlargement (cf. figure 30).

For partitioning of *S*, the seeded tree of *R* with all bucket extents is copied. In the original proposal for the spatial intersection join each object *s* of *S* is assigned to all buckets *b* which are intersected by *s*. This approach can be generalized for the similarity join which has to assign each object $s \in S$ to all buckets *b* with

$$\text{mindist}\,(b,s) \le \varepsilon.$$

This step involves the object replication.

All corresponding bucket pairs (*r*, *s*) are joined by constructing a quadratic split R-tree on *r*. Each object in *s* is probed to the R-tree on *r*.

3.2.4.2 Partition Based Spatial Merge Join

The other spatial join method called *Partition Based Spatial Merge Join* (*PBSM*) also uses object replication. Originally, it has also been proposed as a join method for the intersection join predicate of spatial polygon sets and can be extended to the similarity join. In contrast to the spatial hash join, PBSM does not construct any hierarchical index but decomposes the data space regularly into tiles of the same size. The partitions either directly correspond to such tiles or are determined from the tiles using hashing (cf.

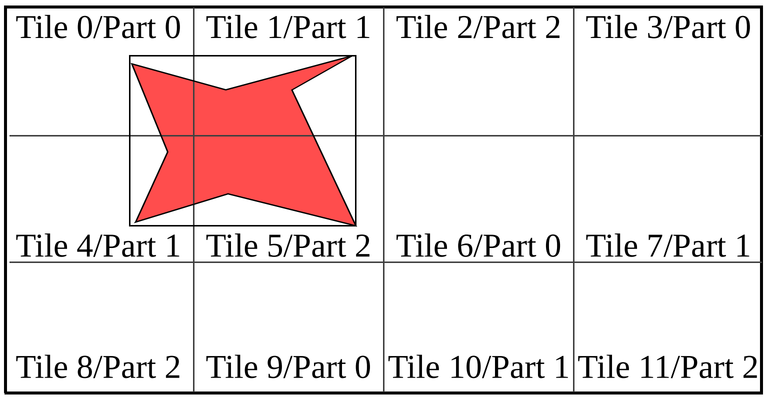


Figure 31: Partition Based Spatial Merge Join)

figure 31 where the tiles are canonically numbered and the partitions are assigned by the "modulo 3" hashing function applied to the tile number).

The Partition Based Spatial Merge Join potentially replicates both the outer set *R* as well as the inner set *S*. A spatially extended objects is assigned to every partition which is intersected by it. Consequently, for the similarity join, we assign each object to all partitions to which it has a distance (mindist) not exceeding ε/2. This is done for the objects of *R* and *S*. The advantage of this approach is that each partition of *R* must be joined with *exactly* one other partition of *S*. In contrast, for the Spatial Hash Join, it is possible that an *S* partition intersects with more than one *R* partitions. The disadvantage, however, is that both sets are subject to object replication. The consequence is in particular that also the object pairs which are generated by the join algorithm may contain some duplicate pairs. This is even a frequent condition because whenever an object pair is hashed to the same two partitions, the join result contains such duplicates. These duplicates must be eliminated e.g. by sorting or hashing according to the pair of object identifiers (OID_r,OID_s).

The initial numbers of partitions is determined according to the following formula:

$$\text{initial} = \left\lceil (|R| + |S|) \cdot \frac{\text{size\_point}}{\text{size\_memory}} \right\rceil$$

This formula intends to choose the number of buckets such that each pair of corresponding buckets fits into the main memory. However, it does not take into account

- object replication
- and data skew

for the estimation which clearly limits the value of this formula.

3.2.5 Join Algorithms Based on Sorting

3.2.5.1 Z-Order

Several approaches are based on the concept of space filling curves such as the Z-order, Gray codes or the Hilbert curve. The principle of space filling curves is depicted on figure 32 (left side). First we start with the data space which must have fixed space boundaries. The complete data space is associated with an empty bitstring ⟨⟩. In the 2-dimensional case, the data space is regularly decomposed into 4 quadrants which are associated with the 4 bitstrings ⟨00⟩, ⟨01⟩, ⟨10⟩, and ⟨11⟩. The exact order of the 4 quadrants depends on the type of the space filling curve (the Z-order has a different assignment of bitstrings than Hilbert or Gray-codes). Each of these quadrants can be further decomposed in a recursive way, which generates longer bitstrings. In the case of a general dimensionality (d), the data space is decomposed into 2^d cells in each step which are described by bit strings of length d. The decomposition stops when a specified resolution is reached.

If space-filling curves are used for the similarity join, it is necessary to consider the points as extended objects (spheres of radius $\varepsilon/2$) which are approximated by the grid cells. Obviously, if two such objects are completely contained in different cells as depicted in the leftmost example in figure 33, then they are guaranteed to have a distance of more than ε and are thus not a join result. Therefore, candidates can be efficiently gener-

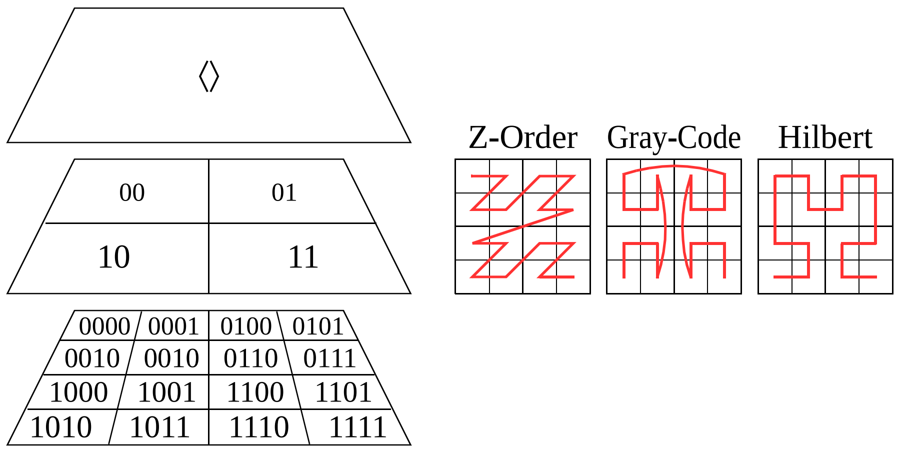


Figure 32: Space filling curves

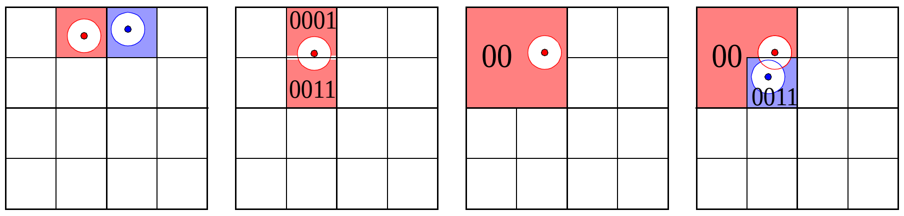


Figure 33: Joins on space-filling curves with and without replication

ated by grouping together objects by cell identifiers, e.g. by sorting according to the bit strings (i.e. a sort-merge join).

Unfortunately, the spheres are not often completely contained in a grid cell. Typically, the grid cell boundaries are intersected as in the second and all following examples of figure 33. In this case, the join partners could be in all cells which are intersected by the sphere. One solution to this problem could be to replicate the objects and store them once for each intersected cell. Note, however, that the number of intersected cells is exponential in the data space dimension.

Another possible solution is depicted in the third example on figure 33. Not only grid cells at the basic resolution are considered as approximations but at every resolution which is generated by the recursive decomposition process. Each sphere is approximated by the smallest cell in which the sphere is completely contained. In the extreme case where a sphere is intersected by the first split line, only the complete data space $\langle\rangle$ can be used. As depicted on the right side of figure 33 two points can be join mates if they are in the same cell or if one of the cells is contained in the other. In terms of bit strings, a cell c_1 is contained in c_2 if the bitstring which is associated with c_2 is a prefix of that of c_1. Therefore, our filter has to match the files such that each bitstrings $b(r)$ of R are matched with all bitstrings of S which are either prefixes of $b(r)$ or of which $b(r)$ is a prefix.

The algorithm of Orenstein [Ore 91] sorts each of the files R and S according to the bit strings in lexicographical order and performs basically a sort merge join. To find not only exact matches between the bit strings but also prefix matches, the algorithm exploits the property of the lexicographical order that a prefix $p(b)$ of a bitstring b appears in the sequence before the bitstring b. When performing the sort merge join, all elements of R and S the bitstrings of which are prefixes of the *current* element of R and S, respectively, are stored temporarily in a suitable data structure. They are deleted from the data structure as soon as they lose the prefix property of the current element. When an element is deleted from the data structure, the files will not contain any further join partners of the deleted element. A suitable data structure is depicted in figure 34. A stack organizes all prefixes of the current element of R and S, respectively. The points which are associated with the corresponding bit strings are stored by linked lists. Whenever the current element changes, the algorithm checks which prefixes have changed (only bit operations are needed for that check) and the corresponding linked lists are discarded. Orenstein also proposed methods to optimize replication in his algorithm [Ore 89]. Another method which is based on a cost model is GESS [DS 01].

3.2.5.2 Multidimensional Spatial Join

Koudas and Sevcik proposed the *Size Separation Spatial Join* (*SSSJ*) for 2D polygon databases [KS 96] and the *Multidimensional Spatial Join* (*MSJ*) for the similarity join

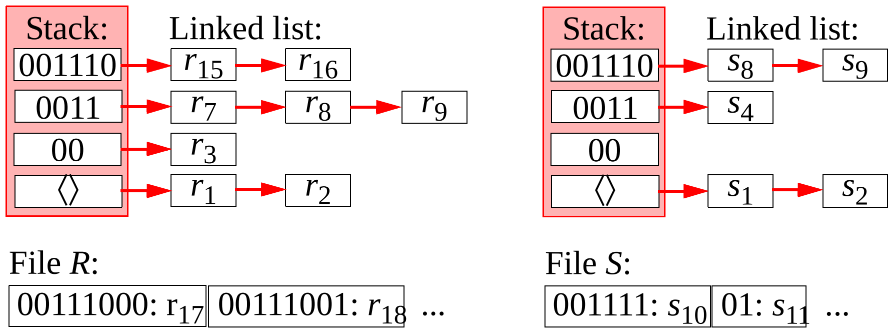


Figure 34: Data Structure for Orenstein's Spatial Join

[KS 97]. Both algorithms are also based on the Z-order of the associated bit strings. In contrast to Orenstein's algorithm, their algorithm first dispatches the points into so-called *level files*. A level file contains all points where the associated bit strings have a defined length, e.g. the level-0 file contains all points which are associated with the bit string of length 0, etc. Then, a multiway sort merge join is performed over all level files of R and S to match all bitstrings with their prefixes in the other data set.

It has been pointed out in [BK 01] that both MSJ as well as Orenstein's algorithm suffer from similar performance problems as the ε-kdB-tree in high-dimensional data spaces. E.g. for uniformly distributed points of an 8-dimensional data space with $\varepsilon = 0.3$, the probability that the first intersection line is intersected, corresponds to 30%. Of the remaining 70%, another 30% intersects the second partitioning line, and so on. This leads to a total expected value of 46% of the data files to be held in main memory. These results could also be experimentally confirmed for real data of a CAD application which needed 26% of all data simultaneously in main memory.

3.3 Nearest Neighbor Based Join Algorithms

The most important drawback of the distance range join is that it is difficult for the user to control the selectivity of this join operation. The distance range join behaves similarly to the range query where the user may get an empty result if ε is chosen too small and the complete database if ε is chosen too large. With increasing data space dimension, the range of ε where the query result corresponds to neither of these two trivial cases becomes more and more narrow. Likewise the result of the similarity join becomes trivial if ε is not suitably chosen. If ε is chosen too small, the join result is (almost) empty. If ε is chosen too large, the join result (nearly) corresponds to the cross-product of the two data sets.

To overcome this drawback of the range distance join, we have defined in chapter 2 the join operations with a nearest neighbor based join predicate. Previously published algorithms are presented in this section. For nearest neighbor based join algorithms, the cardinality of the result is (up to tie situations) defined in the specification of the join query. In chapter 2, we distinguish between closest pair queries (also known as *k*-distance join) and the *k*-nearest neighbor join. We will see that there are no previous publications that concentrate on the latter join operation.

3.3.1 Closest Pair Queries According to Hjaltason&Samet

In [HS 98], Hjaltason and Samet propose the algorithm for three join operations which belong to the group of closest pair queries on both point data as well as polygon data (extended spatial objects):

- *k*-distance join (i.e. *k*-closest pair query)
  corresponds to the original *k*-closest pair query. The user specifies the parameter *k* and the system retrieves those *k* pairs from $R \times S$ having least distance.
- incremental distance join
  similar to the *k*-distance join but the parameter *k* is not previously defined. Rather, the first, second, third, etc. pair is retrieved by repeated calls of a function GetNext. The caller decides according to the results whether or not more pairs are needed.
- *k*-distance semijoin
  similar to the k-distance join but a GROUP BY operation is performed on one of the point sets (*R*). Only the first pair is reported for each point of *R*, subsequent

pairs where the same point of R is reported again, are discarded. Therefore, the k-distance semijoin retrieves those k points of R which have the smallest nn-distance with respect to S. This operation can also be used to implement the special case of the k-nearest neighbor join where k corresponds to 1 (the k in the k-distance join and in the k-nearest neighbor join have a different meaning). For this purpose, the k of the k-distance semijoin must be set to the cardinality of R:

$$k = |R|.$$

Although this is up to future work, we do not believe that this leads to a good efficiency of the 1-nearest neighbor join. The k-nearest neighbor join for $k \neq 1$ cannot be implemented using this technique.

In [HS 98], Hjaltason and Samet extend their algorithms for the usual nearest neighbor query and the distance ranking which have been proposed in [HS 95] (cf. also section 3.1.4) to the *similarity join*. The basic idea is to replace the two priority queues by different ones. For plain similarity search, we have one priority queue which stores the active pages (called *active page list APL*) ordered by increasing distance from the query point. The other priority queue stores the k candidate points ordered by decreasing distance from the query point. For the k-distance join, the APL stores *pairs* of pages ordered by increasing distance from each other. Likewise, the candidate list stores k pairs of points.

The algorithm for the k-distance join in each step takes the top pair of pages (P_i, Q_j) from the APL. Then, one of the two pages of the pair (say P_i) is *expanded*, i.e. loaded into the main memory. If P_i is a directory page, the set of child pages $\{P_{i,1}...P_{i,l}\}$ of P_i is determined and the pairs ($P_{i,1}$, Q_j), ..., ($P_{i,l}$, Q_j) are inserted into the priority queue. This kind of expansion of an APL entry is called the *unidirectional node expansion* because of the top pair (P_i, Q_j) only one page P_i *or* Q_j is expanded (cf. figure 35). The authors also propose an algorithm which performs a simultaneous expansion of P_i and Q_j which is called *bidirectional* node expansion. For the bidirectional node expansion, plane sweeping can be applied. Hjaltason and Samet also propose various strategies for tie breaking (i.e. several page pairs have the same distance) and for the tree traversal.

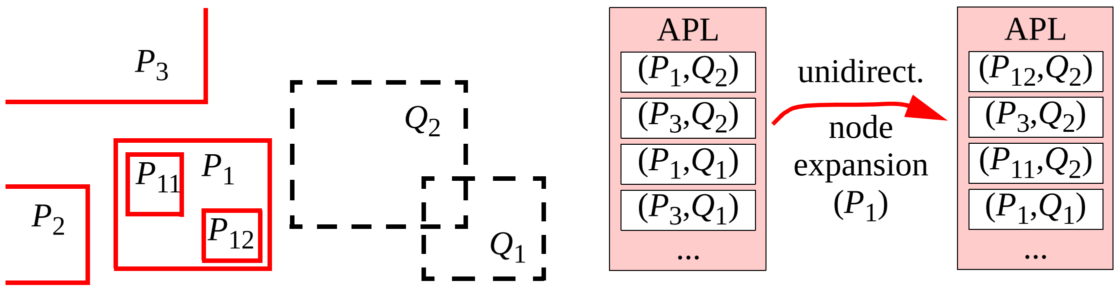


Figure 35: Principle of the *k*-distance join by Hjaltason&Samet

The algorithm for the *incremental* distance join works like the algorithm for the *k*-distance join with some minor modifications: The candidate list is infinite and ordered by increasing distance. The algorithm stops whenever the top candidate pair is validated to be the next closest pair. In fact, the authors store both kinds of pairs, page pairs and object pairs in the same priority queue. The next closest pair is validated whenever an *object pair* appears at the top of the queue.

The distance semijoin is implemented using the incremental distance join as a building block. To retrieve the *k* points of *R* which have the smallest nearest neighbor distance, the incremental distance join is repeatedly called. For each result (o_1,o_2) of such a call the algorithm checks whether o_1 has already been reported before. If so, the pair (o_1,o_2) is discarded, otherwise reported as a result of the *k*-distance semijoin. The algorithm stops when *k* pairs have been reported. Several additional strategies for the tree traversal of the semijoin are mentioned in [HS 98].

3.3.2 Alternative Approaches

Shin et al. [SML 00] propose several modifications of the algorithm of Hjaltason and Samet to improve the performance. They propose a method for selecting the sweep axis and direction for the plane sweep method in bidirectional node expansion which minimizes the computational overhead of this expansion. Moreover, they apply aggressive pruning methods to further optimize the distance join processing. This pruning is based

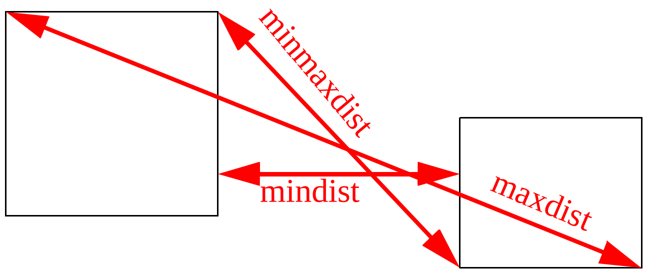


Figure 36: Mindist, maxdist, and minmaxdist of a page pair

on estimated values for the pruning distance. Compensation methods ensure the correctness of the algorithm in the case that this estimation fails. The estimate is initially chosen and then during query processing dynamically corrected.

Corral et al. [CMTV 00] also propose a collection of five algorithms for the *k*-closest pair query. In contrast to Hjaltason and Samet, they consider 5 different algorithms for the nearest neighbor search and systematically transform them such that they implement the *k*-closest pair query. The five approaches are:

- Naive: Traverse both indexes depth first without any pruning. I.e., every possible page pair is formed.
- Exhaustive: Like naive, but prune those page pairs the mindist of which exceed the current candidate distance
- Simple recursive: Additionally prune according to the minmaxdist criterion
- Sorted distance recursive: before descending the tree, sort the pairs of child pages according to their mindist. This algorithm is basically the extension of the RKV-algorithm for nearest neighbor search [RKV 95] to the *k*-closest pair query.
- Heap algorithm: Similar to [HS 98] with some differences of minor importance

Basically the first three approaches are only limited versions of the fourth. Two of the algorithms perform pruning based on the minmaxdist criterion. The minmaxdist of a pair of pages is the maximum pruning distance which can be encountered after processing the corresponding subtrees. The mindist, maxdist, and minmaxdist for a pair of pages are depicted in figure 36. Several new strategies for tie breaking and processing of trees of different height are proposed.

3.4 Conclusions

We have seen that there are several algorithms that implement the distance range join. However, most of them are based on the spatial join and, therefore, do not yield a very high performance in high dimensional data spaces. Even the most important approaches that have exclusively been published for the similarity join such as MDJ [KS 97a] or the ε-kdB tree run into serious problems for high dimensional data spaces. Therefore we see the need for further research in this area. There is some related work for the *k*-closest pair query. We will not consider this kind of join operation in our further work, because this operation is not very important for our applications. The *k*-nearest neighbor join, in contrast, has a high importance. There is almost no related work for this operation. Therefore, the *k*-nn join will also play an important role in our thesis.

Chapter 4
Density Based Clustering on the Distance Range Join

When considering algorithms for KDD, we can observe that many algorithms rely heavily on repeated *similarity queries* (i.e. range queries or nearest neighbor queries among feature vectors) which are a database primitive prevalent in most multimedia database systems. For example, the algorithm for mining spatial association rules (extracting associations between objects based on spatial neighborhood relations) proposed in [KH 95] performs a similarity query for each object of a specified type such as a town. Another example is the algorithm for proximity analysis proposed in [KN 96] which uses the features of neighboring objects in order to explain the existence of known clusters. This algorithm performs a similarity query for each object contained in the considered cluster. For various other KDD algorithms, this situation comes to an extreme: a similarity query has to be answered *for each object in the database* which obviously leads to a considerable computational effort. Examples include algorithms for the identification of outliers in large databases [KN 97, KN 98, BKNS 00] and numerous clustering algorithms [Sib 73, Mur 83, JD88, HT 93, EKSX 96, ABKS 99].

In order to accelerate this massive similarity query load, multidimensional index structures [BKSS 90, LJF 95, BKK 96] are usually applied for the management of the feature vectors. Provided that the index quality is high enough, which can usually be assumed for low and medium dimensional data spaces, such index structures accelerate the similarity queries to a logarithmic complexity. Therefore, the overall runtime complexity of the KDD algorithm is in O(n log n). Unfortunately, the overhead of executing all similarity queries separately is large. The locality of the queries is often not high enough, so that usual caching strategies for index pages such as LRU fail, which results in serious performance degenerations of the underlying KDD algorithms. Several solutions to alleviate this problem have been proposed, e.g. sampling [GRS 98] or dimensionality reduction [FL 95].

The benefits of these kinds of data reduction, however, are limited. Especially clustering algorithms are not insensitive with respect to sampling, because clusters consisting of a very small number of points are lost if too few points are in the sample. In order to maintain the completeness of the result, guidelines for the bounds of the sampling rate have been proposed recently [GRS 98]. Dimensionality reduction of the data can be done either by manual feature selection (which requires substantial domain knowledge) or by some standard method such as Principal Component Analysis, Discrete Fourier Transform, or the FastMap algorithm proposed in [FL 95]. However, the reduction of the dimensionality of the data implies some loss of information and thus may not always be applicable. The introduction of parallelism is also a promising approach in order to support query intensive KDD algorithms efficiently. The development of parallel algorithms, however is complex and expensive. While the benefits of the acceleration techniques mentioned above are limited, they can also be applied in combination with our technique proposed in this chapter to further improve the performance.

The basic intention of our solution is to substitute the great multitude of expensive similarity queries by another database primitive, the *similarity join*, using a distance-based join predicate, without affecting the correctness of the result of the given KDD algorithm: Consider a KDD algorithm that performs a range query (with range ε) in a large database of points P_i ($0<i<n$) for a large set of query points Q_j ($0<j<m$). During the processing of such an algorithm, each point P_i in the database is combined with each query point Q_j which has a distance of no more than ε. This is essentially a join operation between the two point sets P and Q with a distance-based join predicate, a so-called

distance join or *similarity join*. The general idea of our approach is to transform query intensive KDD algorithms such that the transformed algorithms are based on a similarity join instead of repeated similarity queries. In this chapter, we concentrate on algorithms which perform a range query for each point in the database. In this case, the similarity join is a self-join on the set of points stored in the database. Nevertheless, our approach is also applicable for many other KDD algorithms where similarity queries are not issued for each database object, but which are still query intensive. Additionally, since a large variety of efficient processing strategies have been proposed for the similarity join operation, we believe that our approach opens a strong potential for performance improvements.

Note that this idea is not applicable to every KDD algorithm. There is a class of algorithms which is not meant to interact with a database management system and thus is not based on database primitives like similarity queries, but instead works directly on the feature vectors. What we have in mind is the large class of algorithms which are based on repeated similarity queries (or, at least, can be based on similarity queries). Examples of methods where our idea can be applied successfully are the distance based outlier detection algorithm RT [KN 98], the density based outliers LOF [BKNS 00], the clustering algorithms DBSCAN [EKSX 96], DenClue [HK 98], OPTICS [ABKS 99], nearest-neighbor clustering [HT 93], single-link clustering [Sib 73], spatial association rules [KH 95], proximity analysis [KN 96], and other algorithms. In this chapter, we demonstrate our idea on the known clustering algorithm DBSCAN and on the recently proposed hierarchical clustering method OPTICS.

The remainder of this chapter which is the extended version of [BBBK 00] is organized as follows: Section 4.1 describes the most important clustering algorithms in more details. Section 4.2 proposes a schema for transforming KDD algorithms using repeated range queries into equivalent algorithms using similarity joins. The sections 4.2.2 and 4.2.3 describe in detail the transformations of the data mining algorithms *DBSCAN* and *OPTICS*, respectively. In section 4.3, we present a comprehensive experimental evaluation of our technique, and section 4.4 concludes this chapter.

4.1 Clustering Algorithms

Existing clustering algorithms can be classified into *hierarchical* and *partitioning clustering* algorithms (see e.g. [JD 88]). Hierarchical algorithms decompose a database *D* of *n* objects into several levels of nested partitionings (clusterings). Partitioning algorithms, on the other hand, construct a flat (single level) partition of a database *D* of *n* objects into a set of *k* clusters such that the objects in a cluster are more similar to each other than to objects in different clusters. Popular hierarchical algorithms are e.g. the *Single-Link method* [Sib 73] and its variants (see e.g. [JD 88, Mur 83]) or CURE [GRS 98]. Partitioning methods include *k-means* [McQ 67], *k-modes* [Hua 97], *k-medoid* [KR 90] algorithms and *CLARANS* [NH 94]. The basic idea of partitioning methods is to determine the set of pairwise distances among the points in the data set. Points with minimum distances are successively combined into clusters.

Density based approaches apply a local cluster criterion and are popular for the purpose of data mining, because they yield very good quality clustering results. Clusters are regarded as regions in the data space in which the objects are dense, separated by regions of low object density (noise). These regions may have an arbitrary shape and the points inside a region may be arbitrarily distributed. The local densities are determined by *repeated range queries*. We can distinguish between algorithms that execute these range queries directly and algorithms that replace these range queries by a grid approximation.

Repeated range queries are executed directly in the DBSCAN algorithm [EKSX 96]. The basic idea is that for each point of a cluster, the neighborhood of a given radius (ε) has to contain at least a minimum number of points (*MinPts*) where ε and *MinPts* are input parameters. Here the mutual distances between the points are determined by evaluating exactly one range query for each point stored in the database. Such a range query can be processed by the sequential scan approach or by the index approach. The sequential scan reads the whole database and determines all distances in a straightforward way. In this case, the runtime of DBSCAN is quadratic. The index based approach accelerates query processing under certain boundary conditions to a logarithmic complexity. Therefore, the time complexity of DBSCAN is in $O(n \log n)$. While DBSCAN as a partitioning algorithm computes only clusters of one given density, *OPTICS* [ABKS 99] generates a density based cluster-ordering, representing the intrinsic hierarchical cluster structure of the dataset in a comprehensible form. Both DBSCAN and OPTICS execute exactly one ε-range query for every point in the database. They will be presented in more detail in sections 4.2.2 and 4.2.3, respectively.

Due to performance considerations several proposals rely on grid cells [JD 88] to accelerate query processing. The data space is partitioned into a number of non-overlapping regions or cells which can be used as a filter step for the range queries (multi-step query processing [KSF+ 96]). All points in the result set are contained in the cells intersecting the query range. To further improve the performance of the range queries to a constant time complexity, query processing is limited to a constant number of these cells (e.g. the cell covering the query point and the direct neighbor cells) and the refinement step is dropped, thereby trading accuracy for performance. Cells containing a relatively large number of objects are potential cluster centers and the boundaries between clusters fall in cells with fewer points. The success of this method depends on the size of the cells which must be specified by the user. Cells of small volume will give a very “noisy” estimate of the density, whereas large cells tend to overly smooth the density estimate. Additionally, simple grid based methods degenerate in high dimensional spaces. For example, partitioning every dimension in a 20-dimensional space only once, results in $2^{20} > 1{,}000{,}000$ grid cells. Algorithms using grids include *WaveCluster* [SCZ 98], *DenClue* [HK 98] and *CLIQUE* [AGGR 98]. For low dimensional spaces, these algorithms work and perform very well. In order to scale to medium dimensional spaces, they employ sophisticated techniques to find an acceptable trade-off between accuracy and speed. In high dimensional spaces their performance as well as accuracy break down due to the problems mentioned above.

For example, the basic idea of DenClue [HK 98] is that the influence of each point on the density of the data space can be modeled formally using a mathematical function (e.g. the Gaussian function), called influence function. The overall density of the data space can be calculated as the sum of the influence functions of all the data points. In this model, cluster centers are the local maxima of the overall density function, which can be found by a hill-climbing procedure guided by the gradient of the overall density function. The gradient at a point p is approximated by considering the influences of data

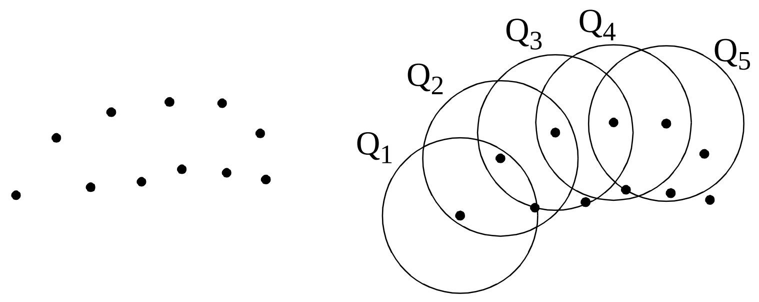


Figure 37: Sequence of Range Queries for A_1

points close to *p* only, as most points do not actually contribute to the overall density function, because they are so far away, that their influence is negligible. Obviously, this basic idea can be very easily and accurately implemented using repeated range queries. Instead, the authors of [HK 98] chose a grid based approach, trading accuracy for speed and limiting the algorithm to moderate dimensions.

4.2 Similarity-Join Based Clustering

4.2.1 General Idea

In section 4.1, we have seen that density based clustering algorithms perform range queries in a multidimensional vector space. Since a range query is executed for each point stored in the database, we can describe those algorithms using the following schema A_1:

Algorithmic Schema A_1:

```
foreach Point $p \in D$ {
    PointSet $S$ := RangeQuery ($p$, $\varepsilon$) ;
    foreach Point $q \in S$
        DoSomething ($p$,$q$) ;
}
```

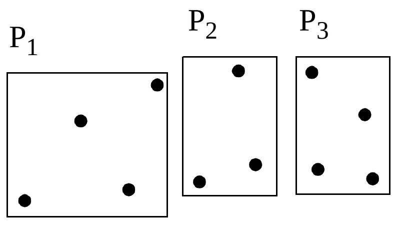


Figure 38: An Index Pagination for the Sample Data Set

In order to illustrate this algorithmic schema, we consider as an example task the determination of the core point property for each point of the database. According to the DBSCAN definition, a point is a core point if there is at least a number of *MinPts* points in its ε-neighborhood (for a formal definition see [EKSX 96]). For this task, the procedure DoSomething (*p,q*) will simply increment a counter and set the core point flag if the threshold *MinPts* is reached. Assume a sample data set with one cluster as depicted on the left side of figure 37. On the right side of figure 37 is the start of a sequence order in which schema A_1 may evaluate the range queries. Since A_1 does not use the information which points belong to which page of the index, the sequence of the range queries does not consider the number of page accesses or even optimize for a low number of page accesses.

Under the assumption of a page capacity of 4 data points, a pagination as depicted in figure 38 is quite typical and, for our sample sequence, the following page accesses must be performed: Query Q_1 accesses page P_1 and the queries Q_2 and Q_3 both access the pages P_1 and P_2. The query Q_4 accesses all three pages P_1, P_2 and P_3, and so on. After processing the upper part of the cluster, range queries for the lower part are evaluated and thus P_1 is accessed once again. But at this point in time, P_1 is eventually discarded from the cache and therefore P_1 must be loaded into main memory again.

However, by considering the assignment of the points to the pages, a more efficient sequence for the range queries can be derived, i.e. loading identical data pages several times into main memory can be avoided: First, determine all pairs of points on page P_1

having a distance no more than ε; then, all pairwise distances of points on page P_2; and afterwards, all cross-distances between points on page P_1 and P_2. Then, P_1 is no longer needed and can be deleted from the cache. Finally, we load page P_3 from secondary storage and determine the pairs on P_3 and the cross-distances between P_2 and P_3. Since the distance between the pages P_1 and P_3 is larger than ε, there is no need to determine the corresponding cross-distances. Processing the data pages in this way clearly changes the order in which data points with a distance no more than ε are combined. The only difference from an application point of view, however, is that we now count the ε-neighborhoods of many points *simultaneously.* Therefore, we simply need an additional attribute for each point which may be a database attribute unless all active counters fit into main memory.

What we have actually done in our example is to transform the algorithmic schema A_1 into a new algorithmic schema A_2 and to replace the procedure DoSomething (*p*,*q*) by a new, but quite similar procedure DoSomething' (*p*,*q*). The only difference between these two procedures is that the counter which is incremented in each call is not a global variable but an attribute of the tuple *p*. The changes in the algorithmic schema A_2 are more complex and can be expressed as follows:

```
Algorithmic Schema A_2:
  foreach DataPage $P$
    LoadAndPinPage ($P$) ;
    foreach DataPage $Q$
      if (mindist ($P,Q$) $\le \varepsilon$) then
        CachedAccess ($Q$) ;
        /* Run Algorithmic Schema A_1 with    */
        /* restriction to the points on $P$ and $Q$:   */
        foreach Point $p \in P$
          foreach Point $q \in Q$
            if (distance ($p,q$) $\le \varepsilon$) then
              DoSomething' ($p,q$) ;
    UnPinPage ($P$) ;
```

Here, mindist (P,Q) is the minimum distance between the page regions of P and Q, i.e.

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

where lb_i and ub_i denote the lower and upper boundaries of the page regions. CachedAccess (...) denotes the access of a page through the cache. Thus, a physical page access is encountered if the page is not available in the cache. In order to show the correctness of this schema transformation, we prove the equivalence of schema A_1 and A_2 in the following lemma.

Lemma 2. Equivalence of A_1 and A_2.

(1) The function DoSomething' is called for each pair (*p*,*q*) in the algorithmic schema A_2 for which DoSomething is called in schema A_1.

(2) DoSomething is called for each pair (*p*,*q*) for which DoSomething' is called.

Proof:

(1) If DoSomething (*p*,*q*) is called in A_1, then *q* is in the ε-neighborhood of *p*, i.e. the distance $|p - q| \le \varepsilon$. The points are either stored on the same page *P* (case **a**) or on two different pages *P* and *Q* (case **b**).

(a) As mindist $(P,P) = 0 \le \varepsilon$ the pair of pages (*P*,*P*) is considered in A_2. The pair of points (*p*,*q*) is then encountered in the inner loop of A_2 and, thus, DoSomething' (*p*,*q*) is called.

(b) As the regions of the pages *P* and *Q* are conservative approximations of the points *p* and *q*, the distance between the page regions cannot exceed the distance of the points, i.e. $\mathrm{mindist}(P,Q) \le |p - q| \le \varepsilon$. Therefore, the pair of pages (*P*,*Q*) is considered in A_2 and DoSomething'(*p*,*q*) is called.

(2) If DoSomething' is called in A_2, then $|p - q| \le \varepsilon$. As *q* is in the ε-neighborhood of *p*, DoSomething (*p*,*q*) is called in A_1.**q.e.d.**

We note without a formal proof that for each pair (*p*,*q*) both DoSomething and DoSomething' are evaluated at most once. Considering the algorithmic schema A_2, we observe that this schema actually represents an implementation of a join-operation which is called *pagewise nested loop join*. More precisely, it is a self-join operation where the join predicate is the distance comparison $|p - q| \le \varepsilon$. Such a join is also called *similarity self-join*. If we hide the actual implementation (i.e. the access strategy of the pages) of the join operation, we could also replace the algorithmic schema A_2 by a more general schema A_3 where $D \bowtie_{|p - q| \le \varepsilon} D$ denotes the similarity self-join:

```
Algorithmic Schema A_3:
    foreach PointPair $(p,q) \in (D \bowtie_{|p - q| \le \varepsilon} D)$
        DoSomething' $(p,q)$ ;
```

This representation allows us not only to use the pagewise nested loop join but any known evaluation strategy for similarity joins. Depending on the existence of an index or other preconditions, we can select the most suitable join implementation.

When transforming a KDD algorithm, we proceed in the following way: First, the considered KDD method is broken up into several subtasks that represent independent runs of the similarity join algorithm. Additional steps for preprocessing (e.g. index generation) and postprocessing (e.g. cleaning-up phases) may be defined. Then, the original algorithm in A_1 notation is transformed such that it operates on a cursor iterating over a similarity join (A_3 notation). Next, we consider how the operations can be further improved by exploiting the knowledge of not only one pair of points but of all points on a pair of index pages. In essence, this means that the original algorithm runs restricted to a pair of data pages.

In summary, our transformation of a KDD algorithm allows us to apply any algorithm for the similarity self-join, be it based on the sequential scan or on an arbitrary index structure. The choice of the join algorithm and the index structure is guided by performance considerations.

4.2.2 Application to DBSCAN

In this section we will shortly introduce the algorithm DBSCAN and then show how to base it on the similarity self-join. The key idea of density based clustering is that for each object of a cluster the neighborhood of a given radius (ε) has to contain at least a minimum number of objects (*MinPts*), i.e. the cardinality of the neighborhood has to exceed a given threshold. For a detailed presentation of the formal definitions see [EKSX 96].

Definition 7 (directly density-reachable)

Object p is *directly density-reachable* from object q wrt. ε and *MinPts* in a set of objects D if

1) $p \in N_\varepsilon(q)$
2) $|N_\varepsilon(q)| \geq MinPts$

where $N_\varepsilon(q)$ denotes the subset of D contained in the ε-neighborhood of q.

The condition $|N_\varepsilon(q)| \geq MinPts$ is called the *core object condition*. If this condition holds for an object p then we call p a *core object*. Other objects can be directly density-reachable only from core objects.

Definition 8 (density-reachable and density-connected)

An object p is *density-reachable* from an object q wrt. ε and *MinPts* in the set of objects D if there is a chain of objects $p_1, ..., p_n$, $p_1 = q$, $p_n = p$ such that $p_i \in D$ and p_{i+1} is directly density-reachable from p_i wrt. ε and *MinPts*.

Object p is *density-connected* to object q wrt. ε and *MinPts* in the set of objects D if there is an object $o \in D$ such that both p and q are density-reachable from o wrt. ε and *MinPts* in D.

Density-reachability is the transitive closure of direct density-reachability. Density-connectivity is a symmetric relation.

A density based cluster is now defined as a set of density-connected objects which is maximal wrt. density-reachability and the noise is the set of objects not contained in any cluster.

Definition 9 (cluster and noise)

Let D be a set of objects. A *cluster* C wrt. ε and *MinPts* in D is a non-empty subset of D satisfying the following conditions:

1) Maximality: $\forall p,q \in D$: if $p \in C$ and q is density-reachable from p wrt. ε and *MinPts,* then also $q \in C$.
2) Connectivity: $\forall p,q \in C$: p is density-connected to q wrt. ε and *MinPts* in D.

Every object not contained in any cluster is *noise*.

Note that a cluster contains not only core objects but also objects that do not satisfy the core object condition. These objects - called *border objects* of the cluster - are directly density-reachable from at least one core object of the cluster (in contrast to noise objects).

The algorithm DBSCAN, which discovers the clusters and the noise in a database according to the above definitions, is based on the fact that a cluster is equivalent to the set of all objects in D which are density-reachable from an arbitrary core object in the cluster (cf. lemma 1 and 2 in [EKSX 96]). The retrieval of density-reachable objects is performed by iteratively collecting directly density-reachable objects. DBSCAN checks the ε-neighborhood of each point in the database. If the ε-neighborhood $N_\varepsilon(p)$ of a point p has more than *MinPts* points, a new cluster C containing the objects in $N_\varepsilon(p)$ is created. Then, the ε-neighborhood of all points q in C, which have not yet been processed, is checked. If $N_\varepsilon(q)$ contains more than *MinPts* points, the neighbors of q, which are not already in C, are added to the cluster and their ε-neighborhood is checked in the next step. This procedure is repeated until no new point can be added to the current cluster C. Then the next point without cluster id is considered.

In contrast to executing exactly one range query for every point in the database, we propose to base DBSCAN on the result of a similarity self-join of the database. Materializing the result of the join, however, requires space potentially quadratic in the size of the database, so we adapt DBSCAN to run directly on the joining pages, using the schema given in section 4.2.1. This algorithm, called **J**-DBSCAN (which returns the same clustering as DBSCAN), consists of three steps. In step 1, the core points are determined, in step 2, a partial clustering of the database is computed and in step 3, these are merged into the final clustering (clean-up phase).

Step 1: To determine whether a point *p* satisfies the core object condition we need the cardinality of *p*'s ε-neighborhood. We keep a counter *p.counter* in *p*'s data page, initialized to zero. We then execute a similarity self-join. For every pair of joining pages $page_1$ and $page_2$ and every pair of points $o \in page_1$, $p \in page_2$ with $dist(o, p) \leq \varepsilon$, we increment *o.counter* and *p.counter* if o ≠ p, or *p.counter* if $o = p$.

Once the join finishes, *q.counter* contains $|N_\varepsilon(q)|$ for every point *q*.

Step 2: We assign tentative cluster ids to the data points, by executing a second similarity join and, in principle, running DBSCAN on every pair of joining data pages. The tentative cluster ids are assigned in such a way, that the following two conditions hold: 1) Points having the same tentative cluster id belong to the same cluster. 2) If two points belonging to the same cluster are assigned different tentative ids, then these two tentative ids will be in the same maximally connected component of the graph represented by the `mergeList`. The `mergeList` is the adjacency list representation of an undirected graph, its nodes are the tentative cluster ids and edges are inserted whenever the two core points with different tentative cluster ids join (cf. case (1) given below).

As an example consider figure 39. Assume that $page_1$ is first joined with itself. Then the five clusters on $page_1$ will be assigned five different cluster ids ① to ⑤. Next, assume that $page_1$ and $page_2$ are joined, such that point *X* from $page_1$ is joined with every point *p* from $page_2$ with $dist(X, p) \leq \varepsilon$ and point *Y* from $page_1$ is joined with every point *p* from $page_2$ with $dist(Y, p) \leq \varepsilon$. Then some points of the cluster on $page_2$ will be assigned cluster id ② and others will be assigned cluster id ④. Finally, $page_2$ is joined with itself, and a point with cluster id ② is joined with a point having cluster id ④. Thus, the pair of cluster ids (②, ④) is added to the `mergeList`. Therefore, the U-shaped cluster will be identified correctly.

According to definition 9, whenever two core points join, they have to be assigned the same cluster id. Whenever a core point and a non-core point join, they should be assigned the same cluster id. Note however, that a non-core point may join with two core

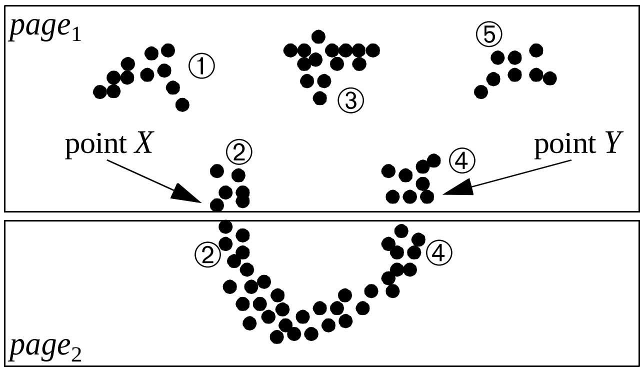


Figure 39: Tentative Cluster Ids

points from different clusters. In this case, the non-core point may be assigned either one of the cluster ids. From this we infer the (symmetric) matrix given in figure 40. For every pair of joining points (p_1, p_2), we execute the action given in figure 40, explained in detail in the following:

(1) If both points are core points and both already have cluster ids, we need to merge these two clusters. The actual merging will be done in step three of the algorithm. Here in step two we only insert the pair of cluster ids into the list of "cluster ids to be merged" (`mergeList`). This is equivalent to adding an edge between the two tentative cluster ids to the graph, and thereby merging the two maximally connected components they belong to.

(2) If both points are core points and only one already has a cluster id *C*, the other one is assigned this cluster id *C*.

(3) If one point, assume p_1, is a core point with a cluster id and the other, p_2, a non-core point with a cluster id, then p_2 is a border point of the cluster to which p_1 belongs. In this case, nothing needs to be done. This is obvious, if their cluster ids are equal. If they have different cluster ids, then there are two possible cases. Case 1: p_2 is a border point of two different clusters, i.e. no point *o* with $o \neq p_1$ and $o \neq p_2$ exists such that p_1 and p_2 are density connected through *o*. Then p_2 may be assigned to either cluster, and nothing needs to be done. Case 2: p_1 and p_2 belong to the same cluster, i.e. a point *o* with $o \neq p_1$ and $o \neq p_2$ exists such that p_1 and p_2 are density connected through *o*. This chain of core

points connecting p_1 and p_2 guarantees that the two cluster ids will end up in the same maximally connected component in the `mergeList` graph. Again, nothing needs to be done.

(4) If one is a core point with a cluster id *C* and the other a non-core point without a cluster id, the non-core point is a border point of the cluster and therefore assigned the cluster id *C*.

(5) If both are core points without cluster ids, they are directly density reachable and belong to the same cluster. A new cluster id is generated and assigned to both points.

(6) If, without loss of generality, p_1 is a non-core point with a cluster id and p_2 a core point without a cluster id, we do nothing. We cannot safely assign p_2 the cluster id of p_1. We could assign a new cluster id to p_2, however, we do not want to do this in order to keep the number of cluster ids as small as possible. If p_2 joins with a core point having a cluster id sometime later, it will be assigned this cluster id. If that does not happen, p_2 will eventually join with itself leading to case (5). Thus we can safely defer assigning a cluster id to p_2.

	P_1	CORE POINT		NON-CORE POINT	
P_2		ID	NULL	ID	NULL
CORE POINT	ID	merge if P_1.ID≠P_2.ID (1)	P_1.ID = P_2.ID (2)	(3)	P_1.ID = P_2.ID (4)
	NULL	P_2.ID = P_1.ID (2)	P_1.ID = P_2.ID = new ID (5)	(6)	P_1.ID = P_2.ID = new ID (7)
NON-CORE POINT	ID	(3)	(6)	(8)	(8)
	NULL	P_2.ID = P_1.ID (4)	P_1.ID = P_2.ID = new ID (7)	(8)	(8)

Figure 40: J-DBSCAN matrix

(7) If one is a core point and the other is not, both having no cluster ids, a new cluster id is generated and assigned to both. We cannot defer this as in case (6) because the non-core point may then end up without a cluster id.

(8) If both points are non-core points, they are not directly density reachable so nothing needs to be done.

In order to keep `mergeList` as short as possible we try to defer cases (5) and (7) as much as possible. For two different joining pages, we do this by making two passes over the joining points, first handling all other cases. Thereby, the number of times we execute cases (5) and (7) is minimized. For a page joining with itself we can further improve this by making depth first passes starting (in turn) with all the core points having cluster ids.

Step 3: The final cluster ids are computed, using the entries in the `mergeList`. Recall that the `mergeList` represents an undirected graph, the nodes are the cluster ids and for every entry (p_1, p_2) in the list there exists an edge between p_1 and p_2. We now determine the maximally connected components of this graph by a depth first search. Each such component is one cluster, so the tentative cluster ids are replaced by final cluster ids, if necessary. In all our experiments the size of the `mergeList` was very small and step three took negligible time.

When **J**-DBSCAN terminates, all points belonging to clusters will have been assigned cluster ids according to their cluster membership and all noise points will have a cluster id of NULL.

4.2.3 Application to OPTICS

While DBSCAN can only identify a “flat” clustering, the newer algorithm OPTICS [ABKS 99] computes an order of the points augmented by additional information (the *core-distance* and a *reachability-distance*) representing the intrinsic hierarchical (nested) cluster structure. The result of OPTICS, the cluster-ordering, can be used as a stand-alone tool to get insight into the distribution of a dataset. Depending on the size of the database it can either be represented graphically (for small datasets) or visualized using an appropriate visualization technique (for large datasets). Thus, it is possible to explore interactively the clustering structure, offering additional insights into the distribution and correlation of the data. Furthermore, not only ‘flat’ clustering information, but also the hierarchical clustering can be automatically extracted using an efficient and effective algorithm.

As in the previous section, we will shortly introduce the definitions and the algorithm and then show how to base OPTICS on the similarity join.

Definition 10 (core-distance)

Let p be an object from a database D, let ε be a distance value, let $N_\varepsilon(p)$ be the ε-neighborhood of p, let *MinPts* be a natural number and let *MinPts-dist(p)* be the distance from p to its *MinPts*-th neighbor. Then, the *core-distance* of p, denoted as $\textit{core-dist}_{\varepsilon,MinPts}(p)$ is defined as *MinPts-dist(p)* if $|N_\varepsilon(p)| \geq MinPts$ and *UNDEFINED* otherwise.

Definition 11 (reachability-distance)

Let p and o be objects from a database D, let $N_\varepsilon(o)$ be the ε-neighborhood of o, let $dist(o,p)$ be the distance between o and p, and let *MinPts* be a natural number. Then, the *reachability-distance* of p with respect to o denoted as $\textit{reachability-dist}_{\varepsilon,MinPts}(p, o)$ is defined as $max(\textit{core-dist}_{\varepsilon,MinPts}(o), dist(o,p))$ if $|N_\varepsilon(o)| \geq MinPts$ and *UNDEFINED* otherwise.

Note that the reachability-distance of an object p depends on the core object with respect to which it is calculated.

The algorithm OPTICS creates an ordering of a database, additionally storing the core-distance and a suitable reachability-distance for each object. Its main data structure is a seedlist, containing tuples of points and reachability-distances. Initially the seedlist is empty and all points are marked as not-done. The algorithm works as follows:

```
Algorithm OPTICS:
  repeat
    if the seedlist is empty then
      if all points are marked “done”, then terminate;
      find “not-done” point q closest to the origin;
```

```
        add (q, infinity) to the seedlist;
    (p,r) := seedlist entry with smallest reachability value;
    remove (p,r) from seedlist;
    mark p as "done";
    output (p,r);
    update-seedlist(p);
```

The function update-seedlist(p) executes an ε-range query around the point *p*. For every point *q* in the result of the range query it computes $r = \textit{reachability-dist}_{\varepsilon,MinPts}(q, p)$. If the seedlist already contains an entry (*q*, *s*), it is updated to (*q*, *min*(*r*, *s*)), otherwise (*q*, *r*) is added to the seedlist.

The similarity join based algorithm **J**-OPTICS, which computes the same result as OPTICS, consists of three steps. In step 1 the core-distances of all points are determined. In step 2 the reachability values from every point to every point in its ε-neighborhood are materialized. As the number of these reachability-distances is quadratic in the number of points, we do not save all of them but prune as many as possible. In step 3 the order of the points is computed.

Step 1: To calculate the core-distance of a point, we need to know if there are at least *MinPts* points in its ε-neighborhood and, if this is the case, the distance of the *MinPts*-th neighbor. For every point, we keep an array of distances of size *MinPts*, in which we record the distances of the closest *MinPts* neighboring points seen so far. Initially all the entries in this array are set to *infinity*. We then execute a similarity join and for every joining pair of points, calculate their distance and update the array accordingly. After the join is finished, the maximal entry in the array is the core-distance, or *infinity* if the core-distance is *UNDEFINED*.

Step 2: Given the core-distance for a point, we can calculate the reachability-distances from this point to every other point in its ε-neighborhood by executing a second similarity join. As the number of reachability-distances is potentially quadratic in the number of points, we employ the following techniques to filter most of them. We have to distinguish the following two cases:

Case 1: A page joining with itself, containing m points. For every pair of joining points on this page, we calculate the reachability distances, leading to a $m \times m$ matrix R, with $R[p,o]=\textit{reachability-dist}_{\varepsilon,MinPts}(o, p)$. Because of the rules OPTICS uses to chose the next point, we can easily prove by a case analysis that if

$$R[p,o] < R[p,q] \text{ and } R[o,q] < R[p,q],$$

then OPTICS will never use $R[p,q]$. In all our experiments, we eliminated approximately 70% of the relevant (i.e. not *infinity*) entries in R using this rule.

All entries $R[p,o]$ that have not been eliminated are added to the reachability-distance-list for point p.

Case 2: Two different pages $page_1$ and $page_2$ joining, containing m_1 and m_2 points, respectively. We compute two matrices, an $m_1 \times m_2$ matrix R and an $m_2 \times m_1$ matrix P with

$$R[o,p] = \textit{reachability-dist}_{\varepsilon,MinPts}(p, o)$$
$$P[p,o] = \textit{reachability-dist}_{\varepsilon,MinPts}(o, p)$$

for $o \in page_1$, $p \in page_2$. Because we do not have access to the reachability distances from one point in a page to another point in the same page, we cannot use the condition given in case 1 to eliminate unnecessary reachability-distances; instead we use the easily provable condition that if $R[p,o] < R[p,q]$ and $P[o,r] < R[p,q]$ and $R[r,q] < R[p,q]$, then OPTICS will never use $R[p,q]$. In our experiments, this rule allowed us to eliminate at least 90% of the relevant entries in P and R. All entries $R[p,o]$ and $P[p,o]$ that have not been eliminated are added to the reachability-distance-list for point p.

Step 3: To compute the order of the points, a modified version of the original OPTICS algorithm is run, in which update-seedlist(p) fetches the reachability-distance-list for p instead of executing a range query around p. Because this list contains all relevant reachability-distances, the result of **J**-OPTICS is guaranteed to be the same as the result of OPTICS.

4.3 Experimental Evaluation

In order to show the practical relevance of our method, we applied the proposed schema transformation to two effective data mining techniques. In particular, we transformed the known clustering algorithm DBSCAN and the hierarchical cluster structure analysis method OPTICS such that both techniques use a similarity self-join instead of repeated range queries. Note again that the resulting cluster structures generated by DBSCAN and OPTICS based on the similarity self-join are identical to the cluster structures received from the original techniques (the only exception are non-core point objects which are density reachable from more than one cluster; for both versions of DBSCAN and OPTICS these points can be arbitrarily assigned to any of the clusters from which they are density reachable). We performed an extensive experimental evaluation using two real data sets: first, an image database containing 64-*d* color histograms of 112,000 TV-snapshots, and second, 300,000 feature vectors in 9-*d* representing weather data. For both data set, we used the Euclidean distance. We used the original version of the R*-tree and a 2-level variant of the X-tree. In all experiments, the R*-tree and the X-tree were allowed to use the same amount of cache (10% of the database size). Additionally, we implemented the similarity query evaluation based on the sequential scan. The join algorithm we used is similar to the algorithm proposed in [BKS 93], i.e. the basic join strategy for R-tree like index structures. Advanced similarity join algorithms can further improve the performance of our approach. All experiments were performed on an HP-C160 under HP-UX B.10.20. In the following, **Q**-DBSCAN denotes the original algorithm, i.e. when DBSCAN is performed with iterative range queries, and **J**-DBSCAN denotes our new approach, i.e. based on a similarity self-join. In the same way we will use **Q**-OPTICS and **J**-OPTICS. In all experiments, we report the total time (i.e. I/O plus CPU time). The sequential scan methods on the file were implemented efficiently, such that the file is scanned in very large blocks. Therefore, the I/O cost of scanning a file is considerably smaller than reading the same amount of data pagewise from an index.

4.3.1 Page Size

In our first set of experiments, we performed DBSCAN and OPTICS with varying page sizes in order to determine the optimal page sizes with respect to the used access method.

In figure 41a, the runtimes of **Q**-DBSCAN and **J**-DBSCAN on 100,000 points from the weather data with ε = 0.005 and *MinPts* = 10 are shown. The page size is given as the average number of points located on a data page. We can observe that for all page sizes the runtime of **Q**-DBSCAN is considerably larger than the runtime of **J**-DBSCAN and this holds for the R*-tree, for the X-tree and for the sequential scan. The speed-up factor of **J**-DBSCAN compared to **Q**-DBSCAN for the optimal page sizes is 20 for both index structures, i.e. **J**-DBSCAN based on the R*-tree is 20 times faster than **Q**-DBSCAN based on the R*-tree (and the same speed-up factor is achieved for the X-tree).

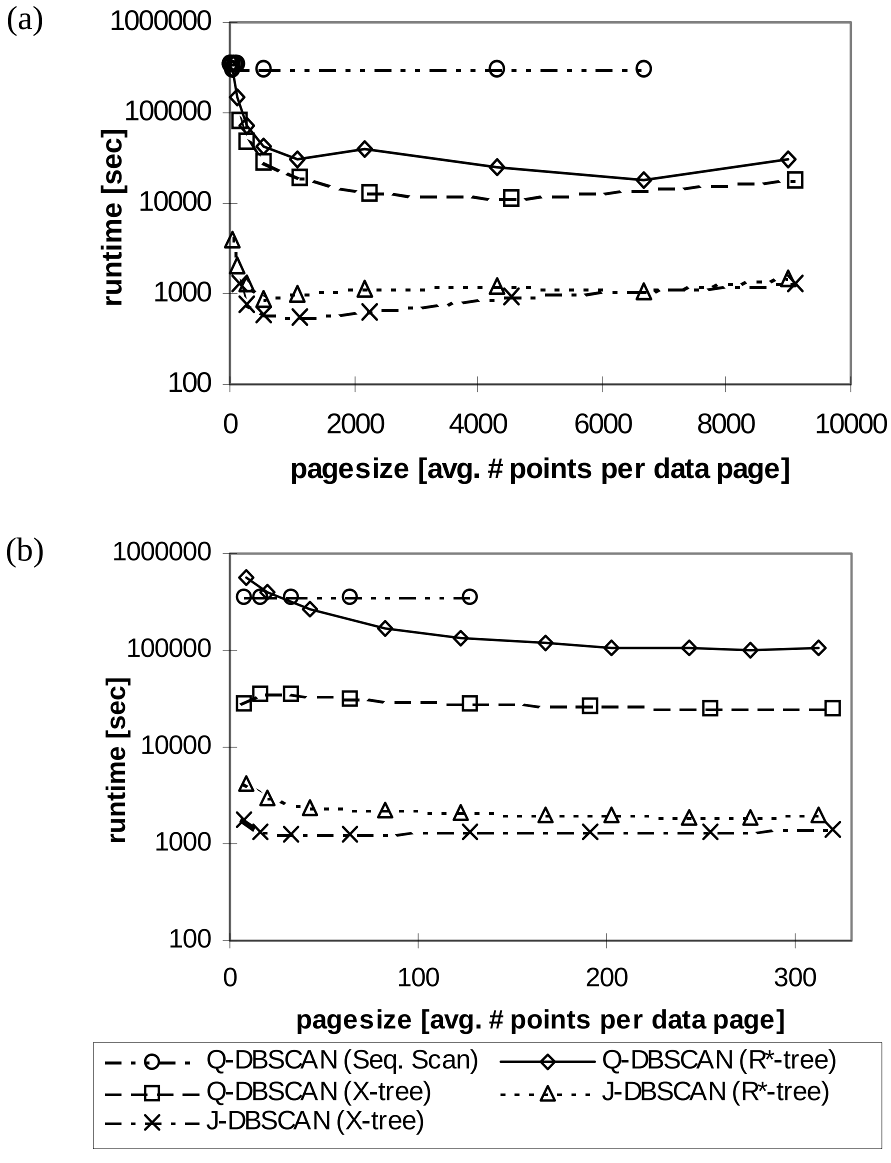


Figure 41: DBSCAN for increasing page size on (a) weather data and (b) image data

Performing **Q**-DBSCAN on the sequential scan clearly yields the worst runtime, which is 556 times the runtime of **J**-DBSCAN using the X-tree. Note that we used a logarithmic scale of the y-axis in figure 41 since otherwise the runtimes of **J**-DBSCAN and **J**-OPTICS would hardly be visible. Figure 41b shows the results for the image data. We clustered 40,000 points with ε = 0.08 and *MinPts* = 10. For this data set, the performance improvement of **J**-DBSCAN compared to **Q**-DBSCAN using the R*-tree is even higher: the speed-up factor is 54 when the R*-tree is the underlying access method and 19 using the X-tree. For small page sizes, performing **Q**-DBSCAN on the sequential scan yields a better runtime than using the R*-tree. However, when the page size of the R*-tree is well adjusted, the **Q**-DBSCAN on the sequential scan again has the slowest runtime. We can also observe, that the **J**-DBSCAN variants on the R*-tree and on the X-tree are relatively insensitive to page size calibrations.

For OPTICS, we observed similar results. Again, we varied the page size and ran **Q**-OPTICS and **J**-OPTICS on both data sets. Figure 42a shows the runtimes of **Q**-OPTICS and **J**-OPTICS on the weather data set for ε = 0.01 and *MinPts* = 10. For the optimal page sizes, all query based approaches are clearly beaten by the join based approaches, and, once again, the X-tree outperforms the R*-tree. The speed-up factor of **J**-OPTICS over **Q**-OPTICS is 3.4 using the R*-tree, and 5.6 using the X-tree. The runtime of **Q**-OPTICS based on the sequential scan is 51 times the runtime of **J**-OPTICS based on the X-tree. Figure 42b presents the results for the image data for ε = 0.1 and *MinPts* = 10. The speed-up factor of **J**-OPTICS over **Q**-OPTICS is 22 using the R*-tree, and 12 using the X-tree. Note that because of the high-dimensionality of the database, the sequential scan outperforms the R*-tree for almost all page sizes.

Independently from the underlying page size, the join based techniques outperform the query based techniques by large factors. Therefore, page size optimization is neither absolutely necessary to achieve good performance, nor is it possible to outperform our new techniques simply by optimizing the page size parameter of the query-based algorithms. Since the X-tree consistently outperformed the R*-tree on both data sets, we focus on the X-tree and on the sequential scan in the remainder of our experimental evaluation.

4.3.2 Database Size

Our next objective was to investigate the scalability of our approach when the database size increases. We ran **Q**-DBSCAN and **J**-DBSCAN on both data sets and increased the number of points from 50,000 to 300,000 (weather data) and from 10,000 to 110,000 (image data). The used parameter values were ε = 0.005, *MinPts* = 10 on the weather

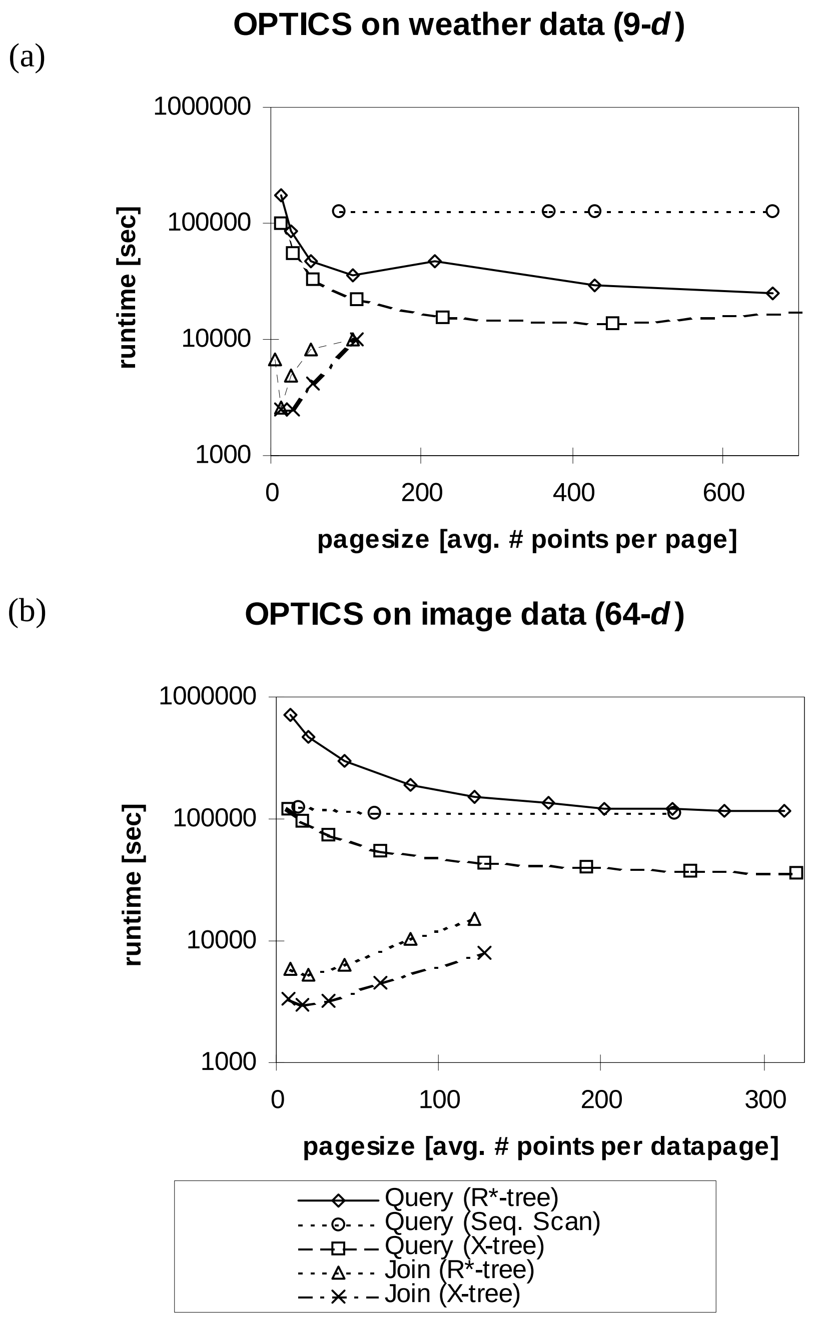


Figure 42: OPTICS for increasing page size on (a) weather data and (b) image data

data and ε = 0.08, *MinPts* = 10 on the image data. As figure 43 depicts, the query based approach **Q**-DBSCAN scales poorly when the iterative range queries are processed by the sequential scan. The reason is that DBSCAN yields a quadratic time complexity when using a sequential scan as the underlying access method.

The scalability of **Q**-DBSCAN on top of the X-tree is obviously better due to the indexing properties of the X-tree. For **J**-DBSCAN, however, we clearly observe the best scalability as the database sizes increase: the speed-up factor compared to **Q**-DBSCAN using the X-tree increases to 23 for 300,000 points of the weather data and the speed-up factor for 110,000 points of the image data is 20.

We also investigated the scalability of **Q**-OPTICS and **J**-OPTICS. The results for the weather data with ε = 0.01 and *MinPts* = 10 are depicted in figure 44a. In this experiment we increased the number of points from 50,000 up to 300,000. As before, the scalability of the query based approach is poor whereas the join based approach scales well. The speed-up factor of **J**-OPTICS over **Q**-OPTICS increases to 6. Figure 44b shows the same experiment for the image database for ε = 0.1 and *MinPts* = 10, where we increased the number of points from 10,000 to 110,000 and again found the scalability of **J**-OPTICS clearly better than the scalability of **Q**-OPTICS. The speed-up factor increases from 7.6 for 10,000 points to 14 for 110,000 points.

4.3.3 Query Range

For the performance of **Q**-DBSCAN and **Q**-OPTICS, the query range ε is a critical parameter when the underlying access method is an index structure. When ε becomes too large, a range query cannot be performed in logarithmic time since almost every data page has to be accessed. Consequently, performing **Q**-DBSCAN and **Q**-OPTICS on the sequential scan can yield better runtimes for large ε-values since the sequential scan does not cause random seeks on the secondary storage. In order to analyze our join based approach when ε becomes large, we ran **J**-DBSCAN and **J**-OPTICS with increasing ε-values.

Figure 45a depicts the results for **Q**-DBSCAN and **J**-DBSCAN on 100,000 points of the weather data and figure 45b shows the results on 40,000 points of the image data. For both data sets we set *MinPts* = 10. We can clearly observe that the runtime of **Q**-DB-SCAN substantially increases with ε whereas the runtime of **J**-DBSCAN shows only moderate growth, thus leading to a speed-up factor of 19 for ε = 0.02 on the weather data and on the image data the speed-up factor was 33 for ε = 0.2. Note, in figure 45 we omitted the runtimes of **Q**-DBSCAN on the sequential scan since even for large values

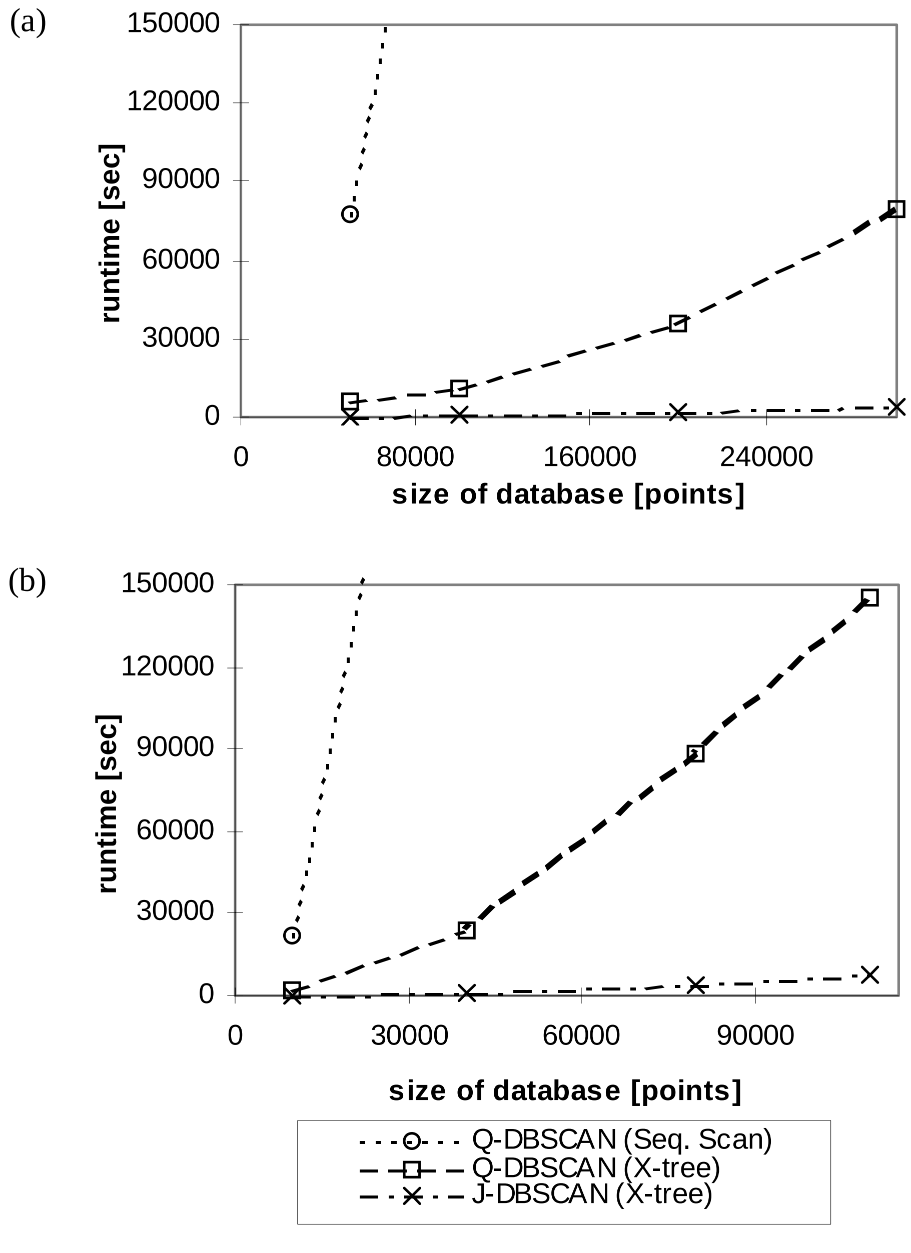


Figure 43: Scalability of DBSCAN on (a) weather data and (b) image data

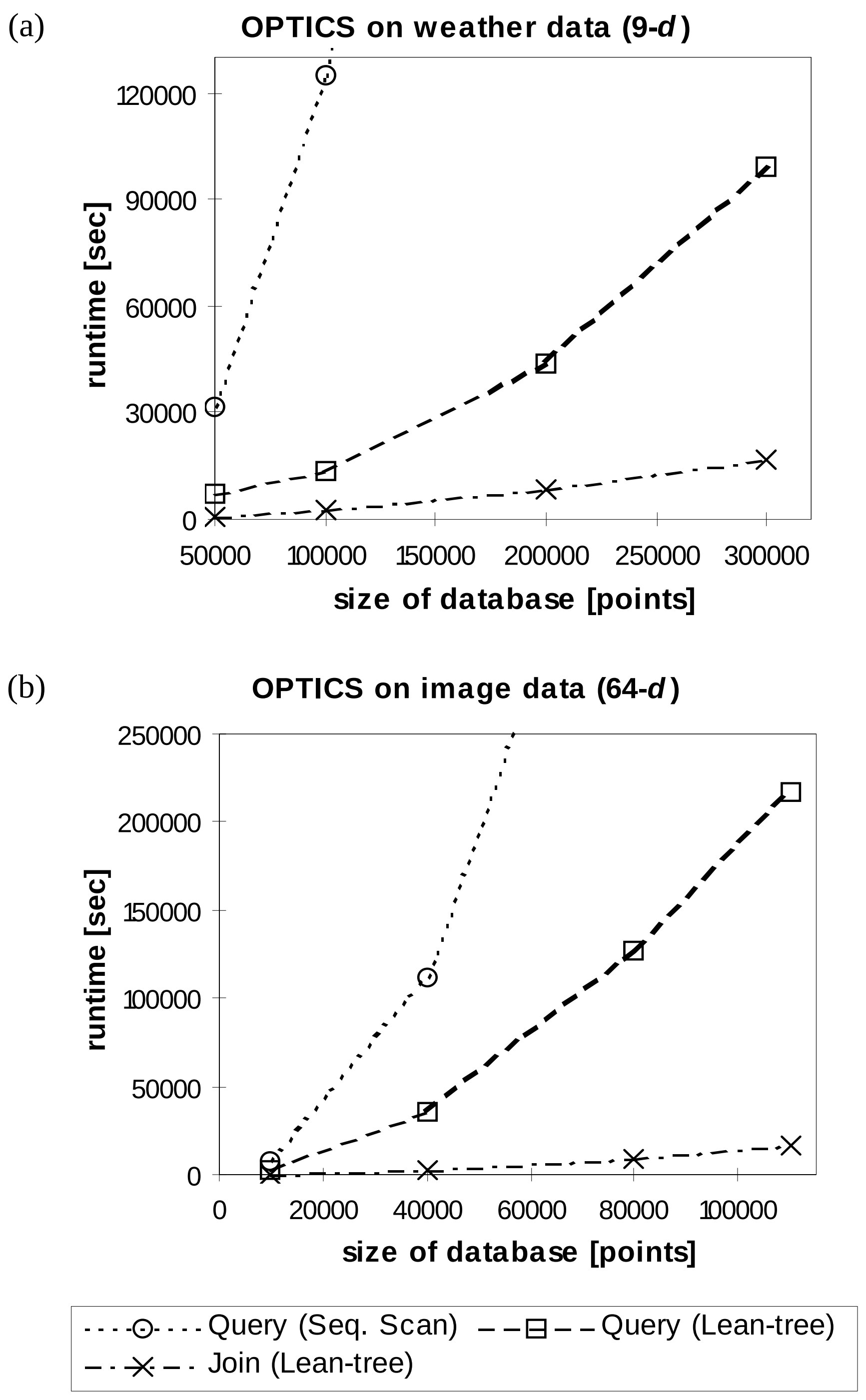
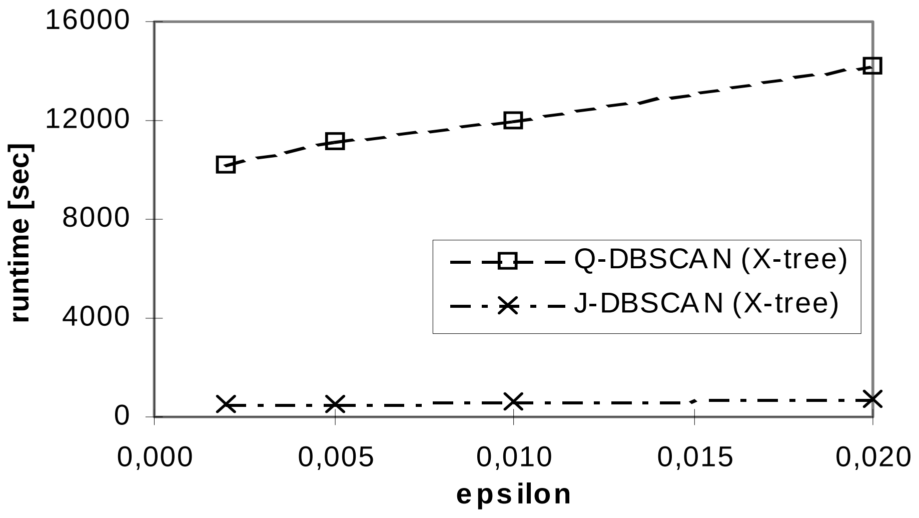


Figure 44: Scalability of OPTICS on (a) weather data and (b) image data

of ε the runtimes applying the sequential scan was by factors larger compared to a processing on top of the X-tree (e.g. for ε = 0.2 and the image data the runtime of **Q**-DB-SCAN on the sequential scan still was about 5 times the runtime of **Q**-DBSCAN based on the X-tree).

We also performed **Q**-OPTICS and **J**-OPTICS with increasing ε on 100,000 points of the weather data set and for 40,000 points of the image data, both for *MinPts* = 10 (cf. figure 46). For the weather database, the runtime for **Q**-OPTICS based on the sequential

(a)



(b)

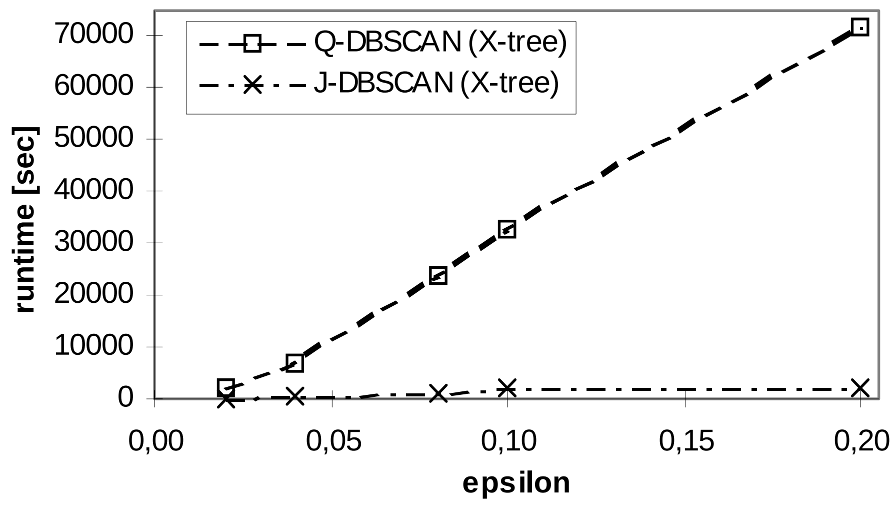
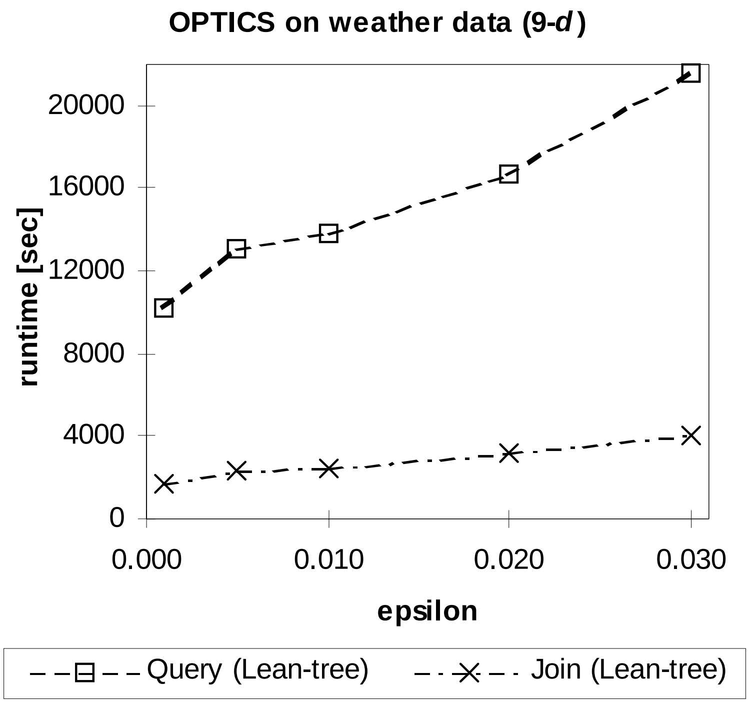


Figure 45: DBSCAN for varying query range on (a) weather data and (b) image data

(a)



(b)

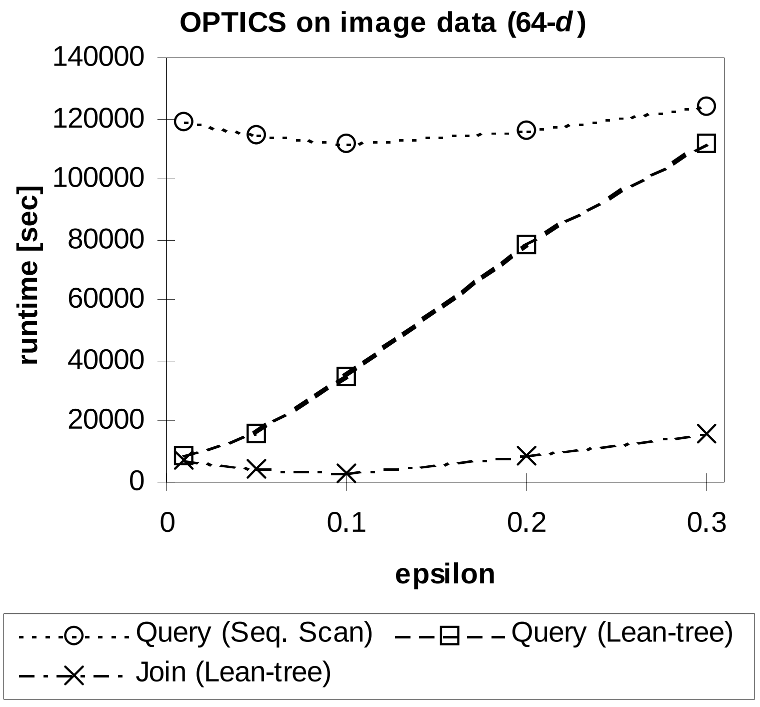


Figure 46: OPTICS for increasing query range on (a) weather data and (b) image data

scan is not shown, as it is about 6 times higher than the runtime of **Q**-OPTICS based on the X-tree even for ε=0.03. For the image data set, the runtime of **Q**-OPTICS based on the X-tree quickly increases and eventually reaches the runtime of **Q**-OPTICS based on the sequential scan. Obviously, when increasing ε further, **Q**-OPTICS on the sequential scan will outperform **Q**-OPTICS based on the X-tree since the X-tree will read too many data pages for each range query while paying expensive random disk seeks. The runtime of **J**-OPTICS for both data sets, on the other hand, shows a comparatively small growth when increasing ε. **J**-OPTICS outperforms **Q**-OPTICS on both the X-tree and the sequential scan by a large factor. This results from the fact that even when the similarity self-join generates all possible data page pairs due to a large ε, these are generated only once whereas **Q**-OPTICS generates these page pairs many times.

4.4 Conclusions

In this chapter, we have presented a general technique for accelerating query-driven algorithms for knowledge discovery in databases. A large class of KDD algorithms depends heavily on repeated range-queries in multidimensional data spaces, which, in the most extreme case, are evaluated for every point in the database. These range queries are expensive database operations which cause serious performance problems when the data set does not fit into main memory. Our solution is the transformation of such a data mining technique into an equivalent form based on a similarity join algorithm. We proposed a general schema for rewriting KDD algorithms which use repeated range queries such that they are based on a similarity join. In order to show the practical relevance of our approach, we applied this transformation schema to the known clustering algorithm DBSCAN and to the hierarchical cluster structure analysis method OPTICS. The result of the transformed techniques are guaranteed to be identical to the result of the original algorithms. In a careful experimental evaluation, we compared our transformed algorithms with the original proposals, where the query evaluation is based on various concepts such as the X-tree, the R*-tree and the sequential scan. The traditional techniques are outperformed by factors of up to 33 for the X-tree and 54 for the R*-tree.

Chapter 5
Further Applications of the Range Distance Join

After the complex case of the transformation of DBSCAN and OPTICS, we give in this chapter a few algorithms for which the evaluation on top of the similarity join is rather straightforward. The applications presented here are robust similarity search in sequence data where the join leads in particular to robustness with respect to noise and scaling. We also present a few generalizations of this technique to similarity of multidimensional sequences (i.e. raster or voxel data) and to partial similarity. Also presented are applications like catalogue matching and duplicate detection. It is worth noting, however, that only few publications from the application areas such as robust similarity search [ALSS 95] or data mining [BBBK 00] are really conscious of using the similarity join as a primitive operation. Others such as astronomy catalogue matching [VCVS 95] merely describe their algorithms without becoming aware of the fact that it is actually a redefinition of a rather simple similarity join algorithm (in most cases the nested loop join). In particular for these applications, we expect a high potential for performance gains if the simple join algorithm is replaced by a more sophisticated and efficient one. It is one of the intentions of this thesis to promote the concept of the similarity join in prospective application domains.

5.1 Robust Similarity Search

The traditional approach for similarity search, range queries or nearest neighbor queries on the associated feature vectors has several clear disadvantages with respect to the robustness of the search. It is difficult to control how flexible the search reacts with respect to mismatches in single attributes and to employ a partial similarity search which reports database objects which resemble the query object only at a certain region. The Euclidean distance (and to more or lesser extent also other well-known distance measures) is rather sensitive with respect to mismatches in a single dimension. The maximum metric even defines the similarity according to the worst matching dimension, so it is the most sensitive among the L_p norms. It is clearly application dependent whether or not such a sensitivity is useful. If it is not desired, then some user adaptable similarity measure [ALSS 95] may be helpful to assign low weights to mismatching dimensions. This approach, however, requires that the dimension of mismatch is known and constant for all objects. If this is not the case, the simple feature based approach will not be very effective. This problem that assigning single vectors to objects is not robust with respect to

- noise or mismatch in single attributes and
- similarity search for partial objects

has been addressed by Agrawal et al. [ALSS 95] for the domain of time sequence analysis. The principle can be generalized also for different domains such as color images, CAD parts, protein molecules and many others.

5.1.1 Query Decomposition

Features of time sequences can be generated in a quite trivial way by using the elements of the sequences as features. The Euclidean distance between two feature vectors corresponds to the *squared approximation error* when approximating one sequence by the other which is a well accepted distance measure in the corresponding technical application domains such as electrical engineering.

To achieve robustness with respect to noise and subsequence matching Aggrawal et al. propose to decompose both components database objects as well as query objects using a given sliding window of size *w*. This is a prominent concept borrowed from information retrieval where usually strings (words) are decomposed in overlapping *n*-lets (typically $n = 3$) to make the search for words robust with respect to spelling errors. Here the similarity between two words is measured by the number of matching *n*-lets, normalized by the length of the searched word. In a similar way, the sliding windows of time sequences can be matched (called window stitching by Agrawal et al. [ALSS 95]). The robustness with respect to noise can be achieved by a low weighting of individual mismatches. Partial similarity search is achieved by not enforcing the complete reconstruction of the complete database object but only of a part of the same length as the query object. In essence, matching of decomposed query vectors with decomposed data vectors is a similarity join. As we have to find those (complete) time sequences which have the maximum number of matching subsequences, we have to apply a *groupby* statement which groups according to the object ID and filters the sequences according to counts of similar subsequences or counts of contiguous similar subsequences.

5.1.2 Application of the Similarity Join

Agrawal et al. [ALSS 95] use the similarity join for matching of the short, elementary subsequences obtained by the sliding window. For this step, overlapping subsequences of the length *w* of the sliding window are matched after a suitable offset translation and amplitude scaling. The similarity of two such elementary subsequences is given by a channel around the query subsequence with a defined width of $2 \cdot \varepsilon$ as depicted in figure 47. In this example, the window size corresponds to $w = 9$. Translation and scaling is achieved by normalizing both database and query subsequences to [0..1] in the *y*-axis (amplitude). Two elementary sequences are called similar if one is in the ε-channel of the other. As being inside the ε-channel means that each component of the two vectors has a difference of no more than ε the similarity measure for elementary subsequences corresponds to the maximum norm (L_∞). Finding all pairs of similar elementary subsequences corresponds to the distance range join between the set of elementary subsequences of the

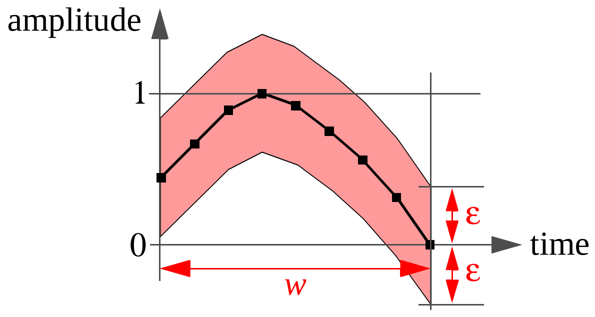


Figure 47: The Similarity Join in Robust Similarity Search

query sequence and the elementary subsequences of the sequences stored in the database:

$$\text{subseq-pairs} := \text{decomposed}\,(q) \underset{L_\infty \le \varepsilon}{\bowtie} \text{decomposed}\,(DB)$$

As the focus of Agrawal et al. [ALSS 95] is on finding pairs of similar time sequences in a database, in this special case a similarity self join on the set of decomposed database objects is applied.

5.1.3 Further Processing

The output of the similarity join are matching pairs of similar elementary subsequences. To determine subsequences of maximum length Agrawal et al. [ALSS 95] apply an operation called *window stitching*. Although we do not feel that it is actually required to apply a particular reconstruction of the subsequences (in information retrieval, this is already sufficiently done by matching *overlapping* character *n*-lets), we briefly mention that the window stitching algorithm constructs a graph of similar elementary subsequences where the vertices correspond to elementary matches and the edges correspond to subsequent matches. Window stitching is equivalent to finding longest path in the match graph as depicted in figure 48. The authors do not show that window stitching yields improvements over the simple approach of counting and maximizing similar elementary subsequences.

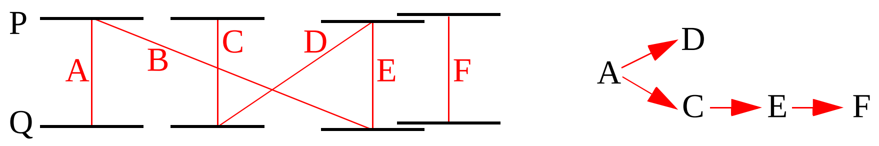


Figure 48: Window Stitching

5.1.4 Generalization

The principle of query decomposition can also be applied to other kinds of objects, in particular objects which have an inherent structure which can be immediately transformed into vectors. Examples are raster images but also three-dimensional objects like CAD parts and protein molecules. For instance, the surface of a protein can be decomposed into areas which are typically subject to biochemical interactions and then, similar molecules or even docking partners may be found using the similarity join. Here the object decomposition and/or the similarity measure for the partial objects must also insure robustness with respect to rotation. A sliding window for raster images is depicted in figure 49.

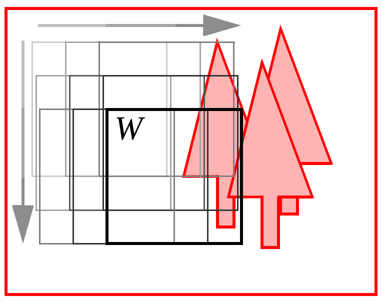


Figure 49: Extension to Raster Images

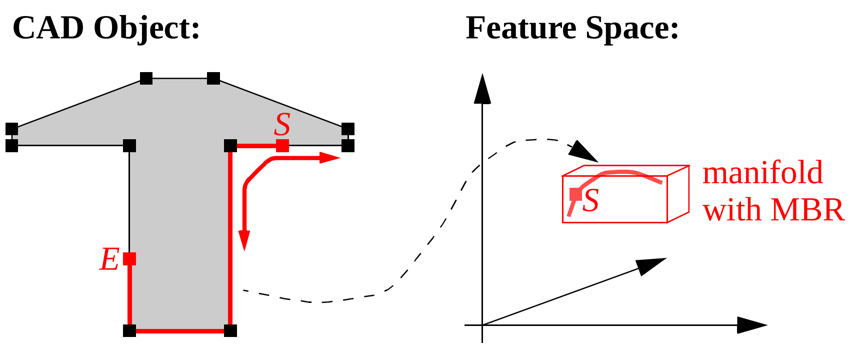


Figure 50: Continuous Feature Transitions

5.1.5 Continuous Feature Transitions

In some approaches for time sequence similarity [FRM 94] or CAD similarity [BK 97] either queries or database objects are not only decomposed into a finite set of points but rather into a continuum of points (manifold, cf. figure 50). An example for partial similarity search on 2D CAD parts is depicted in figure 50. Here, partial search is defined as follows: We are given a part of a contour which is not closed. Find a CAD part in the database which contains the query contour (considering invariance with respect to translation and rotation). The partial contour may start and end at arbitrary points (not necessarily vertices) of the retrieved contour. For partial *similarity*, the database CAD part need not to contain exactly the query contour but a contour which is similar to the query contour. Berchtold and Kriegel [BK 96] solve the problem by basically storing the set of all subcontours of a CAD part in the database. This set is essentially infinite. Therefore, all contours which start and end at a given pair of subsequent vertices are assembled into a manifold. By starting the corresponding subsequence at an arbitrary point *S* on the contour, a one-dimensional line (not necessarily a straight line) is generated in the feature space. As also the ending point *E* varies on the contour, a two-dimensional area is generated (not depicted in figure 50). These manifolds are also decomposed for effectiveness and efficiency. Therefore, we also face a similarity join problem. The corresponding similarity join between two extended objects, however, has not yet been considered and may be subject to future research.

5.2 Catalogue Matching

An application to which the concept of similarity join can be directly applied, is matching of multimedia catalogues stemming from different sources. Examples of such requirements exits in virtually all application domains of feature based similarity. We use as a running example astronomic catalogues which store information about astronomic objects such as planets, stars, galaxies, etc.

5.2.1 Known transformation

Features which are stored in such catalogues are not only the positions of the objects using some appropriate coordinate system but also, for instance, the intensity of the radiation in different frequency bands. All features are measured using both physical devices and complex computations. Therefore, each feature is subject to a measurement error which is bounded by the device tolerance.

The operation of catalogue matching is defined as follows: Two catalogues, *R* and *S* are given which store features about objects which are partially common to both sets (*R* and *S* are generally neither disjoint nor equal). The typical situation is, that both catalogues store different but overlapping regions of the universe and that e.g. one of the catalogues stores fewer objects due to weaker sensitivity of the device. Figure 51 illustrates the two catalogues, *R* and *S*. An appropriate ε for matching can be derived from the device tolerances. Determining the set of pairs which are common to both catalogues corresponds to the distance range join $R \bowtie_{\varepsilon} S$ between the two catalogues as indicated on the right side of figure 51. Our resulting catalogue should also contain information about all objects which are only registered in one of our two catalogues. The corresponding join operation which takes also those objects to the result which have no join partner in the other set (leaving the corresponding attributes NULL) is called *full outer join* in

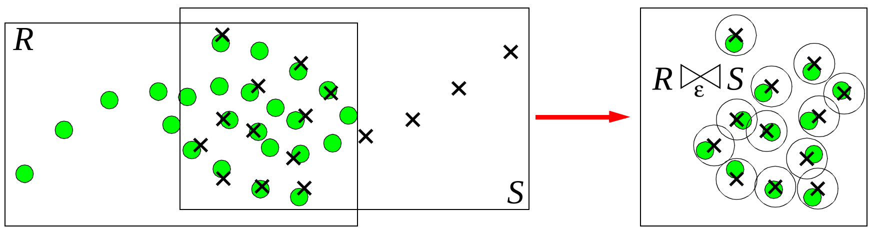


Figure 51: Catalogue Matching

the relational database theory. It is straightforward to extend the similarity join to a *full outer similarity join*.

5.2.2 Unknown Transformation

Until now we have implicitly assumed that positions and the remaining features of both catalogues are stored using a common coordinate system. If the required transformation to bring set *S* into the coordinate system used in set *R* is known, then we simply can apply this transformation to each point of *S* before the similarity join. Particularly for astronomic catalogues it is sometimes unknown what the relative position of two catalogues is. In this case, we can determine the relative position directly from the data sets. For this purpose, in each of the sets triangles are drawn connecting some of the objects which are likely to be included in both sets (e.g. the brightest stars). Then some features of the triangles which are invariant with respect to the transformation to be determined are stored in data sets (such as the ratios between angles etc.). Matching pairs of these derived triangle features can be used to determine the actual coordinate transformation. Matching the triangles again corresponds to a distance range join of the two derived feature sets.

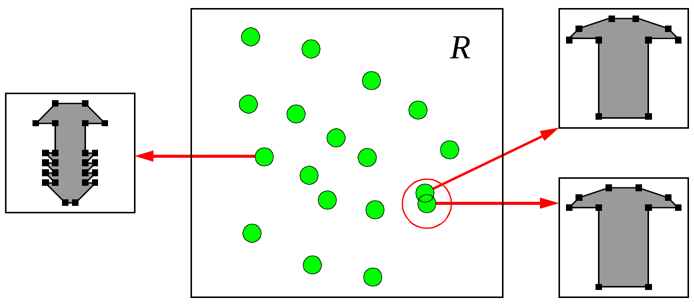


Figure 52: Duplicate Detection

5.3 Duplicate Detection

A problem similar to catalogue matching is the duplicate detection. In contrast to catalogue matching, duplicate detection typically involves only one data set *R* which contains some duplicate feature vectors. Here we also have to assume that the features are superimposed by some noise. Therefore, duplicate elimination in feature databases does not involve exact matches but fuzzy matches. To detect duplicates we determine the self join of the data set using a distance range ε which is derived from the noise level of the feature transformation. If it is not possible to determine an appropriate ε it is also possible to determine a *closest pair ranking*. Duplicate detection is depicted in figure 52.

5.4 Conclusions

In this chapter, we have proposed several further application domains for the similarity join. In contrast to the applications from the data mining domain presented in chapter 4, the transformation of algorithms such as robust similarity search, duplicate detection, catalogue matching is relatively simple as demonstrated throughout this chapter. The major objective of this chapter is to underpin our argumentation for the similarity join as an important database primitive for various application domains.

Chapter 6
A Cost Model for Index Based Similarity Join Algorithms

After the previous chapters 4 and 5 which concentrated on the applications of the similarity join we are now in this and the two following chapters focussing on the *algorithms* to implement the distance range join.

Due to the high impact of the similarity join operation, a considerable number of different algorithms to compute the similarity join have been proposed (cf. chapter 3). From a *theoretical* point of view, however, the similarity join has not been sufficiently analyzed. Our feeling is that the lack of insight into the properties of the similarity join is an obstacle in developing new methods with better performance.

Therefore, we develop a cost model for index based similarity join algorithms in this chapter. The concept used in this cost model is the Minkowski sum which is here modified to estimate the number of page pairs from the corresponding index structures which have to be considered. In contrast to usual similarity search, the concept of the Minkowski sum must be applied twice for the similarity join in order to estimate the number of page pairs which must be joined. We will point out that the index selectivity is the central key to the performance analysis. In section 6.1, we will present the formula for the index selectivity with respect to the similarity join operation. We will further analyze how much selectivity is needed to justify the usage of an index for join processing. The surprising result is, that for the optimization of the CPU operations, a fine-grained index is indispensable. For the I/O operations, however, fine-grained indexes are disastrous. Our conclusion from these results is the necessity of decoupling the CPU optimization from the I/O optimization.

6.1 Problem analysis

In this section, we separately analyze the performance behavior of the similarity join with respect to CPU cost and I/O cost. For this purpose, we assume a simplified version of an R-tree like index structure which consists of a set of data pages on the average filled with a number C_{eff} of points and a flat directory. In our simplified index, similarity joins are processed by first reading the directory in a sequential scan, then determining the qualifying pairs of data pages, and, finally, accessing and processing the corresponding data pages.

When using an index, the only gain is the selectivity, i.e. not all pairs of pages must be accessed and not all pairs of points must be compared. For a join, the index selectivity σ is defined as the number of page pairs to be processed divided by the theoretically possible page pairs:

$$\sigma = \frac{\text{processed page pairs}}{B_R \cdot B_S} \approx \frac{\text{processed point pairs}}{|R| \cdot |S|},$$

where B_R and B_S are the numbers of blocks of the point sets and $|R|$ and $|S|$ are the sizes of the corresponding data sets (number of points). The index selectivity depends on the quality of the index and on the parameter ε of the similarity join. As a matter of fact, using an index for a join computation induces some overhead. We will first determine the possible overhead for the index usage. It is important to limit the overhead to a threshold, say 10%, to avoid that the join algorithm becomes arbitrarily bad in case of a large ε.

The distance calculations in the directory are the most important overhead for the CPU. The calculation of a Euclidean distance between two boxes (time t_{box}) can be assumed to be by a factor α more expensive than a distance calculation between two points (time t_{point}) with

$$\alpha = t_{\text{box}} / t_{\text{point}}\text{, typically } \alpha \approx 5\,,$$

because it requires 2 additional case distinctions per dimension d (since both times t_{point} and t_{box} are linear in d, α does not depend on d). Therefore, the relative CPU overhead when processing a page filled with C_{eff} points is

$$v_{\text{CPU}} = \frac{t_{\text{box}}}{C_{\text{eff}} \cdot t_{\text{point}}} = \frac{\alpha}{C_{\text{eff}}}\,.$$

Limiting the CPU overhead $v_{\text{CPU}} \le 10\%$ requires $C_{\text{eff}} \ge \alpha / v_{\text{CPU}} \ge 10\alpha$. A similar consideration is possible for the I/O. Here, the time for reading the directory is negligible (less than 5%). Important are, however, the seek operations which are necessary because index pages are loaded by random accesses rather than sequentially. The overhead is the time necessary for disk arm positioning (t_{seek}) and for the latency delay (t_{lat}), divided by the "productive" time for reading the actual data from disk (t_{tr} is the transfer time per Byte):

$$v_{\text{I/O}} = \frac{t_{\text{seek}} + t_{\text{lat}}}{C_{\text{eff}} \cdot 4d \cdot t_{\text{tr}}} = \frac{\beta}{C_{\text{eff}} \cdot 4d}$$

with the hardware constant $\beta = (t_{\text{seek}} + t_{\text{lat}}) / t_{\text{tr}}$ ($\beta \approx 40000$ for typical disk drives). We assume 4 bytes for a floating point value. Limiting the I/O overhead $v_{\text{I/O}} \le 10\%$ requires

$$C_{\text{eff}} \ge \beta / (4d \cdot v_{\text{I/O}}) \ge 100000 / d$$

which is even for a high data space dimension $d \ge 100$ orders of magnitude larger than the corresponding CPU limit.

Next we analyze how much selectivity is needed to brake-even with the overhead in index-based query processing. Again, we separately treat the CPU cost and the I/O cost.

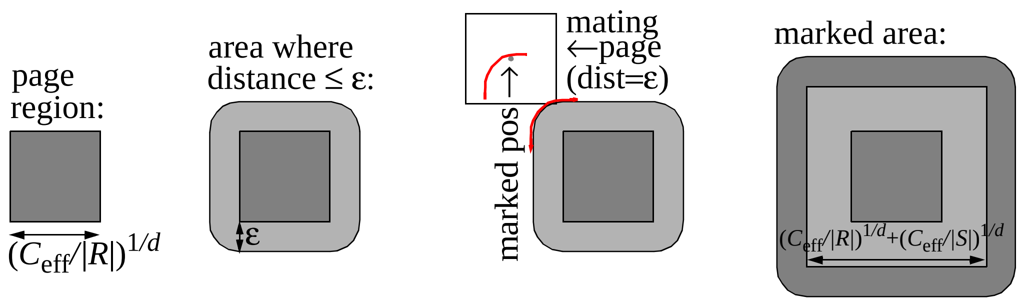


Figure 53: The Minkowski Sum for Page Pairs

For the CPU cost, we know that we have to perform one distance calculation for every pair of pages in the directory. Additionally, for those page pairs which are mates (i.e. $\sigma \cdot |R| \cdot |S| / C_{\text{eff}}{}^2$ pairs) all pairs of the stored points must be distance compared (C_{eff}^2 distance computations). Altogether, we get $\sigma \cdot |R| \cdot |S|$ distance computations for the points. For join processing without index, $|R| \cdot |S|$ distance calculations must be performed. To justify the index, we postulate:

$$\sigma \cdot |R| \cdot |S| \cdot t_{\text{point}} + \frac{|R| \cdot |S|}{C_{\text{eff}}^2} \cdot t_{\text{box}} \le |R| \cdot |S| \cdot t_{\text{point}}$$

$$\text{and thus } \sigma \le 1 - \frac{t_{\text{box}}}{C_{\text{eff}}^2 \cdot t_{\text{point}}} = 1 - \frac{\alpha}{C_{\text{eff}}^2}$$

For each pair of pages which must be processed, we assume that a constant number λ of pages must be loaded from disk. If there is no cache and a random order of processing then $\lambda = 2$. If a cache is available λ is lower, but we assume that λ is not dependent on the page capacity, because the *ratio* of cached pages is constant (e.g. 10 %). We postulate that the cost for the page accesses using a page capacity C_{eff} and a selectivity σ must not exceed the cost for the page accesses for low-overhead pages without selectivity:

$$\sigma\lambda\frac{|R| \cdot |S|}{C_{\text{eff}}^2} \cdot (t_{\text{seek}} + t_{\text{lat}} + 4dC_{\text{eff}} \cdot t_{\text{tr}}) \le \frac{\lambda \cdot |R| \cdot |S|}{\beta^2 / (4dv_{\text{I/O}})^2} \cdot \frac{\beta t_{\text{tr}}}{v_{\text{I/O}}}$$

We obtain the following selectivity which is required to justify an index with respect to the I/O cost:

$$\sigma \le \frac{C_{\mathrm{eff}}^2 \cdot 16d^2 \cdot v_{\mathrm{I/O}}}{\beta \cdot (\beta + 4d \cdot C_{\mathrm{eff}})}$$

The actual selectivity of an index with respect to the similarity join operation can be modeled as follows. As we assume no knowledge about the data set, we model a uniform and independent distribution of the points in a d-dimensional unit hypercube $[0..1]^d$. Furthermore, we assume that the data pages have the side length $\sqrt[d]{C_{\mathrm{eff}} / |R|}$ and $\sqrt[d]{C_{\mathrm{eff}} / |S|}$, respectively because $C_{\mathrm{eff}} / |R|$ is the expected volume of a page region.

The index selectivity can be determined by the concept of the Minkowski sum [BBKK 97]. A pair of pages is processed whenever the minimum distance between the two page regions does not exceed ε. To determine the probability of this event, we fix one page at some place and we (conceptually) move the other page over the data space. Whenever the distance is less than or equal to ε we mark the data space at the position of the center of the second page (cf. figure 53). As we mark the complete area where the variable page is a join mate of the fixed page, the probability of an arbitrary page to be a mate of the fixed page, corresponds to the marked area divided by the area of all possible positions of the page (which is the data space, $[0..1]^d$).

The Minkowski sum is a concept often used in robot motion planning. Understanding two geometric objects A and B each as an infinite number of vectors (points) in the data space (e.g. $A = \{a_1, a_2, ...\}$) the Minkowski sum $A \oplus B$ is defined as the set of the vector sums of all combinations between vectors in A and B, i.e.

$$A \oplus B = \{a_1{+}b_1, a_1{+}b_2, a_2{+}b_1, ...\}.$$

For cost modeling we are only interested in the volume of the Minkowski sum, not in its shape. The example in figure 2 is now constructed, step by step: On the left hand side, simply the fixed page region with side length $\sqrt[d]{C_{\mathrm{eff}} / |R|}$ is depicted. Next we show the complete area of the data space where the distance from the page region does not exceed ε. This corresponds to the Minkowski sum of the page region and a sphere of radius ε. Then, we show an example of a marginally mating page. The center point of the page is

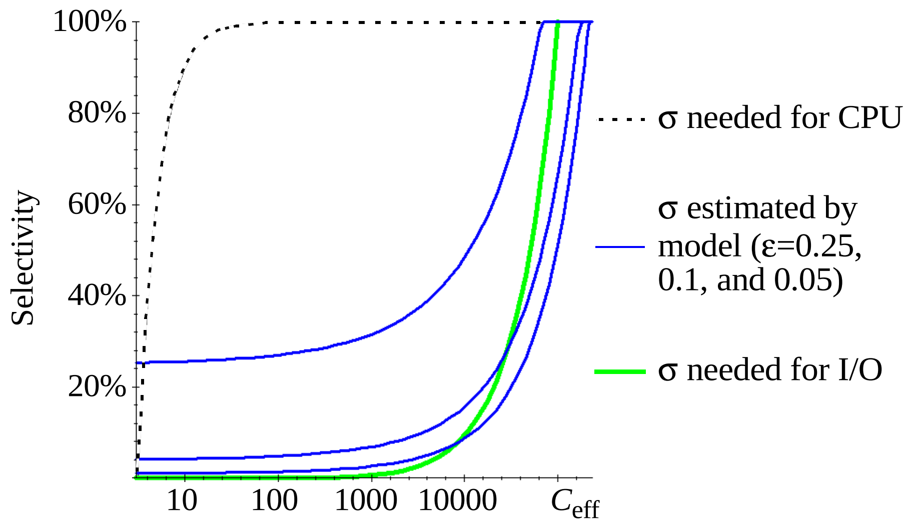


Figure 54: Selectivities Needed to Justify an Index

marked, as depicted. If we move this page around the shaded contour, we obtain the geometric object depicted on the right hand side. It corresponds to the Minkowski sum of three objects, the two page regions and the ε-sphere. The Minkowski sum of the two cubes is a cube with added side length. The Minkowski sum of the resulting cube and the ε-sphere can be determined by a binomial formula which was derived first in [BBKK 97]:

$$\sigma_{\text{Mink}} = \sum_{0 \le i \le d} \binom{d}{i} \cdot \left(\sqrt[d]{\frac{C_{\text{eff}}}{|R|}} + \sqrt[d]{\frac{C_{\text{eff}}}{|S|}} \right)^{d-i} \cdot V_{i\text{-dim. Sphere}}(\varepsilon)$$

In figure 54 we compare the required and the estimated selectivities along with varying block sizes. The thin, dashed line shows σ as it is needed to justify the CPU overhead of the index. The curve is increasing very fast. Therefore, no good (i.e. low) selectivity is needed unless the block size is *very* small (<10). Quite the opposite is true for the I/O cost (thick gray line). Until a block size of at least 10,000 points, an unrealistic good selectivity is needed. Only for block sizes starting at 10,000, index selectivities above

10% are allowed. Also depicted are 3 actual selectivities, estimated by our model. These curves are typical examples to demonstrate the range of possible curves.

The index usage is justified if the actual selectivity is below the needed selectivity. The higher the difference between “actual” and “needed” is, the more the index will outperform non-index joins. Over a wide range of block sizes, the actual selectivity curve is below the curve for the CPU cost. The highest difference is obtained between 10 and 100 points. In contrast the I/O curve needs always a better selectivity than the index has, if the distance parameter ε is high. For lower ε, the index is justified, but only for very large pages. The difference is never high.

It would be possible to determine the optimum block size for the CPU-cost and for the I/O cost. For this purpose we would have to choose a fixed distance parameter ε. As our objective is to create an index which is suitable in every join situation, it would be bad to optimize for a specific ε.

However, we can learn some important lessons from figure 54. Smaller pages are very good for minimizing the CPU cost. But for the I/O cost, small pages of 10..100 points are disastrous. Large pages, in contrast, minimize the I/O cost but are bad for the CPU cost. Gains at one of the sides are always paid by a higher price on the other side. Optimizing the overall-cost can merely bring the two cost components in balance.

To escape from this dilemma, it is necessary to decouple the I/O cost from the CPU cost by a new index architecture which will be proposed in chapter 7. This architecture consists of large blocks which are subject to I/O. These large blocks accommodate a secondary search structure with “subpages” which are used for reducing the computational effort.

6.2 Optimization of the I/O time

We have seen in the previous sections that limiting the overhead of I/O operations requires large pages with C_{eff} in the area of at least some 10,000s points. Additionally, only for such large pages, the actual selectivity is below the needed selectivity (cf. figure 54). When the block size is small, the selectivity which is needed to compensate for the index overhead is much smaller than the actually achievable selectivity of the multidimensional index.

We may ask ourselves whether or not the page size has an influence on the performance if the selectivity is close to 100%. For similarity queries with a bad index selectivity, the sequential scan is optimal, i.e. an infinitely large page size. For joins, however, the situation may be different and at the end of this section, we will know that a careful page size optimization is important.

In chapter 7 we will propose a join algorithm which loads several pages of *R* into the buffer and combines them with those pages of *S* which have a distance less than or equal to ε to at least one of the buffered *S*-pages. For such algorithms, the number of page accesses is

$$A = B_R + \sigma \frac{B_R \cdot B_S}{C-1} = \left\lceil \frac{f_R}{b} \right\rceil + \sigma \frac{\lceil f_R / b \rceil \cdot \lceil f_S / b \rceil}{\lfloor c / b \rfloor - 1},$$

where B_S and B_R are the numbers of blocks into which the point sets are decomposed, *C* is the number of blocks of the buffer, *b* is the block size in Bytes and f_R, f_S, and *c* are the sizes of the point sets and of the buffer in Bytes. The formula states the fact that the point set *R* is scanned once (B_R accesses) and the blocks of *S* are considered $\sigma B_R / (C-1)$ times. As *S* is scanned blockwise we face the following trade-off: If *b* is too large, i.e. close to *c/*2, then *S* must be scanned more often than necessary. In contrast, if *b* is chosen too small (e.g. 1 KByte), then the disk yields a latency delay after each block access.

The total cost of the join can be summarized as follows: For every block access of *S*, we have the corresponding transfer time $b \cdot t_{tr}$ and the latency delay t_{lat}. Additionally, for each of the $B_R / (C-1)$ traversals of *R* we have two disk arm positioning operations (t_{seek}), one more latency delay, and the transfer time:

$$t_{I/O} = f_R t_{tr} + \frac{\lceil f_R / b \rceil}{\lfloor c / b \rfloor - 1} \cdot (2 t_{seek} + t_{lat})$$

$$+ \ \sigma \cdot \frac{\lceil f_R / b \rceil \cdot \lceil f_S / b \rceil}{\lfloor c / b \rfloor - 1} \cdot (t_{lat} + b \cdot t_{tr})$$

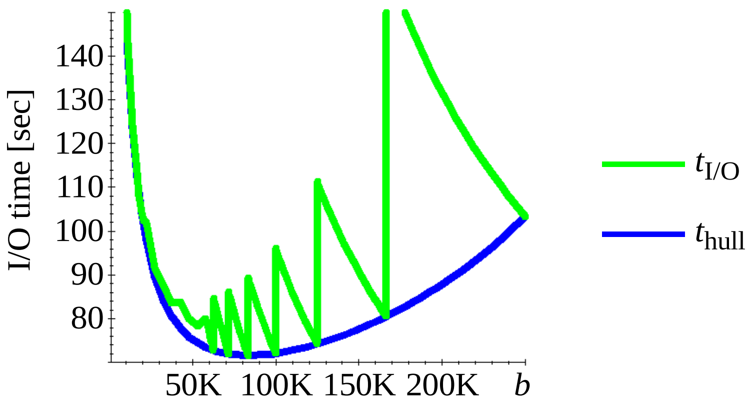


Figure 55: Optimizing the Join for I/O

As R is scanned only once and in larger blocks than the inner point set, we can neglect the cost for that. Further we can omit the ceiling-operator in f_R/b and f_S/b, because the point sets are much larger than the block size, and thus the relative error by this approximation is negligible, too:

$$t_{\mathrm{I/O}} \approx \sigma \cdot \frac{f_R \cdot f_S}{b^2 \cdot (\lfloor c/b \rfloor - 1)} \cdot (t_{\mathrm{lat}} + b \cdot t_{\mathrm{tr}})$$

We are looking for the block size b which minimizes $t_{\mathrm{I/O}}$. The only obstacle in optimization by setting the derivative to 0 is the floor-rounding in c/b which cannot be neglected because $c>>b$ is not guaranteed (we are basically out to determine whether the buffer should be assigned to R and S more balanced or more unbalanced). We solve this problem by first optimizing a hull function t_{hull} with $t_{\mathrm{hull}} = t_{\mathrm{I/O}}$ if b divides c and $t_{\mathrm{hull}} < t_{\mathrm{I/O}}$ otherwise:

$$t_{\mathrm{hull}} = \sigma \cdot \frac{f_R \cdot f_S}{b \cdot (c - b)} \cdot (t_{\mathrm{lat}} + b \cdot t_{\mathrm{tr}})$$

Figure 55 depicts the actual cost function $t_{\mathrm{I/O}}$ and the hull function t_{hull} for a file size of 10 MByte and a buffer of 500 KByte. It is easy to see that the optimum of $t_{\mathrm{I/O}}$ cannot be at some position where $t_{\mathrm{I/O}}$ is continuous, because the remaining term

$$\frac{\sigma \cdot f_R \cdot f_S}{b^2 \cdot \gamma} \cdot (t_{\text{lat}} + b \cdot t_{\text{tr}})$$

(when the floor-expression is some constant γ) is strictly monotonically decreasing. This can be shown by the derivative. So, the minimum of $t_{\text{I/O}}$ must be at a position where b divides c without rest. As t_{hull} meets $t_{\text{I/O}}$ at all such positions, we know that the optimum of $t_{\text{I/O}}$ can only be at the first meeting point ($\text{t}_{\text{I/O}} = \text{t}_{\text{hull}}$) immediately left or right from the minimum of t_{hull}. The minimum of t_{hull} can be determined by setting the derivative to zero which yields two results. Only one is positive and it is a minimum, which can be shown according to the second derivative. The positive solution of

$$\frac{\partial}{\partial b} t_{\text{hull}} = 0$$

is (because $\partial\sigma / \partial b \approx 0$ for large pages):

$$b_{\text{opt,hull}} = \frac{\sqrt{t_{\text{lat}}^2 + t_{\text{tr}} \cdot t_{\text{lat}} \cdot c} - t_{\text{lat}}}{t_{\text{tr}}}$$

The two possible positions of the actual optimum of $t_{\text{I/O}}$ are

$$b_1 = \frac{c}{\left\lfloor \frac{c}{b_{\text{opt,hull}}} \right\rfloor} \qquad b_2 = \frac{c}{\left\lceil \frac{c}{b_{\text{opt,hull}}} \right\rceil} .$$

These two values must be substituted in the cost function to determine the actual minimum.

$$b_{\text{opt,I/O}} = \begin{cases} b_1 \text{ if } t_{\text{hull}}(b_1) \le t_{\text{hull}}(b_2) \\ b_2 \text{ if } t_{\text{hull}}(b_1) \ge t_{\text{hull}}(b_2) \end{cases}$$

As the minimum of t_{hull} is very stable (cf. figure 55), it is also possible to use e.g. b_1 without considering b_2.

Figure 56 depicts b_1 with a buffer size varying from 0 to 10 MByte for a disk drive with a transfer rate of 4 MByte/s and a latency delay of 5 ms. The optimum for a 10 MByte buffer, for instance, is 455903 Bytes (i.e., 23 buffer pages).

6.3 Optimization of the CPU time

The CPU cost are composed of two components: cost of directory processing (i.e. distance computations among page regions) and cost of data level processing (i.e. point distance calculations). In our simplified index structure, the distance between every pair of pages must be calculated, i.e. $|R| \cdot |S| / C_{\text{eff}}^2$ calculations. The number of point distance calculations depends on the index selectivity and is $\sigma \cdot |R| \cdot |S|$. The total CPU cost is:

$$t_{\text{CPU}} = \sigma \cdot |R| \cdot |S| \cdot t_{\text{point}} + \frac{|R| \cdot |S|}{C_{\text{eff}}^2} \cdot t_{\text{box}}$$

As we have $t_{\text{box}} = \alpha \cdot t_{\text{point}}$, we can rewrite this and insert our estimate of the selectivity:

$$t_{\text{CPU}} = |R| \cdot |S| \cdot t_{\text{point}} \cdot \left(\left(\sqrt[d]{\frac{C_{\text{eff}}}{|R|}} + \sqrt[d]{\frac{C_{\text{eff}}}{|S|}} + 2\varepsilon \right)^d + \frac{\alpha}{C_{\text{eff}}^2} \right)$$

We do not want to optimize the index for a specific distance parameter ε, because we must create an index which is good for every similarity join. Therefore, we consider the two extreme situations of very low and very high distance parameters. For small ε, we can rewrite our CPU cost formula to

$$t_{\text{low}\varepsilon} = |R| \cdot |S| \cdot t_{\text{point}} \cdot \left(C_{\text{eff}} \cdot \left(\sqrt[d]{\frac{1}{|R|}} + \sqrt[d]{\frac{1}{|S|}} \right)^d + \frac{\alpha}{C_{\text{eff}}^2} \right)$$

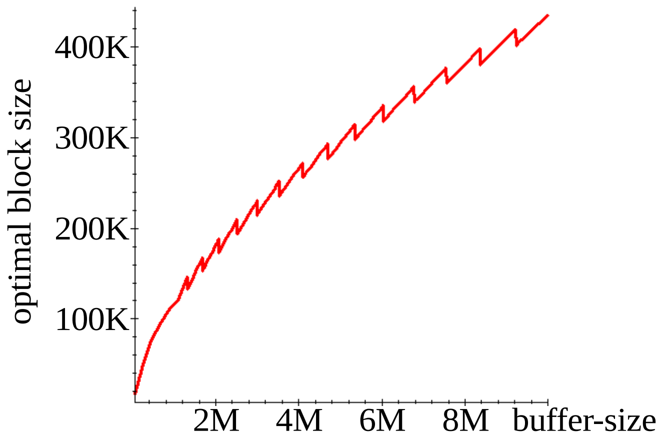


Figure 56: I/O-Optimal Block Size

which is optimized by

$$C_{\mathrm{low}\varepsilon} = \sqrt[3]{2\alpha / \left(\sqrt[d]{\frac{1}{|R|}} + \sqrt[d]{\frac{1}{|S|}}\right)^d} \, .$$

If ε is very large, then the index cannot yield any selectivity. In this case, it is merely necessary to limit the overhead as in the beginning of section 6.1. For a 10% limit at least 10α points must be stored in a data page. Therefore, we have the following value for the effective capacity:

$$C_{\mathrm{opt}} = \max\left\{10\alpha, \sqrt[3]{2\alpha / \left(\sqrt[d]{\frac{1}{|R|}} + \sqrt[d]{\frac{1}{|S|}}\right)^d}\right\}$$

6.4 Conclusions

In this chapter, we have proposed a cost model for the index selectivity of the similarity join. We have given cost formulas for both CPU and I/O cost and have shown how our cost model can be used to optimize the page capacity for maximum CPU and I/O performance, respectively. Our analysis, however, revealed a serious optimization conflict between these two cost factors. While large pages are needed to optimize the I/O performance, large pages ruin the CPU performance and vice versa for small pages. In our next chapter, we propose a solution to this conflict, a new index structure which allows a separate optimization for CPU and I/O.

Chapter 7
MuX: An Index Architecture for the Similarity Join

This chapter is dedicated to the solution of the optimization conflict detected in the analysis of chapter 6. Our objective is to develop a index architecture which allows a separate optimization for CPU and I/O performance. Therefore, we basically need two separate page capacities, one for CPU and one for I/O. This goal is achieved by the multipage index (MuX). This index structure consists of large data and directory pages which are subject to I/O operations. Rather than directly storing points and directory records an these large pages, these pages accommodate a secondary search structure which is used to speed up the CPU operations. To facilitate an effective and efficient optimization, this secondary search structure has again an R-tree like structure with a directory and data pages. Thus, the page capacity of the secondary search structure can be optimized by the cost functions developed in chapter 6, however, for the CPU trade-off. We show that the CPU performance of MuX is similar (equal up to some small dilatational management overhead) to the CPU performance of a traditional index which is purely CPU optimized. Likewise, we show that the I/O performance resembles that of an I/O optimized traditional index. Our experimental evaluation confirms this and demonstrates the clear superiority over the traditional approaches.

7.1 The Multipage Index (MuX)

It has been shown in the previous chapter that it is necessary to decouple the I/O and CPU optimization to achieve a satisfactory performance in multidimensional join processing. It was shown how to optimize join processing with respect to I/O and CPU performance. We now introduce an index architecture and the corresponding algorithms which enable the separate optimization. In essence, our index consists of large I/O pages that are supported by an additional search structure to speed up the main-memory operations. A few index structures with supporting search structures have already been previously proposed. For instance, Lomet and Salzberg propose the hB tree [LS 90] which uses a kd-tree like structure to organize directory pages. Their objective is improve the insert operations in order to achieve an overlap-free space decomposition in their index, not a separate optimization of CPU and I/O operations. Also, some quad tree based structures can be used in such a way. Kornacker [Kor 99] provides an interface for GIST that allows the application of supporting search structures in index pages. Our solution uses a simple R-tree like secondary search structure. In the current chapter, we have not evaluated which kind of search tree serves the best purpose. Our motivation for using minimum bounding rectangles for both, the primary and the secondary search structure, is to be able to apply the same cost model for both optimizations. Using different concepts for the primary and secondary search structure is viable, but requires different cost models and makes the analysis thus more complex. It remains as an issue for future work to evaluate different secondary search structures with respect to high-dimensional indexing and similarity join processing.

7.2 Index architecture

The *Multipage Index* (MuX) is a height-balanced tree with directory pages and data pages (cf. figure 57). Both kinds of pages are assigned to a rectilinear region of the data space and to a block on secondary storage. The block size is optimized for I/O according to the model proposed in chapter 6. The I/O optimized pages are called the *hosting pages*. As in usual R-trees, both kinds of pages store a number of entries (directory entries

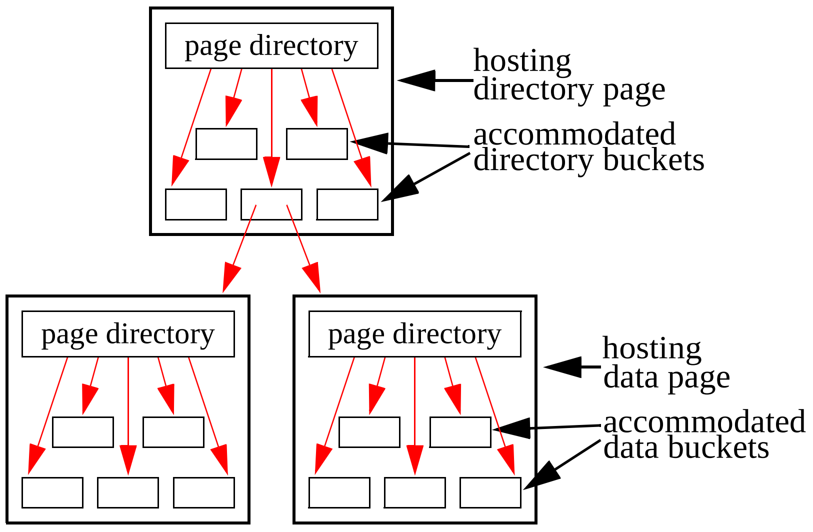


Figure 57: Index architecture of the multipage index

and data points). In contrast to usual R-trees, where the entries of pages are stored in random order in a simple array, MuX uses a secondary search structure to organize the entries. The complete search structure is accommodated in the hosting pages. Therefore, search operations in the secondary search structure do not raise any further I/O operations once the hosting page has been loaded.

For the secondary search structure, we use a degenerated R-tree consisting of a flat directory (called *page directory*) and a constant number of leaves (called *accommodated buckets*). If the hosting page is a data page, the accommodated buckets are data buckets and contain feature vectors. If the hosting page is a directory page, the accommodated buckets are directory buckets which store pairs of a MBR and a pointer to another hosting page. The page directory is flat and consists of an array of MBRs and pointers to the corresponding accommodated buckets. Generally, it would be straightforward to use a hierarchical page directory. The actual number of buckets accommodated on a hosting page, however, is not high enough to justify a deep hierarchy. In our current implementation, the primary directory of MuX also consists of a single level (flat hierarchy), because hierarchical directories often do not pay off in high-dimensional query processing, as it was pointed out e.g. in [BBJ+ 00].

7.3 Construction and maintenance

For a fast index construction, the bottom-up algorithm for X-tree construction [BBK 98] was adopted. The various R-tree algorithms for insertions and deletions can also be adapted to the MuX architecture. Due to space limitations we cannot go into further details at this point.

7.4 Similarity queries

Similarity range queries can be efficiently processed by a depth-first traversal of the multipage index. For nearest neighbor queries, *k*-nearest neighbor queries and ranking queries, we propose to adapt the HS algorithm [HS 95] which uses a priority queue for page scheduling. In our implementation, only the hosting pages are scheduled by the priority queue. Once a hosting page is accessed, the corresponding accommodated buckets are processed in order of decreasing priority. Accommodated buckets can additionally be pruned whenever their query distance exceeds the current pruning distance.

7.5 Join processing

We use the following strategy for join processing: One block of the buffer memory with the size of one hosting page is reserved for *S* (the *S-buffer*). The rest of the buffer (*R-buffer*) is used for caching one or more hosting pages of *R*. In the outermost loop of the algorithm presented in figure 58, the *R*-buffer is filled with a chunk of pages of *R*. In line (*), each hosting page of *S* which is a join mate of (at least) one of the accommodated buckets in the *R*-buffer is accessed. Then each pair of accommodated buckets having a distance of at least ε is processed, i.e. the point pairs fulfilling the join criterion are determined.

```
algorithm MuX_join
    for $i$ := 1 to $B_R$ step $C - 1$ do
        load hosting pages $B_R(i)$ .. $B_R(i + C - 1)$ ;
        for $j$ := 1 to $B_S$ do
(*)         if $B_S(j)$ has some join mate in an accomm.
                    bucket of $B_R(i)$ .. $B_R(i+C-1)$ then
                load hosting page $B_S(j)$ ;
                for each accomm. bucket of
                        $B_R(i)$ .. $B_R(i + C - 1)$ do
                    for each accomm. bucket of $B_S(j)$
                        if distance (buckets) $\le \varepsilon$ then
                            process pair of buckets;
```

Figure 58: Join Processing for the Multipage Index

In line (*) our algorithm considers the *accommodated* buckets of the chunk in the *R*-buffer to exclude *hosting* pages of *S* from consideration. Note that our algorithm could also use the hosting pages of *R* instead of the accommodated buckets. The buckets, however, exclude more *S*-pages from processing (i.e. the index selectivity is improved). It would also be desirable to use the accommodated buckets of *S* for this exclusion test, but the corresponding MBRs of these buckets are not known until the hosting page is loaded.

In the following two claims, we will point out why our MuX structure achieves a separate optimization of CPU and I/O performance and why this leads to a superior performance compared to the conventional R-tree join. For these claims we assume that the capacity of an accommodated bucket is at least 20 data points and that a hosting page stores at least 10 accommodated buckets.

Claim 1: The I/O cost of an R-tree and MuX are very similar if the page capacity of the R-tree corresponds to the capacity of a *hosting page* of MuX.

Claim 2: With respect to CPU cost, the MuX join performs similarly to an R-tree if the page capacity of the R-tree is chosen like the *accommodated buckets* of MuX.

Rationale for claim 1: Provided that the R-tree and the MuX structure apply the same insertion and splitting rules and provided that the page capacities are equal, both techniques lead to identical paginations. Therefore, the same page pairs have to be considered which leads to the same number of page accesses. The main difference is that MuX pages have to store additionally the page directory which increases the cost of a page access. The page directory stores pairs of lower bounds and upper bounds for each accommodated bucket. For each bucket we have to store as much information as for two data points. As the capacity of a bucket is at least 20 data points, the storage size of a MuX hosting page is at most 10% larger than the storage size of the R-tree. Therefore, the I/O cost of MuX is at most 10% higher than that of the R-tree.

Rationale for claim 2: Provided that the page capacity of the R-tree corresponds to the page capacity of the accommodated buckets, and provided that the same insertion and split strategy has been applied, the two structures exactly compare the same point pairs. The number of point distance computations is identical. The MuX structure determines at most as many distances between accommodated buckets as the R-tree determines distances between R-tree pages (in practice even much fewer because not all pairs of accommodated buckets have to be considered; only those located in mating hosting pages). The additional CPU cost in the MuX structure are the distance computations between the hosting pages. Because each hosting page stores more than 10 accommodated buckets there can be only one successful distance calculation per 10^2=100 distance calculations between accommodated buckets. MuX can in the worst case be 1% worse than the corresponding R-tree.

We optimize the capacity of the hosting pages of MuX such that they are I/O optimal. The capacity of the accommodated buckets is optimized such that they are CPU-optimal. Taken claim 1 and claim 2 together, we obtain a CPU performance which resembles a

CPU-optimized R-tree and an I/O performance that resembles an I/O optimal R-tree (for both cases plus the overhead mentioned in the rationales of the claims).

Compared to conventional index join algorithms which traverse the indexes depth-first [BKS 93] or breadth-first [HJR 97], our new algorithm improves the performance with respect to CPU and I/O. The I/O effort is reduced by two ideas: The first idea is to use more cache for the point set *R* which is scanned in the outermost loop. The advantage is that in the case of a bad index selectivity the number of scans of the other point set *S* is minimized. Therefore, the I/O cost cannot become substantially worse than the I/O cost of a nested loop join. In the case of a good index selectivity, in the inner loop only those *S*-pages are loaded which are actually needed. Therefore, the performance cannot become substantially worse than a breadth-first or depth-first index traversal. For these extreme cases, we have always the performance of the best of the two worlds: nested loops or tree traversal. In the cases between these extremes, we combine the advantages of both paradigms and outperform them both clearly. The second idea leading to reduced I/O cost is that we use the page regions of the accommodated *R*-buckets to exclude hosting *S*-pages. While only I/O optimized pages are subject to I/O operations, the more selective bucket regions are used for excluding, leading to a clear advantage in the index selectivity. The CPU effort is minimized due to the optimization of the bucket size for minimum computational cost. Additionally, many distance computations between bucket regions are avoided, because buckets can only mate if their hosting pages mate, too.

7.6 Experimental evaluation

To show the superiority of our proposal over competitive techniques, we have performed an extensive experimental evaluation. For this purpose, we implemented our multipage index join algorithm. For comparison, we also implemented a similarity join algorithm using nested loops and a similarity join algorithm based on the R-tree spatial join (RSJ) algorithm [BKS 93] with three different scheduling and caching schemes.

The cache for the nested loop-join was assigned according to our optimization presented in chapter 6. All RSJ variants used a caching strategy discarding the page which

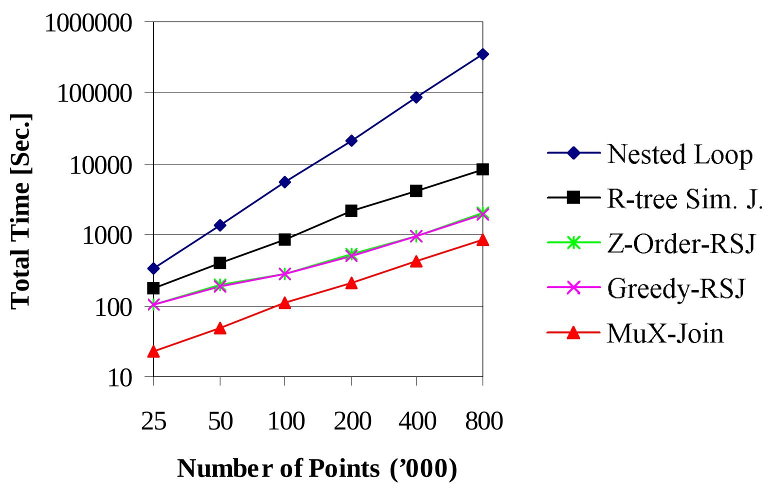


Figure 59: 4D Uniform Data Varying Database Size

will not be used for the longest time in the future. Note that, in contrast to usual paging algorithms applied in general-purpose operating systems, the join algorithm allows to exploit the knowledge of the page schedule in the future. The basic RSJ algorithm accesses the data pages of the index in a random order. The cache hit rate can be improved

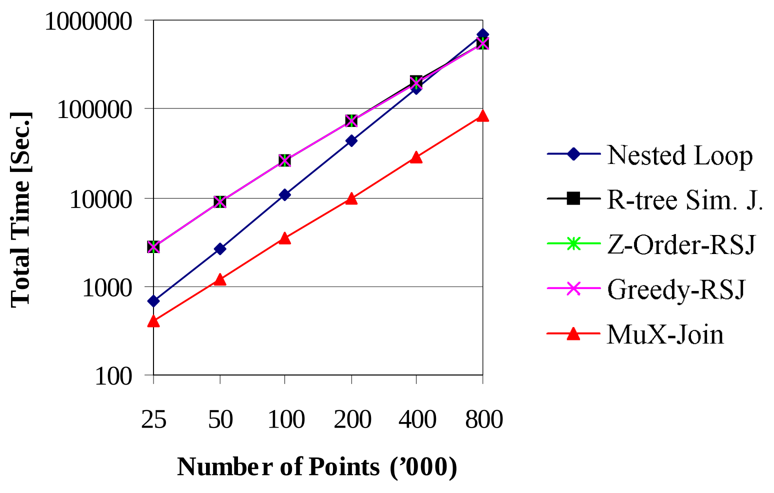


Figure 60: 8D Uniform Data Varying Database Size

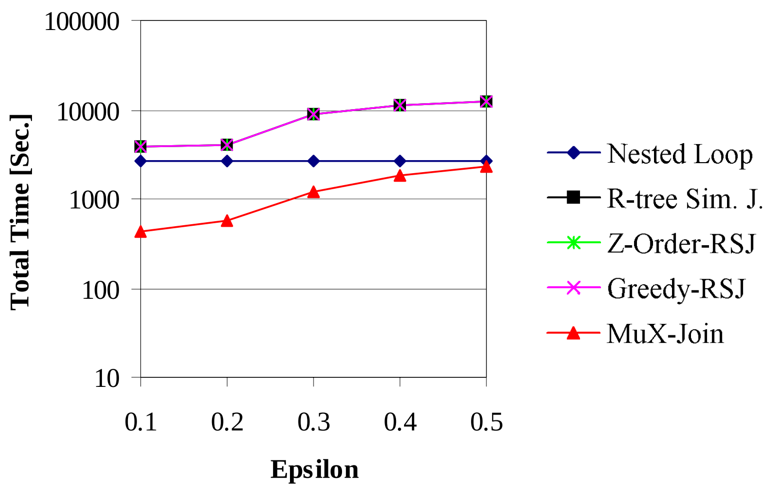


Figure 61: 8D Uniform Data

by accessing the pages of the index in an order preserving the spatial proximity. In [HJR 97], 4 different kinds of page ordering were proposed, including the Hilbert curve, and an improvement of the cache hit ratio of up to 50% was reported. We implemented a page scheduling strategy based on Z-ordering and a greedy optimization strategy which starts with an arbitrary page and accesses in each step the unprocessed page with the smallest distance to the last previously accessed pages. We will refer to the three variants as "*R-tree Similarity Join* (*RSJ*)", "*RSJ with Z-ordering optimization*", and "*RSJ with greedy optimization*." All algorithms were allowed to use the same amount of buffer memory (5% of the database size).

All our experiments were carried out on HP 9000/780 workstations under HPUX-10.20. We used a disk device with a transfer rate of 4 MByte/sec, a seek time of 5 msec, and latency time of 5 msec. Our algorithms do not exploit parallelism between CPU and I/O, which would be possible in all approaches. Therefore, our reported total query time corresponds to the sum of the CPU time and the I/O time. The index construction was not taken into account.

For our experiments, we used synthetic as well as real data. Our synthetic data sets consist of up to 800,000 uniformly distributed points in the unit hypercube with the

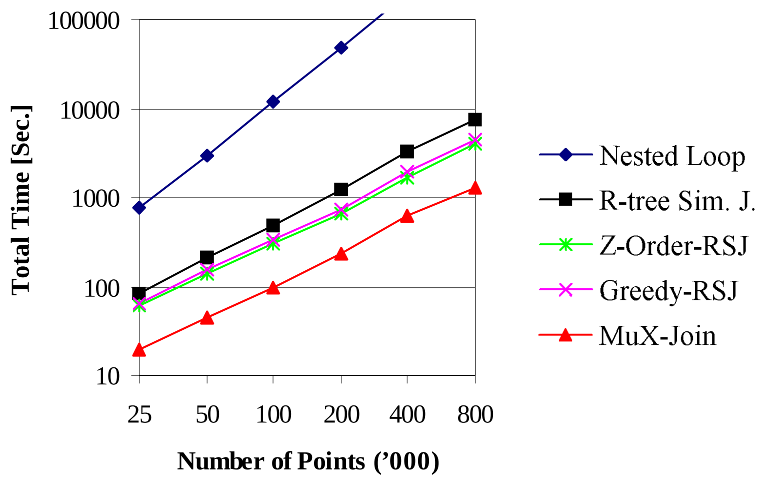


Figure 62: 9D Real Data from a Meteorology Application

dimensions 4 and 8. Our real-world data stem from three application domains: A CAD database with 16-dimensional feature vectors extracted from geometrical parts, a color image database with 64-dimensional feature vectors representing color histograms, and a meteorology database with 9-dimensional feature vectors generated by weather obser-

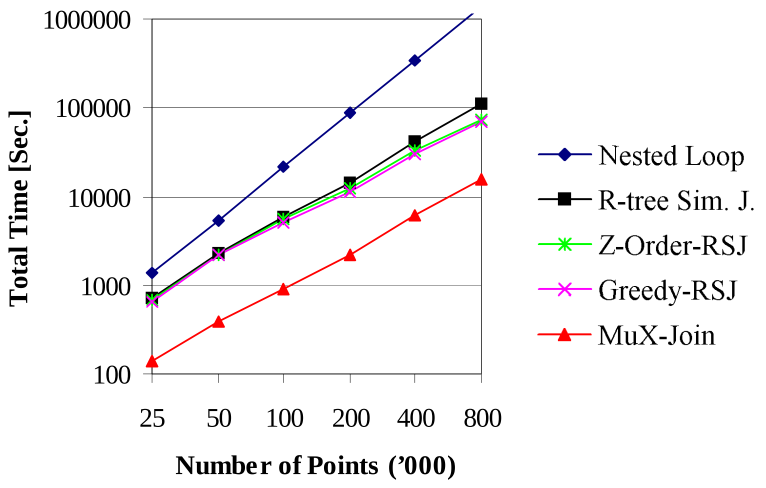


Figure 63: 8D Uniform Data Varying Database Size

vation. In the similarity join, we used the Euclidean distance. Appropriate distance parameters ε for each data set were determined such that they are useful in clustering [EKSX 96] and that each point of the data set is combined with a few other points on the average. That means in particular that we avoided in our experiments the extreme cases of no resulting pair (or in the case of self joins: each point is only a join mate of itself), or each point is combined with every other point.

Figure 59 shows our experiments on uniformly distributed point data. In the left diagram, the data space is 4-dimensional and an appropriate $\varepsilon = 0.05$ (i.e. in the result, each point has an average of 8.5 join mates). The nested loop join has the worst performance over all scales. With increasing database size, this technique is outperformed by all other techniques by increasing factors. For low-dimensional data spaces, the scheduling strategy in the R-tree similarity join plays a relatively important role. Therefore, the more sophisticated strategies which order the page accesses by Z-ordering or a greedy strategy improve the performance of the R-tree similarity join by factors up to 4.2. The clear winner over all database sizes is our new technique, the MuX-join. It outperforms the nested loop join up to 400 times and is up to 10 times faster than the R-tree similarity join. Even the improved R-tree join versions are outperformed with factors between 2.3 and 4.6. The diagram in the middle shows our experiments with an 8-dimensional data space ($\varepsilon = 0.3$; each point has an average of 22.3 join mates). In this dimension, the various R-tree join variants do not differ much. As the index selectivity begins to deteriorate in medium-dimensional data spaces, the nested loop join is much more competitive and is only for the largest database (800,000 points) outperformed by the three R-tree join variants. Our new technique, in contrast, outperforms the other techniques by a factor of 6.3 (over R-trees) and 8.1 (over nested loop) for the largest database size. For 100,000 points, the corresponding factors are 7.4 (over R-trees) and 3.1 (over nested loop). The diagram on the right side depicts the performance of the join algorithms with varying distance parameter ε ($d = 8$; $n = 50{,}000$). It is obvious that for very large ε the nested loop join must be the winner, because the join result combines each point with every other point, and the nested loop join has no additional index overhead. Therefore, the R-tree variants are clearly outperformed. As our new technique strictly limits the index overhead by an appropriate optimization of I/O as well as CPU, it is never clearly

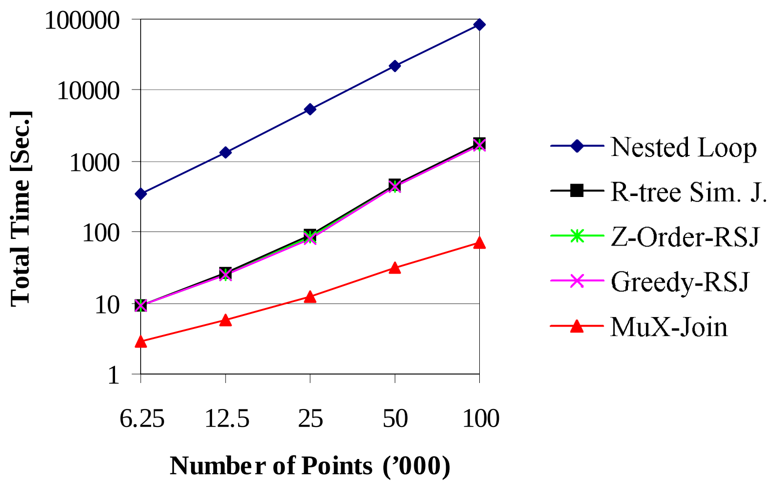


Figure 64: 64D Real Data (Color Histograms) form a Multimedia Application

outperformed. Instead, the performance slowly approaches the performance of the nested loop join with increasing ε.

Our experiments on real application data depicted in figure 64 clearly confirm our experiments on uniform data. Partially, the improvement factors are even higher. The left diagram depicts the results on the 9-dimensional meteorology feature vectors ($\varepsilon = 0.0001$; 3.9 join mates per point). For the largest database size, our technique was 590 times faster than the nested loop join, 5.9 times faster than the R-tree similarity join, and 3.5 times faster than RSJ with the improved scheduling strategies. For the 16-dimensional CAD feature vectors (diagram in the middle; $\varepsilon = 0.01$; 7.5 join mates per point) our technique is up to 87 times faster than the nested loop join and between 6 and 7 times faster than the 3 R-tree similarity join variants. The right diagram shows the results on our color image database ($\varepsilon = 0.0001$; 1.1 join mates per point). For the largest database, our technique yields an improvement factor of 1203 over the nested loop join of 25 over all R-tree similarity join algorithms.

7.7 Conclusions

In the context of chapter 6, a severe optimization conflict between CPU and I/O optimization has been discovered. To solve this conflict, we have proposed an index architecture which allows a separate optimization of the CPU time and the I/O time in this chapter. Our architecture utilizes large primary pages which are subject to I/O processing and optimized for this purpose. The primary pages accommodate a secondary search structure to reduce the computational effort. Our experimental evaluation has shown consistently good performance. Competitive approaches are outperformed by large factors. An open question for future work is the suitability of our secondary search structure. For simplicity, and in order to uniformly apply the same cost model for CPU and I/O optimization, we used minimum bounding rectangles for both, the primary and the secondary search structure. More sophisticated techniques, however, should have the potential to even improve our high speedup factors.

Chapter 8
Epsilon Grid Order: Joining Massive High Dimensional Data

In this chapter, we develop a method for massive data sets of at least 1 GByte operating on a virtual grid partition of the data space. This method is based on the observation that for the distance range join with a given distance parameter ε, a grid partition with a grid distance of ε is an effective means to reduce the search space for join partners of a point *p*. Due to the *curse of dimensionality*, however, the number of grid cells in which potentially joining points are contained explodes with the data space dimension and results in an order of $O(3^d)$ cells. To avoid considering each of the grid cells one by one, we introduce the grid partition only in a virtual way as the basis of a particular sort order, the ε grid order, which orders points according to grid cell containment. The ε grid order is used as ordering criterion in an external memory sort operator. Later, the ε grid order supports effective and efficient algorithms for CPU and I/O processing, particularly for large data sets which cannot be joined in main memory.

8.1 The Epsilon Grid Order

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

Our join algorithm exploits this knowledge of the ε-interval. Assuming a limited cache size, we have to distinguish two cases: The ε-interval of a point fits into the main memory or not. If the ε-interval *of each database point* fits into main memory, then a single scan of the database is sufficient for join processing. We call this kind of database traversal the *gallop mode*. If the ε-intervals of some points do not fit into the main memory, we have to scan the corresponding part of the database more than once. The database is traversed in the so-called *crabstep mode*. These two modes will be explained in section 8.1.2. Finally, we will show in section 8.1.3 how sequences of epsilon-grid ordered points can be joined efficiently with respect to CPU operations. Epsilon grid ordering yields the particular advantage that no directory structure needs to be constructed for this purpose. In contrast to index structures that manage main memory data structures such as MuX or ε-kdB-trees the full buffer size can be used to store point information; nearly no buffer capacity is wasted for management overhead.

8.1.1 Basic Properties of the Epsilon Grid Order

First we give a formal definition of the Epsilon Grid Order ($\cdot <_{\mathrm{ego}} \cdot$). For this order, a regular grid[1] is laid over the data space, anchored in the origin, and with a grid distance of ε. We define a lexicographical order on the grid cells, i.e. the first dimension d_0 has

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

the highest weight; for two grid cells having the same coordinates in d_0, the next dimension d_1 is considered, and so on. This grid cell order is induced to the points stored in the database: For a pair of two points p and q located in different grid cells, we let $p \lessgtr_{\text{ego}} q$ be *true* if the grid cell surrounding p is lexicographically lower than the grid cell surrounding q. Since we want to avoid explicit numbering of grid cells (which would be slightly clumsy unless we assume a previously limited data space), the following definition determines the order for the points directly, without explicitly introducing the grid cells:

Definition 12 Epsilon Grid Order ($\cdot \lessgtr_{\text{ego}} \cdot$).

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

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

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

Our first lemma proves that the epsilon grid order is, indeed, an order. We have not defined the epsilon grid order as a reflexive order due to points which are located in the same grid cell. Such points are not able to fulfill the antisymmetry property which is usually required for an order. Therefore, we have defined the epsilon grid order as an irreflexive or strict order which is required to be irreflexive, asymmetric, and transitive. There are almost no consequences from a practical point of view. For instance, the usual sorting algorithms can cope with an irreflexive order without modification. In the following lemma, we prove the three required properties, one of which (transitivity) is also exploited in lemma 4 and 5.

Lemma 3. The Epsilon Grid Order is an irreflexive order.

Proof:

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

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

Asymmetry ($p \lessgtr_{\text{ego}} q \Rightarrow \neg\, q \lessgtr_{\text{ego}} p$):

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

Transitivity ($p \lessgtr_{\text{ego}} q \wedge q \lessgtr_{\text{ego}} r \Rightarrow p \lessgtr_{\text{ego}} r$):

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

□

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

Lemma 4. A point q with $q \lessgtr_{\text{ego}} p - [\varepsilon,\varepsilon,...,\varepsilon]^T$ cannot be a join mate of p or of any point p' which is **not** $p' \lessgtr_{\text{ego}} p$.

Proof:

Following definition 12, there exists a dimension d_i such that

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

The monotonicity[1] of the floor function insures that $q_i < p_i - \varepsilon$. Because both ε and $(p_i - q_i)$ are positive we can rewrite this as $(p_i - q_i)^2 > \varepsilon^2$. This specific square $(p_i -$

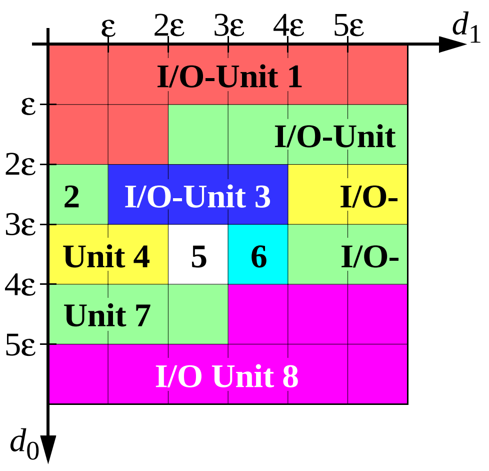


Figure 65: I/O Units in the Data Space

$q_i)^2$ for some i cannot be smaller than the sum of all squares, which corresponds to the distance between p and q:

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

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

Lemma 5. A point q with $p + [\varepsilon,\varepsilon,...,\varepsilon]^T \lesssim_{\text{ego}} q$ cannot be a join mate of p or of any point p' which is **not** $p \lesssim_{\text{ego}} p$'.

Proof. Analogous to lemma 4.

8.1.2 I/O Scheduling Using the ε Grid Order

In the previous section we have shown that our join algorithm must consider all points between $p - [\varepsilon,\varepsilon,...,\varepsilon]^T$ and $p + [\varepsilon,\varepsilon,...,\varepsilon]^T$ to find the join mates of p. In this section we

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

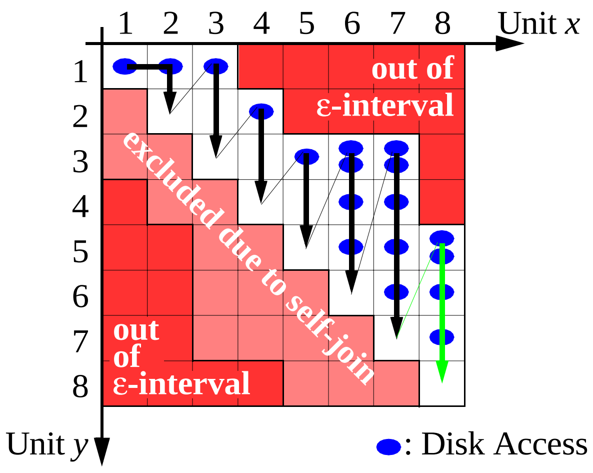


Figure 66: I/O Units in the Schedule

construct an algorithm which schedules the disk I/O operations for a similarity self join on a file of points which is sorted according to the epsilon grid order.

In our algorithm, we want to allow for *unbuffered* I/O operations on raw devices. Therefore, we assume that the block size for the I/O units is a multiple of some hardware given system constant. Generally, an I/O unit does not contain a whole number of data point records. Instead, an I/O unit is allowed to store *fragments* of point records at the beginning and at the end. Our join algorithm solves the corresponding problems by storing the fragments in separate variables. The number of points contained in an I/O unit is to some extent system given. Due to fragmentation, the number of point records per I/O unit may vary by ±1. In general, the points in an I/O unit are not perfectly aligned to rows and columns of the grid, as in the 2-dimensional example depicted in figure 65.

Figure 66 shows which pairs of I/O units must be considered for join processing. Each entry in the matrix stands for one pair of I/O units (taken from the example in figure 65), for instance, the upper left corner for the pair (1,1), i.e. the self join of “I/O-Unit 1”. For the self join operation, our algorithm needs not to consider the lower left triangular matrix due to the symmetry of the pairs. The pair (*x*,*y*) is equivalent to the pair (*y*,*x*), and, therefore, the lower left half is canceled in the figure. A large, but less regular part in the upper right corner is also cancelled. The corresponding pairs, for instance (1,4), are excluded from processing, because the complete I/O-Unit 1 is out of the ε-interval of I/O-Unit 4 (and vice versa, due to the symmetry of $\cdot \lessapprox_{\text{ego}} \cdot$).

In figure 66, a small area of pairs of I/O-Units remains (starting at the diagonal) which must be scheduled efficiently. We indicate one of the most obvious scheduling methods, column-by-column, by arrows in our running example. We start with the pair (1,1), proceed to (1,2), then (2,2), (1,3), and so on. Additionally, we mark the disk accesses caused by the schedule assuming main memory buffers for up to 3 I/O-Units which are replaced using a LRU strategy.

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

We avoid this *I/O thrashing effect* by switching into a different mode of scheduling, the *crabstep mode*. Since the ε-interval does not fit into main memory, obviously, we have to read some I/O units more than once. For those relational joins which have to form all possible pairs of I/O units or at least many of them (e.g. SELECT * FROM *A*,*B* WHERE *A.a*≠*B.b*) it is well known that the strategy of *outer loop buffering* is optimal. We adopt this strategy for the epsilon grid order where we do not have to form all possible pairs of I/O units, but only those in a common ε-interval. Our algorithm reserves in this mode only the main memory buffer for one I/O unit for the inner loop. Most of the buffer space is reserved for the outer loop, and the next I/O units from the outer loop are pinned in the buffer. The inner loop iterates over all I/O units which are in the ε-interval of any of the pinned pages. In figure 67, the two scheduling modes are visualized, assuming buffer space for up to 4 I/O units. Figure 67a shows the gallop mode where enough buffer space is available. Here, 6 disk accesses are enough to form 24 page pairs. Figure 67b shows the case where the gallop mode leads to I/O thrashing (36 disk accesses for 36 page pairs). In contrast, the crabstep mode depicted in figure 67c requires 16

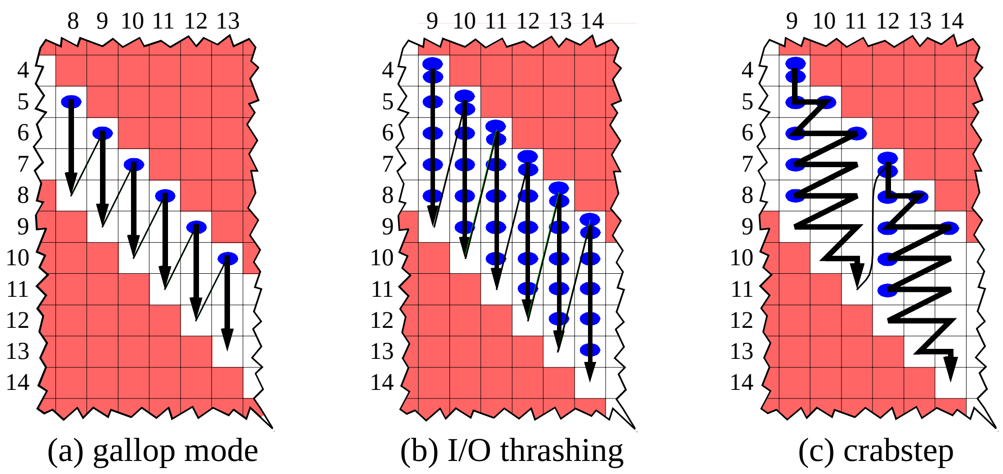


Figure 67: Scheduling Modes

disk accesses for 36 page pairs. The corresponding scheduling algorithm is shown in figure 68. Note that for a clear presentation the algorithm is simplified.

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

8.1.3 Joining Two I/O-Units

It is not optimal to process a pair of I/O units by direct comparisons between the points stored in the I/O units. Instead, our algorithm partitions the point set stored in each I/O unit into smaller subsets. In contrast to other partitioning approaches without preconstructed index, where partitioning requires multiple sorting of the subset according to

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

Figure 68: Scheduling Algorithm

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

Definition 13 (active, inactive and unspecified dimension): For a sequence $\langle p_1, p_2, ..., p_k \rangle$ of k points which are epsilon-grid-ordered (i.e. $p_1 \underset{\text{ego}}{\lessdot} p_2 \underset{\text{ego}}{\lessdot} \cdots \underset{\text{ego}}{\lessdot} p_k$) a dimension d_i is *active* if and only if the following two conditions hold:

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

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

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

The intuitive meaning of definition 13 is: The active dimension of a sequence is the first dimension where the points are extended over more than one grid cell length (if any exists). Due to the properties of the order relation, this can be decided according to the

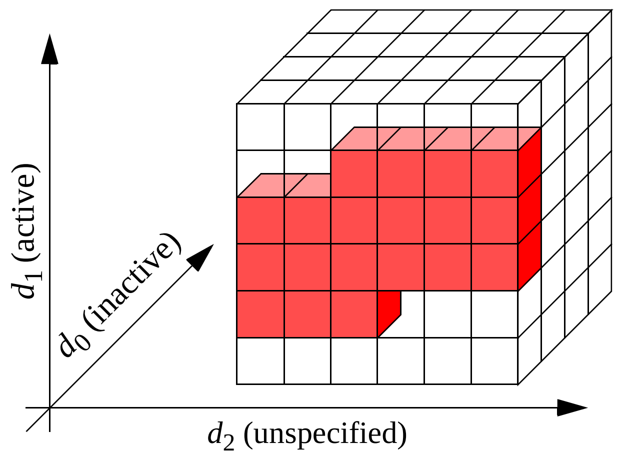


Figure 69: The active dimension of a sequence

first point p_1 and the last point p_k of the sequence. Dimension d_i is the first dimension where p_1 and p_k are different after dividing and rounding.

Figure 69 shows for a 3-dimensional data space an example sequence (shaded area) where d_1 is the active dimension. The particular property of the inactive dimensions is that they can be used very effectively to determine whether two sequences $P = \langle p_1, p_2, ..., p_k \rangle$ and $Q = \langle q_1, q_2, ..., q_m \rangle$ of epsilon-grid-ordered points have to be joined. They need *not* be joined if for at least one of the common inactive dimensions the distance between the cells exceeds ε. Formally: If $\exists\, d_j$ such that d_j is inactive in P and d_j is inactive in Q and

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

Active and unspecified dimensions are not used for excluding a sequence from being join mate. Figure 70 shows our recursive algorithm for the join of two sequences. It has two terminating cases: (^{1}) the rule discussed above applies and (^{2}) both sequences are short enough. The cases where only one sequence has more than *minlen* points are straightforward and left out in figure 70.

8.2 Optimization Potential

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

8.2.1 Separate Optimization of I/O and CPU

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

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

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

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

Figure 70: Algorithm for Joining Sequences

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

For Epsilon Grid Ordering, no directory is explicitly constructed. Instead, the point sequences (stored as arrays) are recursively decomposed. Therefore, the only space overhead of our technique is the recursion stack which is O (log n). Our technique can optimize the final size of the sequences (parameter *minlen* in figure 70) without considering any limiting overhead.

```
function distance_below_eps (Point $p$, Point $q$): boolean
    $distance\_sq := 0.0$ ;
    for $i := 0$ to $d - 1$ do
1       $j := dimension\_order[i]$ ;
        $distance\_sq := distance\_sq + (p[j] - q[j])^2$ ;
        if $distance\_sq > \varepsilon^2$ then return false ;
    return true ;
```

Figure 71: Algorithm for Distance Calculations

8.2.2 Active Dimensions and Distance Calculations

In spite of the CPU optimization proposed in section 8.2.1 the CPU cost is dominated by the final distance calculations between candidate pairs of points. A well-known technique to avoid a considerable number of these distance calculations is to apply the triangle inequality [BEKS 00]. In our experiments, however, the triangle inequality did not yield an improvement of the Epsilon Grid Order due to the use of small, CPU optimized sequences. A more successful way is to determine the distances between two points (dimension by dimension) and testing in each step whether the distance already exceeds ε. The corresponding algorithm is depicted in figure 71.

For this step-by-step test, it is essential that the dimensions are processed in a suitable order, depending on the inactive dimensions, because some dimensions have a rather high probability of adding large values to the distance (a high *distinguishing potential*), others not. Therefore, in the line marked with (**1**) the dimensions are taken from a lookup table which is sorted according to the *distinguishing potential*. The lookup table is filled when starting the join between two minimal sequences. In the following we will show how to estimate the distinguishing potential of the dimensions for a given pair of se-

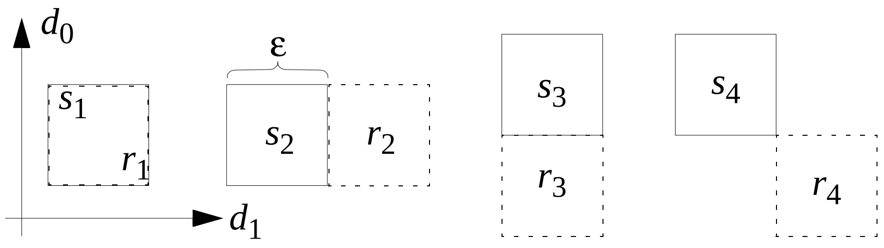


Figure 72: Distinguishing Potential of the Dimensions

quences. For the analysis in this section, we assume that the points of a sequence follow a uniform (not necessarily independent) distribution in the inactive dimensions, i.e. if d_i is inactive in sequence s and the corresponding cell extension in d_i is $[x_i{\cdot}\varepsilon..(x_i{+}1){\cdot}\varepsilon]$, then for the i-th coordinate p_i of each point $p \in s$ every value between $[x_i{\cdot}\varepsilon..(x_i{+}1){\cdot}\varepsilon]$ has the same probability. In the following, we determine the distinguishing potential of the inactive dimensions of a pair of sequences (i.e. the dimensions which are inactive in *both* sequences).

How large the distinguishing potential of a dimension d_i is, depends on the relative position of the two sequences in the data space (cf. figure 72). Since we consider only the inactive dimensions (in the example both dimensions d_0 and d_1), both sequences s_j and r_j have an extension of ε in all considered dimensions. Due to the grid, the sequences are in an inactive dimension d_i either perfectly *aligned* to each other or directly *neighboring*. In figure 72, s_1 and r_1 are aligned in both dimensions; s_4 and r_4 are *neighboring* in both dimensions; s_2 and r_2 are aligned in d_0, and s_3 and r_3 are aligned in d_1, neighboring in the other dimension. Other relationships are not considered, because if the sequences are neither aligned nor neighboring, they are excluded from processing, as described in section 8.1.3.

A single, aligned dimension has no distinguishing power at all, because the difference between two coordinates is at most the cell length ε. It is possible that the combination of several aligned dimensions distinguishes points, but not very likely. In contrast, a dimension where the two sequences are *neighboring* has a high distinguishing power.

Under the above mentioned assumptions the distinguishing power can be determined as follows, according to the sequences s_2 and r_2 in figure 72 for which we determine the distinguishing power of d_1: A point on the left boundary of s_2 cannot have any join mate on r_2 (exclusion probability 1). For points on the right boundary of s_1, no points on r_2 *can be excluded* by only considering d_1 (probability 0). Between these extremes, the exclusion probability (with respect to d_1) decreases linearly from 1 to 0 (e.g. 50% for a point in the middle of s_2). Integrating this linear function yields an overall exclusion probability of 50% for each neighboring dimension.

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

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

This order reveals decreasing distinguishing powers of the dimensions and leads to an exclusion of point pairs as early as possible in the algorithm of figure 71.

8.3 Experimental Evaluation

In order to show the benefits of our technique we implemented the EGO-algorithm and performed an extensive experimental evaluation using database sizes of well beyond 1 GB. For comparison, we applied the original source code of the Multipage Index Join [BK 01] and a similarity join algorithm based on the R-tree spatial join (RSJ) algorithm [BKS 93]. The latter join algorithm, *RSJ with Z-ordering optimization,* employs a page

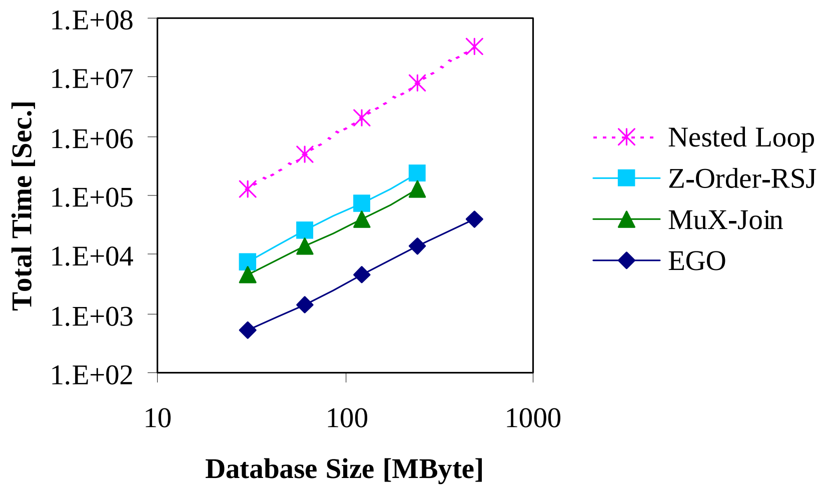


Figure 73: Experimental Results on Uniformly Distributed, 8-Dimensional Data

scheduling strategy based on Z-ordering and will be denoted as *Z-Order-RSJ*. It is very similar to the Breadth-First-R-tree-Join (BFRJ) proposed in [HJR 97]. The values for the well known nested loop join with its quadratic complexity were merely calculated and should give a reference for comparison. All algorithms were allowed to use the same amount of buffer memory (10% of the database size).

For our new technique, EGO, we considered both CPU cost as well as I/O cost, including the sorting phase which was implemented as a mergesort algorithm on secondary storage. As in figure 68 shown, our algorithm switches between the gallop and the crabstep mode on demand.

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

All our experiments were carried out under Windows NT4.0 on Fujitsu-Siemens Celsius 400 machines equipped with a Pentium III 700 MHz processor and 256 MB main memory (128 MB available for the cache). The installed disk device was a Seagate

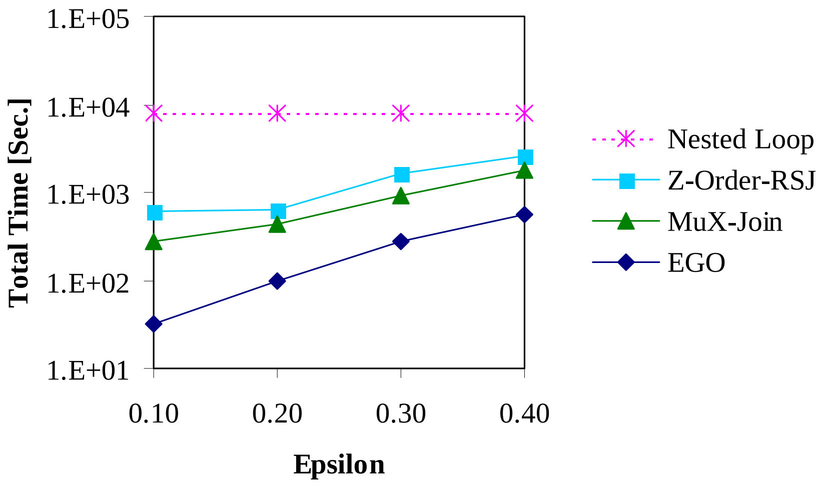


Figure 74: Experimental Results on Uniformly Distributed, 8-Dimensional Data

ST310212A with a sustained transfer rate of about 9 MB/s and an average read access time of 8.9 ms with an average latency time of 5.6 ms.

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

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

Figure 74 shows our experiments using uniformly distributed 8-dimensional point data. In the left diagram, the database size is varied from 0.5 million to 40 million points while on the right side results are compared for varying values of the ε parameter. The largest database was about 1.2 GB. For this size (as well as for the 20 million points) only the results for EGO could be obtained in reasonable time. The nested loop join has the worst performance off all the compared techniques. The Z-Order-RSJ outperforms the nested loop join by factors ranging from 30 to 140 while the MuX-Join still is at least two

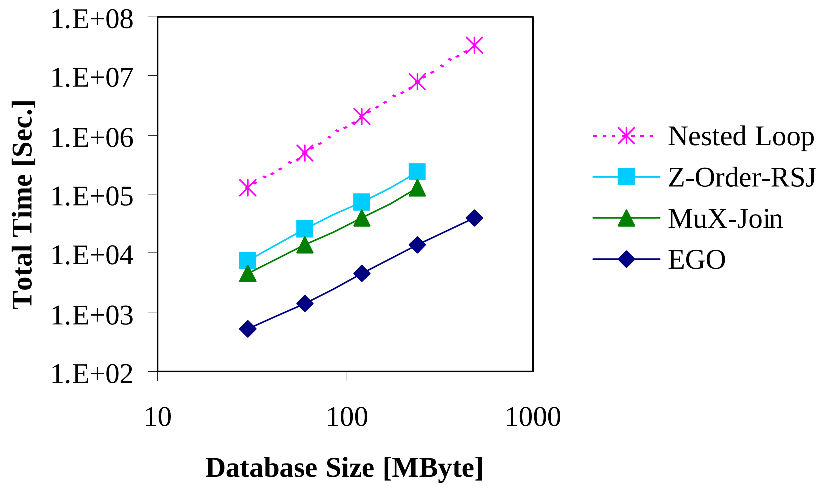


Figure 75: 16-Dimensional Real Data from a CAD-Application (Scalability)

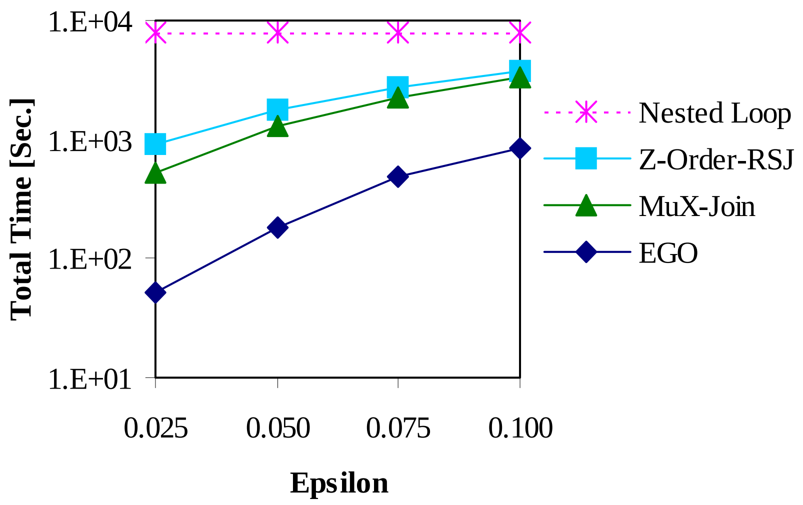


Figure 76: 16-Dimensional Real Data from a CAD-Application (Epsilon)

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

The experiments with real data are depicted in figure 76. The results for the 16-dimensional CAD data set confirm our experiments on uniform data. Again, the left diagram shows performance for varying database size while the right diagram shows performance for varying ε values. EGO was 9 times faster than the MuX-Join for the largest database size and 16 times faster than the Z-Order-RSJ. In the right diagram we can observe, that the performance of the MuX-Join and the Z-Order-RSJ converge for larger ε values while EGO still shows substantially better performance for all values of ε. The improvement factors of our technique varied between 4.0 and 10 over the Multipage Index and between 4.5 and 17 over Z-Order-RSJ.

8.4 Conclusions

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

Chapter 9
k-Nearest Neighbor Joins: Turbo Charging the KDD Process

Our previous chapters have primarily concentrated on the distance range join where the user has to provide a similarity distance ε to define the join operation. Although there are numerous applications to the distance range join such as various clustering algorithms, a general problem of this join operation is the difficulty to determine a suitable query parameter ε. This parameter is not very intuitive to the user because the user has in most cases no concept about typical feature vectors and similarity distances. Feature distances are only meaningful to the user in comparisons with other feature distances.

This is of course not a problem of the similarity join itself but rather of the concept to use such a given radius in the corresponding data mining algorithms. The similarity join inherits the corresponding problems from the data mining algorithm.

It is a consequence of the curse of dimensionality [BGRS 00] that the cardinality of the join result is highly sensitive to a suitable choice of the radius ε. If ε is too small, the join result will be empty. If ε is too large, the join result will be equal to the cartesian product $R \times S$. With increasing dimension, the interval of a *sensible* radius ε where the join result is *non-trivial* is becoming more and more narrow.

The problems of a fixed query range ε can be overcome by replacing the range query based join predicate by a nearest neighbor based join predicate where the user defines a result cardinality parameter *k*. The cardinality parameter *k* is much more intuitive to the user than the distance parameter ε is.

Therefore, Hjaltason and Samet have proposed the *k*-distance join [HS 98] which retrieves those *k* pairs from the cross-product $R \times S$ which have least distance. Up to the rare tie situations, the result cardinality exactly corresponds to *k* which obviously solves the cardinality control problem.

We believe, however, that the applications of the *k*-distance join and its incremental version are rather limited. The authors mention applications in geographical information systems including queries like “find the *k* cities nearest to any river”. Standard tasks of data mining and knowledge discovery in databases are difficult to implement on top of the *k*-distance join.

Many standard tasks of data mining, however, evaluate *k*-nearest neighbor queries for a large number of query points. Examples are clustering algorithms such as *k*-means [McQ 67], or *k*-medoid [KR 90], but also data cleansing and other pre- and postprocessing techniques e.g. when sampling plays a role in data mining.

In this chapter, we propose a third kind of similarity join, the *k*-nearest neighbor similarity join, short *k*-nn join. This operation is motivated by the observation that the vast majority of 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*.

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.

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. Then, the *k*-distance diagram is a technique for a suitable parameter selection for data mining. 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 [BKKS 01]. 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. In this chapter, we show how some of these standard algorithms can be based on top of the *k*-nearest neighbor join. These standard algorithms are

- *k*-means and *k*-medoid clustering
- *k*-nearest neighbor classification
- sample postprocessing
- and *k*-distance diagrams, a method for determining a suitable radius ε in density based clustering methods.

We will evaluate them in the following sections.

9.1 *k*-Means and *k*-Medoid Clustering

The *k*-means method [HK 00] 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 centers such that each database point can be assigned to one of the centers to minimize the overall distance of the database points to their associated center points.

The basic algorithm for *k*-means clustering works as follows: In the initialization, *k* database points are randomly selected as tentative cluster centers. Then, each database point is associated to its closest center point and, thus, a tentative *cluster* is formed. Next, the cluster centers are redetermined as the means point of all points of the center, simply by forming the vector sum of all points of a (tentative) cluster. The two steps (1) point association and (2) cluster center redetermination are repeated until convergence (no more considerable change). It has been shown that (under several restrictions) the algorithm always converges. The cluster centers 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 center.

The *k*-means algorithm is visualized in figure 77 using $k = 3$. At the left side (a) $k = 3$ points (white symbols ◇ △ □) are randomly selected as initial cluster centers. Then in figure 77(b) the remaining data points are assigned to the closest center which is depicted by the corresponding symbols (◆ ▲ ■). The cluster centers are redetermined (moving arrows). The same two operations are repeated in figure 77(c). If the points are finally assigned to their closest center, 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 recognized in the data mining community that the point association step which is performed in each iteration of the algorithm corresponds to a ($k = 1$) near-

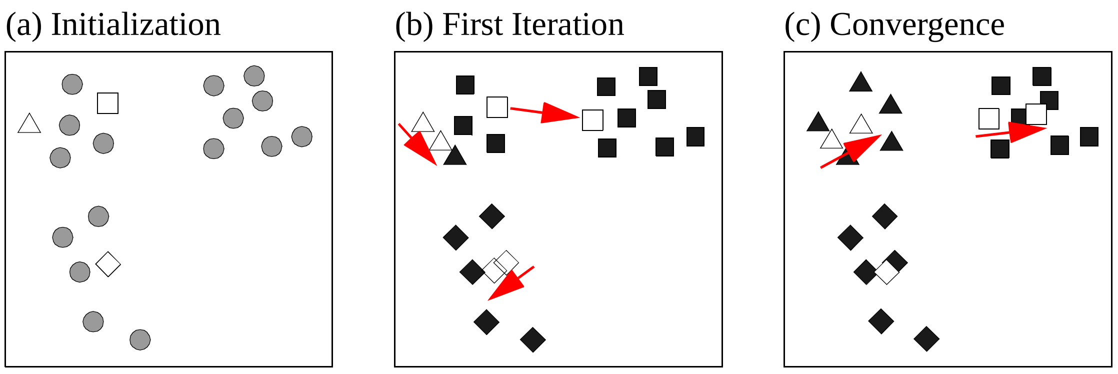


Figure 77: *k*-Means Clustering

est neighbor join between the set of center points (at the right side) and the set of database points (at the left side of the join symbol) because each database point is associated with its nearest neighbor among the center points:

$$\text{database-point-set} \bowtie_{1\text{-nn}} \text{center-point-set}$$

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) $\in$ database-point-set $\bowtie_{1\text{-NN}}$ center-point-set do
    if dp.center $\neq$ cp.id then change := true ;
    dp.center := cp.id ;
    cp.newsum := cp.newsum + dp.point ;
    cp.count := cp.count + 1 ;
  foreach cp $\in$ center-point-set do
    cp.point := cp.newsum / cp.count ;
until $\neg$ change ;
```

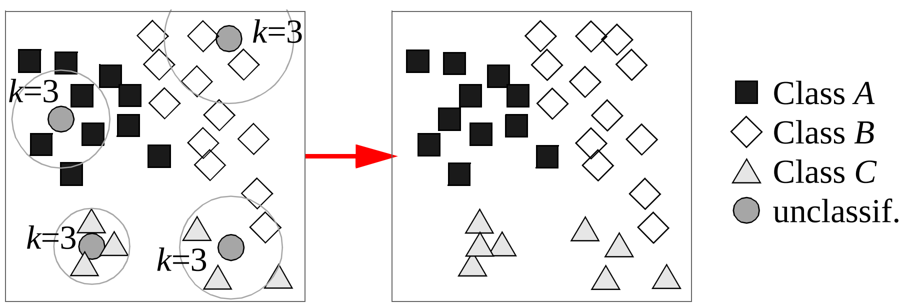


Figure 78: *k*-Nearest Neighbor Classification

9.2 *k*-Nearest Neighbor 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 running example of astronomy databases we have different classes of stars, galaxies, planets etc.). For classification, also a set of objects without class label (newly detected objects) is 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* [HK 00]). Another important approach is *k*-nearest neighbor classification [HT 93]. Here, for each unclassified object, a *k*-nearest neighbor 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 visualized in figure 78. As for each unclassified object a *k*-nn-query on the set of classified objects is evaluated, this corresponds again to a *k*-nearest neighbor join:

$$\text{unclassified-point-set} \ltimes_{k\text{-}nn} \text{classified-point-set}$$

9.3 Sampling Based Data Mining

Data mining methods which are based on sampling often require a *k*-nearest neighbor 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 neighbor join can give hints 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 [BKKS 01]. 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.

9.4 *k*-Distance Diagrams

The most important limitation of the DBSCAN algorithm is the difficult determination of the query radius ε. In [SEKX 98] a method called *k*-distance diagram is proposed to determine a suitable radius ε. For this purpose, a number of objects (typically 5-20 percent of the database) is randomly selected. For these objects, a *k*-nearest neighbor 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 neighbor of each are then sorted and depicted in a diagram (cf. figure 79). Vertical gaps in that plot indicate distances that clearly separate different clusters, because there exist larger *k*-nearest neighbor distances (inter-cluster distances, noise points) and smaller ones (intra-cluster distance). As for each sample point a *k*-nearest neighbor que-

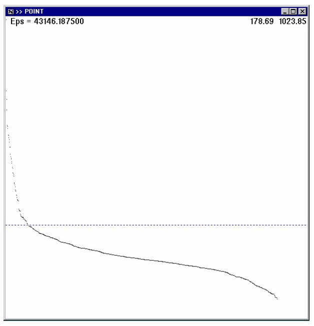


Figure 79: *k*-Distance Diagram

ry 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:

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

9.5 Conclusions

In this chapter, we have proposed 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 shown that the *k*-nn join is 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. Therefore, we believe that the *k*-nearest neighbor join will gain much attention as an important database primitive to speed up the complete KDD process.

Chapter 10
Processing k-Nearest Neighbor Joins Using MuX

In this chapter, we show how the operation of a *k*-nearest neighbor similarity join can be efficiently implemented on top of a multidimensional index structure. In chapter 6 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) (cf. chapter 7) 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 a formal proof is up to future work, we believe that also the *k*-nn join clearly benefits from the separate optimization, because the 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. A simple strategy for the higher index levels is sufficient, because most cost arise in the data level. Therefore, our strategies focus on the last level, the data pages.

10.1 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 [BK 01] 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 80. 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 point, 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 strat-

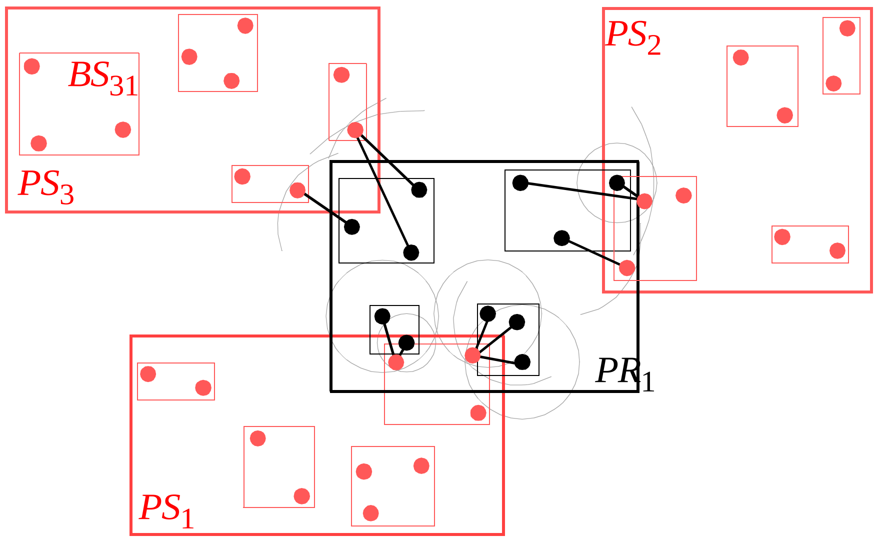


Figure 80: *k-nn* Join on the Multipage Index (here *k*=1)

egy 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.

```
1  **foreach** $PR$ of $R$ **do**
2      $cand$ : PQUEUE [$|PR|, k$] of point := $\{\perp,\perp,...,\perp\}$ ;
3      **foreach** $PS$ of $S$ **do** $PS$.done := false ;
4      **while** $\exists\ i$ **such that** $cand\ [i]$ is not confirmed **do**
5          **while** $\exists$ empty cache frame $\wedge$
6                $\exists\ PS$ with ($\neg PS$.done $\wedge\ \neg$ 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 10.2. 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 10.3. 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.

10.2 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 [BBKK 97]. 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:

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

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 highest quality.

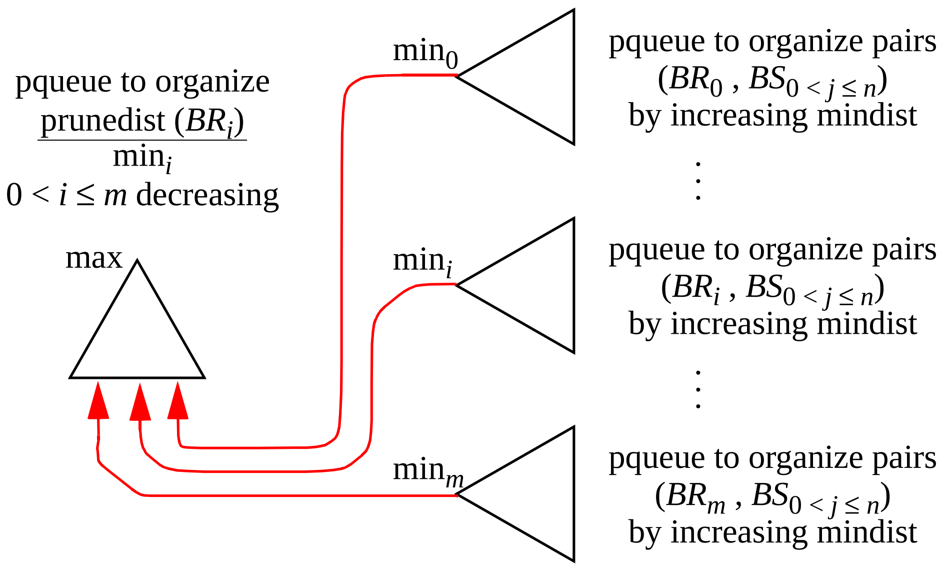
10.3 Processing Strategy

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 10.2, the quality Q(*BR*,*BS*) of a bucket pair rewards a small distance and punishes a small pruning distance:

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

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-cut data structure, a fractionated pqueue (half sorted tree). By *fractionated* we mean a pqueue of pqueues, as depicted in figure 81. 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.

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



pqueue to organize
prunedist (BR_i)
$\min_i$
$0 < i \leq m$ decreasing
max
$\min_0$
$\min_i$
$\min_m$
pqueue to organize pairs (BR_0 , $BS_{0 < j \leq n}$) by increasing mindist
pqueue to organize pairs (BR_i , $BS_{0 < j \leq n}$) by increasing mindist
pqueue to organize pairs (BR_m , $BS_{0 < j \leq n}$) by increasing mindist

Figure 81: Structure of a fractionated pqueue

by *PS* appears. We select that hosting page *PS* with the fewest active pairs (i.e. the hosting page that causes least effort).

10.4 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 [BK 01] 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 optimized for the *k*-nn case) and the *k*-nn algorithm by Hjaltason and Samet [HS95] 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

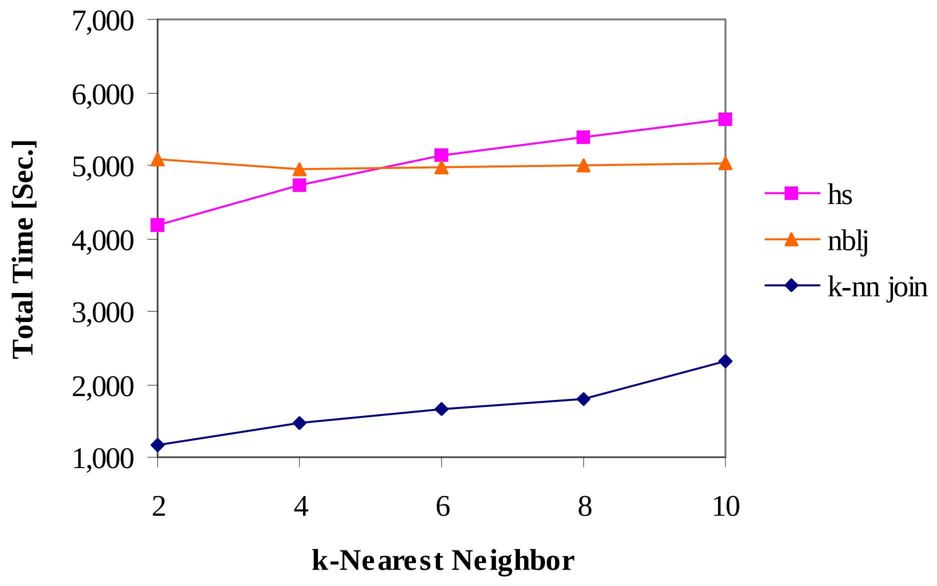


Figure 82: Varying k for 8-dimensional uniform data

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 included figure 82 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 82 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 83 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 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 84. Obviously, the *k*-nn join scales much better with increasing size of the database than the other two techniques.

Figure 85 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 86 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*-

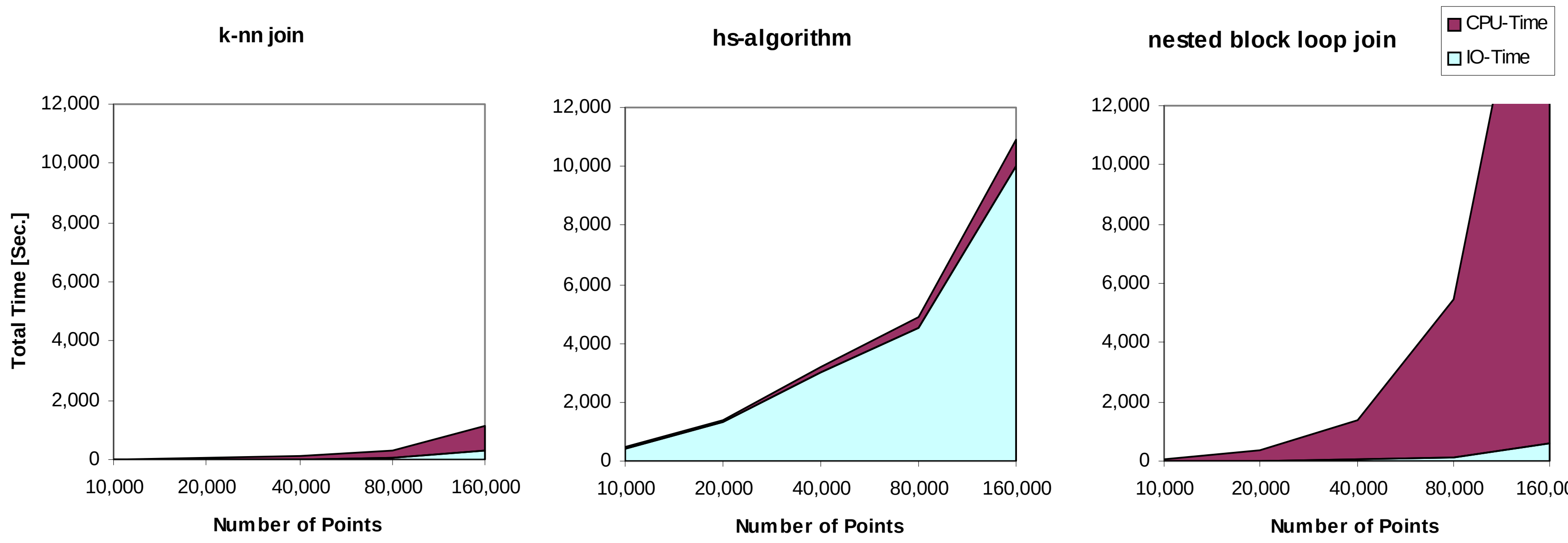


Figure 83: Total Time, CPU-Time and I/O-Time for hs, *k-nn* join and nblj for varying size of the database

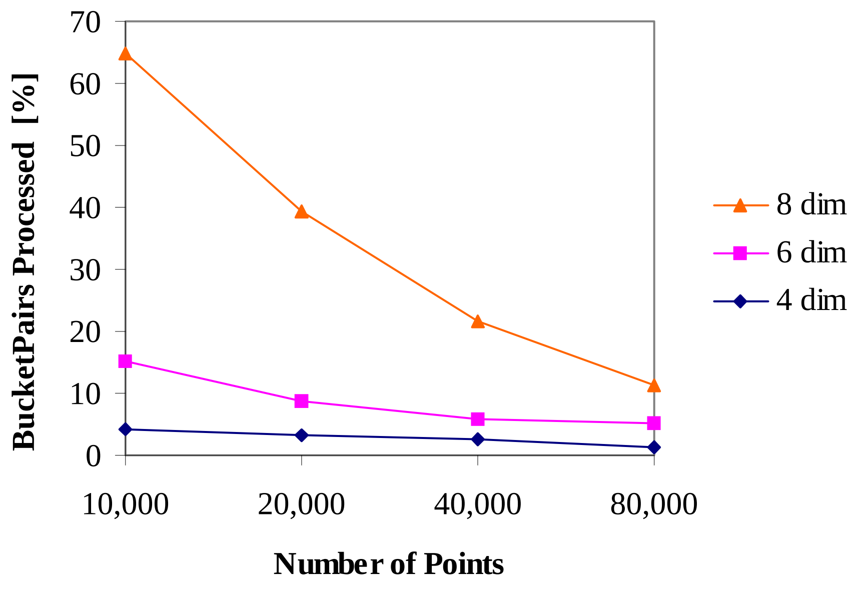


Figure 84: Pruning of bucket pairs for the *k-nn* join

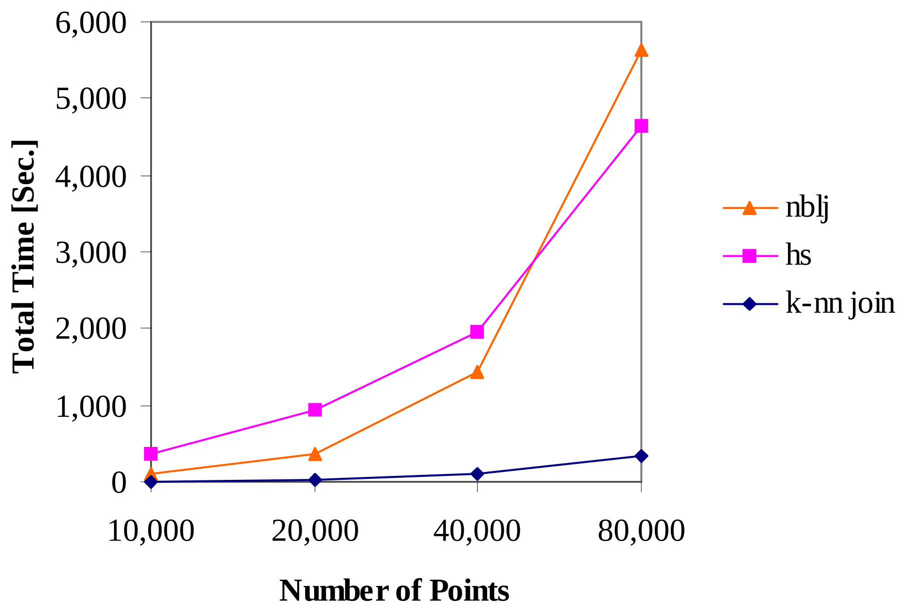


Figure 85: Results for 9-dimensional weather data

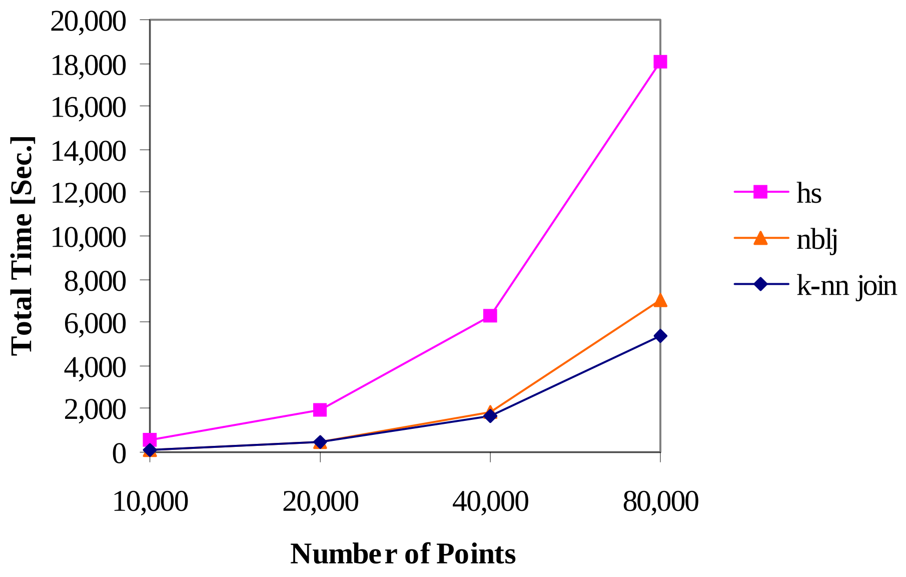


Figure 86: Results for 16-dimensional CAD data

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 5.

10.5 Conclusions

This chapter was dedicated to the efficient implementation of the *k*-nearest neighbor join. We have argued that our Multipage Index (MuX) which has already been introduced in chapter 6 for the distance range join is also an adequate index structure for processing *k*-nearest neighbor joins. Our analysis in chapter 5 of the distance range join is also valid for the *k*-nn join since the optimization tradeoffs are quite similar.

For the *k*-nearest neighbor join on top of a complex index structure, a new algorithm was needed along with some good strategies for accessing pages and processing page pairs in the main memory.

We have proposed strategies for three different tasks:

- **Loading Strategy:**
  This strategy determines the order in which hosting pages of the Multipage Index (MuX) are fetched into the main memory
- **Processing Strategy:**
  This strategy determines the order in which pairs accommodated buckets are formed for joining them in the main memory

We have implemented a *k*-nearest neighbor join algorithm which applies these strategies. We have conducted an extensive experimental evaluation in which the clear superiority over competitive approaches was shown.

Chapter 11
Optimizing the Similarity Join

Due to its high importance, many different algorithms for the similarity join have been proposed, operating on multidimensional index structures [BKS 93, LR 94, HJR 97], multidimensional hashing [LR 96, PD 96], or various sort orders [SSA 97, KS 97, BBKK 01]. In contrast to algorithms for simple similarity queries upon a single data set (such as *range queries* or *nearest neighbor queries*), all of these algorithms are clearly CPU bound. In spite of the filtering capabilities of the above algorithms the evaluation cost are dominated by the final distance calculations between the points. This is even true for index structures which are optimized for minimum CPU cost [BK 01].

Therefore, in the current chapter, we propose a technique for *avoiding* and *accelerating* a high number of the distance calculations between feature vectors. Our methods shows some resemblance to the principle of plane-sweep algorithms [PS 85] which is extended by the determination of an optimal order of dimensions. A design objective of our technique was generality, i.e. our method can be implemented on top of a high number of basic algorithms for the similarity join such as R-tree based joins [BKS 93, LR 94, HJR 97], hashing based methods [LR 96, PD 96], and sort orders [SSA 97, KS 97, BBKK 01]. A precondition for our optimal dimension order is to have some notion of *partitions* to be joined having a position and extension in the data space. This is given for

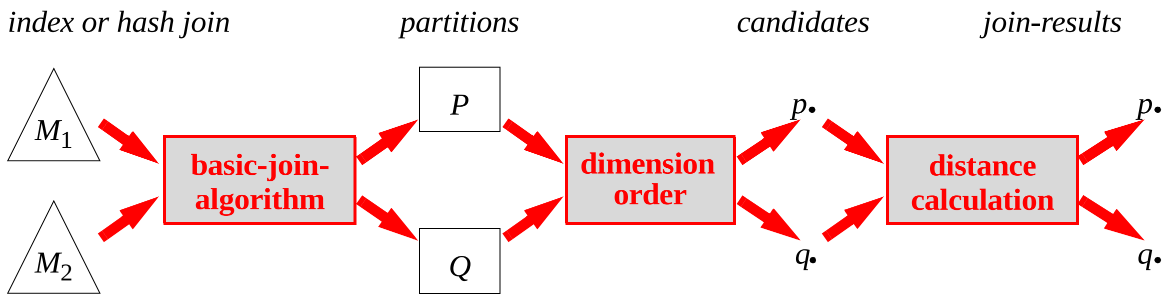


Figure 87: Integration of the dimension-order-algorithm

all techniques mentioned above. Our technique is not meaningful on top of the simple nested loop join [Ull 89].

11.1 Optimal Dimension Order

In this section we will develop a criterion for ordering the dimensions to optimize the distance computations. We assume that our join algorithm with optimal dimension order is preceded by a filter step based on some spatial index structure or spatial hash method which divides the point sets that are to be joined into rectangular partitions and only considers such partitions that have a distance to each other of at most ε. Suitable techniques are depth-first- and breadth-first-R-tree-Join [BKS 93, HJR 97], Spatial Hash Join [PD 96, LR 96], Seeded Trees [LR 94], the ε-kdb-tree [SSA 97], the Multidimensional Join (MDJ) [KS 98], or the ε-grid-order [BBKK 01].

11.2 Algorithm

The integration of the dimension order is shown in figure 87. Our dimension order algorithm receives partition pairs (P, Q) from the basic technique for the similarity join and generates point-pairs as candidates for the final distance calculations. The general idea of the dimension order is as follows: If the points of one partition, say *Q* are sorted by

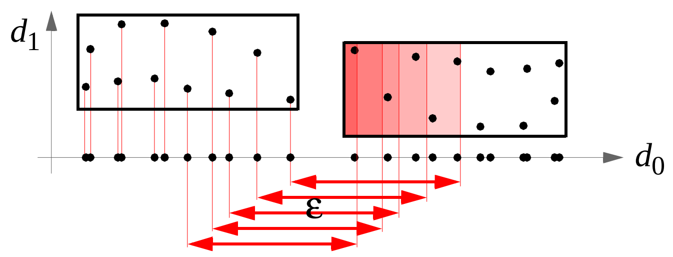


Figure 88: Idea of the Dimension Order

one of the dimensions, then the points of Q which can be join mates of a point p of P form a contiguous sequence in Q (cf. figure 88). A large number of points which are excluded by the sort dimension can be ignored. In most cases, the points which can be ignored, are located at the lower or upper end of the sorted sequence, but it is also possible, that the sequence of points that must be processed are in the middle of the sequence. In the latter case, the start and end of the sequence of relevant points must be searched e.g. by binary search, as depicted in the algorithm in figure 89. In the other cases, it is actually not necessary to determine the first (last) point before entering the innermost loop. Here, we can replace the search by a suitable break operation in the innermost loop.

11.3 Determining the Optimal Sort Dimension

If two partitions are joined together by the basic technique, we can use the following information in order to choose the optimal dimension:

- The distance of the two partitions with respect to each other or the overlap (which we will interpret as *negative* distance from now on) in each case projected on the one single dimension of the data space. We observe that the overall distance of the two partitions as well as the distance projected on each of the dimensions cannot exceed ε as otherwise the whole partition pair would have been eliminated by the *preprocessing step* in the basic technique (see figure 90).

```
algorithm optimal_dimension_order_join (index $M_1$, $M_2$)
        the similarity-join basic method
        generates partition pairs from $M_1$ and $M_2$ ;
        for all parition pairs (P,Q) with dist(P,Q) $\le \varepsilon$
            determine best sort dimension $s$ according to Eq. (12) ;
            sort (indirectly) points in Q according to dimension $s$ ;
            for all points $p \in$ P
                determine the first point $a \in Q: |a_s - p_s| \le \varepsilon$ ;
                determine the last point $b \in Q: |b_s - p_s| \le \varepsilon$ ;
                for all points $q \in$ Q with $a_s \le q_s \le b_s$
                    if dist(p,q) $\le \varepsilon$
                        output $(p,q)$ ;
end ;
```

Figure 89: Algorithmic Scheme

- The extent of the two partitions with respect to each of the single dimensions

In order to demonstrate that both distance as well as extent, really do matter, see figure 91: In both cases the distance of the two partitions is the same. For simplification we show one exemplary point with its ε-neighborhood in partitions Q and Q´ although in our further discussion we assume uniform distribution of points within the two partitions i.e. we will not consider one specific point.

On the left side of figure 91 both of the partitions are roughly square. Let us first look at the projection on the d_0-axis: We observe that about 70% of the projected area of P lies within the projected ε-neighborhood of our sample point in Q. If we were to choose d_0 as sort-dimension only about 30% of the points can be excluded as join-mates for our sample point in the first step. For the remaining 70% we still have to test dimension d_1 i.e. we have to compute the overall point distance. If we now look at the projection on dimension d_1: here only 25% of the area of P lies within the ε-neighborhood of our

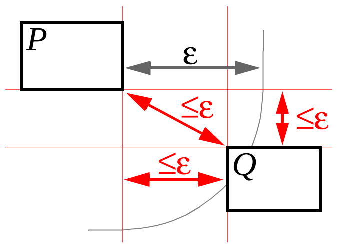


Figure 90: ε for partitions and projections

sample point in Q. If we choose d_1 as start-dimension as much as 75% of the points are already eliminated in the first step of our dimension ordering algorithm. In the case of quadratic partitions it is thus advisable to choose the dimension within which the partitions have the largest distance with respect to each other as this minimizes the area of the projected ε-neighborhood.

The right side of figure 91 shows the two partitions P' and Q' which have a much larger extent in dimension d_0 than in dimension d_1. For this reason the projection on the d_0-axis, with a portion of 33% of the area, is much better than the projection on the d_1-axis (75%). In this case the dimension d_0 should be chosen as sort-dimension.

We can note the following as a first rule of thumb for the selection of the dimension: for approximately square partitions choose the dimension with the greatest distance, otherwise the dimension with the greatest extent.

11.4 Probability Model

In the following we will propose a model which grasps this rule of thumb much more precisely. Our model is based on the assumption that the points within each partition follow a uniform distribution. In our previous work it has already been shown that this assumption is sufficiently fulfilled (e.g. [BBJ+ 00]). The results of our experiments in section 4 will provide even more justification for this assumption.

Our model determines for each dimension the probability $W_i[\varepsilon]$ (termed mating probability in the following) that two points in partitions P and Q with given rectangular boundaries $P.\mathrm{lb}_i$, $P.\mathrm{ub}_i$, $Q.\mathrm{lb}_i$, $Q.\mathrm{ub}_i$ $(0 \le i < d$; lb and ub for lower bound and upper bound, respectively) have at most the distance ε with respect to dimension d_i.

Definition 14

Given two partitions P and Q, let $W_i[\varepsilon]$ denote the probability for each dimension d_i that an arbitrary pair of points (p,q) with $p \in P$ and $q \in Q$ has a maximal distance of ε with respect to d_i:

$$W_i[\varepsilon] := W(|p_i - q_i| \le \varepsilon),\ (p,q) \in (P,Q) \tag{1}$$

If $P.\#$ denotes the number of points within partition P, then the expectation of the number of point pairs which are excluded by the optimal dimension order join equals to

$$E_i[\varepsilon] = P.\# \cdot Q.\# \cdot (1 - W_i[\varepsilon]). \tag{2}$$

This means that exactly the dimension d_i should be chosen as sort dimension that *minimizes* the mating probability $W_i[\varepsilon]$.

We will now develop a universal formula to determine the mating probability. We assume uniform distribution within each of the partitions P and Q. Thus the i-th component p_i of the point $p \in P$ is an arbitrary point from the uniform interval given by $[P.\mathrm{lb}_i .. P.\mathrm{ub}_i]$. The pair (p_i,q_i) is chosen from an independent and uniform distribution within the two-dimensional interval $[P.\mathrm{lb}_i..P.\mathrm{ub}_i] \times [Q.\mathrm{lb}_i..Q.\mathrm{ub}_i]$ because of the independence of the distributions within P and Q, which we can assume for $P \neq Q$. Hence the event space is given by

$$F_i = (P.\mathrm{ub}_i - P.\mathrm{lb}_i) \cdot (Q.\mathrm{ub}_i - Q.\mathrm{lb}_i) \tag{3}$$

$W_i[\varepsilon]$ is therefore given by the ratio of the portion of the area of F_i where p_i and q_i have a distance of at most ε to the whole area F_i. This can be expressed by the following integral:

$$W_i[\varepsilon] = \frac{1}{F_i} \cdot \int_{P.\mathrm{lb}_i}^{P.\mathrm{ub}_i} \int_{Q.\mathrm{lb}_i}^{Q.\mathrm{ub}_i} \begin{cases} 1 & for\ |x-y| \le \varepsilon \\ 0 & otherwise \end{cases} dydx \tag{4}$$

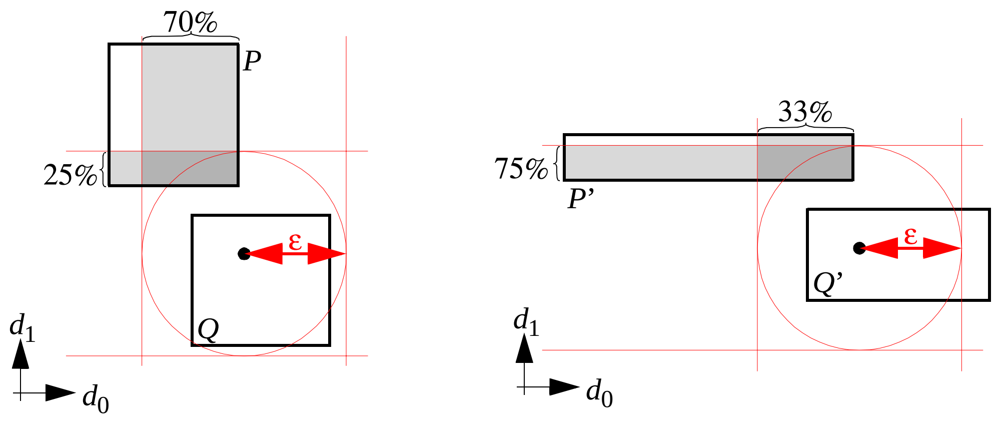


Figure 91: Distance and extension of partitions

In the following, we will show how this formula can be simplified using one exemplary case out of the several possible configurations. A complete list of all formulas for all possible cases will be given (without proof) thereafter.

First we need a consistent notion for the distance and the overlap of two partitions with respect to dimension d_i. We thus define a parameter δ_i whose sign indicates whether the partitions are disjoint in d_i ($\delta_i > 0$) or show some overlap ($\delta_i \leq 0$). The absolute value of δ_i represents either the distance or the overlap.

Definition 15

$$\delta_i := \max\{P.\mathrm{lb}_i, Q.\mathrm{lb}_i\} - \min\{P.\mathrm{ub}_i, Q.\mathrm{ub}_i\} \tag{5}$$

We can now simplify the integral of formula (4) by case analysis looking at the geometric properties of our configuration, i.e. we can transform our problem into d distinct two-dimensional geometric problems. To illustrate this, we look at the join of the two partitions P and Q in two-dimensional space as shown on the left hand side of figure 92. In this case, it is not directly obvious which dimension yields better results. The projection on d_0 which is the transformation that is used to determine $W_0[\varepsilon]$ is shown on the right hand side of figure 92. The range with respect to d_0 of points which can be stored in P is shown on the x-axis while the range with respect to d_0 of points which can be stored in

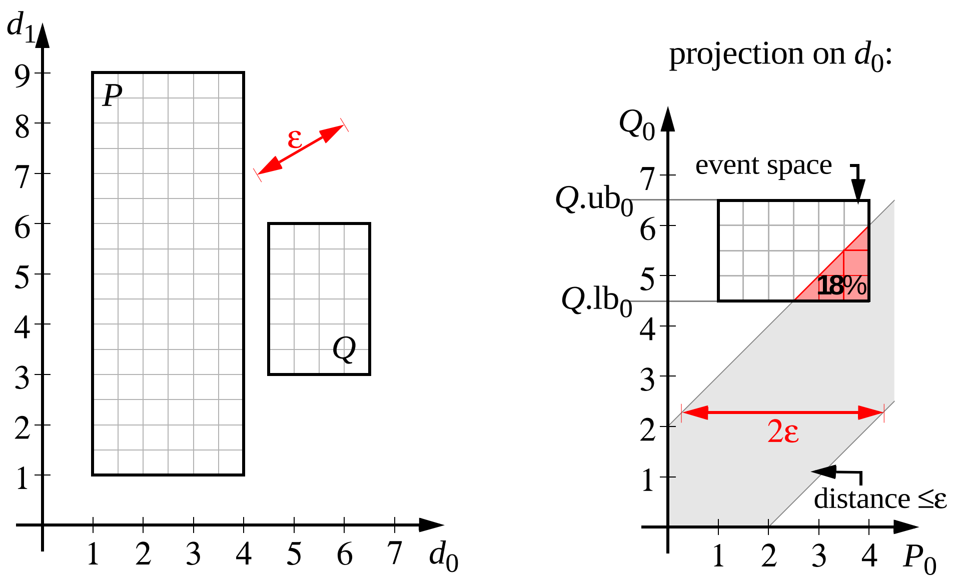


Figure 92: Determining the mating probability $W_0[\varepsilon]$

Q is shown on the y-axis. The projection (p_0,q_0) of an arbitrary pair of points $(p,q) \in (P,Q)$ can only be drawn inside the area denoted as event space (cf. equation 3), as all points of P with respect to dimension d_0 are by definition within $P.\mathrm{lb}_0$ and $P.\mathrm{ub}_0$. The same holds for Q.

The area within which our join condition is true for dimension d_0 i.e. the area within which the corresponding points have a distance of less than ε with respect to d_0 is marked in gray in figure 92. All these projections of pairs of points which fall into the gray area are located within a stripe of width 2ε (the ε-stripe) which is centered around the 45° main diagonal. All projections outside this stripe can be excluded from our search as the corresponding points already have a distance with respect to d_0 that exceeds our join condition. The intersection of this stripe with the event space represents those point pairs that cannot be excluded from our search using d_0 alone. The mating probability is given by the ratio of the intersection to the whole event space which equals 18% in our example.

We will now show one single simple lemma and then give a complete enumeration of relevant cases and discuss their efficient computation in the next section.

Lemma 6.

If the two distinct partitions P and Q, P≠Q, are either disjoint or show overlap of no more than ε i.e. $-\delta_i \le \varepsilon$ then

$$W_i[\varepsilon] = \frac{(\varepsilon - \delta_i)^2}{2F_i} \tag{6}$$

Proof. In this case $-\delta_i \le \varepsilon$. This means the intersection of the event space and the ε-stripe forms an isosceles-rectangular triangle with a lateral side length of $\varepsilon - \delta_i$. □

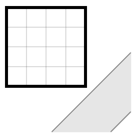
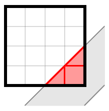
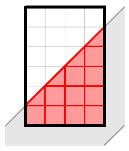
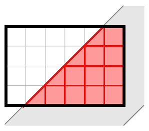
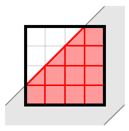
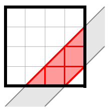
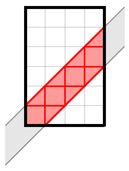
11.5 Efficient Computation

In the previous section, we have seen that the exclusion probability of a dimension d_i corresponds to the proportion of the event space which is covered by the ε-stripe. In this section, we show how this proportion can be efficiently determined. Efficiency is an important aspect here because the exclusion probability must be determined for each pair of mating pages (partitions) and for each dimension d_i.

Throughout this section we will use the shortcut PL for $P.\mathrm{lb}_i$ and similarly PU, QL, and QU. Considering figure 93-95 we can observe that there exists a high number of different shapes that the intersection of the event space and the ε-stripe can have. For each shape, an individual formula for the intersection area applies. We will show

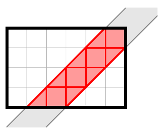
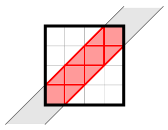
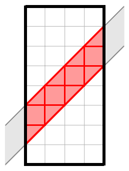
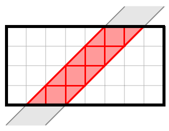
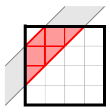
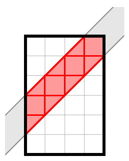
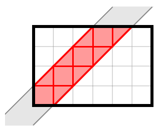
- that exactly 20 different shapes are possible,
- how these 20 cases can be efficiently distinguished, and
- that for each case a simple, efficient formula exists.

Obviously, the shape of the intersection is determined by the relative position of the 4 corners of the event space with respect to the ε-stripe. E.g. if 3 corners of the event space are *above* (or *left* from) the ε-stripe, and 1 corner is *inside* the ε-stripe, the intersection

#	Code	Figure	Formula
1	1111		0.0
2	1112		$\frac{(PU - QL + \varepsilon)^2}{2(PU - PL)(QU - QL)}$
3	1122		$\frac{(PU + PL)/2 + \varepsilon - QL}{QU - QL}$
4	1212		$\frac{PU - (QU + QL)/2 + \varepsilon}{PU - PL}$
5	1222		$1.0 - (\%)$
6	1113		$\frac{(PU - QL + \varepsilon)^2}{2(PU - PL)(QU - QL)} - (*)$
7	1123		$\frac{(PU + PL)/2 + \varepsilon - QL}{QU - QL} - (*)$

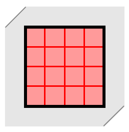
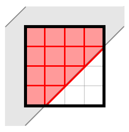
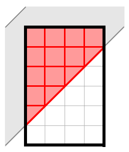
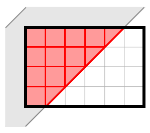
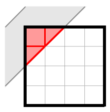
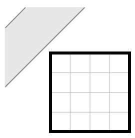
where $(\%) = \frac{(QU - PL - \varepsilon)^2}{2(PU - PL)(QU - QL)}$ and $(*) = \frac{(PU - QL - \varepsilon)^2}{2(PU - PL)(QU - QL)}$

Figure 93: Relative Positions of Event Space and ε-Stripe and Probability Formulas

#	Code	Figure	Formula
8	1213		$\frac{PU-(QU+QL)/2+\varepsilon}{PU-PL}-(*)$
9	1223		$1.0-(\%)-(*)$
10	1133		$\frac{2\varepsilon}{QU-QL}$
11	1313		$\frac{2\varepsilon}{PU-PL}$
12	1333		$\frac{(QU-PL+\varepsilon)^2}{2(PU-PL)(QU-QL)}-(\%)$
13	1233		$\frac{QU-(PU+\mathit{PL})/2+\varepsilon}{QU-QL}-(\%)$
14	1323		$\frac{(QU+QL)/2+\varepsilon-PL}{(PU-PL)}-(\%)$

where $(\%)=\frac{(QU-PL-\varepsilon)^2}{2(PU-PL)(QU-QL)}$ and $(*)=\frac{(PU-QL-\varepsilon)^2}{2(PU-PL)(QU-QL)}$

Figure 94: Relative Positions of Event Space and ε-Stripe and Probability Formulas

#	Code	Figure	Formula
15	2222		1.0
16	2223		1.0 – (*)
17	2233		$\frac{QU-(PU+PL)/2+\varepsilon}{QU-QL}$
18	2323		$\frac{(QU+QL)/2+\varepsilon-PL}{(PU-PL)}$
19	2333		$\frac{(QU-PL+\varepsilon)^2}{2(PU-PL)(QU-QL)}$
20	3333		0.0

where $(\%) = \frac{(QU-PL-\varepsilon)^2}{2(PU-PL)(QU-QL)}$ and $(*) = \frac{(PU-QL-\varepsilon)^2}{2(PU-PL)(QU-QL)}$

Figure 95: Relative Positions of Event Space and ε-Stripe and Probability Formulas

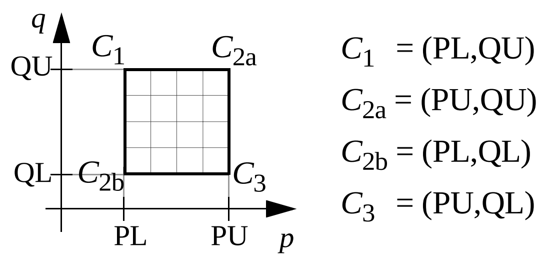


Figure 96: Identifiers for the Corners of the Event Space

shape is always a triangle (as discussed previously). For the relative position of a corner and the ε-stripe, we define the following cornercode *cc* of a point:

Definition 16 Cornercode (*cc*) of a point in the event space

A point (*p*,*q*) in the event space has the corner code *cc*(*p*,*q*) with

$$cc(p, q) = \begin{cases} 1 & \text{if } q > p + \varepsilon \\ 2 & \text{otherwise} \\ 3 & \text{if } q < p - \varepsilon \end{cases} \tag{7}$$

Intuitively, the cornercode is 1 if the point is left (or above) from the ε-stripe, 3 if it is right (or underneath) from the ε-stripe, and 2 if it is inside the ε-stripe (cf. figure 97). For an event space given by its upper and lower bounds (PL,PU,QL,QU), the corners are denoted as C_1, C_{2a}, C_{2b}, and C_3 as depicted in figure 96. We induce the cornercode to the complete event space given by its lower and upper bounds:

Definition 17 Cornercode *cc*(*ES*) of the event space

The cornercode of the event space *ES* given by the lower and upper limits

$$ES = (\text{PL},\text{PU},\text{QL},\text{QU}) \tag{8}$$

is the 4-tuple:

$$cc(ES) = (cc(C_1), cc(C_{2a}), cc(C_{2b}), cc(C_3)) \tag{9}$$

Formally, there exist 3^4=81 different 4-tuples over the alphabet {1,2,3}. However, not all these 4-tuples are geometrically meaningful. For instance it is not possible that simulta-

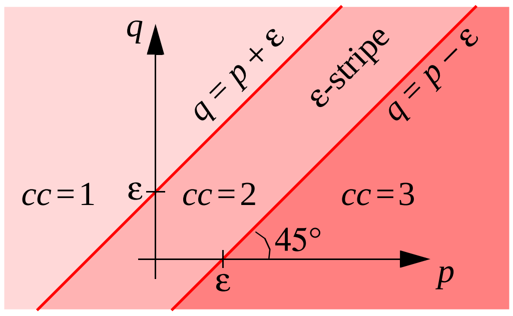


Figure 97: The ε-stripe

neously C_1 is below and C_3 above the ε-stripe. As C_1 is left from C_{2a} and C_{2a} is above C_3 we have the constraint:

$$cc\,(C_1) \le cc\,(C_{2a}) \le cc\,(C_3) \tag{10}$$

And as C_1 is above C_{2b} and C_{2b} is left from C_3 we have the constraint:

$$cc\,(C_1) \le cc\,(C_{2b}) \le cc\,(C_3) \tag{11}$$

The corner code of C_{2a} may be greater than, less than, or equal to the corner code of C_{2b}. The following lemma states that there are 20 4-tuples that fulfill the two constraints above.

Lemma 7. Completeness of Case Distinction

There are 20 different intersection shapes for the event space and the ε-stripe.

Proof. By complete enumeration of all four-tuples: There are 3 tuples where $cc(C_1) = cc(C_3)$: 1111, 2222, and 3333. If the difference between $cc(C_1)$ and $cc(C_3)$ is equal to 1 (i.e. tuples like 1??2 or 2??3), we obtain 2 possibilities for each of the corner codes $cc(C_{2a})$ and $cc(C_{2b})$, i.e. $2 \cdot 2^2 = 8$ different tuples. For a difference of two between $cc(C_1)$ and $cc(C_3)$, which corresponds to tuples like 1??3, we have a choice out of three for each of the corners C_{2a} and C_{2b}, i.e. $2^3 = 9$ tuples. Summarized, we obtain 20 different tuples □

Note that the cornercodes 1111 and 3333 which are associated with a probability of 0.0 actually are never generated because the corresponding partitions have a distance of more than ε and, thus, are excluded by the preceding filter step.

Each corner code of the event space is associated with a geometric shape of the intersection between event space and ε-stripe. The shape varies from a triangle (e.g. *cc* = 1112) to a six-angle (*cc* = 1223). The fact that only 45° and 90° angles occur facilitates a simple and fast computation. Figure 93 shows the complete listing of all 20 shapes along with the corresponding corner codes and the formulas to compute the intersection area. Note that Lemma 6 covers the cases 2 (*cc* = 1112) and 19 (*cc* = 2333).

The concept of the cornercodes is not only a formal means to prove the completeness of our case distinction but also provides an efficient means to implement the area determination. Our algorithm computes the corner code for each of the 4 corners of the event space, concatenates them using arithmetic operations and performs a case analysis between the 20 cases.

11.6 Determining the Optimal Sort Dimension

Our algorithm determines the sort dimension such that the mating probability $W_i[\varepsilon]$ is minimized. Ties are broken by random selection, i.e.

$$d_{\text{sort}} = \text{some } \{d_i \mid 0 \le i < d,\ W_i[\varepsilon] \le W_j[\varepsilon]\ \forall j,\ 0 \le j < d\}. \qquad (12)$$

Thus, we have an easy way to evaluate the formula for the sort dimension. As $W_i[\varepsilon]$ merely is evaluated for each dimension d_i, thus keeping the current minimum and the corresponding dimension in local variables, the algorithm is linear in the dimension d of the data space and independent of all remaining parameters such as the number of points stored in the partitions, the selectivity of the query, etc. Moreover, the formula must be evaluated only once per pair of partitions. This constant (with respect to the capacity of the partition) effort is contrasted by potential savings which are quadratic in the capacity (number of points stored in a partition). The actual savings will be shown in the subsequent section.

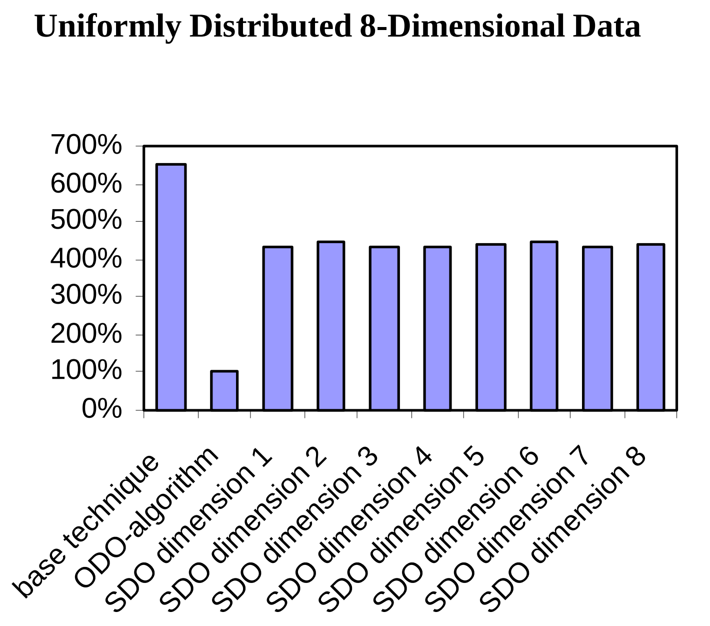
11.7 Experimental Evaluation

In order to show the benefits of our technique we implemented our optimal dimension-ordering algorithm on top of several basic similarity join methods and performed an extensive experimental evaluation using artificial and real data sets of varying size and dimension. For comparison we tested our algorithm not only against plain basic techniques, but also against a simple version of the dimension-ordering algorithm which does not calculate the best dimensions for each partition pair, but chooses one dimension which then is used globally for all partition pairs. In the following we will not only observe that our algorithm can improve CPU-efficiency by an important factor, but we will also see that it is optimal in the sense that it performs much better than the simple dimension-ordering algorithm – even if this algorithm chooses the best global dimension.

We integrated the ODO-algorithm into two index-based techniques, namely the *Multipage Index Join* (*MuX*) [BK 01] and the *Z-order-RSJ* which is based on the *R-tree Spatial Join (RSJ)* [BKS 93] and employs a page scheduling strategy using Z-ordering. The latter is very similar to the *Breadth-First-R-tree-Join* (*BFRJ*) proposed in [HJR 97]. We also implemented the ODO-algorithm into the recently proposed *Epsilon Grid Order* (*EGO*) [BBKK 01] which is a technique operating without preconstructed index.

The Multipage Index (MuX) is an index structure in which each page accommodates a secondary main-memory search structure which effectively improves the CPU performance of the similarity join. We implemented ODO on top of this secondary search structure, i.e. we measured the improvement that ODO brings on top of this secondary search structure. For comparison, we used the original MuX code which also exploited the secondary search structure.

All our experiments were carried out under Windows NT4.0 on Fujitsu-Siemens Celsius 400 machines equipped with a Pentium III 700 MHz processor and 256 MB main memory (128 MB available). The installed disk device was a Seagate ST310212A with



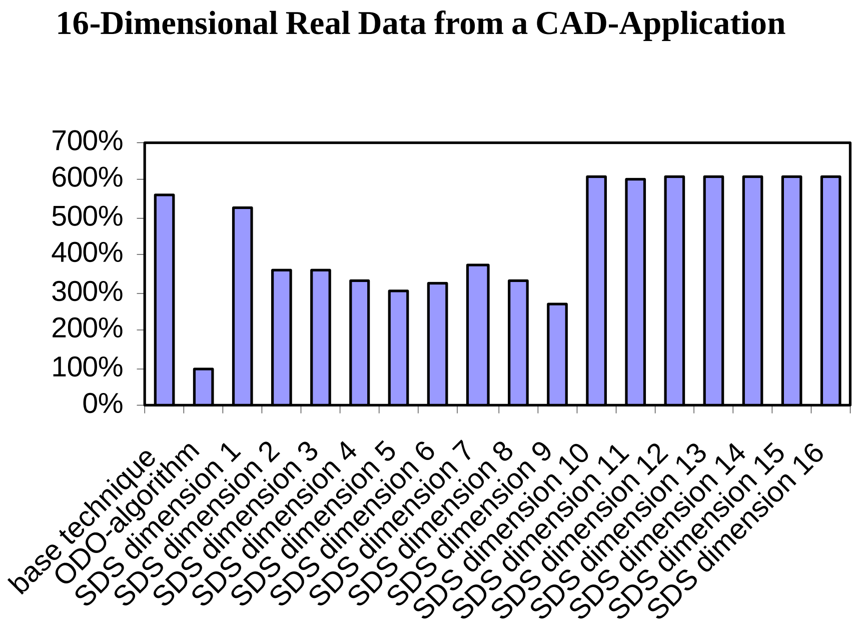
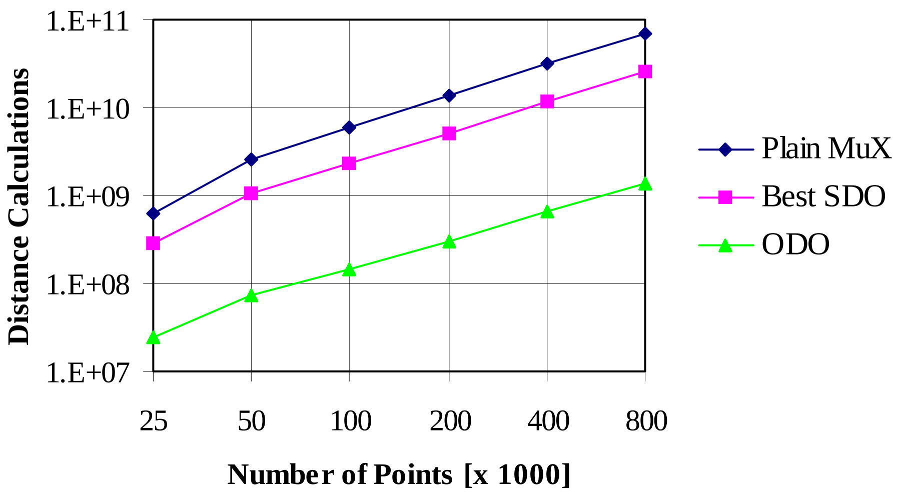


Figure 98: Experimental Results for MuX: Plain Basic Technique, ODO and SDO



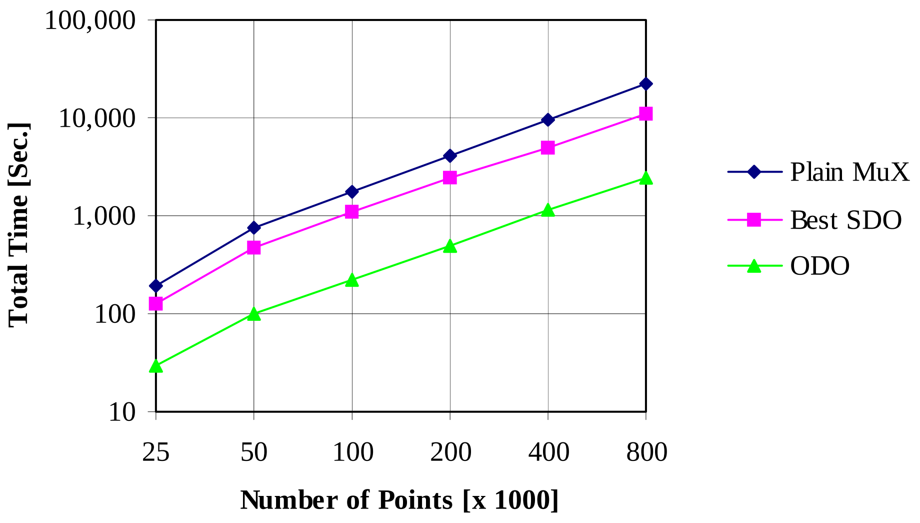
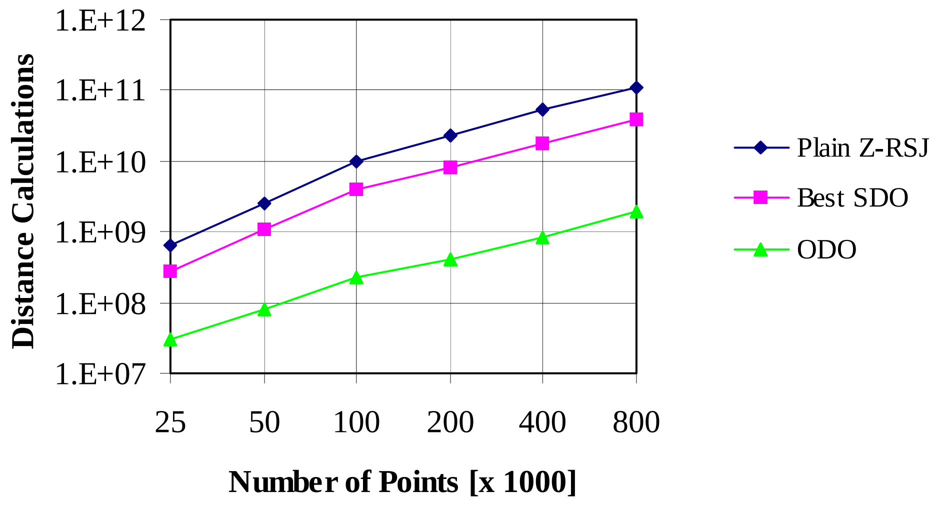


Figure 99: Experimental Results for MuX: Uniformly Distributed 8-D Data



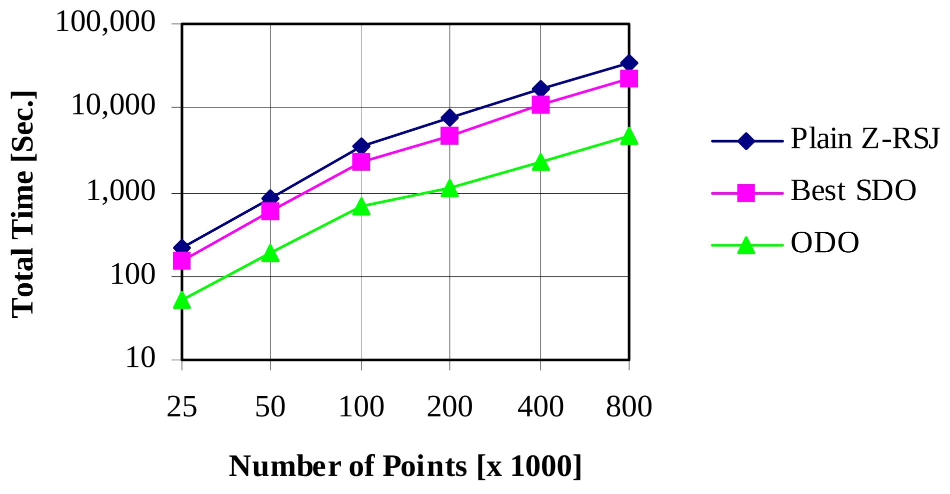


Figure 100: Experimental Results for Z-RSJ: Uniformly Distributed 8-D Data

a sustained transfer rate of about 9 MB/s and an average read access time of 8.9ms with an average latency time of 5.6ms.

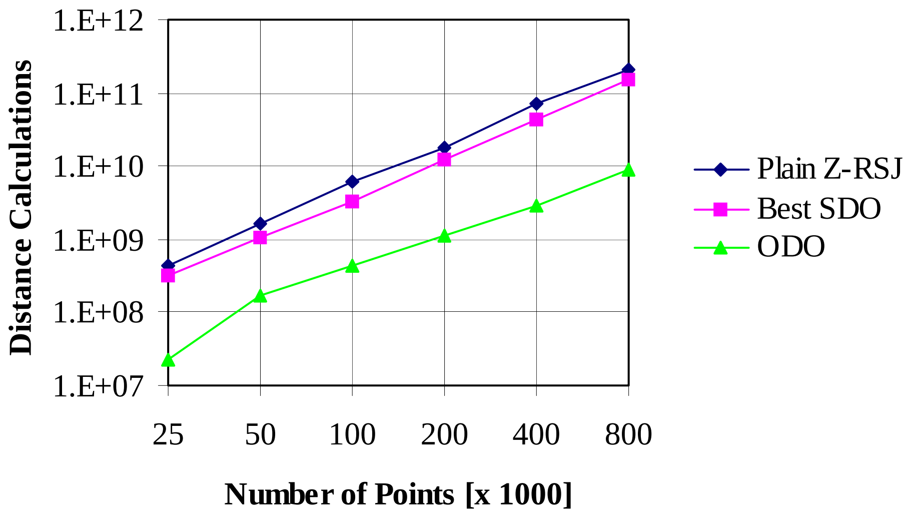
Our 8-dimensional synthetic data sets consisted of up to 800,000 uniformly distributed points in the unit hypercube. Our real-world data set is a CAD database with 16-dimensional feature vectors extracted from geometrical parts.

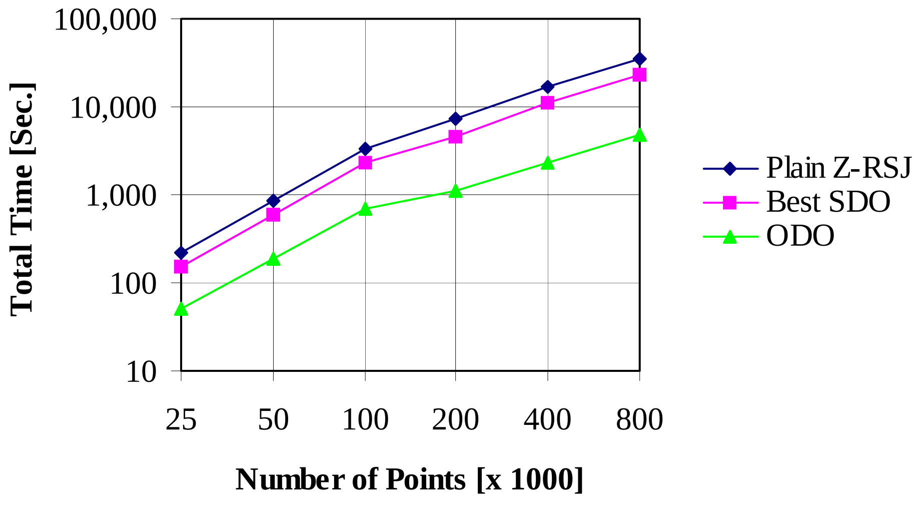
The Euclidean distance was used for the similarity join. We determined the distance parameter ε for each data set such that it is suitable for clustering following the selection criteria proposed in [SEKX 98] obtaining a reasonable selectivity.

Figure 98 shows our experiments comparing the overall runtime i.e. I/O- and CPU-time for the plain basic technique MuX either to MuX with integrated ODO or integrated simple dimension-order (SDO) for all possible start dimensions. The left diagram shows the results for uniformly distributed 8-dimensional artificial data while the right diagram shows results for 16-dimensional real data from a CAD-application. The database contained 100,000 points in each case. The SDO-algorithm depends heavily on the shape of the page regions i.e on the split algorithm used by the index employed by the basic technique.

For uniformly distributed artificial data the loading procedure used by MuX treats all dimensions equally and therefore the results for the simple dimension-sweep algorithm are roughly the same for all start dimensions. ODO performs 6-times faster than plain MuX and 4 times faster than the best SDO while SDO itself is about 1.5 times faster than plain MuX. Note again that our algorithm chooses the most suitable dimension *for each pair of partitions*. Therefore, it is possible that ODO clearly outperforms the simple dimension sweeping technique (SDO) even for its best dimension.

For our real data set SDO shows varying performance with varying start dimension. We can even observe that for some start dimensions the overhead of SDO outweighs the savings and overall performance degrades slightly compared to the plain basic technique. This shows that it can be unfeasible to apply dimension-ordering for one fixed start dimension. MuX with integrated ODO is about 5.5 times faster for the real data set





igure 101: Experimental Results for Z-RSJ: 16-D Real Data from a CAD-Application

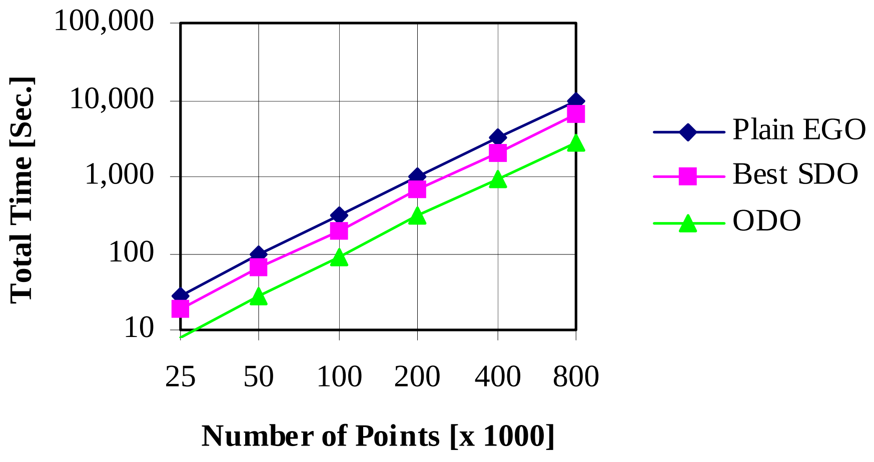


Figure 102: Experimental Results for EGO (16d CAD data)

than plain MuX while it is still 3 times faster than the SDO with the best performance, however it is more than 6 times faster than SDO with the worst performance.

Figure 99 shows all results for the uniformly distributed artificial data set for varying database size, including the diagram with distance calculations. We can see that the plain MuX performs up to 50 times more distance calculations than with ODO. The diagrams for the real data set are left out due to space ristrictions.

In order to show that the optimal dimension-ordering algorithm can be implemented on top of other basic techniques as well, we show the results for the *Z-order-RSJ* with uniformly distributed data in figure 100. Z-order-RSJ without ODO is up to 7 times slower than with integrated ODO and performs up to 58 times more distance calculations. The results for *Z-order-RSJ* with real data are shown in figure 101. We can see a speedup factor of 1.5 for SDO vs. plain *Z-order-RSJ* with respect to total time and of 1.8 with respect to distance calcualtions. ODO performs 3.5 times faster and performs 17 times fewer distance calculations than SDO while it performs 5.5 times faster and up to 25 times less distance calculations than SDO.

EGO was used to demonstrate integration of ODO with a basic technique that does not use a preconstructed index. The results are given in figure 102 where EGO with SDO, as well as plain EGO clearly perform worse than ODO i.e. SDO is about 1.5 times faster than plain EGO, but ODO is twice as fast as SDO and outperforms plain EGO by a factor of 3.5.

11.8 Conclusions

Many different algorithms for the efficient computation of the similarity join have been proposed in the past. While most known techniques concentrate on disk I/O operations, relatively few approaches are dedicated to the reduction of the computational cost, although the similarity join is clearly CPU bound. In this chapter, we have proposed the Optimal Dimension Order, a generic technique which can be applied on top of many different basic algorithms for the similarity join to reduce the computational cost. The general idea is to avoid and accelerate the distance calculations between points by sorting the points according to a specific dimension. The most suitable dimension for each pair of pages is carefully chosen by a probability model. Our experimental evaluation shows substantial performance improvements for several basic join algorithms such as the multipage index, the ε-grid-order and the breadth-first-R-tree join.

Chapter 12
Conclusions

Both data mining and query processing on feature data sets are emerging domains of research. The objective of our thesis was to bridge the gap between these two domains. With the similarity join we have proposed a powerful database primitive to support data analysis and data mining on large databases. The material presented in this thesis has contributed to this goal both theoretically as well as practically. Our work had two focusses: At the one hand, we demonstrated the implementation of important basic algorithms for data mining on top of the similarity join. At the other hand, we proposed new algorithms as well as a cost model and optimization techniques for the similarity join. Our intention was to promote the similarity join in the research community and to achieve awareness of the similarity join as a powerful database primitive to support various prospective applications.

12.1 Contributions

In the literature, there are several different definitions for join operations involving similarity. Therefore, at the beginning of our thesis, we propose a general definition for the similarity join and give a taxonomy of the different similarity join operations. We distinguish between approaches where the join predicate is a range search and approaches where the join condition bases on the *k*-nearest neighbor principle. This taxonomy gives our thesis the main structure. In this summarization, we group together applications and algorithms.

12.1.1 Applications of the Similarity Join

There are many applications for which it is quite straightforward to use the similarity join as a database primitive. In chapter 4, however, we have shown how to transform density based clustering algorithms such that they can use the similarity join as a database primitive. In particular, we demonstrate such a transformation for the density based clustering method DBSCAN and for a density based analysis method for the hierarchical cluster structure of a data set called OPTICS.

For these two methods, the transformation is particularly challenging because in contrast to some other methods presented in this thesis, DBSCAN and OPTICS in their original definitions enforce a certain order in which similarity queries are evaluated. Therefore it is not straightforward to replace the similarity queries by the similarity join. We proposed two methods of transformation: The first, called semantic rewriting first transforms the clustering algorithm semantically to ensure that it is independent of the order in which join pairs are generated. This is done by assigning cluster IDs tentatively, and with a complex action table which handles inconsistent tentative results. The other technique is called join result materialization. The join result is predetermined prior to the run of the clustering algorithm and similarity queries are efficiently answered by lookups to the materialized join result.

We can show for both techniques that the result of the clustering algorithms is identical to that of the original algorithms. Our experimental evaluation yields performance advances of up to a factor of 50 by our techniques.

To demonstrate that using the similarity join is not always complex we also give in chapter 5 a couple of application algorithms for which this transformation is straightforward. The applications presented here are robust similarity search in sequence data where the join leads in particular to robustness with respect to noise and scaling. We also present a few generalizations of this technique to similarity of multidimensional sequences (i.e. raster or voxel data) and to partial similarity. Also presented are applications like catalogue matching and duplicate detection. All these algorithms are based on the database primitive of the *distance range join*.

Chapter 9 is dedicated to the applications of the *k*-Nearest Neighbor Join (*k*-nnj) which combines each point of a point set *R* with its *k* nearest neighbors in another point set *S*. Many standard tasks of data mining evaluate *k*-nearest neighbor queries for a large number of query points. Examples are clustering algorithms such as *k*-means, *k*-medoid and the nearest neighbor method, but also data cleansing and other pre- and postprocessing techniques e.g. when sampling plays a role in data mining. 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. Then, the *k*-distance diagram is a technique for a suitable parameter selection for data mining. In the core step, i.e. data mining, many algorithms such as clustering and classification are based on *k*-nn queries. 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. In chapter 9, we show how some of these standard algorithms can be based on top of the *k*-nearest neighbor join.

12.1.2 Algorithms for the Similarity Join

We have proposed algorithms for both kinds of similarity joins, those based on the range search as well as those based on nearest neighbor search. Additionally we can distinguish our solution to the applied paradigms, i.e. whether or not they operate on multidimensional index structures.

Our most important contribution to the first group, index based join algorithms is a cost model for the distance range join which estimates the index selectivity, i.e. the number of page pairs which must be considered to compute the join result (cf. chapter 6). The index selectivity is the key factor which is responsible for I/O and CPU cost of the join algorithms. We also show how the logical page capacity of these index structures can be optimized in order to minimize CPU and I/O time.

The concept used in this cost model is the Minkowski sum which is here modified to estimate the number of page pairs from the corresponding index structures which have to be considered. In contrast to usual similarity search, the concept of the Minkowski sum must be applied twice for the similarity join in order to estimate the number of page pairs which must be joined.

During this analysis, we discover a serious optimization conflict between I/O and CPU optimization. While large pages optimize the I/O, the CPU performance benefits from small pages. This results in the observation that in traditional index structures only one of these performance factors can be optimized.

To solve the conflict, we propose in chapter 7 a novel index architecture called Multipage Index (MuX). This index structure consists of large data and directory pages which are subject to I/O operations. Rather than directly storing points and directory records an these large pages, these pages accommodate a secondary search structure which is used to speed up the CPU operations. To facilitate an effective and efficient optimization, this secondary search structure has again an R-tree like structure with a directory and data pages. Thus, the page capacity of the secondary search structure can be optimized by the cost functions developed in chapter 6, however, for the CPU tradeoff.

We show that the CPU performance of MuX is similar (equal up to some small dilatational management overhead) to the CPU performance of a traditional index which is purely CPU optimized. Likewise, we show that the I/O performance resembles that of an I/O optimized traditional index. Our experimental evaluation confirms this and demonstrates the clear superiority over the traditional approaches.

The Multipage Index is also applied in chapter 10 to implement the *k*-nearest neighbor join. Join algorithms on the nearest neighbor principle are more difficult to implement as it is not immediately decidable which page pairs must be formed to compute the join result. Therefore, much more strategic decisions must be made to determine a suitable order for the page accesses and for the join between page pairs. We develop two strategies, a *page access strategy* and a *processing strategy* for these purposes.

For join processing without support of any precomputed index structure, we propose in chapter 8 the ε grid order, a sort order which is founded on a virtual grid partition of the data space. This method is based on the observation that for the distance range join with a given distance parameter ε, a grid partition with a grid distance of ε is an effective means to reduce the search space for join partners of a point *p*. Due to the *curse of dimensionality,* however, the number of grid cells in which potentially joining points are contained explodes with the data space dimension ($O(3^d)$ cells). To avoid considering the grid cells one by one, we introduce the grid partition only in a virtual way as the basis of a particular sort order, the ε grid order, which orders points according to grid cell containment. The ε grid order is used as ordering criterion in an external memory sort operator. Later, the ε grid order supports effective and efficient algorithms for CPU and I/O processing, particularly for large data sets which cannot be joined in main memory.

Our last contribution to algorithms for similarity joins is a generic technique to accelerate and partially avoid the finalizing distance computations when computing the similarity join. It can be applied on top of all join algorithms proposed in this thesis and also on most algorithms described in the related work chapter. In spite of all optimization efforts, most of these algorithms are clearly CPU bound, and the most important cost factor are the finalizing distance calculations between the feature vectors. Our optimization technique accelerates these distance calculations by selecting the dimension with the highest selectivity and sorting the points along this optimal dimension. Therefore, we call this technique the optimal dimension order. To select an optimal dimension our technique considers the regions which are assigned to the considered partitions. It is not restricted to index based processing techniques but can also be applied on top of hashing based methods or grid based approaches such as the size separation spatial join, the ε-kdB-tree or our ε Grid Order.

To summarize and overview the contributions made in this thesis cf. figure 103 which gives again a small taxonomy of the similarity join and classifies our contributions according to the two categories *applications* and *algorithms* which are, in turn categorized

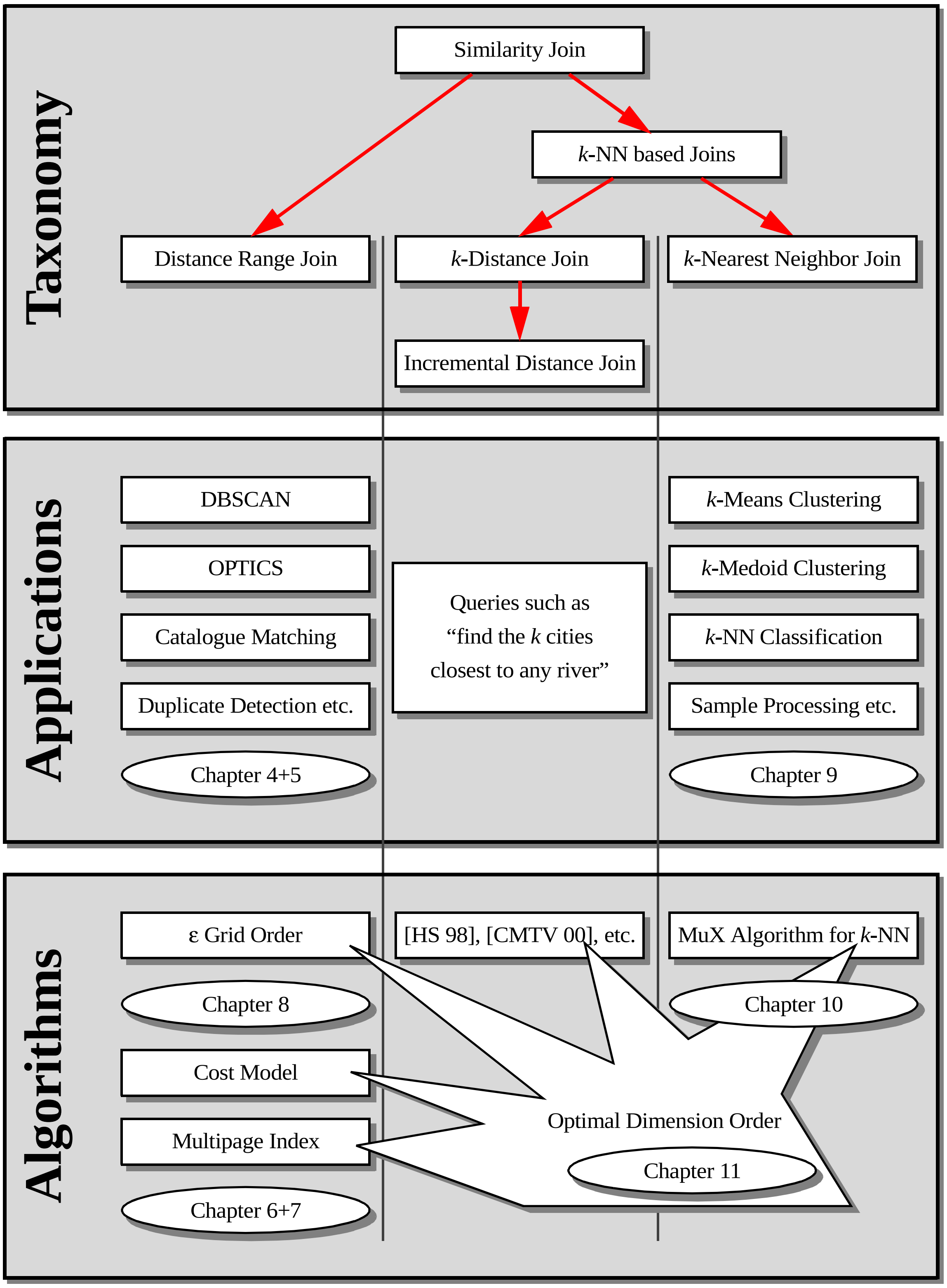


Figure 103: Taxonomy and Overview of the Thesis

into index based and non index based techniques. Our contributions from this thesis are marked with ovals and with links to the corresponding chapters.

12.2 Future Work

Besides complementing and even strengthening our effort in the successful areas of database primitives for similarity search and data mining, we have identified several research directions into which we plan to extend our future work. This includes opening new, innovative application domains with new challenging research potential, a general framework for the development of similarity search systems, and database technology centered research.

12.2.1 New Data Mining Tasks

Due to a complex analysis of the complete data set data mining algorithms are often of a much higher computational complexity than traditional database applications. This has mainly prevented data mining tasks from being strongly integrated into the database environment. Our method of identifying very powerful database primitives such as the similarity join (or as another example, the convex hull operation, cf. [BK 01b]), data mining algorithms may become standard database applications like others. The consequence is a much tighter integration of data mining in the information infrastructure of an enterprise which yields many advantages.

Due to the dramatic increase of performance by our approaches, it will be possible to implement quite new kinds of data mining algorithms which detect new kinds of patterns. An interesting, new challenge is subspace clustering [AGGR 98]. Typically, not all attributes of feature vectors carry information which is useful in data mining. Other attributes may be noisy and should be ignored as they deteriorate the data mining result. Identifying the relevant attributes, however, is a difficult task. Subspace clustering combines the two tasks of selecting attributes and finding clusters. Subspaces, i.e. groups of attributes, are determined such that maximal, distinguishable clusters can be found. First algorithms, however, suffer from the high computational cost. Basing them on top of powerful database primitives could open the potential to make this computation feasible.

Another approach could be to make current data mining much more interactive. The current process is to select parameters, to run a data mining algorithm and, finally, to visualize the result of the algorithm. Our dramatic performance gains could open the potential to make this process so fast that a user may change parameters and immediately see the resulting changes in the visualization. Here, it could be beneficial to apply new concepts in database systems which evaluate queries *approximately* [CP 00] or produce first results in an early stage of processing.

12.2.2 New Application Domains

We have identified three areas of new applications which have only superficially been considered as database applications, in spite of vast data amounts and clear relations to similarity search and data mining.

Electronic Commerce

Many stages in electronic commerce require concepts from similarity search and data mining. In the early stage, marketing, it is essential to perform a customer segmentation, a typical data mining application, to make directed offers to which the customers are maximum responsive.

In the core area of e-commerce, booking and sales systems, users specify their needs in an inexact way. For instance, they have initial ideas about features their product should have and the corresponding price. Then, in an interactive process the system has to find out which of the initial features are how relevant to the specific customer and will find in this way a product which fits best the users notions.

After commitment of the trade the final stage is marketing for additional products. A good (human) salesman develops a sense what additional high-revenue products could be of interest for the customer, based on his experience with previous customers purchasing similar products. This behavior could also be imitated using concepts of similarity search and data mining.

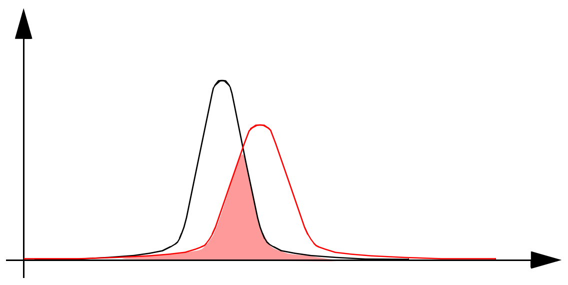


Figure 104: Characteristic of Fuzzy Biometry Data

For applications mentioned above, it is necessary to extend known concepts and to develop new concepts. Classical similarity search takes the basic assumption that the similarity measure is a parameter given by the user. Therefore, weights for the individual features are assumed to be known. Here, we are rather facing the situation that the measures are initially completely unknown and develop during the selection process. Instead of assuming a uniform importance of the features, and ranking the products according to the Euclidean distance, the user should be provided with a selection of products that reveals different weighting of the features. A selection with varying weights of features essentially corresponds to the *convex hull* of a subset of the data [BK 01b]. The products which are further investigated by the customers can be used for a *relevance feedback*, to determine a suitable similarity measure. A first approach to use relevance feedback for this purpose is the MindReader [ISF 98] which determines a quadratic form distance measure [BKS 01]. For electronic commerce, we identify two additional requirements. First, the relevance feedback should be extended to a multi modal model to take into account that users in general do not only like one single "ideal" product but often have a few alternatives in their minds which are not clearly separated in their notion. The second requirement is a seamless integration of the concepts of similarity search, convex hull, and relevance feedback.

Biometry Databases

Biometry applications store human data such as features from face images, fingerprints, the hand geometry, the retina, or even voice and handwriting for identification and au-

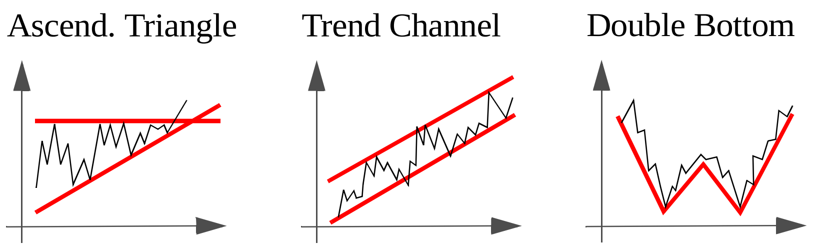


Figure 105: Chart Analysis

thentication purposes. For many applications, a high number of feature vectors are stored and due to the inexactness of the measuring devices, similarity search is needed for identification.

In contrast to traditional similarity search, the uncertainty of the individual features is not uniform among all features and even for a single feature, the uncertainty is not uniform among all stored vectors. Instead, each feature of each feature vector is associated with an individual uncertainty which is stored in the database. With this concept, it is possible to capture problems introduced by different environments and technical devices. The uncertainty of facial features such as the eye distance, for instance, depends on the angle between camera and person, and also on the illumination. The error can be assumed to be taken from a Gaussian distribution, so the uncertainty is measured in terms of a standard deviation.

The challenge here is to develop specialized index structures to store feature vectors with individual uncertainty vectors and query processing algorithms that facilitate a fast and efficient evaluation of queries such as

- determine all persons that match the query person with a probability of at least 10%
- determine the person that matches the query person with maximum probability.

Technical Analysis of Share Price

One of the classical applications of similarity search and data mining is clearly the analysis of time sequences [ALSS 95] such as share price analysis. Various similarity measures have been proposed. For practical analysis, however, quite different concepts are used, such as indicators, i.e. mathematical formulas derived from the time sequence that generate trading signals (*buy, sell*). Another concept for the analysis of a time sequence is the chart analysis (cf. figure 105) which detects typical formations in the share price which are known to indicate a certain direction of the future price. Examples are triangles, trend channels, lines of support and resistance, W-formations (double bottom), head-and-shoulder-formations etc.

For effectively supporting users in their share price analysis, the requirement is to integrate both indicators as well as formation analysis into search systems and into sequence mining algorithms. Suitable index structures and query processing techniques must be developed to facilitate a fast and efficient analysis.

12.2.3 A Framework for the Development of Similarity Search Systems

The problem of similarity search should also be considered in a more general way. Currently, similarity search methods are tailored to specific application domains, and only very basic techniques such as the nearest neighbor search solve general problems that arise in virtually all similarity search systems.

The main difficulty in the development of similarity measures is the communication between domain experts and similarity experts, as the similarity search involves a deep knowledge of the scientific concepts of the domain. Vice versa, domain experts can hardly imagine what a similarity search system may achieve and what concepts must be applied for this purpose.

Our idea is to alleviate this problem by a common framework that bundles concepts which are often applied in similarity search in a toolbox. This toolbox could contain various methods of feature extraction such as histograms, fourier transformation, and moment invariants, and various search methods such as similarity search, query decomposition for making the search robust, search for partial similarity, etc.

This toolbox could be complemented with visualization systems, evaluation methods and the above mentioned data mining techniques such as subspace clustering, convex hull and mind reader which may be used to determine whether the resulting feature transformation is adequate.

References

[ABKS 99] Ankerst M., Breunig M. M., Kriegel H.-P., Sander J.: "*OPTICS*: *Ordering Points To Identify the Clustering Structure*", Proc. ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'99), Philadelphia, PA, 1999, pp. 49-60.

[AFS 93] Agrawal R., Faloutsos C., Swami A.: *'Efficient similarity search in sequence databases'*, Proc. 4th Int. Conf. on Foundations of Data Organization and Algorithms, 1993, in: Lecture Notes in Computer Science, Springer, Vol. 730, pp. 69-84.

[AGGR 98] Agrawal R., Gehrke J., Gunopulos D., Raghavan P.: 'Automatic Subspace Clustering of High-Dimensional Data for Data Mining Applications', Proc. ACM SIGMOD Int. Conf. on Management of Data, Seattle, WA, 1998, pp. 94-105.

[AGMM 90] Altschul S. F., Gish W., Miller W., Myers E. W., Lipman D. J.: *'A Basic Local Alignment Search Tool'*, Journal of Molecular Biology, Vol. 215, No. 3, 1990, pp. 403-410.

[ALSS 95] Agrawal R., Lin K., Shawney H., Shim K.: *'Fast Similarity Search in the Presence of Noise, Scaling, and Translation in Time-Series Databases'*, Proc. 21st Conf. on Very Large Data Bases, 1995, pp. 490-501.

[AMN 95] Arya S., Mount D. M., Narayan O.: '*Accounting for Boundary Effects in Nearest Neighbor Searching*', Proc. 11th Symp. on Computational Geometry, Vancouver, Canada, 1995, pp. 336-344.

[APR+ 98] Arge L., Procopiuc O., Ramaswamy S., Suel T., Vitter J. S.: *'Scalable Sweeping-Based Spatial Join'*, Proc. 24th Int. Conf. on Very Large Data Bases, New York, NY, 1998, pp. 570-581.

[Ary 95] Arya S.: ‘*Nearest Neighbor Searching and Applications’*, Ph.D. thesis, University of Maryland, College Park, MD, 1995.

[AS 83] Abel D. J., Smith J. L.: *‘A Data Structure and Algorithm Based on a Linear Key for a Rectangle Retrieval Problem’*, Computer Vision 24, 1983, pp. 1-13.

[AS 91] Aref W. G., Samet H.: *‘Optimization Strategies for Spatial Query Processing’*, Proc. 17th Int. Conf. on Very Large Data Bases (VLDB’91), Barcelona, Catalonia, 1991, pp. 81-90.

[AS 94] Agrawal R., Srikant R.: *‘Fast Algorithms for Mining Association Rules in Large Databases*’, Proc. 20th Int. Conf. on Very Large Data Bases (VLDB’94), Santiago de Chile, Chile, 1994, pp. 487-499.

[AY 00] Aggarwal C. C., Yu P. S.: *‘Finding Generalized Projected Clusters In High Dimensional Spaces*’, Proc. ACM SIGMOD Int. Conf. on Management of Data (SIGMOD 2000), Dallas, TX, 2000, pp. 70-81.

[BA 96] Brachmann R. and Anand T.: ‘*The Process of Knowledge Discovery in Databases: A Human Centered Approach*”, in: Advances in Knowledge Discovery and Data Mining, AAAI Press, Menlo Park, 1996, pp.37-58.

[BBB+ 97] Berchtold S., Böhm C., Braunmüller B., Keim D. A., Kriegel H.-P.: *‘Fast Parallel Similarity Search in Multimedia Databases’*, Proc. ACM SIGMOD Int. Conf. on Management of Data, 1997, Tucson, AZ, pp. 1-12, SIGMOD BEST PAPER AWARD.

[BBBK 00] Böhm C., Braunmüller B., Breunig M. M., Kriegel H.-P.: *‘High Performance Clustering Based on the Similarity Join*’, 9th Int. Conf. on Information and Knowledge Management (CIKM 2000), Washington DC, 2000, pp. 298-313.

[BBBK 01] Böhm C., Braunmüller B., Breunig M. M., Kriegel H.-P.: *‘The Similarity Join: A Database Primitive for High-Performance Data Mining’*, submitted for publication.

[BBJ+ 00] Berchtold S., Böhm C., Jagadish H. V., Kriegel H.-P., Sander J.: *‘Independent Quantization: An Index Compression Technique for High-Dimensional Data Spaces*’, Proc. Int. Conf. on Data Engineering (ICDE 2000), San Diego, CA, 2000, pp. 577-588.

[BBK 98] Berchtold S., Böhm C., Kriegel H.-P.: *‘Improving the Query Performance of High-Dimensional Index Structures Using Bulk-Load Operations’*, 6th. Int. Conf. on Extending Database Technology, Valencia, Spain, 1998, pp. 216-230.

[BBK 98b] Berchtold S., Böhm C., Kriegel H.-P.: *‘The Pyramid-Technique: Towards indexing beyond the Curse of Dimensionality’*, Proc. ACM SIGMOD Int. Conf. on Management of Data, Seattle, WA, 1998, pp. 142-153.

[BBK+ 00] Berchtold S., Böhm C., Keim D. A., Kriegel H.-P., Xu X.: *‘Optimal Multidimensional Query Processing Using Tree Striping’*, Proc. Int. Conf. on Data Warehousing and Knowledge Discovery (DaWaK 2000), Greenwich, U.K., 2000, pp. 244-257.

[BBK 00] Böhm C., Braunmüller B., Kriegel H.-P.: ‘*The Pruning Power: A Theory of Scheduling Strategies for Multiple k-Nearest Neighbor Queries*’, Proc. Int. Conf. on Data Warehousing and Knowledge Discovery (DaWaK 2000), Greenwich, U.K., 2000, pp. 372-381.

[BBK 01] Böhm C., Berchtold S., Keim D. A.: ‘*Searching in High-dimensional Spaces: Index Structures for Improving the Performance of Multimedia Databases*’, to appear in ACM Computing Surveys, 2001.

[BBKK 97] Berchtold S., Böhm C., Keim D. A., Kriegel H.-P.: ‘*A Cost Model For Nearest Neighbor Search in High-Dimensional Data Space*’, ACM PODS Symp. on Principles of Database Systems, Tucson, AZ, 1997, pp. 78-86.

[BBKK 01] Böhm C., Braunmüller B., Krebs F., Kriegel H.-P.: *‘Epsilon Grid Order: An Algorithm for the Similarity Join on Massive High-Dimensional Data’*, Proc. ACM SIGMOD Int. Conf. on Management of Data, Santa Barbara, CA, 2001.

[BBKM 00] Böhm C., Berchtold S., Kriegel H.-P., Michel U.: ‘*Multidimensional Index Structures in Relational Databases*’, in: Journal of Intelligent Information Systems (JIIS), Vol. 15, No. 1, 2000, pp. 51-70.

[BBKS 00] Böhm C., Braunmüller B., Kriegel H.-P., Schubert M.: ‘*Efficient Similarity Search in Digital Libraries*’, Proc. IEEE Int. Conf. on Advances in Digital Libraries (ADL 2000), Washington DC, 2000, pp. 193-206.

[BEK+ 98] Berchtold S., Ertl B., Keim D. A., Kriegel H.-P., Seidl T.: ‘*Fast Nearest Neighbor Search in High-Dimensional Spaces*’, Proc. 14th Int. Conf. on Data Engineering, Orlando, FL, 1998, pp. 209-218.

[BEKS 00] Braunmüller B., Ester M., Kriegel H.-P., Sander J.: ‘*Efficiently Supporting Multiple Similarity Queries for Mining in Metric Databases’*, Proc. 16th Int. Conf. on Data Engineering (ICDE 2000), San Diego, CA, 2000, pp. 256-267.

[Ben 75] Bentley J. L.: ‘*Multidimensional Search Trees Used for Associative Searching*’, in: Communications of the ACM, Vol. 18, No. 9, 1975, pp. 509-517.

[Ben 79] Bentley J. L.: *‘Multidimensional Binary Search in Database Applications’*, IEEE Trans. Software Engineering, Vol. 4, No. 5, 1979, pp. 397-409.

[Ber 97] Berchtold S.: ‘*Geometry based search of similar parts*’, (in German), Ph.D. thesis, University of Munich, 1997.

[BF 95] Belussi A., Faloutsos C.: *'Estimating the Selectivity of Spatial Queries Using the 'Correlation' Fractal Dimension'*, Proc. 21th Int. Conf. on Very Large Data Bases (VLDB'95), Zurich, Switzerland, 1995, pp. 299-310.

[BFR 98] Bradley P. S., Fayyad U., Reina C.: '*Scaling Clustering Algorithms to Large Databases*', Proc. 4th Int. Conf. on Knowledge Discovery and Data Mining (KDD'98), New York, NY, AAAI Press, 1998, pp. 9-15.

[BGRS 99] Beyer K., Goldstein J., Ramakrishnan R., Shaft U..: *'When Is "Nearest Neighbor" Meaningful?'*, Proc. Int. Conf. on Database Theory (ICDT'99), Jerusalem, Israel, 1999, pp. 217-235.

[BHF 93] Becker L., Hinrichs K., Finke U.: *'A New Algorithm for Computing Joins with Grid Files'*, Proc. Int. Conf. on Data Engineering, Vienna, Austria, 1993, pp. 190-197.

[Big 89] Biggs N. L.: '*Discrete Mathematics*', Oxford Science Publications, Clarendon Press-Oxford, 1989, pp. 172-176.

[BJK 98] Berchtold S., Jagadish H. V., Ross K.: '*Independence Diagrams: A Technique for Visual Data Mining*', Proc. 4th Int. Conf. on Knowledge Discovery and Data Mining (KDD'98), New York, NY, 1998, pp. 139-143.

[BK 97] Berchtold S., Kriegel H.-P.: '*S3: Similarity Search in CAD Database Systems*', Proc. ACM SIGMOD Int. Conf. on Management of Data, Tucson, AZ, 1997, pp. 564-567.

[BK 99] Böhm C., Kriegel H.-P.: '*Efficient Bulk Loading of Large High-Dimensional Indexes*', Proc. Int. Conf. on Data Warehousing and Knowledge Discovery (DaWaK'99), Forence, Italy, 1999, pp. 251-260.

[BK 00] Böhm C., Kriegel H.-P.: '*Dynamically Optimizing High-Dimensional Index Structures*', Proc. Int. Conf. on Extending Database Technology (EDBT 2000), Konstanz, Germany, 2000, pp. 36-50.

[BK 01a] Böhm C., Kriegel H.-P.: *'A Cost Model and Index Architecture for the Similarity Join'*, Proc. 17th Int. Conf. on Data Engineering (ICDE 2001), Heidelberg, Germany, 2001, pp. 411-420.

[BK 01b] Böhm C., Kriegel H.-P.: *'Determining the Convex Hull in Large Multidimensional Databases'*, Proc. Int. Conf. on Data Warehousing and Knowledge Discovery (DaWaK 2001), Munich, Germany, 2001.

[BK 01c] Böhm C., Kriegel H.-P.: *'Optimale Dimensionswahl bei der Bearbeitung des Similarity Join'*, (in German), conditionally accepted in: Informatik: Forschung und Entwicklung, 2001.

[BKK 96] Berchtold S., Keim D. A., Kriegel H.-P.: '*The X-Tree: An Index Structure for High-Dimensional Data*', Proc. 22nd Conf. on Very Large Data Bases, Bombay, India, 1996, pp. 28-39.

[BKK 97] Berchtold S., Keim D. A., Kriegel H.-P.: '*Using Extended Feature Objects for Partial Similarity Retrieval*', VLDB Journal, Vol. 6, No. 4, 1997, pp. 333-348.

[BKK 01a] Böhm C., Krebs F., Kriegel H.-P.: *'The k-Nearest Neighbor Join: Turbo Charging the KDD Process'*, submitted for publication.

[BKK 01b] Böhm C., Krebs F., Kriegel H.-P.: *'Optimal Dimension Sweeping: A Generic Technique for the Similarity Join'*, submitted for publication.

[BKKS 01] Breunig M. M., Kriegel H.-P., Kröger P., Sander J.: *'Data Bubbles: Quality Preserving Performance Boosting for Hierarchical Clustering'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, 2001.

[BKNS 99] Breunig M. M., Kriegel H.-P., Ng R. T., Sander J.: '*OPTICS-OF: Identifying Local Outliers*', Proc. 3rd European Conf. on Principles of Data Mining and Knowledge Discovery (PKDD'99), Prague, Czech Republic, 1999, in: Lecture Notes in Computer Science, Springer, Vol. 1704, 1999, pp. 262-270.

[BKNS 00] Breunig M. M., Kriegel H.-P., Ng R. T., Sander J.: '*LOF: Identifying Density-Based Local Outliers*', Proc. ACM SIGMOD Int. Conf. on Management of Data (SIGMOD 2000), Dallas, TX, 2000, pp. 93-104.

[BKS 93] Brinkhoff T., Kriegel H.-P., Seeger B.: '*Efficient Processing of Spatial Joins Using R-trees*', Proc. ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'93), Washington DC, 1993, pp. 237-246.

[BKS 96] Brinkhoff T., Kriegel H.-P., Seeger B.: '*Parallel Processing of Spatial Joins Using R-trees*', Proc. 12th Int. Conf. on Data Engineering (ICDE'96), New Orleans, LA, 1996, pp. 258-265.

[BKS 00] Breunig M. M., Kriegel H.-P., Sander J.: '*Fast Hierarchical Clustering Based on Compressed Data and OPTICS*', Proc. 4th European Conf. on Principles and Practice of Knowledge Discovery in Databases (PKDD 2000), Lyon, France, 2000, pp 232-242.

[BKSS 90] Beckmann N., Kriegel H.-P., Schneider R., Seeger B.: *'The R*-tree: An Efficient and Robust Access Method for Points and Rectangles'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, Atlantic City, NJ, 1990, pp. 322-331.

[BM 77] Bayer R., McCreight E. M.: '*Organization and Maintenance of Large Ordered Indices'*, Acta Informatica, Vol. 1, No. 3, 1977, pp. 173-189.

[BO 97] Bozkaya T., Özsoyoglu M.: *'Distance-Based Indexing for High-Dimensional Metric Spaces'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, Tucson, AZ, 1997, pp. 357-368.

[Böh 98] Böhm C.: '*Efficiently Indexing High Dimensional Data Spaces*', PhD. Thesis, University of Munich, 1998.

[Böh 01a] Böhm C.: *‘The Similarity Join: A Powerful Database Primitive for High Performance Data Mining’*, Tutorial, 17th Int. Conf. on Data Engineering (ICDE 2001), Heidelberg, Germany, 2001.

[Böh 01b] Böhm C.: *‘Database Systems Supporting Next Decade’s Applications’*, submitted proposal.

[Bra 00] Braunmüller B.: ‘*High Performance Database Mining*’, PhD. Thesis, University of Munich, 2001.

[Bre 01] Breunig M.: *´Quality Driven Database Mining’*, PhD. Thesis, University of Munich, 2001.

[Bri 95] Brin S.: ‘*Near Neighbor Search in Large Metric Spaces’*, Proc. 21st Int. Conf. on Very Large Data Bases (VLDB’95), Zurich, Switzerland, 1995, pp. 574-584.

[BSK 01] Böhm C., Seidl T., Kriegel H.-P.: *‘Adaptable Similarity Queries Using Vector Quantization’*, Proc. Int. Conf. on Data Warehousing and Knowledge Discovery (DaWaK 2001), Munich, Germany, 2001.

[BSS 00] van den Bercken J., Schneider M., Seeger B.: *‘Plug&Join: An Easy-to-Use Generic Algorithm for Efficiently Processing Equi and Non-equi Joins’*, Proc. Int. Conf. on Extending Database Technology (EDBT’00), Konstanz, Germany, 2000, pp. 495-509.

[BSW 97] van den Bercken J., Seeger B., Widmayer P.:, *‘A Generic Approach to Bulk Loading Multidimensional Index Structures’*, 23rd Conf. on Very Large Data Bases, Athens, Greece, 1997, pp. 406-415.

[CD 97] Chaudhuri S., Dayal U.: *‘Data Warehousing and OLAP for Decision Support’*, Tutorial, Proc. ACM SIGMOD Int. Conf. on Management of Data, Tucson, AZ, 1997.

[Chi 94] Chiueh T., ‘*Content-Based Image Indexing*’, Proc. 20th Int. Conf. on Very Large Data Bases (VLDB’94), Santiago de Chile, Chile, 1994, pp. 582-593.

[CH 98] Carter C. L., Hamilton H. J.: ‘*Efficient Attribute-Oriented Generalization for Knowledge Discovery from Large Databases*’, TKDE, Vol. 10, No. 2, 1998, pp. 193-208.

[CHY 96] Chen M.-S., Han J., Yu P. S.: ‘*Data Mining: An Overview from a Database Perspective*’, in: IEEE Trans. on Knowledge and Data Engineering, Vol. 8, No. 6, 1996, IEEE Computer Society Press, Los Alamitos, CA, pp. 866-883.

[Cle 79] Cleary J. G.: ‘*Analysis of an Algorithm for Finding Nearest Neighbors in Euclidean Space*’, ACM Trans. on Mathematical Software, Vol. 5, No. 2, 1979, pp. 183-192.

[CMTV 00] Corral A., Manolopoulos Y., Theodoridis Y., Vassilakopoulos M.: *'Closest Pair Queries in Spatial Databases'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, Dallas, TX, 2000, pp. 189-200.

[CMTV 01] Corral A., Manolopoulos Y., Theodoridis Y., Vassilakopoulos M.: *'Algorithms for Processing Closest Pair Queries in Spatial Databases,* submitted for publication 2001.

[Coh 00] Cohen W. W.: *'Data Integration Using Similarity Joins and a Word-Based Information Representation Language'*, in: ACM Trans. on Information Systems, Vol. 18, No. 3, 2000, pp. 288-321.

[Com 79] Comer D.: *'The Ubiquitous B-tree'*, ACM Computing Surveys, Vol. 11, No. 2, 1979, pp. 121-138.

[CPZ 97] Ciaccia P., Patella M., Zezula P.: *'M-tree: An Efficient Access Method for Similarity Search in Metric Spaces'*, Proc. 23rd Int. Conf. on Very Large Data Bases (VLDB'97), Athens, Greece, 1997, pp. 426-435.

[DH 73] Duda R. O., Hart P. E.: *'Pattern Classification and Scene Analysis'*, Wiley, New York, 1973.

[DHKP 91] Dietzfelbinger M., Hagerup T., Katajainen J., Penttonen M.: *'A Reliable Randomized Algorithm for the Closest-Pair Problem'*, in: Journal of Algorithms, Vol. 25, 1997, pp. 19-51.

[DS 82] Du H. C., Sobolewski J. S.: '*Disk allocation for cartesian product files on multiple Disk systems*', in: ACM TODS Journal of Trans. on Database Systems, 1982, pp. 82-101.

[DS 01] Dittrich J.-P., Seeger B.: *'GESS: a Scalable Similarity-Join Algorithm for Mining Large Data Sets in High Dimensional Spaces*, Proc. Int. Conf. on Knowledge Discovery in Databases (KDD'01), San Francisco, CA, 2001.

[Eas 81] Eastman C. M.: *'Optimal Bucket Size for Nearest Neighbor Searching in k-d Trees'*, Information Processing Letters Vol. 12, No. 4, 1981.

[EFKS 98] Ester M., Frommelt A., Kriegel H.-P., Sander J.: '*Algorithms for Characterization and Trend Detection in Spatial Databases*', Proc. 4th Int. Conf. on Knowledge Discovery and Data Mining (KDD'98), New York, NY, 1998, pp. 44-50.

[EKSX 96] Ester M., Kriegel H.-P., Sander J., Xu X.: '*A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise*', Proc. 2nd Int. Conf. on Knowledge Discovery and Data Mining (KDD'96), Portland, OR, AAAI Press, 1996, pp. 226-231.

[EKSX 98] Ester M., Kriegel H.-P., Sander J., Wimmer M., Xu X.: '*Incremental Clustering for Mining in a Data Warehousing Environment*', Proc. 24th Int. Conf. on Very Large Data Bases (VLDB'98), New York, NY, 1998, pp. 323-333.

[Eva 94] Evangelidis G.: *'The hB$^{\pi}$-Tree: A Concurrent and Recoverable Multi-Attribute Index Structure'*, Ph. D. thesis, Northeastern University, Boston, MA, 1994.

[Fal 85] Faloutsos C.: *'Multiattribute Hashing Using Gray Codes'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, Austin, TX, 1985, pp. 227-238.

[Fal 88] Faloutsos C.: *'Gray Codes for Partial Match and Range Queries'*, in: IEEE Trans. on Software Engineering, Vol. 14, 1988, pp. 1381-1393.

[FB 74] Finkel R, Bentley J. L. *'Quad Trees: A Data Structure for Retrieval of Composite Keys'*, in: Acta Informatica Vol. 4, No. 1, 1974, pp. 1-9.

[FB 93] Faloutsos C., Bhagwat P.: '*Declustering Using Fractals*', in: PDIS Journal of Parallel and Distributed Information Systems, 1993, pp. 18-25.

[FBF 77] Friedman J. H., Bentley J. L., Finkel R. A.: '*An Algorithm for Finding Best Matches in Logarithmic Expected Time*', ACM Trans. on Mathematical Software, Vol. 3, No. 3, September 1977, pp. 209-226.

[FBF+ 94] Faloutsos C., Barber R., Flickner M., Hafner J., Niblack W., Petkovic D., Equitz W.: *'Efficient and Effective Querying by Image Content'*, in: Journal of Intelligent Information Systems, Vol. 3, 1994, pp. 231-262.

[FG 96] Faloutsos C., Gaede V.: *'Analysis of n-Dimensional Quadtrees using the Hausdorff Fractal Dimension'*, Proc. 22th Int. Conf. on Very Large Data Bases (VLDB'96), Mumbai (Bombay), India, 1996, pp. 40-50.

[FK 94] Faloutsos C., Kamel I.: '*Beyond Uniformity and Independence: Analysis of R-trees Using the Concept of Fractal Dimension'*, Proc. 13th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, Minneapolis, MN, 1994, pp. 4-13.

[FL 95] Faloutsos C., Lin K.-I.: *'FastMap: A Fast Algorithm for Indexing, Data-Mining and Visualization of Traditional and Multimedia Data'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, San Jose, CA, 1995, pp. 163-174.

[FR 89] Faloutsos C., Roseman S.: *'Fractals for Secondary Key Retrieval'*, Proc. 8th ACM SIGACT/SIGMOD Symp. on Principles of Database Systems, 1989, pp. 247-252.

[Fre 87] Freeston M.: *'The BANG file: A new kind of grid file'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, San Francisco, CA, 1987, pp. 260-269.

[FPM 91] Frawley W. J., Piatetsky-Shapiro G., Matheus C. J.: '*Knowledge Discovery in Databases: An Overview*', in: Knowledge Discovery in Databases, AAAI Press, 1991, pp. 1-27.

[FP 97] Fawcett T., Provost F.: ‘*Adaptive Fraud Detection*’, in: Data Mining and Knowledge Discovery Journal, Kluwer Academic Publishers, Vol. 1, No. 3, 1997, pp. 291-316.

[FPS 96] Fayyad U., Piatetsky-Shapiro G., Smyth P.: ‘*Knowledge Discovery and Data Mining: Towards a Unifying Framework*’, Proc. 2nd Int. Conf. on Knowledge Discovery and Data Mining (KDD’96), Portland, OR, AAAI Press, Menlo Park, CA, 1996, pp. 82-88.

[FPS 96a] Fayyad U., Piatetsky-Shapiro G., Smyth P.: ‘*From Data Mining to Knowledge Discovery: An Overview*’, in: Advances in Knowledge Discovery and Data Mining, AAAI Press, Menlo Park, 1996, pp. 1-34.

[FRM 94] Faloutsos C., Ranganathan M., Manolopoulos Y.: *‘Fast Subsequence Matching in Time-Series Databases’,* Proc. ACM SIGMOD Int. Conf. on Management of Data, Minneapolis, MN, 1994, pp. 419-429.

[FSR 87] Faloutsos C., Sellis T., Roussopoulos N.: ‘*Analysis of Object-Oriented Spatial Access Methods*’, Proc. ACM SIGMOD Int. Conf. on Management of Data, San Francisco, CA, 1987, pp. 426-439.

[FSTT 00] Faloutsos C., Seeger B., Traina A., Traina Jr. C.: *‘Spatial Join Selectivity Using Power Laws’*, Proc. ACM SIGMOD Int. Conf. on Management of Data, Dallas, TX, 2000, pp. 177-188.

[Fuk 90] Fukunaga K.: *‘Introduction to Statistical Pattern Recognition’*, 2nd edition, Academic Press, 1990.

[Gae 95] Gaede V.: *‘Optimal Redundancy in Spatial Database Systems’,* Proc. 4th International Symp. on Advances in Spatial Databases (SSD’95), Portland, ME, 1995, in: Lecture Notes in Computer Science, Vol. 951, pp. 96-116.

[Gar 82] Gargantini I.: *‘An Effective Way to Represent Quadtrees’,* in: Comm. of the ACM, Vol. 25, No. 12, 1982, pp. 905-910.

[GG 98] Gaede V., Günther O.: *‘Multidimensional Access Methods’*, in: ACM Computing Surveys, Vol. 30, No. 2, 1998, pp. 170-231.

[GL 89] Golub G. H., van Loan C. F.: *‘Matric Computations’*, 2nd edition, John Hopkins Univerity Press, Baltimore, 1989.

[GM 93] Gary J. E., Mehrotra R.: *‘Similar Shape Retrieval using a Structural Feature Index’,* Information Systems, Vol. 18, No. 7, 1993, pp. 525-537.

[Gre 89] Greene D.: *‘An Implementation and Performance Analysis of Spatial Data Access Methods’*, Proc. 5th IEEE Int. Conf. on Data Engineering (ICDE’89), Los Angeles, CA, 1989, pp. 606-615.

[GRG+ 99] Ganti V., Ramakrishnan R., Gehrke J., Powell A., French J.: '*Clustering Large Datasets in Arbitrary Metric Spaces*', Proc. 15th Int. Conf. on Data Engineering (ICDE'99), Sidney, Australia, 1999, pp. 502-511.

[GRS 98] Guha S., Rastogi R., Shim K.: '*CURE: An Efficient Clustering Algorithms for Large Databases*', Proc. ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'98), Seattle, WA, 1998, pp. 73-84.

[Gue 89] Günther O.: *'The Design of the Cell Tree: An Object-Oriented Index Structure for Geometric Databases',* Proc. 5th Int. Conf. on Data Engineering, Los Angeles, CA, 1989, pp. 598-605.

[Gut 84] Guttman A.: *'R-trees: A Dynamic Index Structure for Spatial Searching',* Proc. ACM SIGMOD Int. Conf. on Management of Data, Boston, MA, 1984, pp. 47-57.

[Hen 94] Henrich A.: '*A distance-scan algorithm for spatial access structures*', Proc. 2nd ACM Workshop on Advances in Geographic Information Systems, ACM Press, Gaithersburg, MD, 1994, pp. 136-143.

[Hen 98] Henrich A.: '*The LSD^h-tree: An Access Structure for Feature Vectors*', Proc. 14th Int. Conf. on Data Engineering, Orlando, FL, 1998, pp. 362-369.

[Hin 85] Hinrichs K.: *'Implementation of the Grid File: Design Concepts and Experiance',* BIT 25, pp. 569-592.

[HJR 97] Huang Y.-W., Jing N., Rundensteiner E. A.: '*Spatial Joins Using R-trees: Breadth-First Traversal with Global Optimizations*', Proc. Int. Conf. on Very Large Data Bases (VLDB'97), Athens, Greece, 1997, pp. 396-405.

[HK 98] Hinneburg A., Keim D. A.: '*An Efficient Approach to Clustering in Large Multimedia Databases with Noise*', Proc. 4th Int. Conf. on Knowledge Discovery and Data Mining (KDD'98), New York, NY, 1998, pp. 58-65.

[HK 99] Hinneburg A., Keim D. A.: '*Optimal Grid-Clustering: Towards Breaking the Curse of Dimensionality in High-Dimensional Clustering*', Proc. 25th Int. Conf. on Very Large Data Bases (VLDB'99), Edinburgh, Scotland, 1999, pp. 506-517.

[Hoa 62] C.A.R. Hoare, '*Quicksort*', in: Computer Journal, Vol. 5, No. 1, 1962.

[HS 95] Hjaltason G. R., Samet H.: '*Ranking in Spatial Databases*', Proc. 4th Int. Symp. on Large Spatial Databases (SSD'95), Portland, ME, 1995, pp. 83-95.

[HS 98] Hjaltason G. R., Samet H.: *'Incremental Distance Join Algorithms for Spatial Databases',* Proc. ACM SIGMOD Int. Conf. on Management of Data, Seattle, WA, 1998, pp. 237-248.

[HSW 88a] Hutflesz A., Six H.-W., Widmayer P.: *‘Globally Order Preserving Multidimensional Linear Hashing’*, Proc. 4th IEEE Int. Conf. on Data Engineering, Los Angeles, CA, 1988, pp. 572-579.

[HSW 88b] Hutflesz A., Six H.-W., Widmayer P.: *‘Twin Grid Files: Space Optimizing Acces Schemes’*, Proc. Int. Conf. on Extending Database Technology, Venice, Italy, 1988, pp. 352-363.

[HSW 89] Henrich A., Six H.-W., Widmayer P.: ‘*The LSD-Tree: Spatial Access to Multidimensional Point and Non-Point Objects*’, Proc. 15th Conf. on Very Large Data Bases, Amsterdam, The Netherlands, 1989, pp. 45-53, 1989.

[HT 93] Hattori K., Torii Y.: ‘*Effective algorithms for the nearest neighbor method in the clustering problem*’, Pattern Recognition, 1993, Vol. 26, No. 5, pp. 741-746.

[Jag 90] Jagadish H. V.: *‘Linear Clustering of Objects with Multiple Attributes’*, Proc. ACM SIGMOD Int. Conf. on Management of Data, Atlantic City, NJ, 1990, pp. 332-342.

[Jag 90b] Jagadish H. V.: *‘Spatial Search with Polyhedra’*, Proc. 6th Int. Conf. on Data Engineering, Los Angeles, CA, 1990, pp. 311-319.

[Jag 91] Jagadish H. V.: *‘A Retrieval Technique for Similar Shapes’*, Proc. ACM SIGMOD Int. Conf. on Management of Data, Denver, CO, 1991, pp. 208-217.

[JD 88] Jain A. K., Dubes R. C.: ‘*Algorithms for Clustering Data*’, Prentice-Hall Inc., 1988.

[JW 96] Jain R, White D. A.: *‘Similarity Indexing: Algorithms and Performance’*, Proc. SPIE Storage and Retrieval for Image and Video Databases IV, Vol. 2670, San Jose, CA, 1996, pp. 62-75.

[Kal 86] Kalos M. H., Whitlock P. A.: *‘Monte Carlo Methods’*, Wiley, New York, 1986.

[Kei 97] Keim D. A.: *‘Efficient Similarity Search in Spatial Database Systems’*, habilitation thesis, Institute for Computer Science, University of Munich, 1997.

[KF 93] Kamel I., Faloutsos C.: ‘*On Packing R-trees*’, CIKM, 1993, pp. 490-499.

[KF 94] Kamel I., Faloutsos C.: *‘Hilbert R-tree: An Improved R-tree using Fractals’*. Proc. 20th Int. Conf. on Very Large Data Bases, Santiago de Chile, Chile, 1994, pp. 500-509.

[KH 95] Koperski K., Han J.: ‘*Discovery of Spatial Association Rules in Geographic Information Databases*’, Proc. 4th Int. Symp. on Large Spatial Databases (SSD’95), Portland, ME, 1995, pp. 47-66.

[KKS 98] Kastenmüller G., Kriegel H.-P., Seidl T.: ‘*Similarity Search in 3D Protein Databases*’, Proc. German Conference on Bioinformatics (GCB`98), Köln (Cologne), 1998.

[KM 00] Korn F., Muthukrishnan S.: *‘Influence Sets Based on Reverse Nearest Neighbor Queries’*, Proc. ACM SIGMOD Int. Conf. on Management of Data, Dallas, TX, 2000, pp. 201-212.

[KN 96] Knorr E. M., Ng R. T.: ‘*Finding Aggregate Proximity Relationships and Commonalities in Spatial Data Mining*’, in: IEEE Trans. on Knowledge and Data Engineering, Vol. 8, No. 6, December 1996, pp. 884-897.

[KN 97] Knorr E. M., Ng R. T.: ‘A *Unified Notion of Outliers: Properties and Computation*’, Proc. 4th Int. Conf. on Knowledge Discovery and Data Mining (KDD’97), Newport Beach, CA, 1997, pp. 219-222.

[KN 98] Knorr E. M., Ng R. T.: ‘*Algorithms for Mining Distance-Based Outliers in Large Datasets*’, Proc. 24th Int. Conf. on Very Large Data Bases (VLDB’98), New York, NY, 1998, pp. 392-403.

[KN 99] Knorr E. M., Ng R. T.: ‘*Finding Intensional Knowledge of Distance-Based Outliers*’, Proc. 25th Int. Conf. on Very Large Data Bases (VLDB’99), Edinburgh, Scotland, 1999, pp. 211-222.

[KNT 00] Knorr E. M., Ng R. T., Tucakov V.: ‘*Distance-based outliers: algorithms and applications*’, in: The VLDB Journal, Vol. 8, 2000, pp. 237-253.

[Knu 75] Knuth D.: *´The Art of Computer Programming, Vol. 3*’, Addison-Wesley, 1975.

[KR 90] Kaufman L., Rousseeuw P. J.: ‘*Finding Groups in Data: An Introduction to Cluster Analysis*’, John Wiley & Sons, 1990.

[KS 97] Koudas N., Sevcik C.: ‘*Size Separation Spatial Join*’, Proc. ACM SIGMOD Int. Conf. on Management of Data, Tucson, AZ, 1997, pp. 324-335.

[KS 98] Koudas N., Sevcik C.: ‘*High Dimensional Similarity Joins: Algorithms and Performance Evaluation*’, Proc. 14th Int. Conf on Data Engineering (ICDE’98), Best Paper Award, Orlando, FL, 1998, pp. 466-475.

[KS 97b] Katayama N., Satoh S.: *‘The SR-tree: An Index Structure for High-Dimensional Nearest Neighbor Queries’*, Proc. ACM SIGMOD Int. Conf. on Management of Data, Tucson, AZ, 1997, pp. 369-380.

[KS 98b] Kriegel H.-P., Seidl T.: ‘*Approximation-Based Similarity Search for 3-D Surface Segments*’, in: GeoInformatica Journal, Kluwer Academic Publishers, Vol. 2, No. 2, 1998, pp. 113-147.

[KSF+ 96] Korn F., Sidiropoulos N., Faloutsos C., Siegel E., Protopapas Z.: *'Fast Nearest Neighbor Search in Medical Image Databases',* Proc. 22nd Int. Conf. on Very Large Data Bases (VLDB'96), Mumbai (Bombay), India, 1996, pp. 215-226.

[KSS 97] Kriegel H.-P., Schmidt T., Seidl T.: *'3D Similarity Search by Shape Approximation',* Proc. 5th Int. Symp. on Large Spatial Data Bases (SSD'97), Berlin, Germany, in: Lecture Notes in Computer Science, Vol. 1262, 1997, pp.11-28.

[Kuk 92] Kukich K.: '*Techniques for Automatically Correcting Words in Text*', ACM Computing Surveys, Vol. 24, No. 4, 1992, pp. 377-440.

[KW 85] Krishnamurthy R., Whang K.-Y.: *'Multilevel Grid Files',* IBM Research Center Report, Yorktown Heights, N.Y., 1985.

[LJF 95] Lin K., Jagadish H. V., Faloutsos C.: *'The TV-Tree: An Index Structure for High-Dimensional Data',* in: VLDB Journal, Vol. 3, pp. 517-542, 1995.

[LR 94] Lo M.-L., Ravishankar C. V.: '*Spatial Joins Using Seeded Trees*', Proc. ACM SIGMOD Int. Conf. on Management of Data, Minneapolis, MN, 1994, pp. 209-220.

[LR 96] Lo M.-L., Ravishankar C. V.: '*Spatial Hash Joins*', Proc. ACM SIGMOD Int. Conf. on Management of Data , Montreal, Canada, 1996, pp. 247-258.

[LS 89] Lomet D., Salzberg B.: *'The hB-tree: A Robust Multiattribute Search Structure',* Proc. 5th IEEE Int. Conf. on Data Engineering (ICDE'89), Los Angeles, CA, 1989, pp. 296-304.

[LS 90] Lomet D., Salzberg B.: *'*The *hB-tree: A Multiattribute Indexing Method with Good Guaranteed Performance',* in. ACM Trans. on Data Base Systems, Vol. 15, No. 4, 1990, pp. 625-658.

[Man 77] Mandelbrot B.: '*Fractal Geometry of Nature*', W. H. Freeman and Company, New York, 1977.

[MCP 93] Matheus C. J., Chan P. K., Piatetsky-Shapiro G.: *"Systems for Knowledge Discovery in Databases"*, IEEE Trans, on Knowledge and Data Engineering, Vol. 5, No. 6, 1993, pp. 903-913.

[McQ 67] McQueen J.: '*Some Methods for Classification and Analysis of Multivariate Observation'*, Proc. 5th Berkeley Symp. on Math. Statist. and Prob., Vol. 1, 1965, pp. 281-297.

[MG 93] Mehrotra R., Gary J.: *'Feature-Based Retrieval of Similar Shapes',* Proc. 9th Int. Conf. on Data Engineering, Vienna, Austria, 1993, pp. 108-115.

[MG 95] Mehrotra R., Gary J.: *'Feature-Index-Based Sililar Shape retrieval',* Proc. of the 3rd Working Conf. on Visual Database Systems (VDB'95), Lausanne, Switzerland, 1995, pp. 46-65.

[Mit 97] Mitchel T.: "*Machine Learning*", McGraw-Hill, 1997.

[Mor 66] Morton G.: *‘A Computer Oriented Geodetic Data BAse and a New Technique in File Sequencing’*, IBM Ltd., 1966.

[Mul 71] Mullin, J. K.: ‘Retrieval-Update Speed Tradeoffs Using Combined Indices’, in: Communications of the ACM, Vol. 14, No. 12, December 1971, pp. 775-776.

[Mur 83] Murtagh F.: ‘*A Survey of Recent Advances in Hierarchical Clustering Algorithms*’, in: The Computer Journal, Vol. 26, No. 4, 1983, pp. 354-359.

[NBE 93] Niblack W., Barber R., Equitz W., Flickner M., Glasmann E., Petkovic D., Yanker P., Faloutsos C., Taubin G.: ‘*The QBIC Project: Querying Images by Content Using Color, Texture, and Shape*’, Proc. SPIE 1993 Int. Symp. on Electronic Imaging: Science and Technology Conference 1908, Storage and Retrieval for Image and Video Databases, San Jose, CA, 1993.

[NH 94] Ng R. T., Han J.: ‘*Efficient and Effective Clustering Methods for Spatial Data Mining*’, Proc. 20th Int. Conf. on Very Large Data Bases (VLDB’94), Santiago de Chile, Chile, San Francisco, CA, 1994, pp. 144-155.

[NHS 84] Nievergelt J., Hinterberger H., Sevcik K. C.: *‘The Grid File: An Adaptable, Symmetric Multikey File Structure’*, in: ACM Trans. on Database Systems, Vol. 9, No. 1, 1984, pp. 38-71.

[OM 84] Orenstein J., Merret T. H.: *‘A Class of Data Structures for Associative Searching’*, Proc. 3rd ACM SIGACT-SIGMOD Symp. on Principles of Database Systems, 1984, pp. 181-190.

[Ore 82] Orenstein J. A.: ‘*Multidimensional tries used for associative searching*’, Inf. Proc. Letters, Vol. 14, No. 4, 1982, pp. 150-157.

[Ore 89] Orenstein J. A.: *‘Redundancy in Spatial Databases’*, Proc. ACM SIGMOD Int. Conf. on Management of Data, Portland, OR, 1989, pp. 294-305.

[Ore 90] Orenstein J. A., : *‘A comparison of spatial query processing techniques for native and parameter spaces’*, Proc. ACM SIGMOD Int. Conf. on Management of Data, Atlantic City, NJ, 1990, pp. 326-336.

[Ore 91] Orenstein J. A.: *‘An Algorithm for Computing the Overlay of k-Dimensional Spaces’*, Proc. Symp. on Large Spatial Databases (SSD’91), Zurich, Switzerland, 1991, pp. 381-400.

[Oto 84] Otoo, E. J.: *‘A Mapping Function for the Directory of a Multidimensional Extendible Hashing’*, Proc. 10th. Int. Conf. on Very Large Data Bases, Singapore, 1984, pp. 493-506.

[Ouk 85] Ouksel M.: *‘The Interpolation Based Grid File’*, Proc 4th ACM SIGACT/SIGMOD Symp. on Principles of Database Systems, 1985, pp. 20-27.

[PD 96] Patel J. M., DeWitt D. J., ‘*Partition Based Spatial-Merge Join*’, Proc. ACM SIGMOD Int. Conf. on Management of Data, Montreal, Canada, 1996, pp. 259-270.

[PFTV 88] Press W., Flannery B. P., Teukolsky S. A., Vetterling W. T.: *‘Numerical Recipes in C’*, Cambridge University Press, 1988.

[PH 90] Patterson D. A., Hennessy J. L.: *‘Computer Architecture: A Quantitative Approach’*, Morgan Kaufman, 1990.

[PM 97] Papadopoulos A., Manolopoulos Y.: ‘*Performance of Nearest Neighbor Queries in R-Trees*’, Proc. 6th Int. Conf. on Database Theory, Delphi, Greece, in: Lecture Notes in Computer Science, Vol. 1186, Springer, 1997, pp. 394-408.

[PS 85] Preparata F. P., Shamos M. I.: ‘Computational Geometry’, Chapter 5 (‘Proximity: Fundamental Algorithms’), Springer Verlag New York, 1985, pp. 185-225.

[PSTW 93] Pagel B.-U., Six H.-W., Toben H., Widmayer P.: *‘Towards an Analysis of Range Query Performance in Spatial Data Structures’*, Proc. 12th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, (PODS’93), Washington DC, 1993, pp. 214-221.

[Qui 86] Quinlan J. R.: ‘*Induction of Decision Trees*’, in: Machine Learning Journal, Vol. 1, 1986, pp. 81-106.

[RKV 95] Roussopoulos N., Kelley S., Vincent F.: ‘*Nearest Neighbor Queries*’, Proc. ACM SIGMOD Int. Conf. on Management of Data, San Jose, CA, 1995, pp. 71-79.

[Rob 81] Robinson J. T.: *‘The K-D-B-tree: A Search Structure for Large Multidimensional Dynamic Indexes’*, Proc. ACM SIGMOD Int. Conf. on Management of Data, Ann Arbor, MI, 1981, pp. 10-18.

[RP 92] Ramasubramanian V., Paliwal K. K.: ‘*Fast k-Dimensional Tree Algorithms for Nearest Neighbor Search with Application to Vector Quantization Encoding*’, IEEE Trans. on Signal Processing, Vol. 40, No. 3, March 1992, pp. 518-531.

[SA 97] Shafer J. C., Agrawal R.: *‘Parallel Algorithms for High-dimensional Proximity Joins*, Proc. 23rd Int. Conf. on Very Large Data Bases, Athens, Greece, 1997, pp. 176-185.

[Sag 94] Sagan H.: *‘Space Filling Curves’*, Springer-Verlag Berlin/Heidelberg/New York, 1994.

[Sch 91] Schröder M.: ‘*Fractals, Chaos, Power Laws: Minutes from an Infinite Paradise*’, W. H. Freeman and Company, New York, 1991.

[Sch 95] Schiele O. H.: ‘*Forschung und Entwicklung im Maschinenbau auf dem Weg in die Informationsgesellschaft*’ (in German, translation by the author), Bundesministerium für Bildung, Wissenschaft, Forschung und Technologie, Frankfurt am Main, Germany, 1995, http://www.iid.de/informationen/vdma/infoway3.html.

[SCZ 98] Sheikholeslami G., Chatterjee S., Zhang A.: ‘*WaveCluster: A Multi-Resolution Clustering Approach for Very Large Spatial Databases*’, Proc. 24th Int. Conf. on Very Large Data Bases (VLDB’98), New York, NY, 1998, pp. 428-439.

[Sed 78] Sedgewick R.: ‘*Quicksort*’, Garland, New York, 1978.

[See 91] Seeger B.: ‘*Multidimensional Access Methods and their Applications*’, Tutorial, 1991.

[Sei 97] Seidl T.: ‘*Adaptable Similarity Search in 3-D Spatial Database Systems’*, Ph.D. Thesis, Faculty for Mathematics and Computer Science, University of Munich, 1997.

[Sei 01] Seidl T. ´’ Habilitation Thesis, University of Munich, 2001.

[SEKX 98] Sander J., Ester M., Kriegel H.-P., Xu X.: ‘*Density-Based Clustering in Spatial Databases: The Algorithm GDBSCAN and its Applications*’, in: Data Mining and Knowledge Discovery, An International Journal, Vol. 2, No. 2, Kluwer Academic Publishers, Norwell, MA, 1998, pp. 169-194.

[SH 94] Shawney H., Hafner J.: ‘*Efficient Color Histogram Indexing*’, Proc. Int. Conf. on Image Processing, 1994, pp. 66-70.

[Sib 72] Sibson R.: ‘*SLINK: an optimally efficient algorithm for the single-link cluster method*’, in: The Comp. Journal, Vol. 16, No. 1, 1972, pp. 30-34.

[Sie 90] Sierra H. M.: *‘An Introduction do Direct Access Storage Devices’,* Academic Press, 1990.

[SK 90] Seeger B., Kriegel H.-P.: *‘The Buddy Tree: An Efficient and Robust Access Method for Spatial Data Base Systems’,* Proc. 16th Int. Conf. on Very Large Data Bases (VLDB’90), Brisbane, Australia, 1990, pp. 590-601.

[SK 97] Seidl T., Kriegel H.-P.: *‘Efficient User-Adaptable Similarity Search in Large Multimedia Databases’,* Proc. 23rd Int. Conf. on Very Large Databases (VLDB’97), Athens, Greece, 1997, pp. 506-515.

[SML 00] Shin H., Moon B., Lee S.: *‘Adaptive Multi-Stage Distance Join Processing’,* Proc. ACM SIGMOD Int. Conf. on Management of Data, Dallas, TX, 2000, pp. 343-354.

[SML 01] Shin H., Moon B., Lee S.: ‘*Adaptive and Incremental Processing for Distance Join Queries*’, submitted for publication 2001.

[SPG 91] Silberschatz A., Peterson J., Galvin P.: ‘*Operating Systems Concepts*’, third edition, Addison-Wesley, 1991.

[Spr 91] Sproull R. F.: ‘*Refinements to Nearest Neighbor Searching in k-Dimensional Trees*’, Algorithmica, 1991, pp. 579-589.

[SRF 87] Sellis T., Roussopoulos N., Faloutsos C.: ‘*The R+-Tree: A Dynamic Index for Multi-Dimensional Objects*’, Proc. 13th Int. Conf. on Very Large Data Bases (VLDB’87), Brighton, England, 1987, pp. 507-518.

[SSA 97] Shim K., Srikant R., Agrawal R.: ‘*High-dimensional Similarity Joins*’, Proc. Int. Conf. on Data Engineering (ICDE’97), Birmingham, U.K., 1997, pp. 301-311.

[SSH 86] Stonebreaker M., Sellis T., Hanson E.: ‘*An Analysis of Rule Indexing Implementations in Data Base Systems*’, Proc. 1st Int. Conf. on Expert Data Base Systems, 1986.

[Str 80] Strang G.: ‘*Linear Algebra and its Applications*’, 2nd edition, Academic Press, 1980.

[TC 91] Taubin G., Cooper D. B.: ‘*Recognition and Positioning of Rigid Objects Using Algebraic Moment Invariants*’, in *Geometric Methods in Computer Vision*, Vol. 1570, SPIE, 1991, pp. 175-186.

[TS 96] Theodoridis Y., Sellis T. K.: ‘*A Model for the Prediction of R-tree Performance*’, Proc. 15th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, Montreal, Canada, 1996, pp. 161-171, ACM Press, ISBN 0-89791-781-2.

[Uhl 91] Uhlmann J. K., ‘*Satisfying General Proximity/Similarity Queries with Metric Trees*’, Information Processing Letters, Vol. 40, 1991, pp. 175-179.

[Ull 89] Ullman J. D.: ‘*Database and Knowledge-Base System*’, Vol. II, Computer Science Press, Rockville, MD, 1989.

[VCVS 95] Valdes F. G., Campusano L. E., Velasquez J. D., Stetson P. B.: ‘*FOCAS Automatic Catalogue Matching*’, Publications of the Astronomical Society of the Pacific, Vol. 107, 1995, p. 1119.

[Wel 71] Welch T.: ‘*Bounds on the Information Retrieval Efficiency of Static File Structures*’, Technical Report 88, MIT, 1971.

[WJ 96] White D. A., Jain R.: ‘*Similarity indexing with the SS-tree*’, Proc. 12th Int. Conf on Data Engineering (ICDE’96), New Orleans, LA, 1996, pp. 516-523.

[WSB 98] Weber R., Schek H.-J., Blott S.: ‘*A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces*’, Proc. 24th Int. Conf. on Very Large Data Bases (VLDB’98), New York, NY, 1998, pp. 194-205, Morgan Kaufmann Publishers, San Francisco, CA.

[WW 80] Wallace T., Wintz P.: *‘An Efficient Three-Dimensional Aircraft Recognition Algorithm Using Normalized Fourier Descriptors’*, Computer Graphics and Image Processing, Vol. 13, 1980, pp. 99-126.

[Yia 93] Yiannilos P. N., ‘*Data Structures an Algorithms for Nearest Neighbor Search in General Metric Spaces’*, Proc. ACM-SIAM Symp. on Discrete Algorithms, 1993, pp. 311-321.

[YY 85] Yao A. C., Yao F. F.: ‘*A General Approach to D-Dimensional Geometric Queries*’, Proc. ACM Symp. on Theory of Computing, 1985.

[WYM 97] Wang W., Yang J., Muntz R.: ‘*STING: A Statistical Information Grid Approach to Spatial Data Mining*’, Proc. 23th Int. Conf. on Very Large Data Bases (VLDB’97), Athens, Greece, 1997, pp. 186-195, Morgan Kaufmann Publishers, San Francisco, CA.

[ZRL 96] Zhang T., Ramakrishnan R., Linvy M.: ‘*BIRCH: An Efficient Data Clustering Method for Very Large Databases*’, Proc. ACM SIGMOD Int. Conf. on Management of Data , 1996, pp. 103-114, ACM Press, New York.

Curriculum Vitae

Christian Böhm was born on September 28, 1968 in Rosenheim, Germany. He attended primary and secondary school from 1975 to 1988.

He entered the *Technische Universität München* (*TUM*) in November 1988 for his study in Computer Science. During this time, he worked as a self-employed software engineer and consultant for various companies. In April 1994, he passed the final examination with distinction and received the diploma degree. His diploma thesis was titled '*Management of Biological Sequence Data in an Object-Oriented Database System*' (in German) which was supervised by Professor R. Bayer, Ph.D. and Professor Dr. J. Christoph Freytag of *Digital Equipment* (*DEC*).

In July 1994, he entered the research group for knowledge bases of the *FORWISS* institute (*Bayerisches Forschungszentrum für wissensbasierte Systeme*) where he was responsible for a nation-wide digital library project.

In January 1996, he transferred to the *Ludwig-Maximilians-Universität München* (*LMU*) where he is working as a research and teaching assistant with Professor Dr. Hans-Peter Kriegel, the chair of the teaching and research unit for database systems at the Institute for Computer Science of the *LMU*. Böhm received his doctoral degree in December, 1998. He gained the SIGMOD Best-Paper-Award in 1997 for a joint publication with Dr. Stefan Berchtold, Bernhard Braunmüller, Professor Dr. Daniel Keim and Professor Dr. Hans-Peter Kriegel.